

У. Р. Стивенс,
Б. Феннер, Э. М. Рудофф



UNIX разработка сетевых приложений

3-е издание



Addison-Wesley

 ПИТЕР®

Annotation

Новое издание книги, посвященной созданию веб-серверов, клиент-серверных приложений или любого другого сетевого программного обеспечения в операционной системе UNIX, — классическое руководство по сетевым программным интерфейсам, в частности сокетам. Оно основано на трудах Уильяма Стивенса и полностью переработано и обновлено двумя ведущими экспертами по сетевому программированию. В книгу включено описание ключевых современных стандартов, реализаций и методов, она содержит большое количество иллюстрирующих примеров и может использоваться как учебник по программированию в сетях, так и в качестве справочника для опытных программистов.

- [У. Р. Стивенс, Б. Феннер, Э. М. Рудофф](#)
 - [Вступительное слово](#)
 - [Предисловие](#)
 - [Часть 1](#)
 - [Глава 1](#)
 - [1.1. Введение](#)
 - [1.2. Простой клиент времени и даты](#)
 - [1.3. Независимость от протокола](#)
 - [1.4. Обработка ошибок: функции-обертки](#)
 - [Значение системной переменной Unix `errno`](#)
 - [1.5. Простой сервер времени и даты](#)
 - [1.6. Таблица соответствия примеров технологии клиент-сервер](#)
 - [1.7. Модель OSI](#)
 - [1.8. История сетевого обеспечения BSD](#)
 - [1.9. Сети и узлы, используемые в примерах](#)
 - [Определение топологии сети](#)
 - [1.10. Стандарты Unix](#)
 - [История POSIX](#)
 - [История Open Group](#)
 - [Объединение стандартов](#)
 - [Internet Engineering Task Force](#)
 - [1.11. 64-разрядные архитектуры](#)
 - [1.12. Резюме](#)
 - [Упражнения](#)
 - [Глава 2](#)
 - [2.1. Введение](#)
 - [2.2. Обзор протоколов TCP/IP](#)
 - [2.3. UDP: протокол пользовательскихдейтаграмм](#)
 - [2.4. TCP: протокол контроля передачи](#)
 - [2.5. SCRIPT: протокол управления передачей потоков](#)
 - [2.6. Установление и завершение соединения TCP](#)
 - [Трехэтапное рукопожатие](#)
 - [Параметры TCP](#)
 - [Завершение соединения TCP](#)
 - [Диаграмма состояний TCP](#)
 - [Обмен пакетами](#)
 - [2.7. Состояние TIME_WAIT](#)
 - [2.8. Установление и завершение ассоциации SCRIPT](#)
 - [Четырехэтапное рукопожатие](#)

- [Завершение ассоциации](#)
- [Диаграмма состояний SCRIPT](#)
- [Обмен пакетами](#)
- [Параметры SCRIPT](#)
- [**2.9. Номера портов**](#)
 - [Пара сокетов](#)
- [**2.10. Номера портов TCP и параллельные серверы**](#)
- [**2.11. Размеры буфера и ограничения**](#)
 - [Отправка по TCP](#)
 - [Отправка по UDP](#)
 - [Отправка по SCRIPT](#)
- [**2.12. Стандартные службы Интернета**](#)
- [**2.13. Использование протоколов типичными приложениями Интернета**](#)
- [**2.14. Резюме**](#)
- [Упражнения](#)
- [**Часть 2**](#)
 - [**Глава 3**](#)
 - [3.1. Введение](#)
 - [3.2. Структуры адреса сокетов](#)
 - [Структура адреса сокета IPv4](#)
 - [Универсальная структура адреса сокета](#)
 - [Структура адреса сокета IPv6](#)
 - [Новая универсальная структура адреса сокета](#)
 - [Сравнение структур адреса сокетов](#)
 - [3.3. Аргументы типа «значение-результат»](#)
 - [3.4. Функции определения порядка байтов](#)
 - [3.5. Функции управления байтами](#)
 - [3.6. Функции `inet_aton`, `inet_addr` и `inet_ntoa`](#)
 - [3.7. Функции `inet_pton` и `inet_ntop`](#)
 - [Пример](#)
 - [3.8. Функция `sock_ntop` и связанные с ней функции](#)
 - [3.9. Функции `readn`, `writen` и `readline`](#)
 - [3.10. Резюме](#)
 - [Упражнения](#)
 - [**Глава 4**](#)
 - [4.1. Введение](#)
 - [4.2. Функция `socket`](#)
 - [AF_xxx и PF_xxx](#)
 - [4.3. Функция `connect`](#)
 - [4.4. Функция `bind`](#)
 - [4.5. Функция `listen`](#)
 - [4.6. Функция `accept`](#)
 - [Пример: аргументы типа «значение-результат»](#)
 - [4.7. Функции `fork` и `exec`](#)
 - [4.8. Параллельные серверы](#)
 - [4.9. Функция `close`](#)
 - [Счетчик ссылок дескриптора](#)
 - [4.10. Функции `getsockname` и `getpeername`](#)
 - [Сравнение функций](#)

- [Пример: получение семейства адресов сокета](#)
- [4.11. Резюме](#)
- [Упражнения](#)
- [Глава 5](#)
 - [5.1. Введение](#)
 - [5.2. Эхо-сервер TCP: функция main](#)
 - [5.3. Эхо-сервер TCP: функция str_echo](#)
 - [5.4. Эхо-клиент TCP: функция main](#)
 - [5.5. Эхо-клиент TCP: функция str_cli](#)
 - [5.6. Нормальный запуск](#)
 - [5.7. Нормальное завершение](#)
 - [5.8. Обработка сигналов POSIX](#)
 - [Функция signal](#)
 - [Семантика сигналов POSIX](#)
 - [5.9. Обработка сигнала SIGCHLD](#)
 - [Обработка зомбиированных процессов](#)
 - [Обработка прерванных системных вызовов](#)
 - [5.10. Функции wait и waitpid](#)
 - [Различия между функциями wait и waitpid](#)
 - [5.11. Прерывание соединения перед завершением функции accept](#)
 - [5.12. Завершение процесса сервера](#)
 - [5.13. Сигнал SIGPIPE](#)
 - [5.14. Сбой на узле сервера](#)
 - [5.15. Сбой и перезагрузка на узле сервера](#)
 - [5.16. Выключение узла сервера](#)
 - [5.17. Итоговый пример TCP](#)
 - [5.18. Формат данных](#)
 - [Пример: передача текстовых строк между клиентом и сервером](#)
 - [Пример: передача двоичных структур между клиентом и сервером](#)
 - [5.19. Резюме](#)
 - [Упражнения](#)
- [Глава 6](#)
 - [6.1. Введение](#)
 - [6.2. Модели ввода-вывода](#)
 - [Модель блокируемого ввода-вывода](#)
 - [Модель неблокируемого ввода-вывода](#)
 - [Модель мультиплексирования ввода-вывода](#)
 - [Модель ввода-вывода, управляемого сигналом](#)
 - [Модель асинхронного ввода-вывода](#)
 - [Сравнение моделей ввода-вывода](#)
 - [Сравнение синхронного и асинхронного ввода-вывода](#)
 - [6.3. Функция select](#)
 - [При каких условиях дескриптор становится готовым?](#)
 - [Максимальное число дескрипторов для функции select](#)
 - [6.4. Функция str_cli \(продолжение\)](#)
 - [6.5. Пакетный ввод](#)
 - [6.6. Функция shutdown](#)
 - [6.7. Функция str_cli \(еще раз\)](#)
 - [6.8. Эхо-сервер TCP \(продолжение\)](#)
 - [При каких условиях дескриптор становится готовым?](#)
 - [Максимальное число дескрипторов для функции select](#)

- [Атака типа «отказ в обслуживании»](#)
- [6.9. Функция pselect](#)
- [6.10. Функция poll](#)
- [6.11. Эхо-сервер TCP \(еще раз\)](#)
- [6.12. Резюме](#)
- [Упражнения](#)
- **Глава 7**
 - [7.1. Введение](#)
 - [7.2. Функции getsockopt и setsockopt](#)
 - [7.3. Проверка наличия параметра и получение значения по умолчанию](#)
 - [7.4. Состояния сокетов](#)
 - [7.5. Общие параметры сокетов](#)
 - [Параметр сокета SO_BROADCAST](#)
 - [Параметр сокета SO_DEBUG](#)
 - [Параметр сокета SO_DONTROUTE](#)
 - [Параметр сокета SO_ERROR](#)
 - [Параметр сокета SO_KEEPALIVE](#)
 - [Параметр сокета SO_LINGER](#)
 - [Параметр сокета SO_OOBINLINE](#)
 - [Параметры сокета SO_RCVBUF и SO_SNDBUF](#)
 - [Параметры сокета SO_RCVLOWAT и SO SNDLOWAT](#)
 - [Параметры сокета SO_RCVTIMEO и SO_SNDTIMEO](#)
 - [Параметры сокета SO_REUSEADDR и SO_REUSEPORT](#)
 - [Параметр сокета SO_TYPE](#)
 - [Параметр сокета SO_USELOOPBACK](#)
 - [7.6. Параметры сокетов IPv4](#)
 - [Параметр сокета IP_HRDINCL](#)
 - [Параметр сокета IP_OPTIONS](#)
 - [Параметр сокета IP_RECVDSTADDR](#)
 - [Параметр сокета IP_RECVIF](#)
 - [Параметр сокета IP_TOS](#)
 - [Параметр сокета IP_TTL](#)
 - [7.7. Параметр сокета ICMPv6](#)
 - [Параметр сокета ICMP6_FILTER](#)
 - [7.8. Параметры сокетов IPv6](#)
 - [Параметр сокета IPV6_CHECKSUM](#)
 - [Параметр сокета IPV6_DONTFRAG](#)
 - [Параметр сокета IPV6_NEXTHOP](#)
 - [Параметр сокета IPV6_PATHMTU](#)
 - [Параметр сокета IPV6_RECVDSTOPTS](#)
 - [Параметр сокета IPV6_RECVHOPLIMIT](#)
 - [Параметр сокета IPV6_RECVHOPOPTS](#)
 - [Параметр сокета IPV6_RECVPATHMTU](#)
 - [Параметр сокета IPV6_RECVPKTINFO](#)
 - [Параметр сокета IPV6_RECVRTHDR](#)
 - [Параметр сокета IPV6_RECVTCLASS](#)
 - [Параметр сокета IPV6_UNICAST_HOPS](#)
 - [Параметр сокета IPV6_USE_MIN_MTU](#)
 - [Параметр сокета IPV6_V6ONLY](#)
 - [Параметры сокета IPV6_XXX](#)
 - [7.9. Параметры сокетов TCP](#)
 -

- [Параметр сокета TCP_MAXSEG](#)
- [Параметр сокета TCP_NODELAY](#)
- [**7.10. Параметры сокетов SCTP**](#)
 - [Параметр сокета SCTP_ADAPTION_LAYER](#)
 - [Параметр сокета SCTP_ASSOCINFO](#)
 - [Параметр сокета SCTP_AUTOCLOSE](#)
 - [Параметр сокета SCTP_DEFAULT_SEND_PARAM](#)
 - [Параметр сокета SCTP_DISABLE_FRAGMENTS](#)
 - [Параметр сокета SCTP_EVENTS](#)
 - [Параметр сокета SCTP_GET_PEER_ADDR_INFO](#)
 - [Параметр сокета SCTP_I_WANT_MAPPED_V4_ADDR](#)
 - [Параметр сокета SCTP_INITMSG](#)
 - [Параметр сокета SCTP_MAXBURST](#)
 - [Параметр сокета SCTP_MAXSEG](#)
 - [Параметр сокета SCTP_NODELAY](#)
 - [Параметр сокета SCTP_PEER_ADDR_PARAMS](#)
 - [Параметр сокета SCTP_PRIMARY_ADDR](#)
 - [Параметр сокета SCTP_RTOINFO](#)
 - [Параметр сокета SCTP_SET_PEER_PRIMARY_ADDR](#)
 - [Параметр сокета SCTP_STATUS](#)
- [7.11. Функция fcntl](#)
- [7.12. Резюме](#)
- [Упражнения](#)
- [**Глава 8**](#)
 - [8.1. Введение](#)
 - [8.2. Функции recvfrom и sendto](#)
 - [8.3. Эхо-сервер UDP: функция main](#)
 - [8.4. Эхо-сервер UDP: функция dg_echo](#)
 - [8.5. Эхо-клиент UDP: функция main](#)
 - [8.6. Эхо-клиент UDP: функция dg_cli](#)
 - [8.7. Потерянные дейтаграммы](#)
 - [8.8. Проверка полученного ответа](#)
 - [8.9. Запуск клиента без запуска сервера](#)
 - [8.10. Итоговый пример клиент-сервера UDP](#)
 - [8.11. Функция connect для UDP](#)
 - [Многократный вызов функции connect для сокета UDP](#)
 - [Производительность](#)
 - [8.12. Функция dg_cli \(продолжение\)](#)
 - [8.13. Отсутствие управления потоком в UDP](#)
 - [Приемный буфер сокета UDP](#)
 - [8.14. Определение исходящего интерфейса для UDP](#)
 - [8.15. Эхо-сервер TCP и UDP, использующий функцию select](#)
 - [8.16. Резюме](#)
 - [Упражнения](#)
- [**Глава 9**](#)
 - [9.1. Введение](#)
 - [9.2. Модели интерфейса](#)
 - [Сокет типа «один-к-одному»](#)
 - [Сокет типа «один-ко-многим»](#)
 - [9.3. Функция sctp_bindx](#)
 - [9.4. Функция sctp_connectx](#)
 - [9.5. Функция sctp_getpaddrs](#)

- [9.6. Функция sctp_freepaddrs](#)
- [9.7. Функция sctp_getladdrs](#)
- [9.8. Функция sctp_freeladdrs](#)
- [9.9. Функция sctp_sendmsg](#)
- [9.10. Функция sctp_recvmsg](#)
- [9.11. Функция sctp_opt_info](#)
- [9.12. Функция sctp_peeloff](#)
- [9.13. Функция shutdown](#)
- [9.14. Уведомления](#)
- [9.15. Резюме](#)
- [Упражнения](#)
- [Глава 10](#)
 - [10.1. Введение](#)
 - [10.2. Потоковый эхо-сервер SCTP типа «один-ко-многим»: функция main](#)
 - [10.3. Потоковый эхо-клиент SCTP типа «один-ко-многим»: функция main](#)
 - [10.4. Потоковый эхо-клиент SCTP: функция str_cli](#)
 - [Запуск программы](#)
 - [10.5. Блокирование очереди](#)
 - [Запуск программы](#)
 - [Запуск измененной программы](#)
 - [10.6. Управление количеством потоков](#)
 - [10.7. Управление завершением соединения](#)
 - [10.8. Резюме](#)
 - [Упражнения](#)
- [Глава 11](#)
 - [11.1. Введение](#)
 - [11.2. Система доменных имен](#)
 - [Записи ресурсов](#)
 - [Распознаватели и серверы имен](#)
 - [Альтернативы DNS](#)
 - [11.3. Функция gethostbyname](#)
 - [Пример](#)
 - [11.4. Функция gethostbyaddr](#)
 - [11.5. Функции getservbyname и getservbyport](#)
 - [Пример: использование функций gethostbyname и getservbyname](#)
 - [11.6. Функция getaddrinfo](#)
 - [11.7. Функция gai_strerror](#)
 - [11.8. Функция freeaddrinfo](#)
 - [11.9. Функция getaddrinfo: IPv6](#)
 - [11.10. Функция getaddrinfo: примеры](#)
 - [11.11. Функция host_serv](#)
 - [11.12. Функция tcp_connect](#)
 - [Пример: клиент времени и даты](#)
 - [11.13. Функция tcp_listen](#)
 - [Пример: сервер времени и даты](#)
 - [Пример: сервер времени и даты с указанием протокола](#)
 - [11.14. Функция udp_client](#)
 - [Пример: не зависящий от протокола UDP-клиент времени и даты](#)

- [11.15. Функция `udp_connect`](#)
- [11.16. Функция `udp_server`](#)
 - [Пример: не зависящий от протокола UDP-сервер времени и даты](#)
- [11.17. Функция `getnameinfo`](#)
- [11.18. Функции, допускающие повторное вхождение](#)
- [11.19. Функции `gethostbyname_r` и `gethostbyaddr_r`](#)
- [11.20. Устаревшие функции поиска адресов IPv6](#)
 - [Константа `RES_USE_INET6`](#)
 - [Функция `gethostbyname2`](#)
 - [Функция `getipnodebyname`](#)
- [11.21. Другая информация о сетях](#)
- [11.22. Резюме](#)
- [Упражнения](#)
- [Часть 3](#)
 - [Глава 12](#)
 - [12.1. Введение](#)
 - [12.2. Клиент IPv4, сервер IPv6](#)
 - [12.3. Клиент IPv6, сервер IPv4](#)
 - [Резюме: совместимость IPv4 и IPv6](#)
 - [12.4. Макроопределения проверки адреса IPv6](#)
 - [12.5. Переносимость исходного кода](#)
 - [12.6. Резюме](#)
 - [Упражнения](#)
 - [Глава 13](#)
 - [13.1. Введение](#)
 - [13.2. Демон `syslogd`](#)
 - [13.3. Функция `syslog`](#)
 - [13.4. Функция `daemon_init`](#)
 - [Пример: сервер времени и даты в качестве демона](#)
 - [13.5. Демон `inetd`](#)
 - [13.6. Функция `daemon_inetd`](#)
 - [Пример: сервер времени и даты, активизированный демоном `inetd`](#)
 - [13.7. Резюме](#)
 - [Упражнения](#)
 - [Глава 14](#)
 - [14.1. Введение](#)
 - [14.2. Тайм-ауты сокета](#)
 - [Тайм-аут для функции `connect` \(сигнал `SIGALRM`\)](#)
 - [Тайм-аут для функции `recvfrom` \(сигнал `SIGALRM`\)](#)
 - [Тайм-аут для функции `recvfrom` \(функция `select`\)](#)
 - [Тайм-аут для функции `recvfrom` \(параметр сокета `SO_RCVTIMEO`\)](#)
 - [14.3. Функции `recv` и `send`](#)
 - [14.4. Функции `ready` и `writen`](#)
 - [14.5. Функции `recvmsg` и `sendmsg`](#)
 - [14.6. Вспомогательные данные](#)
 - [14.7. Сколько данных находится в очереди?](#)
 - [14.8. Сокеты и стандартный ввод-вывод](#)
 - [Пример: функция `str_echo`, использующая стандартный ввод-вывод](#)
 - [14.9. Расширенный опрос](#)

- [Интерфейс /dev/poll](#)
 - [Интерфейс kqueue](#)
 - [Рекомендации](#)
 - [14.10. Резюме](#)
 - [Упражнения](#)
- [Глава 15](#)
 - [15.1. Введение](#)
 - [15.2. Структура адреса доменного сокета Unix](#)
 - [Пример: функция bind и доменный сокет Unix](#)
 - [15.3. Функция socketpair](#)
 - [15.4. Функции сокетов](#)
 - [15.5. Клиент и сервер потокового доменного протокола Unix](#)
 - [15.6. Клиент и сервер дейтаграммного доменного протокола Unix](#)
 - [15.7. Передача дескрипторов](#)
 - [Пример передачи дескриптора](#)
 - [15.8. Получение информации об отправителе](#)
 - [Пример](#)
 - [15.9. Резюме](#)
 - [Упражнения](#)
- [Глава 16](#)
 - [16.1. Введение](#)
 - [16.2. Неблокируемые чтение и запись: функция str_cli \(продолжение\)](#)
 - [Более простая версия функции str_cli](#)
 - [Сравнение времени выполнения различных версий функции str_cli](#)
 - [16.3. Неблокируемая функция connect](#)
 - [16.4. Неблокируемая функция connect: клиент времени и даты](#)
 - [Прерванная функция connect](#)
 - [16.5. Неблокируемая функция connect: веб-клиент](#)
 - [Эффективность одновременных соединений](#)
 - [16.6. Неблокируемая функция accept](#)
 - [16.7. Резюме](#)
 - [Упражнения](#)
- [Глава 17](#)
 - [17.1. Введение](#)
 - [17.2. Функция ioctl](#)
 - [17.3. Операции с сокетами](#)
 - [17.4. Операции с файлами](#)
 - [17.5. Конфигурация интерфейса](#)
 - [17.6. Функция get_ifi_info](#)
 - [17.7. Операции с интерфейсами](#)
 - [17.8. Операции с кэшем ARP](#)
 - [Пример: вывод аппаратного адреса узла](#)
 - [17.9. Операции с таблицей маршрутизации](#)
 - [17.10. Резюме](#)
 - [Упражнения](#)
- [Глава 18](#)
 - [18.1. Введение](#)
 - [18.2. Структура адреса сокета канального уровня](#)

- [18.3. Чтение и запись](#)
 - [Пример: получение и вывод записи из таблицы маршрутизации](#)
- [18.4. Операции функции sysctl](#)
 - [Пример: определяем, включены ли контрольные суммы UDP](#)
- [18.5. Функция get_if_info \(повтор\)](#)
- [18.6. Функции имени и индекса интерфейса](#)
 - [Функция if_nameToIndex](#)
 - [Функция if_indexToString](#)
 - [Функция if_nameIndex](#)
 - [Функция if_freeNameIndex](#)
- [18.7. Резюме](#)
- [Упражнения](#)
- [Глава 19](#)
 - [19.1. Введение](#)
 - [19.2. Чтение и запись](#)
 - [19.3. Дамп базы соглашений о безопасности](#)
 - [19.4. Создание статического соглашения о безопасности](#)
 - [Пример](#)
 - [19.5. Динамическое управление SA](#)
 - [Пример](#)
 - [19.6. Резюме](#)
 - [Упражнения](#)
- [Глава 20](#)
 - [20.1. Введение](#)
 - [20.2. Широковещательные адреса](#)
 - [20.3. Направленная и широковещательная передачи](#)
 - [20.4. Функция dg_cli при использовании широковещательной передачи](#)
 - [Фрагментация IP-пакетов и широковещательная передача](#)
 - [20.5. Ситуация гонок](#)
 - [Блокирование и разблокирование сигнала с помощью функции pselect](#)
 - [Использование функций sigsetjmp и siglongjmp](#)
 - [Применение IPC в обработчике сигнала функции](#)
 - [20.6. Резюме](#)
 - [Упражнения](#)
- [Глава 21](#)
 - [21.1. Введение](#)
 - [21.2. Адрес многоадресной передачи](#)
 - [Адреса IPv4 класса D](#)
 - [Адреса многоадресной передачи IPv6](#)
 - [Область действия адресов многоадресной передачи](#)
 - [Сеансы многоадресной передачи](#)
 - [21.3. Сравнение многоадресной и широковещательной передачи в локальной сети](#)
 - [21.4. Многоадресная передача в глобальной сети](#)
 - [21.5. Многоадресная передача от отправителя](#)
 - [21.6. Параметры сокетов многоадресной передачи](#)
 - [21.7. Функция mcast_join и родственные функции](#)
 - [Пример: функция mcast_join](#)

- [Пример: функция mcast_set_loop](#)
- [21.8 Функция dg_cli, использующая многоадресную передачу](#)
 - [Фрагментация IP и многоадресная передача](#)
- [21.9. Получение анонсов сеансов многоадресной передачи](#)
- [21.10. Отправка и получение](#)
 - [Пример](#)
- [21.11. SNTP: простой синхронизирующий сетевой протокол](#)
- [21.12. Резюме](#)
- [Упражнения](#)
- [Глава 22](#)
 - [22.1. Введение](#)
 - [22.2. Получение флагов, IP-адреса получателя и индекса интерфейса](#)
 - [Пример: вывод IP-адреса получателя и флага обрезки дейтаграммы](#)
 - [22.3. Обрезанные дейтаграммы](#)
 - [22.4. Когда UDP оказывается предпочтительнее TCP](#)
 - [22.5. Добавление надежности приложению UDP](#)
 - [Пример](#)
 - [22.6. Связывание с адресами интерфейсов](#)
 - [22.7. Параллельные серверы UDP](#)
 - [22.8. Информация о пакетах IPv6](#)
 - [Исходящий и входящий интерфейсы](#)
 - [Адрес отправителя и адрес получателя IPv6](#)
 - [Задание и получение предельного количества транзитных узлов](#)
 - [Задание адреса следующего транзитного узла](#)
 - [Задание и получение класса трафика](#)
 - [22.9. Управление транспортной MTU IPv6](#)
 - [Отправка с минимальной MTU](#)
 - [Получение сообщений об изменении транспортной MTU](#)
 - [Определение текущей транспортной MTU](#)
 - [Отключение фрагментации](#)
 - [22.10. Резюме](#)
 - [Упражнения](#)
- [Глава 23](#)
 - [23.1. Введение](#)
 - [23.2. Сервер типа «один-ко-многим» с автоматическим закрытием](#)
 - [23.3. Частичная доставка](#)
 - [23.4. Уведомления](#)
 - [Запуск программы](#)
 - [23.5. Неупорядоченные данные](#)
 - [23.6. Связывание с подмножеством адресов](#)
 - [23.7. Получение адресов](#)
 - [Выполнение программы](#)
 - [23.8. Определение идентификатора ассоциации по IP-адресу](#)
 - [23.9. Проверка соединения и ошибки доступа](#)
 - [23.10. Выделение ассоциации](#)
 - [23.11. Управление таймерами](#)
 - [23.12. Когда SCTP оказывается предпочтительнее TCP](#)
 - [23.13. Резюме](#)

- [Упражнения](#)
- [Глава 24](#)
 - [24.1. Введение](#)
 - [24.2. Внеполосные данные протокола TCP](#)
 - [Простой пример использования сигнала SIGURG](#)
 - [Простой пример использования функции select](#)
 - [24.3. Функция sockatmark](#)
 - [Пример: особенности отметки внеполосных данных](#)
 - [Пример: дополнительные свойства внеполосных данных](#)
 - [Пример: единственность отметки внеполосных данных в TCP](#)
 - [24.4. Резюме по теме внеполосных данных TCP](#)
 - [24.5. Резюме](#)
 - [Упражнения](#)
- [Глава 25](#)
 - [25.1. Введение](#)
 - [25.2. Управляемый сигналом ввод-вывод для сокетов](#)
 - [Сигнал SIGIO и сокеты UDP](#)
 - [Сигнал SIGIO и сокеты TCP](#)
 - [25.3. Эхо-сервер UDP с использованием сигнала SIGIO](#)
 - [25.4. Резюме](#)
 - [Упражнения](#)
- [Глава 26](#)
 - [26.1. Введение](#)
 - [26.2. Основные функции для работы с потоками: создание и завершение потоков](#)
 - [Функция pthread_create](#)
 - [Функция pthread_join](#)
 - [Функция pthread_self](#)
 - [Функция pthread_detach](#)
 - [Функция pthread_exit](#)
 - [26.3. Использование потоков в функции str_cli](#)
 - [26.4. Использование потоков в эхо-сервере TCP](#)
 - [Передача аргументов новым потокам](#)
 - [Функции, безопасные в многопоточной среде](#)
 - [26.5. Собственные данные потоков](#)
 - [Пример: функция readline, использующая собственные данные потока](#)
 - [26.6. Веб-клиент и одновременное соединение \(продолжение\)](#)
 - [26.7. Взаимные исключения](#)
 - [26.8. Условные переменные](#)
 - [26.9. Веб-клиент и одновременный доступ](#)
 - [26.10. Резюме](#)
 - [Упражнения](#)
- [Глава 27](#)
 - [27.1. Введение](#)
 - [27.2. Параметры IPv4](#)
 - [27.3. Параметры маршрута от отправителя IPv4](#)
 - [Пример](#)
 - [Уничтожение полученного маршрута от отправителя](#)
 - [27.4. Заголовки расширения IPv6](#)
 - [27.5. Параметры транзитных узлов и параметры получателя IPv6](#)

- [27.6. Заголовок маршрутизации IPv6](#)
- [27.7. «Закрепленные» параметры IPv6](#)
- [27.8. История развития интерфейса IPv6](#)
- [27.9. Резюме](#)
- [Упражнения](#)
- [Глава 28](#)
 - [28.1. Введение](#)
 - [28.2. Создание символьных сокетов](#)
 - [28.3. Вывод на символьном сокете](#)
 - [Особенности символьного сокета версии IPv6](#)
 - [Параметр сокета IPV6_CHECKSUM](#)
 - [28.4. Ввод через символьный сокет](#)
 - [Фильтрация по типу сообщений ICMPv6](#)
 - [28.5. Программа ping](#)
 - [28.6. Программа traceroute](#)
 - [Пример](#)
 - [28.7. Демон сообщений ICMP](#)
 - [Эхо-клиент UDP, использующий демон icmpd](#)
 - [Примеры эхо-клиента UDP](#)
 - [Демон icmpd](#)
 - [28.8. Резюме](#)
 - [Упражнения](#)
- [Глава 29](#)
 - [29.1. Введение](#)
 - [29.2. BPF: пакетный фильтр BSD](#)
 - [29.3. DLPI: интерфейс поставщика канального уровня](#)
 - [29.4. Linux: SOCK_PACKET и PF_PACKET](#)
 - [29.5. Libcap: библиотека для захвата пакетов](#)
 - [29.6. Libnet: библиотека создания и отправки пакетов](#)
 - [29.7. Анализ поля контрольной суммы UDP](#)
 - [Пример](#)
 - [Функции libnet](#)
 - [29.8. Резюме](#)
 - [Упражнения](#)
- [Глава 30](#)
 - [30.1. Введение](#)
 - [30.2. Альтернативы для клиента TCP](#)
 - [30.3. Тестовый клиент TCP](#)
 - [30.4. Последовательный сервер TCP](#)
 - [30.5. Параллельный сервер TCP: один дочерний процесс для каждого клиента](#)
 - [30.6. Сервер TCP с предварительным порождением процессов без блокировки для вызова accept](#)
 - [Реализация 4.4BSD](#)
 - [Эффект наличия слишком большого количества дочерних процессов](#)
 - [Распределение клиентских соединений между дочерними процессами](#)
 - [Коллизии при вызове функции select](#)
 - [30.7. Сервер TCP с предварительным порождением процессов и защитой вызова accept блокировкой файла](#)
 - [Эффект наличия слишком большого количества дочерних процессов](#)

- [Распределение клиентских соединений между дочерними процессами](#)
- [30.8. Сервер TCP с предварительным порождением процессов и защитой вызова accept при помощи взаимного исключения](#)
- [30.9. Сервер TCP с предварительным порождением процессов: передача дескриптора](#)
- [30.10. Параллельный сервер TCP: один поток для каждого клиента](#)
- [30.11. Сервер TCP с предварительным порождением потоков, каждый из которых вызывает accept](#)
- [30.12. Сервер с предварительным порождением потоков: основной поток вызывает функцию accept](#)
- [30.13. Резюме](#)
- [Упражнения](#)
- [Глава 31](#)
 - [31.1. Введение](#)
 - [31.2. Обзор](#)
 - [Типы сообщений](#)
 - [31.3. Функции getmsg и putmsg](#)
 - [31.4. Функции getpmsg и putpmsg](#)
 - [31.5. Функция ioctl](#)
 - [31.6. TPI: интерфейс поставщика транспортных служб](#)
 - [31.7. Резюме](#)
 - [Упражнения](#)
- [Приложения](#)
 - [Приложение А](#)
 - [A.1. Введение](#)
 - [A.2. Заголовок IPv4](#)
 - [A.3. Заголовок IPv6](#)
 - [A.4. Адресация IPv4](#)
 - [Адреса подсетей](#)
 - [Адрес закольцовки](#)
 - [Неопределенный адрес](#)
 - [Частные адреса](#)
 - [Многоинтерфейсность и псевдонимы адресов](#)
 - [A.5. Адресация IPv6](#)
 - [Объединяемые глобальные индивидуальные адреса](#)
 - [Тестовые адреса 6bone](#)
 - [Адреса IPv4, преобразованные к виду IPv6](#)
 - [Адреса IPv6, совместимые с IPv4](#)
 - [Адрес закольцовки](#)
 - [Неопределенный адрес](#)
 - [Адрес локальной связи](#)
 - [Адрес, локальный на уровне сайта](#)
 - [A.6. ICMPv4 и ICMPv6: протоколы управляющих сообщений в сети Интернет](#)
 - [Приложение Б](#)
 - [B.1. Введение](#)
 - [B.2. MBone](#)
 - [B.3. 6bone](#)
 - [B.4. Переход на IPv6: 6to4](#)
 - [Приложение В](#)
 - [B.1. Трассировка системных вызовов](#)
 - [Сокеты ядра BSD](#)
 - [Сокеты ядра Solaris 9](#)

- [B.2. Стандартные службы Интернета](#)
 - [B.3. Программа sock](#)
 - [B.4. Небольшие тестовые программы](#)
 - [B.5. Программа tcpdump](#)
 - [B.6. Программа netstat](#)
 - [B.7. Программа lsof](#)
 - [Приложение Г](#)
 - [Г.1. Заголовочный файл upr.h](#)
 - [Г.2. Заголовочный файл config.h](#)
 - [Г.3. Стандартные функции обработки ошибок](#)
 - [Приложение Д](#)
 - [Глава 1](#)
 - [Глава 2](#)
 - [Глава 3](#)
 - [Глава 4](#)
 - [Глава 5](#)
 - [Глава 6](#)
 - [Глава 7](#)
 - [Глава 8](#)
 - [Глава 9](#)
 - [Глава 10](#)
 - [Глава 11](#)
 - [Глава 12](#)
 - [Глава 13](#)
 - [Глава 14](#)
 - [Глава 15](#)
 - [Глава 16](#)
 - [Глава 17](#)
 - [Глава 18](#)
 - [Глава 20](#)
 - [Глава 21](#)
 - [Глава 22](#)
 - [Глава 24](#)
 - [Глава 25](#)
 - [Глава 26](#)
 - [Глава 27](#)
 - [Глава 28](#)
 - [Глава 29](#)
 - [Глава 30](#)
 - [Глава 31](#)
 - [Литература](#)
 - [notes](#)
 - [1](#)
 - [2](#)
 - [3](#)
 - [4](#)
 - [5](#)
 - [6](#)
 - [7](#)
-

У. Р. Стивенс, Б. Феннер, Э. М. Рудофф

UNIX

Разработка сетевых приложений

3-е издание

*Piкуу
Aloha nui loa*

Вступительное слово

Вышедшее в 1990 году первое издание этой книги было признано лучшим учебником для программистов, изучающих технологии сетевого программирования. С тех пор сеть претерпела серьезнейшие изменения. Достаточно взглянуть на адрес автора, указанный в том издании: «*unet!hs1!netbook*». Вряд ли любой читатель сможет сказать, что это адрес в сети UUCP, которая была популярна в 1980-х.

Сейчас сети UUCP стали раритетом, а новые технологии, подобные беспроводным сетям, получают повсеместное распространение. Разрабатываются новые протоколы и парадигмы программирования. И программисты начинают ощущать потребность в учебнике, который помог бы им освоить все тонкости новых методик.

Книга, которую вы держите в руках, заполняет этот пробел. Тем, у кого на книжной полке стоит истрапанное первое издание, она даст возможность узнать о новых технологиях программирования и о протоколах следующего поколения, таких как IPv6. Эта книга нужна всем, потому что она представляет собой соединение практического опыта, исторической перспективы и глубины понимания.

Я уже получил удовольствие и новые знания благодаря этой книге, и не сомневаюсь, что вы сможете сделать то же самое.

Сэм Леффлер

Предисловие

Введение

Эта книга предназначена для тех, кто хочет писать программы, взаимодействующие друг с другом посредством интерфейса сокетов. Некоторые читатели, возможно, уже достаточно хорошо знакомы с сокетами, потому что сейчас сетевое программирование фактически немыслимо без них. Другим придется начинать с самых азов. Цель этой книги — предоставить руководство по сетевому программированию для начинающих и для профессионалов, тех, кто разрабатывает новые сетевые приложения и тех, кто поддерживает существующий код. Будет полезна она и тем, кто хочет понимать, как работают сетевые компоненты их систем.

Все примеры в этой книге относятся к операционной системе Unix, хотя основные понятия и концепции сетевого программирования практически не зависят от операционной системы. Почти все операционные системы предоставляют набор сетевых приложений, таких как браузеры, почтовые клиенты и файловые серверы. Мы обсудим обычное деление таких приложений на клиентскую и серверную части и напишем множество примеров собственных клиентов и серверов.

Ориентируясь на Unix, мы не могли не рассказать о самой этой системе и о TCP/IP. В тех случаях, когда читателю могут оказаться интересными более подробные сведения, мы отсылаем его к другим книгам:

- *Advanced Programming in the Unix Environment* [110];
- *TCP/IP Illustrated*, том 1 [111];
- *TCP/IP Illustrated*, том 2 [128];
- *TCP/IP Illustrated*, том 3 [112].

В первую очередь читателю следует обращаться к книге [128], в которой представлена реализация 4.4BSD функций сетевого программирования для API сокетов (`socket`, `bind`, `connect` и т.д.). При понимании того, как реализована та или иная функциональная возможность, ее применение в приложениях становится более осмысленным.

Изменения по сравнению со вторым изданием

Сокеты в нынешней их форме существовали с 1980-х годов. Благодаря совершенству архитектуры этого интерфейса, он продолжает оставаться оптимальным для большинства приложений. Возможно, вы будете удивлены, узнав, как много изменилось в этом интерфейсе с 1998 года, когда было опубликовано второе издание этой книги. Эти изменения были отражены в новом издании. Их можно сгруппировать следующим образом:

- Новое издание содержит обновленные сведения об IPv6, который на момент публикации второго издания существовал только в черновом варианте, и за прошедшие годы был усовершенствован.
- Определения функций и примеры их использования были изменены в соответствии с последней спецификацией POSIX (POSIX 1003.1–2001), которая известна под названием «Единая спецификация UNIX версии 3».
- Описание транспортного интерфейса X/Open было исключено из книги, потому что этот интерфейс вышел из широкого употребления и последняя спецификация POSIX не описывает его.
- Также исключено было описание протокола TCP для транзакций (T/TCP).
- Были добавлены три новые главы, посвященные относительно новому транспортному протоколу SCRIPT. Этот надежный протокол, ориентированный на передачу сообщений, предоставляет поддержку многопоточной передачи и обеспечивает работу с несколькими интерфейсами. Изначально он был предназначен для Интернет-телефонии, но его функции могут оказаться полезными многим другим приложениям.
- Также была добавлена глава, посвященная сокетам управления ключами, которые могут использоваться с протоколом IPSec и другими сервисами сетевой безопасности.
- Все примеры тестировались на новых компьютерах с новыми версиями операционных систем. Во многих случаях это оказывалось необходимым по той причине, что производители устранили ошибки и

добавляли новые функции, правда, время от времени, нам удавалось обнаруживать новые ошибки. Для тестирования использовались следующие компьютеры (см. рис. 1.17):

- Apple Power PC с MacOS/X 10.2.6
- HP PA-RISC с HP-UX 11i
- IBM Power PC с AIX 5.1
- Intel x86 с FreeBSD 4.8
- Intel x86 с Linux 2.4.7
- Sun SPARC с FreeBSD 5.1
- Sun SPARC с Solaris 9

Второй том под названием «Взаимодействие процессов» рассказывает о передаче сообщений, синхронизации, разделяемой памяти и удаленном вызове процедур.

Кому адресована эта книга

Эту книгу можно использовать и как учебное пособие по сетевому программированию, и как справочник для более опытных программистов. При использовании его как учебника или для ознакомления с сетевым программированием следует уделить особое внимание второй части («Элементарные сокеты», главы 3–11), после чего можно переходить к чтению тех глав, которые представляют наибольший интерес. Во второй части рассказывается об основных функциях сокетов как для TCP, так и для UDP; кроме того, рассматривается мультиплексирование ввода-вывода, параметры сокетов и основные преобразования имен и адресов. Всем читателям следует прочесть главу 1, в особенности раздел 1.4, так как в нем описаны некоторые функции-обертки, используемые далее во всей книге. Глава 2 и, возможно, приложение А могут быть использованы по мере необходимости для получения справочных сведений в зависимости от уровня подготовки читателя. Большинство глав в третьей части («Дополнительные возможности сокетов») могут быть прочитаны независимо от других, содержащихся в этой же части.

Для тех, кто собирается использовать эту книгу в качестве справочного пособия, имеется подробный предметный указатель. Для тех, кто будет читать только выборочные главы в произвольном порядке, в книге имеются ссылки на те места, где обсуждаются близкие темы.

Исходный код и замеченные опечатки

Исходный код для всех примеров расположен на моей домашней странице^[1], адрес которой указан в конце предисловия. Чтобы научиться сетевому программированию, лучше всего будет взять эти программы, изменить их и расширить. На самом деле написание программ таким образом является единственным способом овладеть изученными технологиями. В конце каждой главы приводятся упражнения, а ответы на большинство из них содержатся в приложении Г.

Список найденных опечаток по этой книге также находится на моей домашней странице.

Благодарности

Первое и второе издания этой книги были написаны У. Ричардом Стивенсом, который скончался 1 сентября 1999 г. Его книги стали образцом учебной литературы по сетевому программированию и считаются яркими, тщательно проработанными и необычайно популярными произведениями искусства. Авторы новой редакции постарались сохранить качество книги на прежнем уровне.

Без поддержки семьи и друзей написать книгу невозможно. Билл Феннер хотел бы поблагодарить свою жену Пегги (чемпионку в беге на четверть мили) и соседа по дому Кристофера Бойда за то, что они взяли на себя все тяготы домашнего труда на время его работы над этим проектом. Нужно поблагодарить и Джерри Виннера, чья поддержка была незаменима. Энди Рудофф благодарен своей жене Эллен и дочерям Джо и Кэти за понимание и поощрение. Без вашей помощи мы бы не справились с этим.

Рэндолл Стоарт из Cisco Systems предоставил большую часть материала по SCRIPT и заслуживает отдельной благодарности за свой бесценный вклад. Без помощи Рэндолла мы не смогли бы рассказать ничего на эту новую интересную тему.

Многочисленные рецензенты помогли цennыми замечаниями и указаниями, обращая внимание на многочисленные ошибки и те области, которые требовали более подробного изложения, а также предложили альтернативные варианты формулировок, изложения материала и самих программ. Авторы хотели бы поблагодарить Джеймса Карлсона, Ву-Чана Фена, Рика Джонса, Брайана Кернигана, Сэма Леффлера, Джона МакКанна, Крейга Метца, Яна Ланса Тейлора, Дэвида Шварца и Гари Райта.

Многие люди и те организации, в которых они работали, шли мне навстречу, предоставляя программное обеспечение или доступ к системе, необходимые для тестирования некоторых примеров к книге.

- Джесси Хог из IBM Austin предоставила систему AIX и компиляторы.
- Рик Джонс и Уильям Гиллэм из Hewlett-Packard предоставили доступ ко множеству систем под управлением HP-UX.

Истинным удовольствием было работать с персоналом Addison Wesley: Норин Региной, Кэтлин Кэрэн, Дэном де Паскуале, Энтони Гемелларо и Мэри Франц, нашим редактором, которая заслуживает отдельных благодарностей.

Продолжая традиции Рика Стивенса (но в противоположность общепринятым технологиям), мы подготовили оригинал-макет книги, используя замечательный пакет *groff*, написанный Джеймсом Кларком (James Clark), создали иллюстрации с помощью программы *grc* (используя многие из макросов Гари Райта), сделали таблицы с помощью программы *tbl*, составили предметный указатель и подготовили окончательный макет страниц. Программа Дейва Хансона (Dave Hanson) *loom* и некоторые сценарии Гари Райта (Gary Wright) использовались для включения кода программ в книгу. Набор сценариев на языке *awk*, написанный Джоном Бентли (Jon Bentley) и Брайаном Керниганом (Brian Kernighan), помогал в создании предметного указателя.

Авторы с нетерпением ждут комментарии, предложения и сообщения о замеченных опечатках.

authors@unpbook.com

<http://www.unpbook.com>

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Исходные коды всех программ, приведенных в книге, вы можете найти по адресу <http://www.piter.com>.

На веб-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

Часть 1

Введение. TCP/IP

Глава 1

Введение в сетевое программирование

1.1. Введение

Чтобы писать программы, рассчитанные на взаимодействие в компьютерных сетях, необходимо сначала изобрести протокол — соглашение о порядке взаимодействия таких программ. Прежде чем углубляться в детальное проектирование протокола, нужно принять некоторые высокоуровневые решения о том, какая программа будет инициировать передачу данных и в каких случаях можно ожидать ответной передачи. Например, веб-сервер обычно рассматривается как долгоживущая программа (или *демон* — *daemon*), которая отправляет сообщения исключительно в ответ на запросы, поступающие по сети. Другой стороной является веб-клиент, например браузер, который всегда начинает взаимодействие с сервером первым. Деление на клиенты и серверы характерно для большинства сетевых приложений. И протокол, и программы обычно упрощаются, если возможность отправки запросов предоставляется только клиенту. Конечно, некоторые сетевые приложения более сложной структуры требуют поддержки *асинхронного обратного вызова* (*asynchronous callback*), то есть инициации передачи сообщений сервером, а не клиентом. Однако гораздо чаще приложения реализуются в базовой модели клиент-сервер, изображенной на рис. 1.1.



Рис. 1.1. Сетевое приложение: клиент и сервер

Клиенты обычно устанавливают соединение с одним сервером за один раз, хотя, если в качестве примера говорить о веб-браузере, мы можем соединиться со множеством различных веб-серверов, скажем, в течение 10 минут. Сервер, напротив, в любой момент времени может быть соединен со множеством клиентов. Это отражено на рис. 1.2. Далее в этой главе будут рассмотрены различные возможности взаимодействия сервера одновременно со множеством клиентов.

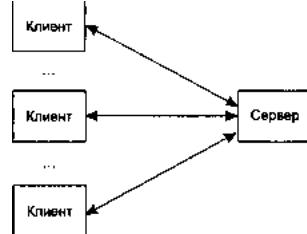


Рис. 1.2. Сервер, который одновременно обслуживает множество клиентов

Не будет большой ошибкой сказать, что клиентское и серверное приложения взаимодействуют по сетевому протоколу, однако фактически в большинстве случаев используется несколько протоколов различных уровней. В этой книге мы сосредоточимся на наборе (стеке) протоколов TCP/IP, также называемом набором протоколов Интернета. Так, например, клиенты и веб-серверы устанавливают соединения, используя протокол управления передачей (Transmission Control Protocol, TCP). TCP, в свою очередь, использует протокол Интернета (Internet Protocol, IP), а протокол IP устанавливает соединение с тем или иным протоколом канального уровня. Если и клиент, и сервер находятся в одной сети Ethernet, взаимодействие между ними будет осуществляться по схеме, изображенной на рис. 1.3.

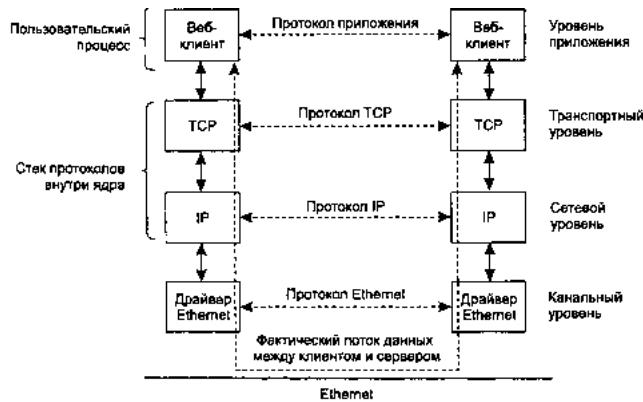


Рис. 1.3. Клиент и сервер в одной сети Ethernet, соединенные по протоколу TCP

Хотя клиент и сервер устанавливают соединение с использованием протокола уровня приложений, транспортные уровни устанавливают соединение, используя TCP. Обратите внимание, что действительный поток информации между клиентом и сервером идет вниз по стеку протоколов на стороне клиента, затем по сети и, наконец, вверх по стеку протоколов на стороне сервера.

Заметьте, что клиент и сервер являются типичными пользовательскими процессами, в то время как TCP и протоколы IP обычно являются частью стека протоколов внутри ядра. Четыре уровня протоколов обозначены на рис. 1.3 справа.

Мы будем обсуждать не только протоколы TCP и IP. Некоторые клиенты и серверы используют протокол пользовательских дейтаграмм (User Datagram Protocol, UDP) вместо TCP; оба эти протокола более подробно обсуждаются в главе 2. Мы часто пользуемся термином «IP», но на самом деле протокол, который мы при этом подразумеваем, называется «IP версии 4» (IP version 4, IPv4). Новая версия этого протокола, IP версии 6 (IPv6), была разработана в середине 90-х и, возможно, со временем заменит протокол IPv4. В этой книге описана разработка сетевых приложений как под IPv4, так и под IPv6. В приложении A приводится сравнение протоколов IPv4 и IPv6 наряду с другими протоколами, с которыми мы встретимся.

Клиент и сервер не обязательно должны быть присоединены к одной и той же локальной сети (*local area network, LAN*), как в примере на рис. 1.3. Вместо этого, как показано на рис. 1.4, клиент и сервер могут относиться к разным локальным сетям, при этом обе локальные сети должны быть соединены в глобальную сеть (*wide area network, WAN*) с использованием маршрутизаторов.

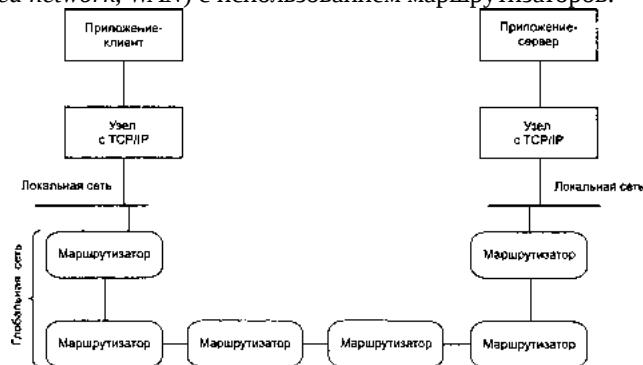


Рис. 1.4. Клиент и сервер в различных локальных сетях, соединенных через глобальную сеть

Маршрутизаторы — это «кирпичи», из которых строится глобальная сеть. На сегодня наибольшей глобальной сетью является Интернет, хотя многие компании создают свои собственные глобальные сети, и эти частные сети могут быть, а могут и не быть подключены к Интернету.

Оставшаяся часть этой главы представляет собой введение и обзор различных тем, которые более подробно раскрываются далее по тексту книги. Мы начнем с полного, хотя и простого, примера клиента TCP, на котором демонстрируются вызовы многих функций и понятия, с которыми мы встретимся далее. Клиент работает только с протоколом IPv4, и мы покажем изменения, необходимые для работы с протоколом IPv6. Разумнее всего создавать независимые от протокола клиенты и серверы, и такое решение будет рассмотрено нами в главе 11. Мы приводим также код полнофункционального сервера TCP, работающего с нашим клиентом.

Чтобы упростить написанный нами код, мы определяем наши собственные функции-обертки (*wrapper functions*) для большинства вызываемых системных функций. Функции-обертки в большинстве случаев служат для проверки кода возврата. В случае ошибки функция-обертка печатает соответствующее сообщение и завершает работу программы.

В этой же главе мы подробно расскажем о сети, в которой тестировались все примеры этой книги, приведем имена узлов, их IP-адреса и названия операционных систем, под управлением которых они работают.

В разговорах о Unix широко используется термин «X», обозначающий стандарт, принятый большинством производителей. Мы опишем историю стандарта POSIX и то, каким образом он определяет интерфейсы программирования приложений (Application Programming Interfaces, API), рассматриваемые в этой книге, наряду с другими конкурирующими стандартами.

1.2. Простой клиент времени и даты

Рассмотрим конкретный пример, на котором мы введем многие понятия и термины, используемые в этой книге. В листинге 1.1^[1] представлена реализация TCP-клиента времени и даты. Этот клиент устанавливает TCP-соединение с сервером, а сервер просто посылает клиенту время и дату в текстовом формате.

Листинг 1.1. Клиент TCP для определения времени и даты

```
//intro/daytimetcpccli.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd, n;
6     char recvline[MAXLINE + 1];
7     struct sockaddr_in servaddr;

8     if (argc != 2)
9         err_quit("usage: a.out <Ipaddress>");

10    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
11        err_sys("socket error");

12    bzero(&servaddr, sizeof(servaddr));
13    servaddr.sin_family = AF_INET;
14    servaddr.sin_port = htons(13); /* сервер времени и даты */
15    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
16        err_quit("inet_nton error for %s", argv[1]);

17    if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) < 0)
18        err_sys("connect error");

19    while ((n = read(sockfd, recvline, MAXLINE)) > 0) {
20        recvline[n] = 0; /* завершающий нуль */
21        if (fputs(recvline, stdout) == EOF)
22            err_sys("fputs error");
23    }
24    if (n < 0)
25        err_sys("read error");

26    exit(0);
27 }
```

ПРИМЕЧАНИЕ

Это формат, который мы используем для всего исходного кода в тексте. Каждая непустая строка пронумерована. Абзац, описывающий некоторую часть кода, начинается с двух номеров — начального и конечного номеров тех строк, о которых идет речь в данном абзаце. Как правило, абзацу предшествует короткий заголовок, в котором кратко резюмируется содержание описываемого кода.

В начале фрагмента кода указано имя файла исходного кода: в данном примере это файл `daytimetccli.c` в каталоге `intro`. Поскольку исходный код всех примеров этой книги можно свободно скачать из Сети (см. предисловие), вы можете найти соответствующие исходные файлы по их названиям. Наилучший способ изучить концепции сетевого программирования — компилировать, запускать и особенно модифицировать эти программы в ходе изучения книги.

ПРИМЕЧАНИЕ

Мы будем использовать примечания наподобие этого для описания особенностей реализации и исторических справок.

Если мы откомпилируем эту программу в определенный по умолчанию файл `a.out` и выполним ее, на выходе мы получим следующее:

```
solaris % a.out 206.168.112.96 наш ввод  
Mon May 26 20:58:40 2003 вывод программы
```

ПРИМЕЧАНИЕ

Отображая интерактивный ввод и вывод, мы показываем то, что мы вводим, полужирным шрифтом; вывод же компьютера показываем моноширинным шрифтом. Мы всегда указываем название системы как часть приглашения интерпретатора (в данном примере `solaris`), чтобы показать, на каком узле выполняется команда. Системы, используемые для выполнения большинства примеров этой книги, показаны на рис. 1.7. Имена узлов обычно соответствуют операционным системам.

В этой программе, состоящей из 27 строк, есть много важных особенностей, нуждающихся в обсуждении. Мы кратко рассмотрим их на тот случай, если это первая сетевая программа, с которой вы встретились, а более подробные сведения по соответствующим вопросам вы сможете получить в других главах.

Подключение собственного заголовочного файла

1 Мы подключаем наш собственный заголовочный файл, `upr.h`, текст которого приведен в разделе Г.1. Этот заголовочный файл, в свою очередь, подключает различные системные заголовочные файлы, которые необходимы большинству сетевых программ, и определяет используемые нами константы (например, `MAXLINE`).

Аргументы командной строки

2-3 Определение функции `main` вместе с аргументами командной строки. Везде в данной книге при написании кода подразумевалось, что для его компиляции должен использоваться компилятор ANSI C (American National Standards Institute — Национальный институт стандартизации США), который также называют ISO C.

Создание сокета TCP

10-11 Функция `socket` создает потоковый сокет (`SOCK_STREAM`) Интернета (`AF_INET`) — это красивое название для обычного TCP-сокета). Функция возвращает дескриптор (небольшое целое число), который мы используем для идентификации сокета во всех последующих вызовах (например, `connect` и `read`).

ПРИМЕЧАНИЕ

Оператор `if` содержит вызов функции `socket`, присваивание возвращаемого значения переменной `sockfd` и последующую проверку, является ли это присвоенное значение меньшим нуля. Мы могли разбить этот оператор на два оператора С следующим образом:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
```

но использованная в листинге 1.1 запись является типичным для языка C способом объединения двух строк. Поскольку в языке C оператор «меньше» (`<`) имеет более высокий приоритет, чем оператор присваивания, необходимо заключить в скобки операции присваивания и вызова функции (как это и сделано в листинге 1.1, в строке 10). Между двумя открывающими скобками мы всегда вставляем пробел как указание на то, что левая часть операции сравнения содержит также операцию присваивания. (Этот стиль позаимствован из исходного кода Minix [120].) Мы используем этот же прием в операторе `while` дальше в нашей программе.

Мы будем встречать множество различных вариантов использования термина *сокет* (*socket*). Во-первых, используемый нами API называется *API сокетов*. Во-вторых, в предыдущем абзаце мы упоминали функцию `socket`, которая входит в API сокетов. В-третьих, там же мы ссылались и на «сокет TCP», который является синонимом *конечной точки TCP* (*TCP endpoint*).

Если вызов функции `socket` оказывается неудачным, мы прерываем выполнение программы с помощью вызова функции `err_sys`. Она выдает сообщение об ошибке с ее описанием (например, «Протокол не поддерживается» — одна из возможных ошибок функции `socket`) и прерывает выполнение процесса. Эта функция создана нами, как и некоторые другие, начинающиеся с `err_`. Мы будем широко использовать их в примерах в последующих главах. Описание функций приводится в разделе Г.4.

Задание IP-адреса и порта сервера

12-16 Мы заполняем структуру адреса сокета Интернета (структурой типа `sockaddr_in` с именем `servaddr`) IP-адресом и номером порта сервера. Сначала мы инициализируем всю структуру нулями, используя функцию `bzero`, затем устанавливаем номер порта в 13 (который является номером *заранее известного порта* (*well-known port*) сервера времени и даты на любом узле TCP/IP, поддерживающем соответствующую службу — см. табл. 2.1), после чего устанавливаем IP-адрес равным значению, определенному первым аргументом командной строки (`argv[1]`). В этой структуре поля IP-адреса и номера порта должны иметь определенный формат: мы вызываем библиотечную функцию `htons` (*host to network short*), чтобы преобразовать двоичный номер порта в требуемый формат, и вызываем библиотечную функцию `inet_nton` (*presentation to numeric*), чтобы преобразовать аргумент командной строки в символах ASCII (например, `206.168.112.96` при выполнении данного примера) в двоичный формат.

ПРИМЕЧАНИЕ

Функция `bzero` не является функцией ANSI C. Она происходит от более раннего кода сетевого программирования Беркли. Тем не менее мы используем именно ее, а не функцию ANSI C `memset`, потому что с функцией `bzero` работать проще: она вызывается с двумя аргументами, а `memset` — с тремя. Почти каждый производитель, поддерживающий API сокетов, также реализует и функцию `bzero`, а если и не реализует, мы определяем ее через макрос в нашем заголовочном файле `inpr.h`.

Автор [112] в первом издании сделал десять ошибок, поменяв местами аргументы `memset`. Компилятор C не может распознать эту ошибку, поскольку оба аргумента принадлежат одному типу. В действительности второй аргумент принадлежит типу `int`, а третий — `size_t` — обычно имеет тип `unsigned int` (то есть целое без знака), но заданные значения, соответственно, 0 и 16, являются допустимыми для обоих типов аргумента. Вызов функции `memset` все равно осуществлялся, но реально функция ничего не делала, поскольку задавалось нулевое число инициализируемых байтов. Программа работала, потому что только некоторые функции сокетов действительно требуют, чтобы последние 8 байт структуры адреса сокета Интернета были установлены в 0. Тем не менее это ошибка, и ее можно избежать при использовании функции `bzero`, поскольку перестановка двух аргументов функции `bzero` всегда будет выявлена компилятором C, если используются прототипы функций.

Возможно, вы впервые встречаете функцию `inet_pton`. Она появилась вместе с протоколом IPv6 (о котором более подробно мы поговорим в приложении A). В старых программах для преобразования точечно-десятичной записи (dotted-decimal string) ASCII в необходимый формат использовалась функция `inet_addr`, но у нее есть ряд ограничений, которых не имеет функция `inet_pton`. Не беспокойтесь, если ваша система (еще) не поддерживает эту функцию; реализация ее приведена в разделе 3.7.

Установка соединения с сервером

17-18 Функция `connect`, применяемая к сокету TCP, устанавливает соединение по протоколу TCP с сервером, адрес сокета которого содержится в структуре, на которую указывает второй аргумент. Мы также должны задать длину структуры адреса сокета в качестве третьего аргумента функции `connect`, а для структур адреса интернет-сокета мы всегда предоставляем вычисление длины компилятору, используя оператор C `sizeof`.

ПРИМЕЧАНИЕ

В заголовочном файле `shp.h` мы используем директиву `#define SA`, чтобы определить `SA` как `struct sockaddr`, что соответствует общей структуре адреса сокета. Каждый раз, когда одна из функций сокетов требует указателя на структуру адреса сокета, этот указатель должен быть преобразован к указателю на общую структуру адреса сокета. Это происходит потому, что функции сокетов появились раньше, чем стандарт ANSI C. Соответственно, тип указателя `void*` не был доступен в начале 80-х, когда эти функции были разработаны. Проблема состоит в том, что "struct sockaddr" занимает 15 символов и часто заставляет выходить строку исходного кода за правую границу экрана (или за страницу книги), поэтому мы сократили ее до `SA`. Более подробно мы исследуем общие структуры адресов сокетов на примере листинга 3.2.

Чтение и отображение ответа сервера

19-25 Мы читаем ответ сервера и отображаем результат, используя стандартную функцию ввода-вывода `fputs`. Нужно быть внимательным при использовании TCP, поскольку это потоковый (byte-stream) протокол без границ записей. Обычно ответом сервера является 26-байтовая строка следующей формы:

`Fri Jan 12 14:27:52 1996\r\n`

где `\r` — это возврат каретки, а `\n` — перевод строки (в символах ASCII). В случае потокового протокола эти 26 байт можно получить в нескольких вариантах: в виде отдельного сегмента TCP, содержащего все 26 байт данных, либо в виде 26 сегментов, каждый из которых содержит по одному байту данных, или в виде любой другой комбинации, в сумме дающей 26 байт. Обычно возвращается один сегмент, содержащий все 26 байт, но при больших объемах данных нельзя рассчитывать, что ответ сервера будет получен с помощью одного вызова `read`. Следовательно, при чтении из сокета TCP нужно всегда вызывать функцию `read` циклически и прерывать цикл либо когда функция возвращает 0 (например, соединение было разорвано другой стороной), либо когда возвращенное значение оказывается меньше нуля (ошибка).

В приведенном примере конец записи обозначается сервером, закрывающим соединение. Эта технология используется также версией 1.0 протокола передачи гипертекста (Hypertext Transfer Protocol, HTTP). Существуют и другие способы обозначения конца записи. Например, протокол передачи файлов (File Transfer Protocol, FTP) и простой протокол передачи почты (Simple Mail Transfer Protocol, SMTP) обозначают конец записи 2-байтовой последовательностью, состоящей из символов ASCII возврата каретки и перевода строки. Служба вызова удаленных процедур (Remote Procedure Call, RPC) и система именования доменов (Domain Name System, DNS) помещают перед каждой записью, отсылаемой по протоколу TCP, двоичное число, соответствующее длине этой записи. Здесь важно осознать, что протокол TCP сам по себе не предоставляет никаких меток записей: если приложение хочет отделять записи одну от другой, оно должно делать это самостоятельно, и для этого имеются стандартные методы.

Завершение программы

26 Функция `exit` завершает программу. Unix всегда закрывает все открытые дескрипторы при завершении процесса, поэтому теперь наш сокет TCP закрыт.

Как уже говорилось, пока мы лишь выделили наиболее важные моменты, детальным исследованием которых займемся в дальнейшем.

1.3. Независимость от протокола

Наша программа, представленная в листинге 1.1, является *зависимой от протокола* (*protocol dependent*) IPv4. Мы выделяем и инициализируем структуру `sockaddr_in`, определяем адрес как относящийся к семейству AF_INET и устанавливаем первый аргумент функции `socket` равным AF_INET.

Если мы хотим изменить программу так, чтобы она работала по протоколу IPv6, мы должны изменить код. В листинге 1.2 показана новая версия программы с соответствующими изменениями, отмеченными полужирным шрифтом.

Листинг 1.2. Версия листинга 1.1 для IPv6

```
//intro/daytimetcpccliv6.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd, n;
6     char recvline[MAXLINE + 1];
7     struct sockaddr_in6 servaddr;

8     if (argc != 2)
9         err_quit("usage: a.out <Ipadress>");

10    if ((sockfd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
11        err_sys("socket error");

12    bzero(&servaddr, sizeof(servaddr));
13    servaddr.sin6_family = AF_INET6;
14    servaddr.sin6_port = htons(13); /* сервер времени и даты */
15    if (inet_pton(AF_INET6, argv[1], &servaddr.sin6_addr) <= 0)
16        err_quit("inet_nton error for %s", argv[1]);

17    if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) < 0)
18        err_sys("connect error");

19    while ((n = read(sockfd, recvline, MAXLINE)) > 0) {
20        recvline[n] = 0; /* символ конца строки */
21        if (fputs(recvline, stdout) == EOF)
```

```

22     err_sys("fputs error");
23 }
24 if (n < 0)
25 err_sys("read error");

26 exit(0);
27 }

```

Изменились только пять строк, но в результате мы все равно получили программу, зависимую от протокола, в данном случае — от протокола IPv6. Лучше сделать программу *независимой от протокола* (*protocol independent*). В листинге 11.3 представлена независимая от протокола версия этого клиента, основанная на вызове `getaddrinfo` из `tcp_connect`.

Другим недостатком наших программ является то, что пользователь должен вводить IP-адрес сервера в точечно-десятичной записи (например, 206.168.112.219 для версии IPv4). Людям проще работать с именами, чем с числами (например, `www.unpbook.com`). В главе 11 мы обсудим функции, обеспечивающие преобразование имен узлов в IP-адреса и имен служб в порты. Мы специально откладываем описание этих функций, продолжая использовать IP-адреса и номера портов, чтобы иметь ясное представление о том, что именно входит в структуры адресов сокетов, которые мы должны заполнить и проверить. Это также упрощает наши объяснения сетевого программирования, снимая необходимость описывать в подробностях еще один набор функций.

1.4. Обработка ошибок: функции-обертки

В любой реальной программе существенным моментом является проверка *каждого* вызова функции на предмет возвращаемой ошибки. В листинге 1.1 мы проводим поиск ошибок в вызовах функций `socket`, `inet_pton`, `connect`, `read` и `fputs`, и когда ошибка случается, мы вызываем свои собственные функции `err_quit` и `err_sys` для печати сообщения об ошибке и для прерывания выполнения программы. В отдельных случаях, когда функция возвращает ошибку, бывает нужно сделать еще что-либо помимо прерывания программы, как показано в листинге 5.9, когда мы должны проверить прерванный системный вызов.

Поскольку прерывание программы из-за ошибки — типичное явление, мы сократим наши программы, определив *функции-обертки*, которые будут вызывать соответствующие рабочие функции, проверять возвращаемые значения и прерывать программу при возникновении ошибки. Соглашение, используемое нами, заключается в том, что название функции-обертки пишется с заглавной буквы, например:

```
sockfd = Socket(AF_INET, SOCK_STREAM, 0);
```

Наша функция-обертка для функции `socket` показана в листинге 1.3.

Листинг 1.3. Наша функция-обертка для функции `socket`

```

//lib/wrapsoc.c

172 int
173 Socket(int family, int type, int protocol)
174 {
175     int n;

176     if ((n = socket(family, type, protocol)) < 0)
177         err_sys("socket error");
178     return (n);
179 }
```

Хотя вы можете решить, что использование этих функций-оберток не обеспечивает большой экономии, на самом деле это не так. Обсуждая потоки (*threads*) в главе 26, мы обнаружим, что, когда происходит какая-либо ошибка, функции потоков не устанавливают значение стандартной переменной Unix `errno` равным определенной константе, специфической для произошедшей ошибки. Вместо этого значение переменной `errno` просто возвращается функцией. Это значит, что каждый раз, когда мы вызываем одну из функций `pthread`, мы должны разместить в памяти переменную, сохранить возвращаемое значение в этой переменной и установить `errno` равной этому значению перед вызовом `err_sys`. Чтобы избежать загромождения кода скобками, мы можем использовать оператор языка C *запятая* для объединения присваивания значения переменной `errno` и вызова `err_sys` в отдельное выражение следующим образом:

```
int n;
if ((n = pthread_mutex_lock(&ndone_mutex)) != 0)
    errno = n, err_sys("pthread_mutex_lock error");
```

ВНИМАНИЕ

В тексте книги вам будут встречаться функции, имена которых начинаются с заглавной буквы. Это наши собственные функции-обертки. Функция-обертка вызывает функцию, имеющую такое же имя, но начинающееся со строчной буквы.

При описании исходного кода, представленного в тексте книги, мы всегдасылаемся на вызываемую функцию низшего уровня (например, socket), но не на функцию-обертку (например, Socket).

В качестве альтернативы мы можем определить новую функцию выдачи сообщений об ошибках, которая в качестве аргумента получает системный код ошибки. Однако проще всего текст будет выглядеть с использованием функции-обертки, определенной в листинге 1.4:

```
Pthread_mutex_lock(&ndone_mutex);
```

Листинг 1.4. Наша собственная функция-обертка для функции pthread_mutex_lock

```
//lib/wrappthread.c
72 void
73 Pthread_mutex_lock(pthread_mutex_t *mptr)
74 {
75     int n;

76     if ((n = pthread_mutex_lock(mptr)) == 0)
77         return;
78     errno = n;
79     err_sys("pthread_mutex_lock error");
80 }
```

ПРИМЕЧАНИЕ

Если аккуратно программировать на С, можно использовать макросы вместо функций, что обеспечивает небольшой выигрыш в производительности, однако функции-обертки редко, если вообще когда-нибудь бывают причиной недостаточной производительности программ.

Наш выбор — первая заглавная буква в названии функции — является компромиссом. Было предложено множество других стилей: подстановка префикса e перед названием функции (как сделано в [67, с. 182]), добавление _e к имени функции и т.д. Наш вариант кажется наименее отвлекающим внимание и одновременно дающим визуальное указание на то, что вызывается какая-то другая функция.

Эта технология имеет, кроме того, полезный побочный эффект: она позволяет проверять возникновение ошибок при выполнении таких функций, ошибки в которых часто остаются незамеченными, например close и listen.

На протяжении всей книги мы будем использовать эти функции-обертки, кроме тех случаев, когда нам нужно проверить ошибку явно и обрабатывать ее другим, отличным от прерывания программы, способом. Мы не приводим исходный код для всех наших собственных функций-оберток, но он свободно доступен в Интернете (см. предисловие).

Значение системной переменной Unix errno

Когда при выполнении функции Unix (например, одной из функций сокетов) происходит ошибка, глобальной переменной errno присваивается положительное значение, указывающее на тип ошибки, а возвращаемое значение функции обычно равно -1. Наша функция err_sys проверяет значение переменной

`errno` и печатает строку с соответствующим сообщением об ошибке (например, «Время соединения истекло», если значение переменной `errno` равно `ETIMEDOUT`).

Переменная `errno` устанавливается равной определенному значению, только если при выполнении функции произошла какая-либо ошибка. Ее значение не определено, если функция не возвращает ошибки. Все положительные значения ошибок являются константами с именами в верхнем регистре, начинающимися на «E», и обычно определяются в заголовке `<sys/errno.h>`. Ни одна ошибка не имеет кода 0.

Переменную `errno` нельзя хранить как глобальную переменную в случае множества потоков, у которых все глобальные переменные являются общими. О решении этой проблемы мы расскажем в главе 23.

На протяжении всего текста книги мы использовали фразы типа «функция `connect` возвращает `ECONNREFUSED`» для сокращенного обозначения того, что при выполнении функции произошла ошибка (обычно при этом возвращаемое значение функции равно `-1`), и значение переменной `errno` стало равным указанной константе.

1.5. Простой сервер времени и даты

Мы можем написать простую версию сервера TCP для определения времени и даты, который будет работать с клиентом, описанным в разделе 1.2. Мы используем функции-обертки, описанные в предыдущем разделе. Код сервера приведен в листинге 1.5.

Листинг 1.5. TCP-сервер времени и даты

```
//intro/daytimetcpsrv.c
1 #include "unp.h"
2 #include <time.h>

3 int
4 main(int argc, char **argv)
5 {
6     int listenfd, connfd;
7     struct sockaddr_in servaddr;
8     char buff[MAXLINE];
9     time_t ticks;

10    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
11    bzero(&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14    servaddr.sin_port = htons(13); /* сервер времени и даты */

15    Bind(listenfd, (SA*)&servaddr, sizeof(servaddr));

16    Listen(listenfd, LISTENQ);

17    for (;;) {
18        connfd = Accept(listenfd, (SA*)NULL, NULL);

19        ticks = time(NULL);
20        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
21        Write(connfd, buff, strlen(buff));

22        Close(connfd);
23    }
24 }
```

10 Создание сокета TCP выполняется так же, как и в клиентском коде.

Связывание заранее известного порта сервера с сокетом

11-15 Заранее известный порт сервера (13 в случае сервера времени и даты) связывается с сокетом путем заполнения структуры адреса интернет-сокета и вызова функции `bind`. Мы задаем IP-адрес как `INADDR_ANY`, что позволяет серверу принимать соединение клиента на любом интерфейсе в том случае, если узел сервера имеет несколько интерфейсов. Далее мы рассмотрим, как можно ограничить прием соединений одним-единственным интерфейсом.

Преобразование сокета в прослушиваемый сокет

16 С помощью вызова функции `listen` сокет преобразуется в прослушиваемый, то есть такой, на котором ядро принимает входящие соединения от клиентов. Эти три этапа, `socket`, `bind` и `listen`, обычны для любого сервера TCP при создании того, что мы называем *прослушиваемым дескриптором* (*listening descriptor*) (в нашем примере это переменная `listenfd`).

Константа `LISTENQ` взята из нашего заголовочного файла `inpr.h`. Она задает максимальное количество клиентских соединений, которые ядро ставит в очередь на прослушиваемом сокете. Более подробно мы расскажем о таких очередях в разделе 4.5.

Прием клиентского соединения, отправка ответа

17-21 Обычно процесс сервера блокируется при вызове функции `accept`, ожидая принятия подключения клиента. Для установки TCP-соединения используется *трехэтапное рукопожатие* (*three-way handshake*). Когда рукопожатие состоялось, функция `accept` возвращает значение, и это значение является новым дескриптором (`connfd`), который называется *присоединенным дескриптором* (*connected descriptor*). Этот новый дескриптор используется для связи с новым клиентом. Новый дескриптор возвращается функцией `accept` для каждого клиента, соединяющегося с нашим сервером.

ПРИМЕЧАНИЕ

Стиль, используемый в книге для обозначения бесконечного цикла, выглядит так:

```
for (;;) {  
    ...  
}
```

Библиотечная функция `time` возвращает количество секунд с начала эпохи Unix: 00:00:00 1 января 1970 года UTC (Universal Time Coordinated — универсальное синхронизированное время, среднее время по Гринвичу). Следующая библиотечная функция, `ctime`, преобразует целочисленное значение секунд в строку следующего формата, удобного для человеческого восприятия:

```
Fri Jan 12 14:27:52 1996
```

Возврат каретки и пустая строка добавляются к строке функцией `snprintf`, а результат передается клиенту функцией `write`.

ПРИМЕЧАНИЕ

Если вы еще не выработали у себя привычку пользоваться функцией `snprintf` вместо устаревшей `sprintf`, сейчас самое время заняться этим. Функция `sprintf` не в состоянии обеспечить проверку переполнения буфера получателя. Функция `snprintf`, наоборот, требует, чтобы в качестве второго аргумента указывался размер буфера получателя, переполнение которого таким образом предотвращается.

Функция `sprintf` была добавлена в стандарт ANSI C относительно нравно, в версии ISO C99. Практически все поставщики программного обеспечения уже сейчас включают эту функцию в стандартную библиотеку языка С. Существуют и свободно распространяемые реализации. В нашей книге мы используем функцию `sprintf` и рекомендуем вам пользоваться ею в своих программах для повышения их надежности.

Удивительно много сетевых атак было реализовано хакерами с использованием незащищенности `sprintf` от переполнения буфера. Есть еще несколько функций, с которыми нужно быть аккуратными: `gets`, `strcat` и `strcpy`. Вместо них лучше использовать `fgets`, `strncat` и `strncpy`. Еще лучше работают более современные функции `strlcat` и `strlcpy`, возвращающие в качестве результата правильно завершенную строку. Полезные советы, касающиеся написания надежных сетевых программ, можно найти в главе 23 книги [32].

Завершение соединения

22 Сервер закрывает соединение с клиентом, вызывая функцию `close`. Это инициирует обычную последовательность прерывания соединения TCP: пакет FIN посыпается в обоих направлениях, и каждый пакет FIN распознается на другом конце соединения. Более подробно трехэтапное рукопожатие и четыре пакета TCP, используемые для прерывания соединения, будут описаны в разделе 2.6.

Сервер времени и даты был рассмотрен нами достаточно кратко, как и клиент из предыдущего раздела. Запомните следующие моменты.

■ Сервер, как и клиент, зависим от протокола IPv4. В листинге 11.7 мы покажем версию, не зависящую от протокола, которая использует функцию `getaddrinfo`.

■ Наш сервер обрабатывает только один запрос клиента за один раз. Если приблизительно в одно время происходит множество клиентских соединений, ядро ставит их в очередь, максимальная длина которой регламентирована, и передает эти соединения функции `accept` по одному за один раз. Наш сервер времени и даты, который требует вызова двух библиотечных функций, `time` и `ctime`, является достаточно быстрым. Но если у сервера обслуживание каждого клиента занимает больше времени (допустим, несколько секунд или минуту), нам будет необходимо некоторым образом организовать одновременное обслуживание нескольких клиентов.

Сервер, показанный в листинге 1.5, называется *последовательным сервером (iterative server)*, поскольку он обслуживает клиентов последовательно, по одному клиенту за один раз. Существует несколько технологий написания *параллельного сервера (concurrent server)*, который обслуживает множество клиентов одновременно. Самой простой технологией является вызов функции Unix `fork` (раздел 4.7), когда создается по одному дочернему процессу для каждого клиента. Другой способ — использование программных потоков (*threads*) вместо функции `fork` (раздел 26.4) или предварительное порождение фиксированного количества дочерних процессов с помощью функции `fork` в начале работы (раздел 30.6).

■ Запуская такой сервер из командной строки, мы обычно рассчитываем, что он будет работать достаточно долго, поскольку часто серверы работают, пока работает система. Поэтому мы должны модифицировать код сервера таким образом, чтобы он корректно работал как *демон (daemon)* Unix, то есть процесс, функционирующий в фоновом режиме без подключения к терминалу. Это решение подробно описано в разделе 13.4.

1.6. Таблица соответствия примеров технологии клиент-сервер

Технологии сетевого программирования иллюстрируются в этой книге на двух основных примерах:

- клиент-сервер времени и даты (описание которого мы начали в листингах 1.1, 1.2 и 1.5), и
- эхо-клиент-сервер (который появится в главе 5).

Чтобы обеспечить удобный поиск различных тем, которых мы касаемся в этой книге, мы объединили разработанные нами программы и сопроводили их номерами листингов, в которых приведен исходный код. В табл. 1.1 перечислены версии клиента времени и даты (две из них мы уже видели). В табл. 1.2 перечисляются версии сервера времени и даты. В табл. 1.3 представлены версии эхо-клиента, а в табл. 1.4 — версии эхо-сервера.

Таблица 1.1. Различные версии клиента времени и даты

Листинг Описание

- 1.1 TCP/Ipv4, зависимый от протокола
- 1.2 TCP/Ipv6, зависитый от протокола
- 11.2 TCP/Ipv4, зависимый от протокола, вызывает функции gethostbyname и getservbyname
- 11.5 TCP, независимый от протокола, вызывает функции getaddrinfo и tcp_connect
- 11.10 UDP, независимый от протокола, вызывает функции getaddrinfo и udp_connect
- 16.7 TCP, использует неблокирующую функцию connect
- 31.2 TCP/IPV4, зависимый от протокола
- Д.1 TCP, зависимый от протокола, генерирует SIGPIPE
- Д.2 TCP, зависимый от протокола, печатает размер буфера сокета и MSS
- Д.5 TCP, зависимый от протокола, допускает использование имени узла (функция gethostbyname) или IP-адреса
- Д.6 TCP, независимый от протокола, допускает использование имени узла (функция gethostbyname).

Таблица 1.2. Различные версии сервера времени и даты, рассматриваемые в данной книге

Листинг Описание

- 1.5 TCP/IPV4, зависимый от протокола
- 11.7 TCP, независимый от протокола, вызывает getaddrinfo и tcp_listen
- 11.8 TCP, независимый от протокола, вызывает getaddrinfo и tcp_listen
- 11.13 UDP, независимый от протокола, вызывает getaddrinfo и udp_server
- 13.2 TCP, независимый от протокола, выполняется как автономный демон
- 13.4 TCP, независимый от протокола, порожденный демоном inetd

Таблица 1.3. Различные версии эхо-клиента, рассматриваемые в данной книге

Листинг Описание

- 5.3 TCP/IPV4, зависимый от протокола
- 6.1 TCP, использует функцию select
- 6.2 TCP, использует функцию select и работает в пакетном режиме
- 8.3 UDP/IPV4, зависимый от протокола
- 8.5 UDP, проверяет адрес сервера
- 8.7 UDP, вызывает функцию connect для получения асинхронных ошибок
- 14.2 UDP, тайм-аут при чтении ответа сервера с использованием сигнала SIGALRM
- 14.4 UDP, тайм-аут при чтении ответа сервера с использованием функции select
- 14.5 UDP, тайм-аут при чтении ответа сервера с использованием опции сокета SO_RCVTIMEO
- 14.7 TCP, использует интерфейс /dev/poll
- 14.8 TCP, использует интерфейс kqueue
- 15.4 Поток домена Unix, зависит от протокола
- 15.6 Дейтаграмма домена Unix, зависит от протокола
- 16.1 TCP, использует неблокируемый ввод-вывод
- 16.6 TCP, использует два процесса (функцию fork)
- 16.14 TCP, устанавливает соединение, затем посыпает пакет RST
- 20.1 UDP, широковещательный, ситуация гонок
- 20.2 UDP, широковещательный, ситуация гонок
- 20.3 UDP, широковещательный, для устранения ситуации гонок используется функция pselect
- 20.5 UDP, широковещательный, для устранения ситуации гонок используются функции sigsetjmp и siglongjmp
- 20.6 UDP, широковещательный, для устранения ситуации гонок в обработчике сигнала используется IPC
- 22.4 UDP, увеличение надежности протокола за счет применения повторной передачи, тайм-аутов и порядковых номеров

- 26.1 TCP, использование двух потоков
- 27.4 TCP/IPv4, задание маршрута от отправителя
- 27.5 UDP/IPv6, задание маршрута от отправителя

Таблица 1.4. Различные версии эхо-сервера, рассматриваемые в данной книге

Листинг Описание

- 5.1 TCP/IPv4, зависимый от протокола
- 5.9 TCP/IPv4, зависимый от протокола, корректно обрабатывает завершение всех дочерних процессов
- 6.3 TCP/IPv4, зависимый от протокола, использует функцию select, один процесс обрабатывает всех клиентов
- 6.5 TCP/IPv4, зависимый от протокола, использует функцию poll, один процесс обрабатывает всех клиентов
- 8.1 UDP/IPv4, зависимый от протокола
- 8.14 TCP и UDP/IPv4, зависимый от протокола, использует функцию select
- 14.6 TCP, использует стандартный ввод-вывод
- 15.3 Доменный сокет Unix, зависимый от протокола
- 15.5 Дейтаграмма домена Unix, зависит от протокола
- 15.13 Доменный сокет Unix, с передачей данных, идентифицирующих клиента
- 22.3 UDP, печатает полученный IP-адрес назначения и имя полученного интерфейса, обрезает дейтаграммы
- 22.13 UDP, связывает все адреса интерфейсов
- 25.2 UDP, использование модели ввода-вывода, управляемого сигналом
- 26.2 TCP, один поток на каждого клиента
- 26.3 TCP, один поток на каждого клиента, машинонезависимая (переносимая) передача аргумента
- 27.4 TCP/IPv4, печатает полученный маршрут от отправителя
- 27.6 UDP/IPv4, печатает полученный маршрут от отправителя и обращает его
- 28.21 UDP, использует функцию icmpd для получения асинхронных ошибок
- Д.9 UDP, связывает все адреса интерфейсов

1.7. Модель OSI

Распространенным способом описания уровней сети является предложенная Международной организацией по стандартизации (International Standards Organization, ISO) модель взаимодействия открытых систем (*open systems interconnection, OSI*). Эта семиуровневая модель показана на рис. 1.5, где она сравнивается со стеком протоколов Интернета.

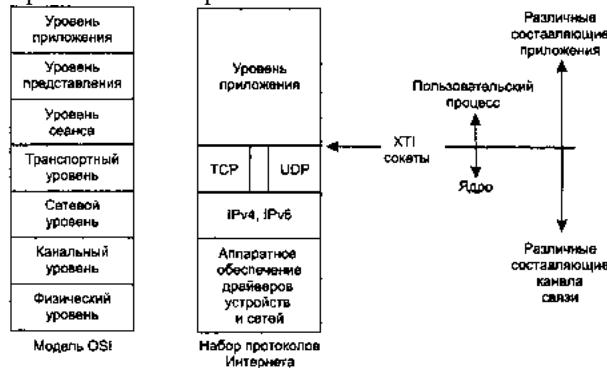


Рис. 1.5. Уровни модели OSI и набор протоколов Интернета

Мы считаем, что два нижних уровня модели OSI соответствуют драйверу устройства и сетевому оборудованию, которые имеются в системе. Обычно нам не приходится беспокоиться об этих уровнях, за

исключением того, что мы должны знать некоторые свойства канального уровня — например, что MTU (максимальная единица передачи) Ethernet, которая описывается в разделе 2.11, имеет размер 1500 байт.

Сетевой уровень управляется протоколами IPv4 и IPv6, оба они описываются в приложении А. Из протоколов транспортного уровня мы можем выбирать TCP, UDP и SCRIPT, они описываются в главе 2. На рис. 1.5 изображен разрыв между TCP и UDP; это означает, что приложение может обойти транспортный уровень и использовать IPv4 или IPv6 непосредственно. В таких случаях речь идет о *символьных сокетах* (*raw socket*), которые будут описаны в главе 28.

Три верхних уровня модели OSI соответствуют уровню приложений. Приложением может быть веб-клиент (браузер), клиент Telnet, веб-сервер, сервер FTP или любое другое используемое нами приложение. В случае протоколов Интернета три верхние уровня модели OSI разделяются очень редко.

Описанный в этой книге API сокетов является интерфейсом между верхними тремя уровнями («приложением») и транспортным уровнем. Это один из важнейших вопросов книги: как создавать приложения, используя сокеты TCP и UDP. Мы уже упоминали о символьных сокетах, и в главе 29 мы увидим, что можем даже полностью обойти уровень IP, чтобы читать и записывать свои собственные кадры канального уровня.

Почему сокеты предоставляют интерфейс между верхними тремя уровнями модели OSI и транспортным уровнем? Для подобной организации модели OSI имеются две причины, которые мы отобразили на правой стороне рис. 1.5. Прежде всего, три верхних уровня отвечают за все детали, имеющие отношение к приложению (например, FTP, Telnet, HTTP), но знают мало об особенностях взаимодействия по сети. Четыре же нижних уровня знают мало о приложении, но отвечают за все, что связано с коммуникацией: отправку данных, ожидание подтверждения, упорядочивание данных, приходящих не в должном порядке, расчет и проверку контрольных сумм и т.д. Второй же причиной является то, что верхние три уровня часто формируют так называемый *пользовательский процесс* (*user process*), в то время как четыре нижних уровня обычно поставляются как часть ядра операционной системы. Unix, как и многие современные операционные системы, обеспечивает разделение пользовательского процесса и ядра. Следовательно, интерфейс между уровнями 4 и 5 является естественным местом для создания API.

1.8. История сетевого обеспечения BSD

API сокетов происходит от системы 4.2BSD (Berkeley Software Distribution — программное изделие Калифорнийского университета, в данном случае — адаптированная для Интернета реализация операционной системы Unix, разрабатываемая и распространяющаяся этим университетом), выпущенной в 1983 году. На рис. 1.6 показано развитие различных реализаций BSD и отмечены главные этапы развития TCP/IP. Некоторые изменения API сокетов также имели место в 1990 году в реализации 4.3BSD Reno, когда протоколы OSI были включены в ядро BSD.

Вертикальная цепочка систем на рис. 1.6 от 4.2BSD до 4.4BSD включает версии, созданные группой исследования компьютерных систем (Computer System Research Group, CSRG) университета Беркли. Для использования этих реализаций требовалось, чтобы у получателя уже была лицензия на исходный код для Unix. Однако весь код сетевых программ — и поддержка на уровне ядра (например, стек протоколов TCP/IP и доменные сокеты Unix, а также интерфейс сокетов), и приложения (такие, как клиенты и серверы Telnet и FTP), были разработаны независимо от кода Unix, созданного AT&T. Поэтому начиная с 1989 года университет Беркли начал выпускать реализации системы BSD, не ограниченные лицензией на исходный код Unix. Эти реализации распространялись свободно и, в конечном итоге, стали доступны через анонимные FTP-серверы фактически любому пользователю Интернета.

Последними реализациями Беркли стали 4.4BSD-Lite в 1994 году и 4.4BSD-Lite2 в 1995 году. Нужно отметить, что эти две реализации были затем использованы в качестве основы для других систем: BSD/OS, FreeBSD, NetBSD и OpenBSD, и все четыре до сих пор активно развиваются и совершенствуются. Более подробную информацию о различных реализациях BSD, а также общую историю развития различных систем Unix можно найти в главе 1 книги [74].

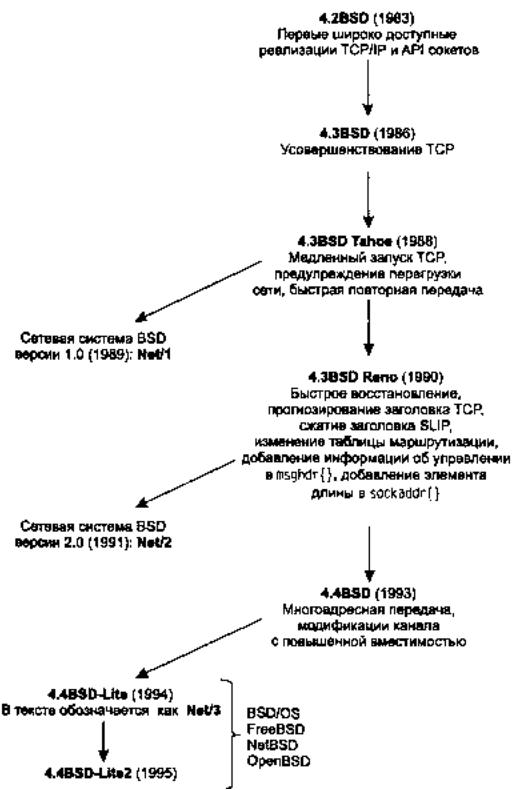


Рис. 1.6. История различных реализаций BSD

Многие системы Unix начинались с некоторой версии сетевого кода BSD, включавшей API сокетов, и мы называем их *реализациями, происходящими от Беркли*, или *Беркли-реализациями (Berkeley-derived implementations)*. Многие коммерческие версии Unix основаны на Unix System V Release 4 (SVR4). Некоторые из них включают сетевой код из Беркли-реализаций (например, UnixWare 2.x), в то время как сетевой код других систем, основанных на SVR4, был разработан независимо (например, Solaris 2.x). Мы также должны отметить, что система Linux, популярная и свободно доступная реализация Unix, *не* относится к классу происходящих от Беркли: ее сетевой код и API сокетов были разработаны «с нуля».

1.9. Сети и узлы, используемые в примерах

На рис. 1.7 показаны различные сети и узлы, используемые нами в примерах. Для каждого узла мы указываем операционную систему и тип компьютера (потому, что некоторые операционные системы могут работать на компьютерах разных типов). Внутри прямоугольников приведены имена узлов, появляющиеся в тексте.

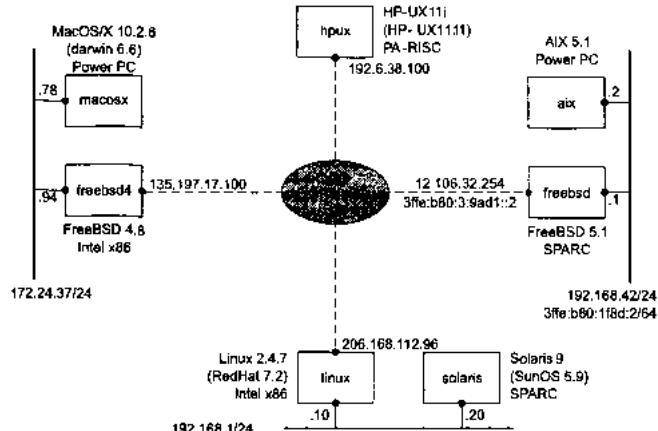


Рис. 1.7. Сети и узлы, используемые в примерах

Топология, приведенная на рис. 1.7, интересна для наших примеров, но на практике физическая топология сети оказывается не столь важной, поскольку взаимодействующие компьютеры обычно связываются через Интернет. Виртуальные частные сети (virtual private network, VPN) и защищенные подключения интерпретатора (secure shell connections, SSH) обеспечивают соединение, не зависящее от физического размещения компьютеров.

Обозначение «/24» указывает количество последовательных битов адреса начиная с крайнего левого, задающих сеть и подсеть. В разделе А.4 об этом формате рассказывается более подробно.

ПРИМЕЧАНИЕ

Хотим подчеркнуть, что настоящее имя операционной системы Sun — SunOS 5.x, а не Solaris 2.x, однако все называют ее Solaris.

Определение топологии сети

На рис. 1.7 мы показываем топологию сети, состоящей из узлов, используемых в качестве примеров в этой книге, но вам нужно знать топологию вашей собственной сети, чтобы запускать в ней примеры и выполнять упражнения. Хотя в настоящее время не существует стандартов Unix в отношении сетевой конфигурации и администрирования, большинство Unix-систем предоставляют две основные команды, которые можно использовать для определения подробностей строения сети: netstat и ifconfig. Мы приводим примеры в различных системах, представленных на рис. 1.7. Изучите руководство, где описаны эти команды для ваших систем, чтобы понять различия в той информации, которую вы получите на выходе. Также имейте в виду, что некоторые производители помещают эти команды в административный каталог, например /sbin или /usr/sbin, вместо обычного /usr/bin, и эти каталоги могут не принадлежать обычному пути поиска (PATH).

1. netstat - i предоставляет информацию об интерфейсах. Мы также задаем флаг -n для печати численных адресов, а не имен сетей. При этом показываются интерфейсы с их именами.

```
linux % netstat -ni
Kernel Interface table
Iface  MTU Met      RX-OK RX-ERR RX-DRP RX-OVR      TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0   1500  0 49211085      0      0      0 40540958      0      0      0 BMRU
lo    16436  0 98613572      0      0      0 98613572      0      0      0 LRU
```

Интерфейс закольцовки называется lo, а Ethernet называется eth0. В следующем примере показан узел с поддержкой IPv6.

```
freebsd % netstat -ni
Name  Mtu Network      Address          Ipkts Ierrs      Opkts Oerrs Coll
hme0  1500 <Link#1>  08:00:20:a7:68:6b 29100435    35 46561488      0      0
hme0  1500 12.106.32/24 12.106.32.254    28746630     - 46617260      -      -
hme0  1500 fe80:1::a00:20ff:fea7 686b/64
                           fe80:1::a00:20ff:fea7:68b
                                         0      -      0      -      -
hme0  1500 3ffe:b80:1f8d:1::1/64
                           3ffe:b80:1f8d:1::1      0      -      0      -      -
hme1  1500 <Link#2>  08:00:20:a7:68:6b 51092      0 31537      0      0
hme1  1500 fe80:2::a00:20ff:fea7:686b/64
                           fe80:2::a00:20ff:fea7:68b
                                         0      -      90      -      -
hme1  1500 192.168.42  192.168.42.1    43584      - 24173      -      -
hme1  1500 3ffe:b80:1f8d:2::1/64
                           3ffe:b80:1f8d:2::1      78      -      8      -      -
lo0   16384 <Link#6>
                           ::1/128      10      -      10      -      -
lo0   16384 fe80:6::1/64 fe80:6::1      0      -      0      -      -
lo0   16384 127          127.0.0.1     10167      - 10167      -      -
```

```

gif0 1280 <Link#8>          6      0      5      0      0
gif0 1280 3ffe:b80:3:9ad1::2/128
                           3ffe:b80:3:9ad1::2      0      -      0      -      -
gif0 1280 fe80:8::a00:20ff:fea7:686b/64
                           fe80:8::a00:20ff:fea7:686b      0      -      0      -      -

```

Мы разбили некоторые длинные строки на несколько частей, чтобы сохранить ясность представления.

2. netstat -r показывает таблицу маршрутизации, которая тоже позволяет определить интерфейсы. Обычно мы задаем флаг -n для печати численных адресов. При этом также приводится IP-адрес маршрутизатора, заданного по умолчанию:

```
freebsd % netstat -nr
Routing tables
```

Internet:

Destination	Gateway	Flags	Refs	Use	Netif	Expire
default	12.106.32.1	UGSc	10	6877	hme0	
12.106.32/24	link#1	UC	3	0	hme0	
12.106.32.1	00:b0:8e:92:2c:00	UHLW	9	7	hme0	1187
12.106.32.253	08:00:20:b8:f7:e0	UHLW	0	1	hme0	140
12.106.32.254	08:00:20:a7:68:b6	UHLW	0	2	lo0	
127.0.0.1	127.0.0.1	UH		1	10167	lo0
192.168.42	link#2	UC	2	0	hme1	
192.168.42.1	08:00:20:a7:68:6b	UHLW	0	11	lo0	
192.168.42.2	00:04:ac:17:bf:38	UHLW	2	24108	hme1	210

Internet6:

Destination	Gateway	Flags	Netif	Expire
::/96	::1	UGRSc	lo0	=>
default	3ffe:b80:3:9ad1::1	UGSc	gif0	
::1	::1	UH	lo0	
::ffff:0.0.0.0/96	::1	UGRSc	lo0	
3ffe:b80:3:9ad1::1	3ffe:b80:3:9ad1::2	UH	gif0	
3ffe:b80:3:9ad1::2	link#8	UHL	lo0	
3ffe:b80:1f8d::/48	lo0	USC	lo0	
3ffe:b80:1f8d::1::/64	link#1	UC	hme0	
3ffe:b80:1f8d::1::1	08:00:20:a7:68:6b	UHL	lo0	
3ffe:b80:1f8d::2::/64	link#2	UC	hme1	
3ffe:b80:1f8d::2::1	08:00:20:a7:68:6b	UHL	lo0	
3ffe:b80:1f8d::2:204:acff:fe17:bf38	00:04.ac:17:bf:38	UHLW	hme1	
fe80::/10	::1	UGRSc	lo0	
fe80::%hme0/64	link#1	UC	hme0	
fe80::a00:20ff:fea7:686b%hme0	08:00:20:a7:68:6b	UHL	lo0	
fe80::%hme1/64	link#2	UC	hme1	
fe80::a00:20ff:fea7:686b%hme1	08:00:20:a7:68:6b	UHL	lo0	
fe80::%lo0/64	fe80::%lo0	Uc	lo0	
fe80::1%lo0	link#6	UHL	lo0	
fe80::%gif0/64	link#8	UC	gif0	
fe80::a00:20ff:fea7:686b%gif0	link#8	UHL	lo0	
ff01::/32	::1	U	lo0	
ff02::/16	::1	UGRS	lo0	
ff02::%hme0/32	link#1	UC	hme0	
ff02::%hem1/32	link#2	UC	hme1	
ff02::%lo0/32	::1	UC	lo0	
ff02::%gif0/32	link#8	UC	gif0	

3. Имея имена интерфейсов, мы выполняем команду ifconfig, чтобы получить подробную информацию для каждого интерфейса:

```
linux % ifconfig eth0
eth0 Link encap:Ethernet HWaddr 00:C0:9F:06:B0:E1
      inet addr:206.168.112.96 Bcast:206.168.112.127 Mask:255.255.255.128
            UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
            RX packets:49214397 errors:0 dropped:0 overruns:0 frame:0
            TX packets:40543799 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:100
            RX bytes:1098069974 (1047.2 Mb) TX bytes:3360546472 (3204.8 Mb)
            Interrupt:11 Base address:0x6000
```

При этом мы получаем IP-адрес, маску подсети и широковещательный адрес. Флаг **MULTICAST** указывает на то, что узел поддерживает широковещательную передачу. В некоторых реализациях поддерживается флаг **-a**, при указании которого печатается информация обо всех сконфигурированных интерфейсах.

4. Одним из способов определить IP-адрес нескольких узлов локальной сети является проверка широковещательного адреса (найденного нами на предыдущем шаге) с помощью утилиты ping.

```
linux % ping -b 206.168.112.127
WARNING: pinging broadcast address
PING 206.168.112.127 (206.168.112.127) from 206.168.112.96 : 56 (84) bytes of data.
64 bytes from 206.168.112.96: icmp_seq=0 ttl=255 time=241 usec
64 bytes from 206.168.112.40: icmp_seq=0 ttl=255 time=2 566 msec (DUP!)
64 bytes from 206.168.112.118: icmp_seq=0 ttl=255 time=2.973 msec (DUP!)
64 bytes from 206.168.112.14: icmp_seq=0 ttl=255 time=3.089 msec (DUP!)
64 bytes from 206.168.112.126: icmp_seq=0 ttl=255 time=3.200 msec (DUP!)
64 bytes from 206.168.112.71: icmp_seq=0 ttl=255 time=3.311 msec (DUP!)
64 bytes from 206.168.112.31: icmp_seq=0 ttl=255 time=3.541 msec (DUP!)
64 bytes from 206.168.112.7: icmp_seq=0 ttl=255 time=3.636 msec (DUP!)
...
```

1.10. Стандарты Unix

Когда писалась эта книга, наибольший интерес в сфере стандартизации Unix вызывала деятельность группы Остина по пересмотру общих стандартов (The Austin Common Standards Revision Group, CSRG). Ими было написано в общей сложности около 4000 страниц спецификаций, описывающих более 1700 интерфейсов программирования. Эти спецификации являются одновременно стандартами IEEE POSIX и The Open Group. Поэтому один и тот же стандарт может встретиться вам под разными названиями, например ISO/IEC 9945:2002, IEEE Std 1003.1-2001 и Single Unix Specification Version 3. В нашей книге мы будем называть этот стандарт просто: спецификация POSIX, за исключением разделов, подобных этому, где обсуждаются особенности различных более старых стандартов.

Проще всего получить копию этого консолидированного стандарта, заказав ее на компакт-диске или скачав из Сети (бесплатно). В любом случае начинать следует с <http://www.UNIX.org/version3>.

История POSIX

Слово «POSIX» представляет собой сокращение от «Portable Operating System Interface» (интерфейс переносимой операционной системы). POSIX — целое семейство стандартов, разрабатываемых организацией IEEE (Institute of Electrical and Electronics Engineers — Институт инженеров по электротехнике и радиоэлектронике). Стандарты POSIX также приняты в качестве международных стандартов ISO (International Standards Organization — Международная организация по стандартизации) и IEC (International Electrotechnical Commission — Международная комиссия по электротехнике), называемых ISO/IEC. История стандартов POSIX достаточно интересна, но мы рассмотрим ее кратко.

■ Первым из стандартов POSIX был IEEE Std 1003.1-1988 (317 страниц), и он определял интерфейс между языком C и оболочкой ядра типа Unix в следующих областях: примитивы процесса (`fork`, `exec`, сигналы, таймеры), среда процесса (идентификаторы пользователя, группы процессов), файлы и каталоги (все функции ввода-вывода), ввод-вывод на терминал, системные базы данных (файлы паролей и групп) и архивные форматы `tar` и `cpio`.

ПРИМЕЧАНИЕ

Первый стандарт POSIX был пробной версией, выпущенной в 1986 году и известной как IEEE-IX. Название «POSIX» было предложено Ричардом Столлмэном (Richard Stallman).

- Следующим был IEEE Std 1003.2-1990 (356 страниц), который стал международным стандартом (ISO/IEC 9945-1:1990). По сравнению с версией 1988 году в версии 1990 года были внесены минимальные изменения. К названию было добавлено «Часть 1: Системный программный интерфейс приложений [язык C]», что указывало, что этот стандарт являлся интерфейсом API, написанным на языке C.
- Затем был выпущен двухтомный стандарт IEEE Std 1003.2-1992 (около 1300 страниц). Второй том был озаглавлен «Часть 2: интерпретатор и утилиты» и описывал интерпретатор команд (Основанный на интерпретаторе System V Bourne Shell) и порядка сотни утилит (программ, запускаемых из интерпретатора, от awk и basename до vi и yacc). В тексте мы будем называть этот стандарт *POSIX.2*.
- IEEE Std 1003.1b-1993 (590 страниц) изначально назывался IEEE 1003.4. Он стал дополнением стандарта 1003.1-1990 и включал расширения реального времени, разработанные группой Р1003.4. Стандарт 1003.1b-1993 добавил к стандарту 1990 года следующие пункты: синхронизацию файлов, асинхронный ввод-вывод, семафоры, управление памятью (вызов `mmap` и разделяемая память), планирование выполнения, часы, таймеры и очереди сообщений.
- Следующий стандарт POSIX — IEEE Std 1003.1, редакция 1996 года [50], включил в себя 1003.1-1990 (базовый API), 1003.1b-1993 (расширения реального времени), 1003.1c-1995 (функции управления потоками) и 1003.1i-1995 (технические исправления 1003.1b). Этот стандарт также называется ISO/IEC 9945-1:1996. Были добавлены три главы, посвященные программным потокам, и общий объем стандарта составил 743 страницы. В тексте мы будем называть его *POSIX.1*. В стандарт включено предисловие, где говорится, что стандарт ISO/IEC 9945 состоит из следующих частей:

- Часть 1. Системный API [язык C].
- Часть 2. Оболочка и утилиты.
- Часть 3. Системное администрирование (в стадии разработки).

Части 1 и 2 — это именно то, что мы называем *POSIX.1* и *POSIX.2*.

ПРИМЕЧАНИЕ

Более четверти из 743 страниц отводится приложению, названному «Обоснование и замечания» («Rationale and Notes»). Это обоснование содержит историческую информацию и причины, по которым те или иные функции были включены или опущены. Часто обоснование бывает столь же информативным, как и официальный стандарт.

- Стандарт IEEE Std 1003.1g: Protocol Independent Interfaces (PII) (интерфейсы, не зависящие от протокола) был принят в 2000 году. До появления единой спецификации Unix версии 3 этот стандарт имел наибольшее отношение к тематике данной книги, потому что он определяет сетевые API (называя их DNI — Detailed Network Interfaces, подробные сетевые интерфейсы):
 - 1) DNI/Socket, основанный на API сокетов 4.4BSD;
 - 2) DNI/XTI, основанный на спецификации X/Open XPG4.

Работа над этим стандартом началась в 80-х (рабочая группа Р1003.12, позже переименованная в Р1003.1g). В тексте мы будем называть его *POSIX.1g*.

Текущее состояние различных стандартов POSIX можно получить в Интернете по адресу <http://www.pasc.org/standing/sd11.html>.

История Open Group

The Open Group (Открытая группа) была сформирована в 1996 году объединением организаций X/Open Company (основана в 1984 году) и Open Software Foundation (OSF, основан в 1988 году). Эта группа представляет собой международный консорциум производителей и потребителей из промышленности, правительства и образовательных учреждений. Их стандарты тоже выходили в нескольких версиях.

- В 1989 году X/Open опубликовала третий выпуск X/Open Portability Guide (Руководство по разработке переносимых программ) — XPG3.
- В 1992 году был опубликован четвертый выпуск (Issue 4), а в 1994 — вторая его версия (Issue 4, Version 2). Последняя известна также под названием Spec 1170, где магическое число 1170 представляет собой сумму количества интерфейсов системы (926), заголовков (70) и команд (174). Есть и еще два названия: X/Open Single Unix Specification (Единая спецификация Unix) и Unix 95.
- В марте 1997 года было объявлено о выходе второй версии Единой спецификации Unix. Этот стандарт программного обеспечения называется также Unix 98, и именно так мы называем эту спецификацию далее в тексте книги. Количество интерфейсов в Unix 98 возросло с 1170 до 1434, хотя для рабочей станции это количество достигает 3030, поскольку сюда входит CDE (Common Desktop Environment — общее окружение рабочего стола), которое, в свою очередь, требует системы X Window System и пользовательского интерфейса Motif. Подробно об этом написано в книге [55]. Полезную информацию можно также найти по адресу <http://www.UNIX.org/version2>. Сетевые службы, входящие в Unix 98, определяются как для API сокетов, так и для XTI. Эта спецификация практически идентична POSIX.1g.

ПРИМЕЧАНИЕ

К сожалению, X/Open обозначает свои сетевые стандарты с помощью аббревиатуры «XNS» — X/Open Networking Services. Например, версия этого документа, в которой определяются сокеты и технологии XTI для Unix 98 [86], называется «XNS Issue 5*». Дело в том, что в мире сетевых технологий аббревиатура «XNS» всегда служила акронимом для «Xerox Network Systems» (сетевые системы Xerox). Поэтому мы избегаем использования акронима «XNS» и называем соответствующий документ X/Open просто стандартом сетевого API Unix 98.

Объединение стандартов

Краткую историю POSIX и The Open Group продолжает опубликованная CSRG третья версия единой спецификации Unix (The Single Unix Specification Version 3), о которой уже шла речь в начале раздела. Добиться принятия единого стандарта пятьюдесятью производителями — заметная веха в истории Unix. Большинство сегодняшних Unix-систем отвечают требованиям какой-либо версии POSIX.1 и POSIX.2, а многие уже соответствуют третьей единой спецификации Unix.

Исторически для большинства Unix-систем четко прослеживалось родство либо с BSD, либо с SVR4, но различия между современными системами постепенно стираются по мере того, как производители принимают новые стандарты. Наиболее существенные из оставшихся отличий связаны с администрированием систем, которое пока не охватывается никакими стандартами.

Эта книга основана на третьей версии единой спецификации Unix, причем основное внимание уделяется API сокетов. Везде, где это возможно, мы используем исключительно стандартные функции.

Internet Engineering Task Force

IETF (Internet Engineering Task Force — группа, отвечающая за решение сетевых инженерных задач) — это большое открытое международное сообщество сетевых разработчиков, операторов, производителей и исследователей, работающих в области развития архитектуры Интернета и более стабильной его работы. Это сообщество открыто для всех желающих.

Стандарты Интернета документированы в RFC 2026 [13]. Обычно стандарты Интернета описывают протоколы, а не интерфейсы API. Тем не менее два документа RFC (RFC 3493 [36] и RFC 3542 [114]) определяют API сокетов для протокола IP версии 6. Это информационные документы RFC, а не стандарты, и они были выпущены для того, чтобы ускорить применение переносимых приложений различными производителями, работающими с более ранними реализациями IPv6. Разработка текстов стандартов занимает много времени, но в третьей версии единой спецификации многие API были успешно стандартизованы.

1.11. 64-разрядные архитектуры

С середины до конца 90-х годов развивается тенденция к переходу на 64-разрядные архитектуры и 64-разрядное программное обеспечение. Одной из причин является более значительная по размеру адресация внутри процесса (например, 64-разрядные указатели), которая необходима в случае использования больших объемов памяти (более 2^{32} байт). Обычная модель программирования для существующих 32-разрядных систем Unix называется *ILP32*. Ее название указывает на то, что целые числа (I), длинные целые числа (L) и указатели (P) занимают 32 бита. Модель, которая получает все большее распространение для 64-разрядных систем Unix, называется *LP64*. Ее название говорит о том, что 64 бита требуется только для длинных целых чисел (L) и указателей (P). В табл. 1.5 приводится сравнение этих двух моделей.

Таблица 1.5. Сравнение количества битов для хранения различных типов, данных в моделях ILP32 и LP64

Тип данных Модель ILP32 Модель LP64

Char	8	8
Short	16	16
Int	32	32
Long	32	64
Указатель	32	64

С точки зрения программирования модель LP64 означает, что мы не можем рассматривать указатель как целое число. Мы также должны учитывать влияние модели LP64 на существующие API.

В ANSI C введен тип данных `size_t`, который используется, например в качестве аргумента функции `malloc` (количество байтов, которое данная функция выделяет в памяти для размещения какого-либо объекта), а также как третий аргумент для функций `read` и `write` (число считываемых или записываемых байтов). В 32-разрядной системе значение типа `size_t` является 32-разрядным, но в 64-разрядной системе оно должно быть 64-разрядным, чтобы использовать преимущество большей модели адресации. Это означает, что в 64-разрядной системе, возможно, `size_t` будет иметь тип `unsigned long` (целое число без знака, занимающее 32 разряда). Проблемой сетевого интерфейса API является то, что в некоторых проектах по POSIX.1g было определено, что аргументы функции, содержащие размер структур адресов сокета, должны иметь тип `size_t` (например, третий аргумент в функциях `bind` и `connect`). Некоторые поля структуры XTI также имели тип данных `long` (например, структуры `t_info` и `t_opthdr`). Если бы стандарты остались неизменными, в обоих случаях 32-разрядные значения должны были бы смениться 64-разрядными при переходе с модели ILP32 на LP64. В обоих случаях нет никакой необходимости в 64-разрядных типах данных: длина структуры адресов сокета занимает максимум несколько сотен байтов, а использование типа данных `long` для полей структуры XTI было просто ошибкой.

Решение состоит в том, чтобы использовать типы данных, разработанные специально для борьбы с подобными проблемами. Интерфейс API сокетов использует тип данных `socklen_t` для записи длины структур адресов сокетов, а XTI использует типы данных `t_scalar_t` и `t_ustralar_t`. Причина, по которой эти 32-разрядные значения не заменяются на 64-разрядные, заключается в том, что таким образом упрощается двоичная совместимость с новыми 64-разрядными системами для приложений, скомпилированных под 32-разрядные системы.

1.12. Резюме

В листинге 1.1 показан полностью рабочий, хотя и простой, клиент TCP, который получает текущее время и дату с заданного сервера. В листинге 1.5 представлена полная версия сервера. На этих примерах вводятся многие термины и понятия, которые далее рассматриваются более подробно. Наш клиент был зависим от протокола, и мы изменили его, чтобы он использовал IPv6. Но при этом мы получили всего лишь еще одну зависимую от протокола программу. В главе 11 мы разработаем некоторые функции, которые позволят нам написать код, не зависящий от протокола. Это важно, поскольку в Интернете начинает использоваться протокол IPv6. По ходу книги мы будем использовать функции-обертки, созданные в разделе 1.4, для уменьшения размера нашего кода, хотя по-прежнему каждый вызов функции будет проходить проверку на предмет возвращения ошибки. Все имена наших функций-оберток начинаются с заглавной буквы.

Третья версия единой спецификации Unix, известная также под несколькими другими названиями (мы называем ее просто «Спецификация POSIX»), представляет собой результат слияния двух стандартов,

осуществленного The Austin Group.

Читатели, которых интересует история сетевого программирования в Unix, должны изучить прежде всего историю развития Unix, а история TCP/IP и Интернета представлена в книге [106].

Упражнения

1. Проделайте все шаги, описанные в конце раздела 1.9, чтобы получить информацию о топологии вашей сети.

2. Отыщите исходный код для примеров в тексте (см. предисловие). Откомпилируйте и протестируйте клиент времени и даты, представленный в листинге 1.1. Запустите программу несколько раз, задавая каждый раз различные IP-адреса в командной строке.

3. Замените первый аргумент функции `socket`, представленной в листинге 1.1, на 9999. Откомпилируйте и запустите программу. Что происходит? Найдите значение `errno`, соответствующее выданной ошибке. Как вы можете получить дополнительную информацию по этой ошибке?

4. Измените листинг 1.1: поместите в цикл `while` счетчик, который будет считать, сколько раз функция `read` возвращает значение, большее нуля. Выведите значение счетчика перед завершением. Откомпилируйте и запустите новую программу-клиент.

5. Измените листинг 1.5 следующим образом. Сначала поменяйте номер порта, заданный функции `sin_port`, с 13 на 9999. Затем замените один вызов функции `write` на циклический, при котором функция `write` вызывается для каждого байта результирующей строки. Откомпилируйте полученный сервер и запустите его в фоновом режиме. Затем измените клиент из предыдущего упражнения (в котором выводится счетчик перед завершением программы), изменив номер порта, заданный функции `sin_port`, с 13 на 9999. Запустите этот клиент, задав в качестве аргумента командной строки IP-адрес узла, на котором работает измененный сервер. Какое значение клиентского счетчика будет напечатано? Если это возможно, попробуйте также запустить клиент и сервер на разных узлах.

Глава 2

Транспортный уровень: TCP, UDP и SCRIPT

2.1. Введение

В этой главе приводится обзор протоколов семейства TCP/IP, которые используются в примерах на всем протяжении книги. Наша цель — как можно подробнее описать эти протоколы с точки зрения сетевого программирования, чтобы понять, как их использовать, а также дать ссылки на более подробные описания фактического устройства, реализации и истории протоколов.

В данной главе речь пойдет о транспортном уровне: протоколах TCP, UDP и протоколе управления передачей потоков (Stream Control Transmission Protocol, SCRIPT). Большинство приложений, построенных по архитектуре клиент-сервер, используют либо TCP, либо UDP. Протоколы транспортного уровня, в свою очередь, используют протокол сетевого уровня IP — либо IPv4, либо IPv6. Хотя и возможно использовать IPv4 или IPv6 непосредственно, минуя транспортный уровень, эта технология (символьные сокеты) используется гораздо реже. Поэтому мы даем более подробное описание IPv4 и IPv6 наряду с ICMPv4 и ICMPv6 в приложении А.

UDP представляет собой простой и ненадежный протокол передачидейтаграмм, в то время как TCP является сложным и надежным потоковым протоколом. SCRIPT тоже обеспечивает надежность передачи, как и TCP, но помимо этого он позволяет задавать границы сообщений, обеспечивает поддержку множественной адресации на транспортном уровне, а также минимизирует блокирование линии в начале передачи. Нужно понимать, какие сервисы предоставляют приложениям транспортные протоколы, какие задачи решаются протоколами, а что необходимо реализовывать в приложении.

Есть ряд свойств TCP, которые при должном понимании упрощают написание надежных клиентов и серверов. Знание этих особенностей облегчит нам отладку наших клиентов и серверов с использованием общеупотребительных средств, таких как netstat. В этой главе мы коснемся различных тем, попадающих в эту категорию: трехэтапное рукопожатие TCP, последовательность прерывания соединения TCP, состояние TCP TIME_WAIT, четырехэтапное рукопожатие и завершение соединения SCRIPT, буферизация TCP, UDP и SCRIPT уровнем сокетов и так далее.

2.2. Обзор протоколов TCP/IP

Хотя набор протоколов и называется «TCP/IP», это семейство состоит не только из собственно протоколов TCP и IP. На рис. 2.1 представлен обзор этих протоколов.

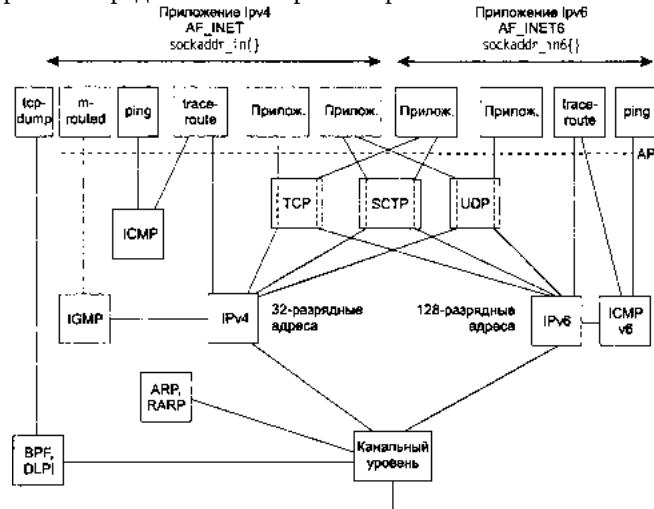


Рис. 2.1. Обзор протоколов семейства TCP/IP

На этом рисунке представлены и IPv4, и IPv6. Если рассматривать этот рисунок справа налево, то пять приложений справа используют IPv6. О константе AF_INET6 и структуре sockaddr_in6 мы говорим в главе 3. Следующие шесть приложений используют IPv4.

Приложение, находящееся в самой левой части рисунка, `tcpdump`, соединяется непосредственно с канальным уровнем, используя либо BPF (BSD Packet Filter — фильтр пакетов BSD), либо DLPI (Data Link Provider Interface — интерфейс канального уровня). Мы обозначили штриховую горизонтальную линию под девятью приложениями (интерфейс) как *API*, что обычно соответствует сокетам или XTI. Интерфейс и к BPF, и к DLPI не использует сокетов или XTI.

ПРИМЕЧАНИЕ

Здесь существует исключение, описанное нами в главе 25: Linux предоставляет доступ к канальному уровню при помощи специального типа сокета, называемого SOCK PACKET.

На рис. 2.1 мы также отмечаем, что программа `traceroute` использует два сокета: один для IP, другой для ICMP. В главе 25 мы создадим версии IPv4 и IPv6 утилит `ping` и `traceroute`.

А сейчас мы опишем каждый из протоколов, представленных на рисунке.

■ **Протокол Интернета версии 4.** IPv4 (Internet Protocol, version 4), который мы часто обозначаем просто как IP, был «рабочей лошадкой» набора протоколов Интернета с начала 80-х. Он использует 32-разрядную адресацию (см. раздел А.4). IPv4 предоставляет сервис доставки пакетов для протоколов TCP, UDP, SCRIPT, ICMP и IGMP.

■ **Протокол Интернета версии 6.** IPv6 (Internet Protocol, version 6) был разработан в середине 90-х как замена протокола IPv4. Главным изменением является увеличение размера адреса, в случае IPv6 равного 128 бит (см. раздел А.5) для работы с бурно развивавшимся в 90-е годы Интернетом. IPv6 предоставляет сервис доставки пакетов для протоколов TCP, UDP, SCRIPT и ICMPv6.

Мы часто используем аббревиатуру «IP» в словосочетаниях типа «IP-адрес», «IP-уровень», когда нет необходимости различать IPv4 и IPv6.

■ **Протокол управления передачей.** TCP (Transmission Control Protocol) является протоколом, ориентированным на установление соединения и предоставляющим надежный двусторонний байтовый поток использующим его приложениям. Сокеты TCP — типичный пример *потоковых сокетов* (*stream sockets*). TCP обеспечивает отправку и прием подтверждений, обработку тайм-аутов, повторную передачу и тому подобные возможности. Большинство прикладных программ в Интернете используют TCP. Заметим, что TCP может использовать как IPv4, так и IPv6.

■ **Протокол пользовательских дейтаграмм.** UDP (User Datagram Protocol) — это протокол, не ориентированный на установление соединения. Сокеты UDP служат примером дейтаграммных сокетов (*datagram sockets*). В отличие от TCP, который является надежным протоколом, в данном случае отнюдь не гарантируется, что дейтаграммы UDP когда-нибудь достигнут заданного места назначения. Как и в случае TCP, протокол UDP может использовать как IPv4, так и IPv6.

■ **Протокол управления передачей потоков.** SCRIPT (Stream Control Transmission Protocol) — ориентированный на установление соединения протокол, предоставляющий надежную двустороннюю ассоциацию. Соединение по протоколу SCRIPT называется *ассоциацией* (*association*), потому что это многоканальный протокол, позволяющий задать несколько IP-адресов и один порт для каждой стороны соединения. SCRIPT предоставляет также сервис сообщений, то есть разграничение отдельных записей в передаваемом потоке. Как и другие транспортные протоколы, SCRIPT может использовать IPv4 и IPv6, но он отличается тем, что может работать с обеими версиями IP на одной и той же ассоциации.

■ **Протокол управляющих сообщений Интернета.** ICMP (Internet Control Message Protocol) обеспечивает передачу управляющей информации и сведений об ошибках между маршрутизаторами и узлами. Эти сообщения обычно генерируются и обрабатываются самостоятельно сетевым программным обеспечением TCP/IP, а не пользовательскими процессами, хотя мы и приводим в качестве примера программы `ping` и `traceroute`, использующие ICMP. Иногда мы называем этот протокол «ICMPv4», чтобы отличать его от ICMPv6.

■ **Протокол управления группами Интернета.** IGMP (Internet Group Management Protocol) используется для многоадресной передачи (см. главу 21), поддержка которой не является обязательной для IPv4.

■ **Протокол разрешения адресов.** ARP (Address Resolution Protocol) ставит в соответствие аппаратному адресу (например, адресу Ethernet) адрес IPv4. ARP обычно используется в широковещательных сетях, таких как Ethernet, Token-ring и FDDI, но не нужен в сетях типа «точка-точка» (point-to-point).

- *Протокол обратного разрешения адресов.* RARP (Reverse Address Resolution Protocol) ставит в соответствие адресу IPv4 аппаратный адрес. Он иногда используется, когда загружается бездисковый узел.
- *Протокол управляющих сообщений Интернета, версия 6.* ICMPv6 (Internet Control Message Protocol, version 6) объединяет возможности протоколов ICMPv4, IGMP и ARP.

■ *Фильтр пакетов BSD.* Этот интерфейс предоставляет доступ к канальному уровню для процесса. Обычно он поддерживается ядрами, произошедшими от BSD.

■ *Интерфейс провайдера канального уровня.* DLPI (Datalink Provider Interface) предоставляет доступ к канальному уровню и обычно предоставляется SVR4 (System V Release 4).

Все протоколы Интернета определяются в документах *RFC (Request For Comments)*, которые играют роль формальной спецификации. Решение к упражнению 2.1 показывает, как можно получить документы RFC.

Мы используем термины узел *IPv4/IPv6 (IPv4/IPv6 host)* и узел с *двойным стеком (dual-stack host)* для определения узла, поддерживающего как IPv4, так и IPv6.

Дополнительные подробности собственно по протоколам TCP/IP можно найти в [111]. Реализация TCP/IP в 4.4BSD описывается в [128].

2.3. UDP: протокол пользовательских дейтаграмм

UDP — это простой протокол транспортного уровня. Он описывается в документе RFC 768 [93]. Приложение записывает в сокет UDP дейтаграмму (*datagram*), которая *инкапсулируется (encapsulate)* или, иначе говоря, упаковывается либо в дейтаграмму IPv4, либо в дейтаграмму IPv6, и затем посыпается к пункту назначения. При этом не гарантируется, что дейтаграмма UDP когда-нибудь дойдет до указанного пункта назначения.

Проблема, с которой мы сталкиваемся в процессе сетевого программирования с использованием UDP, заключается в его недостаточной надежности. Если мы хотим быть уверены в том, что дейтаграмма дошла до места назначения, мы должны встроить в наше приложение множество функций: подтверждение приема, тайм-ауты, повторные передачи и т.п.

Каждая дейтаграмма UDP имеет конкретную длину, и мы можем рассматривать дейтаграмму как *запись (record)*. Если дейтаграмма корректно доходит до места назначения (то есть пакет приходит без ошибки контрольной суммы), длина дейтаграммы передается принимающему приложению. Мы уже отмечали, что TCP является *потоковым (byte-stream)* протоколом, без каких бы то ни было границ записей (см. раздел 1.2), что отличает его от UDP.

Мы также отметили, что UDP предоставляет сервис, не ориентированный на *установление соединения (connectionless)*, поскольку нет необходимости в установлении долгосрочной связи между клиентом и сервером UDP. Например, клиент UDP может создать сокет и послать дейтаграмму данному серверу, а затем сразу же послать через тот же сокет дейтаграмму другому серверу. Аналогично, сервер UDP может получить пять дейтаграмм подряд через один и тот же сокет UDP от пяти различных клиентов.

2.4. TCP: протокол контроля передачи

Сервис, предоставляемый приложению протоколом TCP, отличается от сервиса, предоставляемого протоколом UDP. TCP описывается в документах RFC 793 [96], RFC 1323 [53], RFC 2581 [4], RFC 2988 [91] и RFC 3390 [2]. Прежде всего, TCP обеспечивает установление *соединений (connections)* между клиентами и серверами. Клиент TCP устанавливает соединение с выбранным сервером, обменивается с ним данными по этому соединению и затем разрывает соединение.

TCP также обеспечивает *надежность (reliability)*. Когда TCP отправляет данные на другой конец соединения, он требует, чтобы ему было выслано подтверждение получения. Если подтверждение не приходит, TCP автоматически передает данные повторно и ждет в течение большего количества времени. После некоторого числа повторных передач TCP оставляет эти попытки. В среднем суммарное время попыток отправки данных занимает от 4 до 10 минут (в зависимости от реализации).

ПРИМЕЧАНИЕ

TCP не гарантирует получение данных адресатом, поскольку это в принципе невозможно. Если доставка оказывается невозможной, TCP уведомляет об этом пользователя, прекращая повторную передачу и разрывая соединение. Следовательно, TCP нельзя считать протоколом, надежным на 100%: он обеспечивает надежную доставку данных или надежное уведомление о неудаче.

TCP содержит алгоритмы, позволяющие динамически прогнозировать время (*период обращения*) (round-trip time, RTT) между клиентом и сервером, и таким образом определять, сколько времени необходимо для получения подтверждения. Например, RTT в локальной сети может иметь значение порядка миллисекунд, в то время как для глобальной сети (WAN) эта величина может достигать нескольких секунд. Более того, TCP постоянно пересчитывает величину RTT, поскольку она зависит от сетевого трафика.

TCP также упорядочивает (*sequences*) данные, связывая некоторый порядковый номер с каждым отправляемым им байтом. Предположим, например, что приложение записывает 2048 байт в сокет TCP, что приводит к отправке двух сегментов TCP. Первый из них содержит данные с порядковыми номерами 1-1024, второй — с номерами 1025-2048. (*Сегмент (segment)* — это блок данных, передаваемых протоколом TCP протоколу IP.) Если какой-либо сегмент приходит вне очереди (то есть если нарушается последовательность сегментов), принимающий TCP заново упорядочит сегменты, основываясь на их порядковых номерах, перед тем как отправить данные принимающему приложению. Если TCP получает дублированные данные (допустим, компьютер на другом конце ошибочно решил, что сегмент был потерян, и передал его заново, когда на самом деле он потерян не был, просто сеть была перегружена), он может определить, что данные были дублированы (исходя из порядковых номеров), и дублированные данные будут проигнорированы.

ПРИМЕЧАНИЕ

Протокол UDP не обеспечивает надежности. UDP сам по себе не имеет ничего похожего на описанные подтверждения передачи, порядковые номера, определение RTT, тайм-ауты или повторные передачи. Если дейтаграмма UDP дублируется в сети, на принимающий узел могут быть доставлены два экземпляра. Также, если клиент UDP отправляет две дейтаграммы в одно и то же место назначения, их порядок может быть изменен сетью, и они будут доставлены с нарушением исходного порядка. Приложения UDP должны самостоятельно обрабатывать все подобные случаи, как это показано в разделе 22.5.

TCP обеспечивает управление потоком (*flow control*). TCP всегда сообщает своему собеседнику, сколько именно байтов он хочет получить от него. Это называется объявлением окна (*window*). В любой момент времени окно соответствует свободному пространству в буфере получателя. Управление потоком гарантирует, что отправитель не переполнит этот буфер. Окно изменяется динамически с течением времени: по мере того как приходят данные от отправителя, размер окна уменьшается, но по мере считывания принимающим приложением данных из буфера окно увеличивается. Возможно, что окно станет нулевым: если принимающий буфер TCP для данного сокета заполнен, отправитель должен подождать, когда приложение считает данные из буфера.

ПРИМЕЧАНИЕ

UDP не обеспечивает управления потоком. Быстрый отправитель UDP может передавать дейтаграммы с такой скоростью, с которой не может работать получатель UDP, как это показано в разделе 8.13.

Наконец, соединение TCP также является двусторонним (*full-duplex*). Это значит, что приложение может отправлять и принимать данные в обоих направлениях на заданном соединении в любой момент времени. Иначе говоря, TCP должен отслеживать состояние таких характеристик, как порядковые номера и размеры окна, для каждого направления потока данных: отправки и приема. После установления двустороннее соединение может быть преобразовано в одностороннее (см. раздел 6.6).

ПРИМЕЧАНИЕ

UDP может быть (а может и не быть) двусторонним.

2.5. SCRIPT: протокол управления передачей потоков

Сервисы, предоставляемые SCRIPT, имеют много общего с сервисами TCP и UDP. Протокол SCRIPT описывается в RFC 2960 [118] и RFC 3309 [119]. Введение в SCRIPT приводится в RFC 3286 [85]. SCRIPT ориентирован на создание ассоциаций между клиентами и серверами. Кроме того, SCRIPT предоставляет приложениям надежность, упорядочение данных, управление передачей и двустороннюю связь, подобно TCP. Слово «ассоциация» используется вместо слова «соединение» намеренно, потому что соединение всегда устанавливалось между двумя IP-адресами. Ассоциация означает взаимодействие двух систем, которые могут иметь по несколько адресов (это называется multihoming — множественная адресация).

В отличие от TCP, протокол SCRIPT ориентирован не на поток байтов, а на сообщения. Он обеспечивает упорядоченную доставку отдельных записей. Как и в UDP, длина сообщения, записанная отправителем, передается приложению-получателю.

SCRIPT может поддерживать несколько потоков между конечными точками ассоциации, для каждого из которых надежность и порядок сообщений контролируются отдельно. Утрата сообщения в одном из потоков не блокирует доставку сообщений по другим потокам. Этот подход прямо противоположен тому, что имеется в TCP, где потеря единственного байта блокирует доставку всех последующих байтов по соединению до тех пор, пока ситуация не будет исправлена.

Кроме того, SCRIPT поддерживает множественную адресацию, что позволяет единственной конечной точке SCRIPT иметь несколько IP-адресов. Эта функция обеспечивает дополнительную устойчивость в случае отказов сети. Конечная точка может иметь избыточные IP-адреса, каждый из которых может соответствовать собственному соединению с инфраструктурой Интернета. В такой конфигурации SCRIPT позволит обойти проблему, возникшую на одном из адресов, благодаря переключению на другой адрес, заранее связанный с соответствующей ассоциацией SCRIPT.

ПРИМЕЧАНИЕ

Подобной устойчивости можно достичь и с TCP, если воспользоваться протоколами маршрутизации. Например, BGP-соединения внутри домена (iBGP) часто используют адреса, назначаемые виртуальному интерфейсу маршрутизатора в качестве конечных точек соединения TCP. Протокол маршрутизации домена гарантирует, что если между двумя маршрутизаторами будет хоть какой-то доступный путь, он будет использован, что было бы невозможно, если бы используемые адреса принадлежали интерфейсу в сети, где возникли проблемы. Функция множественной адресации SCRIPT позволяет узлам (а не только маршрутизаторам) использовать аналогичный подход, причем даже с подключениями через разных провайдеров, что невозможно при использовании TCP с маршрутизацией.

2.6. Установление и завершение соединения TCP

Чтобы облегчить понимание функций `connect`, `accept` и `close` и чтобы нам было легче отлаживать приложения TCP с помощью программы `netstat`, мы должны понимать, как устанавливаются и завершаются соединения TCP. Мы также должны понимать диаграмму перехода состояний TCP.

Трехэтапное рукопожатие

При установлении соединения TCP действия развиваются по следующему сценарию.

1. Сервер должен быть подготовлен для того, чтобы принять входящее соединение. Обычно это достигается вызовом функций `socket`, `bind` и `listen` и называется *пассивным открытием* (*passive open*).

2. Клиент выполняет *активное открытие* (*active open*), вызывая функцию `connect`. Это заставляет клиента TCP послать сегмент SYN (от слова synchronize — синхронизировать), чтобы сообщить серверу

начальный порядковый номер данных, которые клиент будет посыпать по соединению. Обычно с сегментом SYN не посыпается никаких данных: он содержит только заголовок IP, заголовок TCP и, возможно, параметры TCP (о которых мы вскоре поговорим).

3. Сервер должен подтвердить получение клиентского сегмента SYN, а также должен послать свой собственный сегмент SYN, содержащий начальный порядковый номер для данных, которые сервер будет посыпать по соединению. Сервер посыпает SYN и ACK — подтверждение приема (от слова acknowledgment) клиентского SYN — в виде единого сегмента.

4. Клиент должен подтвердить получение сегмента SYN сервера.

Для подобного обмена нужно как минимум три пакета, поэтому он называется *трехэтапным рукопожатием TCP* (*TCP three-way handshake*). На рис. 2.2 представлена схема такого обмена.

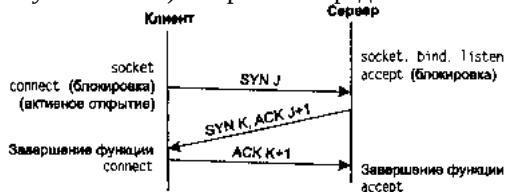


Рис. 2.2. Трехэтапное рукопожатие TCP

Мы обозначаем начальный порядковый номер клиента как J , а начальный порядковый номер сервера как K . Номер подтверждения в сегменте ACK — это следующий предполагаемый порядковый номер на том конце связи, который отправил сегмент ACK. Поскольку сегмент SYN занимает 1 байт пространства порядковых номеров, номер подтверждения в сегменте ACK каждого сегмента SYN — это начальный порядковый номер плюс один. Аналогично сегмент ACK каждого сегмента FIN — это порядковый номер сегмента FIN плюс один.

ПРИМЕЧАНИЕ

Повседневной аналогией установления соединения TCP может служить система телефонной связи [81]. Функция `socket` эквивалентна включению используемого телефона. Функция `bind` дает возможность другим узнать ваш телефонный номер, чтобы они могли позвонить вам. Функция `listen` включает звонок, и вы можете услышать, когда происходит входящий звонок. Функция `connect` требует, чтобы мы знали чей-то номер телефона и могли до него дозвониться. Функция `accept` — аналогия ответа на входящий звонок. Получение идентифицирующих данных, возвращаемых функцией `accept` (где идентифицирующие данные — это IP-адрес и номер порта клиента), аналогично получению информации, идентифицирующей вызывающего по телефону — его телефонного номера. Однако имеется отличие, и состоит оно в том, что функция `accept` возвращает идентифицирующие данные клиента только после того, как соединение установлено, тогда как во время телефонного звонка после указания номера телефона звонящего мы можем выбрать, отвечать на звонок или нет. Служба DNS (см. главу 11) предоставляет сервис, аналогичный телефонной книге. Вызов `getaddrinfo` — поиск телефонного номера в книге; `getnameinfo` — поиск имени по телефонному номеру (правда, такая книга должна быть отсортирована по номерам, а не по именам).

Параметры TCP

Каждый сегмент SYN может содержать параметры TCP. Ниже перечислены наиболее общеупотребительные параметры TCP.

■ *Параметр MSS*. Этот параметр TCP позволяет узлу, отправляющему сегмент SYN, объявить свой максимальный размер сегмента (maximum segment size, MSS) — максимальное количество данных, которое он будет принимать в каждом сегменте TCP на этом соединении. Мы покажем, как получить и установить этот параметр TCP с помощью параметра сокета `tcp_MAXSEG` (см. раздел 7.9).

■ *Параметр масштабирования окна* (*Window scale option*). Максимальный размер окна, который может быть установлен в заголовке TCP, равен 65 535, поскольку соответствующее поле занимает 16 бит. Но высокоскоростные соединения (45 Мбит/с и больше, как описано в документе RFC 1323 [53]) или линии с большой задержкой (спутниковые сети) требуют большего размера окна для получения

максимально возможной пропускной способности. Этот параметр, появившийся не так давно, определяет, что объявленная в заголовке TCP величина окна должна быть отмасштабирована — сдвинута влево на 0-14 разрядов, предостав员я максимально возможное окно размером почти гигабайт ($65\,535 \times 2^{14}$). Для использования параметра масштабирования окна в соединении необходима его поддержка обоими связывающимися узлами. Мы увидим, как задействовать этот параметр с помощью параметра сокета SO_RCVBUF (см. раздел 7.5).

ПРИМЕЧАНИЕ

Чтобы обеспечить совместимость с более ранними реализациями, в которых не поддерживается этот параметр, применяются следующие правила. TCP может отправить параметр со своим сегментом SYN в процессе активного открытия сокета. Но он может масштабировать свое окно, только если другой конец связи также отправит соответствующий параметр со своим сегментом SYN. Эта логика предполагает, что недоступные в данной реализации параметры просто игнорируются. Это общее и необходимое требование, но, к сожалению, его выполнение не гарантировано для всех реализаций.

■ *Временная метка (Timestamp option)*. Этот параметр необходим для высокоскоростных соединений, чтобы предотвратить возможное повреждение данных, вызванное приходом устаревших, задержавшихся и дублированных пакетов. Поскольку это один из недавно появившихся параметров, его обработка производится аналогично параметру масштабирования окна. С точки зрения сетевого программиста, этот параметр не должен вызывать беспокойства.

Перечисленные выше параметры поддерживаются большинством реализаций. Последние два параметра иногда называются «параметрами RFC 1323», они были описаны именно этим стандартом [53]. Они также часто именуются параметрами для «канала с повышенной пропускной способностью», поскольку сеть с широкой полосой пропускания или с большой задержкой называется *каналом с повышенной пропускной способностью*, или, если перевести дословно, *длинной толстой трубой (long fat pipe)*. В главе 24 [111] эти новые параметры описаны более подробно.

Завершение соединения TCP

В то время как для установления соединения необходимо три сегмента, для его завершения требуется четыре сегмента.

1. Одно из приложений первым вызывает функцию *close*, и мы в этом случае говорим, что конечная точка TCP выполняет *активное закрытие (active close)*. TCP этого узла отправляет сегмент FIN, обозначающий прекращение передачи данных.

2. Другой узел, получающий сегмент FIN, выполняет *пассивное закрытие (passive close)*. Полученный сегмент FIN подтверждается TCP. Получение сегмента FIN также передается приложению как признак конца файла (после любых данных, которые уже стоят в очереди, ожидая приема приложением), поскольку получение приложением сегмента FIN означает, что оно уже не получит никаких дополнительных данных по этому соединению.

3. Через некоторое время после того как приложение получило признак конца файла, оно вызывает функцию *close* для закрытия своего сокета. При этом его TCP отправляет сегмент FIN.

4. TCP системы, получающей окончательный сегмент FIN (то есть того узла, на котором произошло активное закрытие), подтверждает получение сегмента FIN.

Поскольку сегменты FIN и ACK передаются в обоих направлениях, обычно требуется четыре сегмента. Мы используем слово «обычно», поскольку в ряде сценариев сегмент FIN на первом шаге отправляется вместе с данными. Кроме того, сегменты, отправляемые на шаге 2 и 3, исходят с узла, выполняющего пассивное закрытие, и могут быть объединены. Соответствующие пакеты изображены на рис. 2.3.

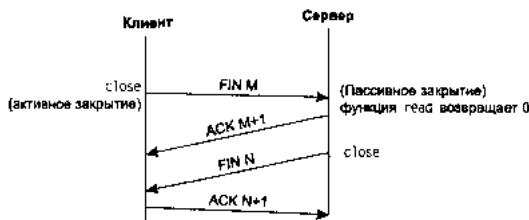


Рис. 2.3. Обмен пакетами при завершении соединения TCP

Сегмент FIN занимает 1 байт пространства порядковых номеров аналогично SYN. Следовательно, сегмент ACK каждого сегмента FIN — это порядковый номер FIN плюс один.

Возможно, что между шагами 2 и 3 какие-то данные будут переданы от узла, выполняющего пассивное закрытие, к узлу, выполняющему активное закрытие. Это состояние называется *частичным закрытием* (*half-close*), и мы рассмотрим его во всех подробностях вместе с функцией shutdown в разделе 6.6.

Отправка каждого сегмента FIN происходит при закрытии сокета. Мы говорили, что для этого приложение вызывает функцию close, но нужно понимать, что когда процесс Unix прерывается либо произвольно (при вызове функции exit или при возврате из функции main), либо непривольно (при получении сигнала, прерывающего процесс), все его открытые дескрипторы закрываются, что также вызывает отправку сегмента FIN любому соединению TCP, которое все еще открыто.

Хотя на рис. 2.3 мы продемонстрировали, что активное закрытие выполняет клиент, на практике активное закрытие может выполнять любой узел: и клиент, и сервер. Часто активное закрытие выполняет клиент, но с некоторыми протоколами (особенно HTTP) активное закрытие выполняет сервер.

Диаграмма состояний TCP

Последовательность действий TCP во время установления и завершения соединения можно определить с помощью *диаграммы состояний TCP* (*state transition diagram*). Ее мы изобразили на рис. 2.4.

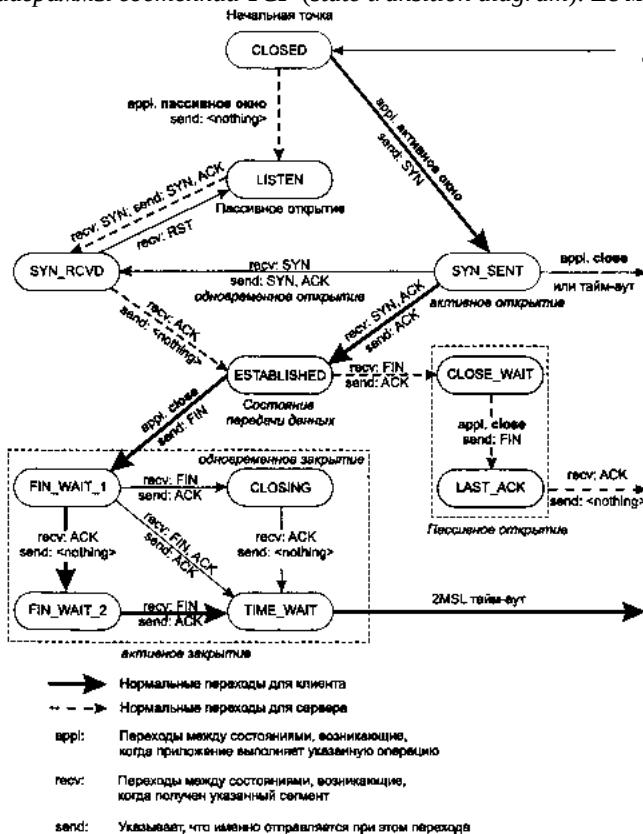


Рис. 2.4. Диаграмма состояний TCP

Для соединения определено 11 различных состояний, а правила TCP предписывают переходы от одного состояния к другому в зависимости от текущего состояния и сегмента, полученного в этом состоянии. Например, если приложение выполняет активное открытие в состоянии CLOSED (Закрыто), TCP отправляет сегмент SYN, и новым состоянием становится SYN_SENT (Отправлен SYN). Если затем TCP получает сегмент SYN с сегментом ACK, он отправляет сегмент ACK, и следующим состоянием становится ESTABLISHED (Соединение установлено). В этом последнем состоянии проходит большая часть обмена данными.

Две стрелки, идущие от состояния ESTABLISHED, относятся к разрыву соединения. Если приложение вызывает функцию close перед получением признака конца файла (активное закрытие), происходит переход к состоянию FIN_WAIT_1 (Ожидание FIN 1). Но если приложение получает сегмент FIN в состоянии ESTABLISHED (пассивное закрытие), происходит переход в состояние CLOSE_WAIT (Ожидание закрытия).

Мы отмечаем нормальные переходы клиента с помощью более толстой сплошной линии, а нормальные переходы сервера — с помощью штриховой линии. Мы также должны отметить, что существуют два перехода, о которых мы не говорили: одновременное открытие (когда оба конца связи отправляют сегменты SYN приблизительно в одно время, и эти сегменты пересекаются в сети) и одновременное закрытие (когда оба конца связи отправляют сегменты FIN). В главе 18 [111] содержатся примеры и описания обоих этих сценариев, которые хотя и возможны, но встречаются достаточно редко.

Одна из причин, по которым мы приводим здесь диаграмму перехода состояний, — мы хотим показать все 11 состояний TCP и их названия. Эти состояния отображаются программой netstat, которая является полезным средством отладки клиент-серверных приложений. Мы будем использовать программу netstat для отслеживания изменений состояния в главе 5.

Обмен пакетами

На рис. 2.5 представлен реальный обмен пакетами, происходящий во время соединения TCP: установление соединения, передача данных и завершение соединения. Мы также показываем состояния TCP, через которые проходит каждый узел.

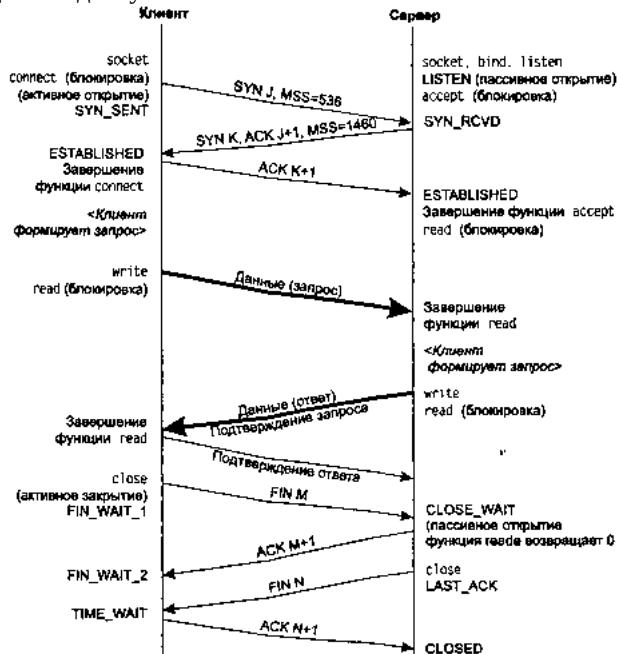


Рис. 2.5. Обмен пакетами для соединения TCP

В этом примере клиент объявляет размер сегмента (MSS) равным 536 байт (это означает, что его реализация работает с минимальным размером буфера сборки пакетов), а сервер — 1460 байт (типичное значение для IPv4 в Ethernet). Как видно, MSS в каждом направлении передачи вполне могут отличаться (см. также упражнение 2.5).

Как только соединение установлено, клиент формирует запрос и посыпает его серверу. Мы считаем, что этот запрос соответствует одиночному сегменту TCP (то есть его размер меньше 1460 байт — анонсированного размера MSS сервера). Сервер обрабатывает запрос и отправляет ответ, и мы также считаем, что ответ соответствует одиночному сегменту (в данном примере меньше 536 байт). Оба сегмента данных мы отобразили более жирными линиями. Заметьте, что подтверждение запроса клиента отправляется с ответом сервера. Это называется *вложенным подтверждением* (*piggybacking*) и обычно происходит, когда сервер успевает обработать запрос и подготовить ответ меньше, чем за 200 мс или около того. Если серверу требуется больше времени, скажем, 1 с, ответ будет приходить после подтверждения. (Динамика потока данных TCP подробно описана в главах 19 и 20 [111].)

Затем мы показываем четыре сегмента, закрывающих соединение. Заметьте, что узел, выполняющий активное закрытие (в данном сценарии клиент), входит в состояние TIME_WAIT. Мы рассмотрим это в следующем разделе.

На рис. 2.5 важно отметить, что если целью данного соединения было отправить запрос, занимающий один сегмент, и получить ответ, также занимающий один сегмент, то при использовании TCP всего будет задействовано восемь сегментов. Если же используется UDP, произойдет обмен только двумя сегментами: запрос и ответ. Но при переходе от TCP к UDP теряется надежность, которую TCP предоставляет приложению, и множество задач по обеспечению надежности транспортировки данных переходит с транспортного уровня (TCP) на уровень приложения. Другое важное свойство, предоставленное TCP, — это управление в условиях перегрузки, которое в случае использования протокола UDP должно принимать на себя приложение. Тем не менее важно понимать, что многие приложения используют именно UDP, потому что они обмениваются небольшими объемами данных, а UDP позволяет избежать накладных расходов, возникающих при установлении и разрыве соединения TCP.

2.7. Состояние TIME_WAIT

Без сомнений, самым сложным для понимания аспектом TCP в отношении сетевого программирования является состояние TIME_WAIT (время ожидания). На рис. 2.4 мы видим, что узел, выполняющий активное закрытие, проходит это состояние. Продолжительность этого состояния равна двум MSL (*maximum segment lifetime* — максимальное время жизни сегмента), иногда этот период называется 2MSL.

В каждой реализации TCP выбирается какое-то значение MSL. Рекомендуемое значение, приведенное в документе RFC 1122 [10], равно 2 мин, хотя Беркли-реализации традиционно использовали значение 30 с. Это означает, что продолжительность состояния TIME_WAIT — от 1 до 4 мин. MSL — это максимальное количество времени, в течение которого дейтаграмма IP может оставаться в сети. Это время ограничено, поскольку каждая дейтаграмма содержит 8-разрядное поле *пределного количества прыжков* (*hop limit*) (поле TTL IPv4 на рис. A.1 и поле «Предельное количество транзитных узлов» IPv6 на рис. A.2), максимальное значение которого равно 255. Хотя этот предел ограничивает количество транзитных узлов, а не время пребывания пакета в сети, считается, что пакет с максимальным значением этого предела (которое равно 255) не может существовать в сети более MSL секунд.

Пакеты в объединенных сетях обычно теряются в результате различных аномалий. Маршрутизатор отключается, или нарушается связь между двумя маршрутизаторами, и им требуются секунды или минуты для стабилизации и нахождения альтернативного пути. В течение этого периода времени могут возникать петли маршрутизации (маршрутизатор A отправляет пакеты маршрутизатору B, а маршрутизатор B отправляет их обратно маршрутизатору A), и пакеты теряются в этих петлях. В этот момент, если потерянный пакет — это сегмент TCP, истекает установленное время ожидания отправляющего узла, и он снова передает пакет, и этот заново переданный пакет доходит до конечного места назначения по некоему альтернативному пути. Но если спустя некоторое время (не превосходящее количества секунд MSL после начала передачи потерянного пакета) петля маршрутизации исправляется, пакет, потерянный в петле, отправляется к конечному месту назначения. Начальный пакет называется *потерянной копией* или *дубликатом* (*lost duplicate*), а также *блуждающей копией* или *дубликатом* (*wandering duplicate*). TCP должен обрабатывать эти дублированные пакеты.

Есть две причины существования состояния TIME_WAIT:

- необходимо обеспечить надежность разрыва двустороннего соединения TCP;
- необходимо подождать, когда истечет время жизни в сети старых дублированных сегментов.

Первую причину можно объяснить, рассматривая рис. 2.5 в предположении, что последний сегмент ACK потерян. Сервер еще раз отправит свой последний сегмент FIN, поэтому клиент должен сохранять

информацию о своем состоянии, чтобы отправить завершающее подтверждение ACK повторно. Если бы клиент не сохранял информацию о состоянии, он ответил бы серверу сегментом RST (еще один вид сегмента TCP), что сервер интерпретировал бы как ошибку. Если ответственность за корректное завершение двустороннего соединения в обоих направлениях ложится на TCP, он должен правильно обрабатывать потерю любого из четырех сегментов. Этот пример объясняет, почему в состоянии TIME_WAIT остается узел, выполняющий активное закрытие: именно этому узлу может потребоваться повторно передать подтверждение.

Чтобы понять вторую причину, по которой необходимо состояние TIME_WAIT, давайте считать, что у нас имеется соединение между IP-адресом 12.106.32.254, порт 1500 и IP-адресом 206.168.112.219, порт 21. Это соединение закрывается, и спустя некоторое время мы устанавливаем другое соединение между теми же IP-адресами и портами: 12.106.32.254, порт 1500 и 206.168.112.219, порт 21. Последнее соединение называется новым *воплощением* (*incarnation*) предыдущего соединения, поскольку IP-адреса и порты те же. TCP должен предотвратить появление старых дубликатов, относящихся к данному соединению, в новом воплощении этого соединения. Чтобы гарантировать это, TCP запрещает установление нового воплощения соединения, которое в данный момент находится в состоянии TIME_WAIT. Поскольку продолжительность состояния TIME_WAIT равна двум MSL, это позволяет удостовериться, что истечет и время жизни пакетов, посланных в одном направлении, и время жизни пакетов, посланных в ответ. Используя это правило, мы гарантируем, что в момент успешного установления соединения TCP время жизни в сети всех старых дубликатов от предыдущих воплощений этого соединения уже истекло.

ПРИМЕЧАНИЕ

Из этого правила существует исключение. Реализации, происходящие от Беркли, инициируют новое воплощение соединения, которое в настоящий момент находится в состоянии TIME_WAIT, если приходящий сегмент SYN имеет порядковый номер «больше» конечного номера из предыдущего воплощения. На с. 958-959 [128] об этом рассказано более подробно. Для этого требуется, чтобы сервер выполнил активное закрытие, поскольку состояние TIME_WAIT должно существовать на узле, получающем следующий сегмент SYN. Эта возможность используется командой rsh. В документе RFC 1185 [54] рассказывается о некоторых ловушках, которые могут вас подстерегать при этом.

2.8. Установление и завершение ассоциации SCRIPT

Протокол SCRIPT ориентирован на установление соединения, подобно TCP, поэтому он также имеет собственные процедуры рукопожатия и завершения. Однако рукопожатия SCRIPT отличаются от рукопожатий TCP, поэтому мы описываем их отдельно.

Четырехэтапное рукопожатие

При установлении ассоциации SCRIPT выполняется приведенная далее последовательность действий, подобная трехэтапному рукопожатию TCP.

1. Сервер должен быть готов к приему входящего соединения. Подготовка обычно осуществляется последовательным вызовом функций `socket`, `bind` и `listen` и называется *пассивным открытием* (*passive open*).

2. Клиент начинает *активное открытие* (*active open*), вызывая функцию `connect` или сразу отправляя сообщение, что также приводит к установлению ассоциации. При этом клиент SCRIPT передает сообщение INIT (от слова «инициализация»), в котором серверу отправляется список IP-адресов клиента, начальный порядковый номер, идентификационная метка, позволяющая отличать пакеты данной ассоциации от всех прочих, количество исходящих потоков, запрашиваемых клиентом, и количество входящих потоков, поддерживаемых клиентом.

3. Сервер подтверждает получение сообщения INIT от клиента сообщением INIT-ACK, которое содержит список IP-адресов сервера, начальный порядковый номер, идентификационную метку, количество исходящих потоков, запрашиваемых сервером, количество входящих потоков, поддерживаемых сервером, и cookie с данными о состоянии. Cookie содержит все сведения о состоянии, которые нужны

серверу для того, чтобы гарантировать действительность ассоциации. В cookie включается цифровая подпись, подтверждающая аутентичность.

4. Клиент отсылает cookie обратно серверу сообщением COOKIE-ECHO. Это сообщение уже может содержать пользовательские данные.

5. Сервер подтверждает правильность приема cookie и установление ассоциации сообщением COOKIE-ACK. Это сообщение также может включать полезные данные.

Минимальное количество пакетов для установления ассоциации SCRIPT равно четырем, поэтому описанная процедура называется *четырехэтапным рукопожатием SCRIPT*. Эти четыре пакета, передаваемые между клиентом и сервером, показаны на рис. 2.6.

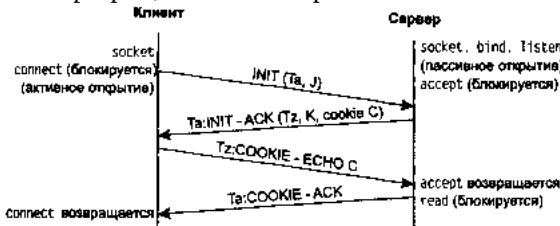


Рис. 2.6. Четырехэтапное рукопожатие SCRIPT

Во многих отношениях четырехэтапное рукопожатие SCRIPT подобно трехэтапному рукопожатию TCP, за исключением всего, что связано с cookie. Сообщение INIT включает (помимо множества параметров) контрольную метку T_a (*verification tag*) и начальный порядковый номер J . Метка T_a должна присутствовать во всех пакетах, отправляемых собеседнику по данной ассоциации. Начальный порядковый номер используется для нумерации сообщений DATA (порций данных — DATA chunks). Собеседник тоже выбирает собственную метку T_z , которая должна присутствовать во всех его пакетах. Помимо контрольной метки и начального порядкового номера K получатель сообщения INIT отправляет cookie C . Пакет cookie содержит все сведения о состоянии, необходимые для установления ассоциации SCRIPT, так что стеку SCRIPT сервера не приходится хранить сведения о клиенте, с которым устанавливается ассоциация. Более подробные сведения о настройке ассоциаций SCRIPT вы можете найти в главе 4 книги [117].

В заключение рукопожатия каждая сторона выбирает основной адрес назначения. На этот адрес передаются все данные в отсутствие неполадок в сети.

Четырехэтапное рукопожатие используется в SCRIPT для того, чтобы сделать невозможной одну из атак типа «отказ в обслуживании» (см. раздел 4.5).

ПРИМЕЧАНИЕ

Четырехэтапное рукопожатие SCRIPT с использованием cookie формализует метод защиты от атак типа «отказ в обслуживании». Многие реализации TCP используют аналогичный метод. Отличие в том, что при работе с TCP данные cookie приходится кодировать в начальный порядковый номер, длина которого составляет всего 32 разряда. В SCRIPT используется поле произвольной длины и криптографическая защита.

Завершение ассоциации

В отличие от TCP, SCRIPT не имеет состояния, соответствующего частично закрытой ассоциации. Когда один узел закрывает ассоциацию, второй узел должен перестать отправлять новые данные. Получатель запроса на закрытие ассоциации отправляет те данные, которые уже были помещены в очередь, после чего завершает процедуру закрытия. Обмен пакетами изображен на рис. 2.7.

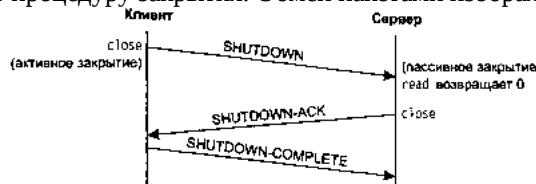


Рис. 2.7. Обмен пакетами при завершении ассоциации SCRIPT

SCTP не нуждается в состоянии TIME_WAIT благодаря контрольным меткам. Все порции данных помечаются так, как было оговорено при обмене сегментами INIT. Задержавшаяся порция от предыдущего соединения будет иметь неправильную метку. Вместо того, чтобы поддерживать в состоянии ожидания TIME_WAIT целое соединение, SCRIPT помещает в это состояние значения контрольных меток.

Диаграмма состояний SCRIPT

Порядок работы SCRIPT при установлении и завершении ассоциаций может быть проиллюстрирован диаграммой состояний (рис. 2.8).

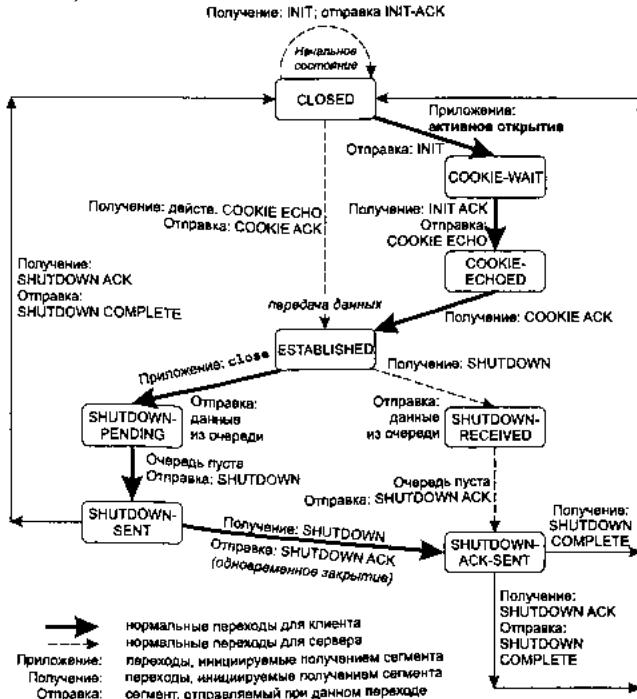


Рис. 2.8. Диаграмма состояний SCRIPT

Как и на рис. 2.4, переходы из одного состояния в другое регулируются правилами SCRIPT и определяются текущим состоянием и порцией данных, полученной в этом состоянии. Например, если приложение выполняет активное открытие в состоянии CLOSED (Закрыто), SCRIPT отправляет пакет INIT и переходит в состояние COOKIE-WAIT (Ожидание cookie). Если затем SCRIPT получает пакет INIT-ACK, он отправляет пакет COOKIE-ECHO и новым состоянием становится COOKIE-ECHOED (Cookie отправлен обратно). Если после этого SCRIPT принимает COOKIE ACK, он переходит в состояние ESTABLISHED (Соединение установлено). В этом состоянии осуществляется передача основного объема данных. Порции данных могут передаваться совместно с пакетами COOKIE ECHO и COOKIE ACK.

Две стрелки из состояния ESTABLISHED на рис. 2.8 соответствуют двум сценариям завершения ассоциации. Если приложение вызывает функцию close до получения пакета SHUTDOWN (активное закрытие), переход осуществляется в состояние SHUTDOWN-PENDING (Ожидание завершения). Если же приложение получает пакет SHUTDOWN, находясь в состоянии ESTABLISHED (пассивное закрытие), переход осуществляется в состояние SHUTDOWN-RECEIVED (Получен сигнал о завершении).

Обмен пакетами

На рис. 2.9 показан реальный обмен пакетами для ассоциации SCRIPT. Рисунок включает установление ассоциации, передачу данных и завершение ассоциации. Мы также показываем состояния SCRIPT, через которые проходит каждый из узлов.

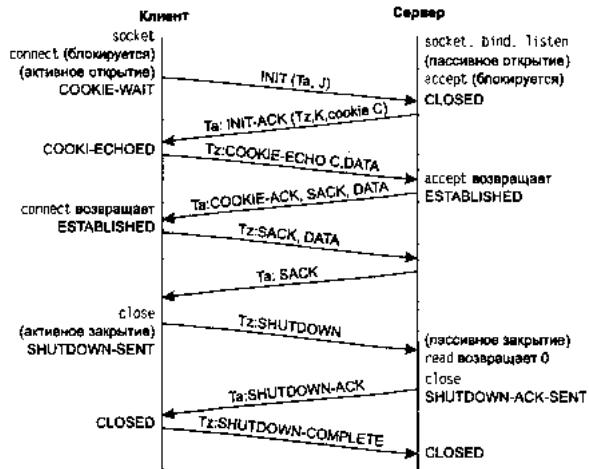


Рис. 2.9. Обмен пакетами для ассоциации SCRIPT

В этом примере первая порция данных включается клиентом в COOKIE ECHO, а сервер включает данные в порцию COOKIE ACK. В общем случае в пакет COOKIE ECHO может включаться и несколько порций данных, если приложение использует интерфейс типа «один-ко-многим» (о разных типах интерфейсов речь пойдет в разделе 9.2).

Блок информации, передаваемый в пакете SCRIPT, называется *порцией (chunk)*. Порция информации самодостаточна, она включает сведения о типе данных, флаги и поле длины. Этот подход облегчает упаковку нескольких порций в один исходящий пакет (подробнее об упаковке порций и нормальном режиме передачи данных рассказывается в главе 5 [117]).

Параметры SCRIPT

SCTP использует параметры для облегчения использования дополнительных возможностей. Функции SCRIPT могут расширяться добавлением новых типов порций или новых параметров. При этом стандартные реализации SCRIPT имеют возможность сообщать о неизвестных параметрах и порциях данных. Старшие два бита пространства параметров и пространства порций определяют, что именно должен сделать получатель SCRIPT с неизвестным параметром или порцией (подробнее см. в разделе 3.1 [117]).

В настоящий момент разрабатываются два расширения SCRIPT:

1. Динамическое расширение адресов, позволяющее взаимодействующим узлам добавлять и удалять IP-адреса из существующей ассоциации.
2. Поддержка частичной надежности, позволяющая взаимодействующим узлам по указанию от приложения ограничивать повторную передачу данных. Если сообщение становится слишком старым (это решает приложение), оно пропускается, и никаких попыток отправить его еще раз не делается. Это означает, что доставка всех данных адресату уже не гарантируется.

2.9. Номера портов

В любой момент времени каждый транспортный протокол (UDP, TCP, SCRIPT) может использоваться несколькими процессами. Все три протокола различают эти процессы при помощи 16-разрядных целых чисел — *номеров портов (port numbers)*.

Когда клиент хочет соединиться с сервером, клиент должен идентифицировать этот сервер. Для TCP, UDP и SCRIPT определена группа *заранее известных портов (well-known ports)* для идентификации известных служб. Например, каждая реализация TCP/IP, поддерживающая FTP, присваивает заранее известный порт 21 (десятичный) серверу FTP. Серверам TFTP (Trivial File Transfer Protocol — упрощенный протокол передачи файлов) присваивается порт UDP 69.

С другой стороны, клиенты используют *динамически назначаемые, или эфемерные (ephemeral)* порты, то есть порты с непродолжительным временем жизни. Эти номера портов обычно присваиваются клиенту автоматически протоколами UDP или TCP. Клиенту обычно не важно фактическое значение динамически

назначаемого порта; клиент лишь должен быть уверен, что динамически назначаемый порт является уникальным на клиентском узле. Реализации транспортного уровня гарантируют такую уникальность.

IANA (Internet Assigned Numbers Authority — агентство по выделению имен и уникальных параметров протоколов Интернета) ведет список назначенных номеров портов. Раньше они публиковались в документах RFC; последним в этой серии был RFC 1700 [103]. В документе RFC 3232 [102] указан адрес базы данных, заменившей RFC 1700: <http://www.iana.org/>. Номера портов делятся на три диапазона.

1. *Заранее известные порты*: от 0 до 1023. Эти номера портов управляются и присваиваются агентством IANA. Когда это возможно, один и тот же номер порта присваивается данному сервису и для TCP, и для UDP. Например, порт 80 присваивается веб-серверу для обоих протоколов, хотя в настоящее время все реализации используют только TCP.

ПРИМЕЧАНИЕ

Когда веб-серверу был назначен порт 80, протокол SCRIPT еще не существовал. Новые порты назначаются всем трем протоколам, и в RFC 2960 отмечено, что все существующие номера портов TCP могут использоваться теми же службами, работающими по протоколу SCRIPT.

2. *Зарегистрированные порты*: от 1024 до 49 151. Они не управляются IANA, но IANA регистрирует и составляет списки использования этих портов для удобства потребителей. Когда это возможно, один и тот же порт выделяется одной и той же службе и для TCP, и для UDP. Например, порты с номерами от 6000 до 6063 присвоены серверу X Window для обоих протоколов, хотя в настоящее время все реализации используют только TCP. Верхний предел 49 151 для этих портов был установлен для того, чтобы оставить часть диапазона адресов для динамических портов. В документе RFC 1700 [103] верхний предел был 65 535.

3. *Динамические, или частные порты*: от 49 152 до 65 535. IANA ничего не говорит об этих портах. Эти порты мы иногда называем *эфемерными*. (Магическое число 49 152 составляет три четверти от 65 536.)

Разделение портов на диапазоны и общее распределение номеров портов показано на рис. 2.10.

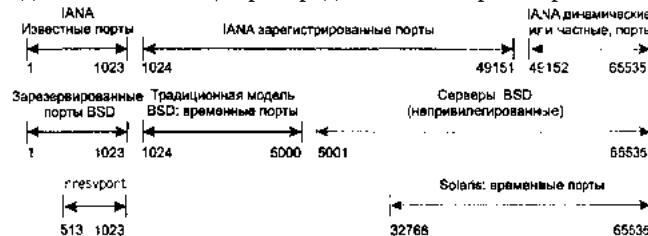


Рис. 2.10. Распределение номеров портов

На этом рисунке мы отмечаем следующие моменты:

■ В системах Unix имеется понятие *зарезервированного порта* (*reserved port*), и это порт с номером меньше 1024. Эти порты может присвоить сокету только процесс, обладающий соответствующими привилегиями. Все заранее известные порты IANA являются зарезервированными портами; следовательно, сервер, желающий использовать этот порт (такой, как сервер FTP), должен обладать правами привилегированного пользователя.

■ Исторически сложилось так, что Беркли-реализации (начиная с 4.3BSD) позволяют динамически выделять порты в диапазоне от 1024 до 5000. Это было хорошо в начале 80-х, когда серверы не могли обрабатывать много клиентов одновременно, но сегодня можно легко найти сервер, поддерживающий более 3977 клиентов в любой момент времени. Поэтому некоторые системы выделяют динамически назначаемые порты по-другому, либо из диапазона, определенного IANA, либо из еще более широкого диапазона (например, Solaris, как показано на рис. 2.6), чтобы предоставить больше динамически назначаемых портов.

ПРИМЕЧАНИЕ

Как выяснилось, значение 5000 для верхнего предела динамически назначаемых портов, реализованное в настоящее время во многих системах, было типографской ошибкой [7]. Этот предел должен был быть равен 50 000.

- Существуют несколько клиентов (не серверов), которые запрашивают зарезервированный порт для аутентификации в режиме клиент-сервер: типичным примером могут служить клиенты rlogin и rsh. Эти клиенты вызывают библиотечную функцию `resvport` для создания сокета TCP и присваивают сокету неиспользованный номер порта из диапазона от 513 до 1023. Эта функция обычно пытается связаться с портом 1023, если попытка оказывается неудачной — с портом 1022, и так далее, пока не будет достигнут желаемый результат или пока не будут перебраны все порты вплоть до порта 513.

ПРИМЕЧАНИЕ

И зарезервированные порты BSD, и порты функции `resvport` частично перекрывают верхнюю половину заранее известных портов IANA. Это происходит потому, что известные порты IANA когда-то заканчивались на 255. В документе RFC 1340 под названием «Assigned numbers» в 1992 году началось присваивание заранее известных портов в диапазоне от 256 до 1023. В предыдущем документе RFC под названием «Assigned numbers» за номером 1060 от 1990 году эти порты назывались стандартными службами Unix (Unix Standard Services). Существует множество Беркли-серверов, номера портов которых были заданы в 80-х годах и начинались с 512 (таким образом, номера с 256 по 511 были пропущены). Функция `resvport` начинает выбор с верхней границы диапазона 512-1023 и направляется вниз.

Пара сокетов

Пара сокетов (socket pair) для соединения TCP — это кортеж (группа взаимосвязанных элементов данных или записей) из четырех элементов, определяющий две конечные точки соединения: локальный IP-адрес, локальный порт TCP, удаленный IP-адрес и удаленный порт TCP. В SCRIPT ассоциация определяется набором локальных IP-адресов, локальным портом, набором удаленных IP-адресов и удаленным портом. В простейшем варианте без множественной адресации получается точно такой же четырехэлементный кортеж, как и для TCP. Однако если хотя бы один из узлов, составляющих ассоциацию, использует множественную адресацию, одной и той же ассоциации может сопоставляться несколько четырехэлементных кортежей (с разными IP-адресами, но одинаковыми номерами портов).

Два значения, идентифицирующих конечную точку, — IP-адрес и номер порта — часто называют *сокетом*.

Мы можем распространить понятие пары сокетов на UDP, даже учитывая то, что этот протокол не ориентирован на установление соединения. Когда мы будем говорить о функциях сокетов (`bind`, `connect`, `getpeername` и т.д.), мы увидим, какими функциями задаются конкретные элементы пары сокетов. Например, функция `bind` позволяет приложению задавать локальный IP-адрес и локальный порт для сокетов TCP, UDP и SCRIPT.

2.10. Номера портов TCP и параллельные серверы

Представим себе параллельный сервер, основной цикл которого порождает дочерний процесс для обработки каждого нового соединения. Что случится, если дочерний процесс будет продолжать использовать заранее известный номер порта при обслуживании длительного запроса? Давайте проанализируем типичную последовательность. Пусть сервер запускается на узле freebsd, поддерживающем множественную адресацию (IP-адреса 12.106.32.254 и 192.168.42.1), и выполняет пассивное открытие, используя свой заранее известный номер порта (в данном примере 21). Теперь он ожидает запрос клиента. Эта ситуация изображена на рис. 2.11.

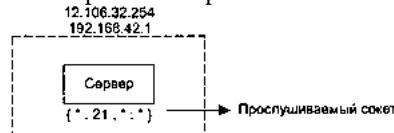


Рис. 2.11. Сервер TCP с пассивным открытием на порте 21

Мы используем обозначение $(*:21, *:*)$ для указания пары сокетов сервера. Сервер ожидает запроса соединения на любом локальном интерфейсе (первая звездочка) на порт 21. Удаленный IP-адрес и удаленный порт не определены, поэтому мы обозначаем их как $*.*$. Такая структура называется *прослушиваемым сокетом (listening socket)*.

ПРИМЕЧАНИЕ

Мы отделяем IP-адрес от номера порта символом «`:`», потому что это обозначение используется в HTTP и часто встречается в других местах. Программа netstat отделяет номер порта от IP-адреса точкой, но иногда это приводит к затруднениям, потому что точки используются как в доменных именах (freebsd.unpbook.com.21), так и в записи IPv4 (12.106.32.254.21).

Когда мы обозначаем звездочкой локальный IP-адрес, такое обозначение называется *универсальным адресом*, а звездочки — *символом подстановки (wildcard)*. Если узел, на котором запущен сервер, поддерживает множественную адресацию (как в нашем примере), сервер может указать, что он хочет принимать входящие соединения, которые приходят только для одного определенного локального интерфейса. Сервер должен выбрать либо один определенный интерфейс, либо принимать запросы от всех интерфейсов, то есть сервер не может задать список, состоящий из нескольких адресов. Локальный адрес, заданный с помощью символа подстановки, соответствует выбору произвольного адреса из определенного множества. В листинге 1.5 перед вызовом функции bind произвольный IP-адрес в структуре адреса сокета задан с помощью константы INADDR_ANY.

Через некоторое время на узле с IP-адресом 206.168.112.219 запускается клиент и выполняет активное открытие соединения с IP-адресом сервера 12.106.32.254. В этом примере мы считаем, что динамически назначаемый порт, выбранный клиентом TCP, — это порт 1500, что отражено на рис. 2.12. Под клиентом мы показываем его пару сокетов.

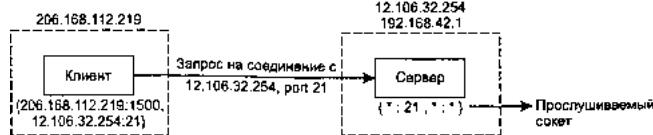


Рис. 2.12. Запрос на соединение от клиента к серверу

Когда сервер получает и принимает соединение клиента, он с помощью функции fork создает свою копию, давая возможность дочернему процессу обработать запрос клиента, как показано на рис. 2.13 (функцию fork мы описываем в разделе 4.7).

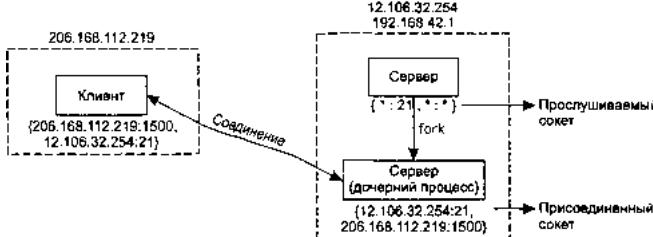


Рис. 2.13. Параллельный сервер, дочерний процесс которого обрабатывает запрос клиента

На этом этапе мы должны провести различие между прослушиваемым сокетом и присоединенным сокетом на сервере. Заметьте, что присоединенный сокет использует тот же локальный порт (21), что и прослушиваемый сокет. Также заметьте, что на многоадресном сервере локальный адрес заполняется для присоединенного сокета (206.62.226.35), как только устанавливается соединение.

При выполнении следующего шага предполагается, что другой клиентский процесс на клиентском узле запрашивает соединение с тем же сервером. Код TCP клиента задает новому сокету клиента неиспользованный номер динамически назначаемого порта, скажем 1501. Мы получаем сценарий, представленный на рис. 2.14. На сервере различаются два соединения: пара сокетов для первого соединения отличается от пары сокетов для второго соединения, поскольку TCP клиента выбирает неиспользованный порт (1501) для второго соединения.

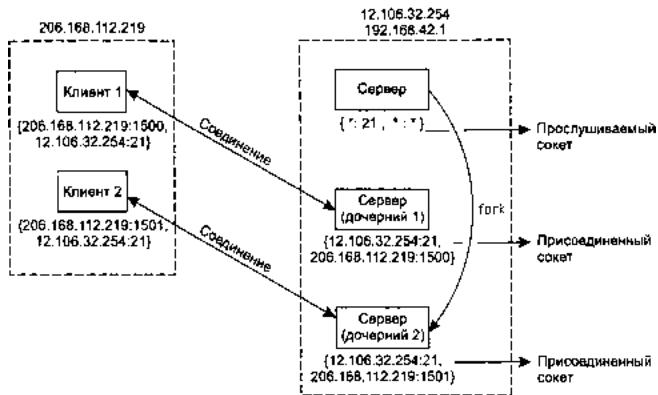


Рис. 2.14. Второе соединение клиента с тем же сервером

Из этого примера видно, что TCP не может демультиплексировать входящие сегменты, просматривая только номера портов назначения. TCP должен обращать внимание на все четыре элемента в паре сокетов, чтобы определить, какая конечная точка получает приходящий сегмент. На рис. 2.14 представлены три сокета с одним и тем же локальным портом (21). Если сегмент приходит с IP-адреса 206.168.112.219, порт 1500 и предназначен для IP-адреса 12.106.32.254, порт 21, он доставляется первому дочернему процессу. Если сегмент приходит с IP-адреса 206.168.112.219, порт 1501 и предназначен для IP-адреса 12.106.32.254, порт 21, он доставляется второму дочернему процессу. Все другие сегменты TCP, предназначенные для порта 21, доставляются исходному серверу с прослушиваемым сокетом.

2.11. Размеры буфера и ограничения

Существуют несколько ограничений, устанавливающих максимальный размер дейтаграмм IP. Сначала мы опишем эти ограничения, а затем свяжем их вместе, чтобы показать, как они влияют на данные, которые может передавать приложение.

- Максимальный размер дейтаграммы IPv4 — 65 535 байт, включая заголовок IPv4. Это связано с тем, что размер дейтаграммы ограничен 16-разрядным полем общей длины (см. рис. А.1).
- Максимальный размер дейтаграммы IPv6 — 65 575 байт, включая 40-байтовый заголовок IPv6. Это ограничение связано с 16-разрядным полем длины полезных данных на рис. А.2. Заметьте, что поле длины IPv6 не включает размер заголовка IPv6, в то время как в случае IPv4 длина заголовка включается.

IPv6 поддерживает возможность передачи *полезных данных увеличенного объема* (*jumbo payload*), при этом поле длины полезных данных расширяется до 32 бит, но эта функция поддерживается только на тех канальных уровнях, на которых максимальная единица передачи (MTU) превышает 65 535. Это свойство разработано для соединений между двумя узлами, таких как HIPPI (High-Performance Parallel Interface — высокоскоростной параллельный интерфейс), у которых часто нет собственных ограничений на MTU.

■ Во многих сетях определена MTU (maximum transmission unit — максимальная единица передачи), величина которой диктуется возможностями оборудования. Например, размер MTU для Ethernet равен 1500 байт. Другие канальные уровни, такие как соединения «точка-точка» с использованием протокола PPP, имеют конфигурируемую MTU. Более ранние соединения по протоколу SLIP (Serial Line Internet Protocol — межсетевой протокол для последовательного канала) часто использовали MTU, равную 296 или 1006 байт.

Минимальная величина *канальной MTU* (*link MTU*) для IPv4 — 68 байт. Это сумма размера заголовка IPv4 максимальной длины (20 байт фиксированных полей и 30 байт параметров) и фрагмента минимального размера (сдвиг фрагмента должен быть кратен 8 байтам). Минимальная величина MTU для IPv6 — 1280 байт. IPv6 может работать и в сетях с меньшей MTU, но при условии фрагментации и последующей сборки на канальном уровне, чтобы извне сеть казалась имеющей большую MTU (RFC 2460 [27]).

■ Наименьшая величина MTU в пути между двумя узлами называется *транспортной MTU* (*path MTU*). В настоящее время MTU Ethernet, равная 1500 байт, часто является и транспортной MTU. Величина транспортной MTU между любыми двумя узлами не обязательно должна быть одинаковой в обоих направлениях, поскольку маршрутизация в Интернете часто асимметрична [90]. То есть маршрут от А к В может отличаться от маршрута от В к А.

- Если размер дейтаграммы превышает канальную MTU, и IPv4 и IPv6 выполняют фрагментацию (*fragmentation*). Сборка (*reassemble*) фрагментов обычно не выполняется, пока они не достигнут конечного места назначения. Узлы IPv4 выполняют фрагментацию дейтаграмм, которые они генерируют, а маршрутизаторы IPv4 выполняют фрагментацию передаваемых ими дейтаграмм. Но в случае IPv6 дейтаграммы фрагментируются только узлами, а маршрутизаторы IPv6 фрагментацией не занимаются.
- Если в заголовке IPv4 (см. рис. А.1) установлен бит DF (don't fragment — не фрагментировать), это означает, что данная дейтаграмма не должна быть фрагментирована ни отправляющим узлом, ни любым маршрутизатором на ее пути. Маршрутизатор, получающий дейтаграмму IPv4 с установленным битом DF, размер которой превышает MTU исходящей линии, генерирует сообщение об ошибке ICMPv4 «Необходима фрагментация, но установлен бит DF» (см. табл. А.5).

Поскольку маршрутизаторы IPv6 не выполняют фрагментации, можно считать, что во всех дейтаграммах IPv6 установлен бит DF. Когда маршрутизатор IPv6 получает дейтаграмму, размер которой превышает MTU исходящей линии, он генерирует сообщение об ошибке ICMPv6 «Слишком большой пакет» (см. табл. А.6).

ПРИМЕЧАНИЕ

Будьте внимательны при использовании данной терминологии. Узел, помеченный как маршрутизатор IPv6, может все равно выполнять фрагментацию, но только для дейтаграмм, которые этот маршрутизатор генерирует сам. Он никогда не фрагментирует передаваемые им дейтаграммы. Когда этот узел генерирует дейтаграммы IPv6, он на самом деле выступает в роли узла (а не маршрутизатора). Например, большинство маршрутизаторов поддерживают протокол Telnet, используемый администраторами для настройки. Дейтаграммы IP, генерируемые сервером Telnet маршрутизатора, считаются порождаемыми маршрутизатором, поэтому он может выполнять их фрагментацию.

Вы можете заметить, что в заголовке IPv4 (см. рис. А.1) существуют поля для выполнения IPv4-фрагментации, но в заголовке IPv6 (см. рис. А.2) полей для фрагментации нет. Поскольку фрагментация скорее исключение, чем правило, IPv6 может содержать дополнительный заголовок с информацией о фрагментации.

Некоторые межсетевые экраны, обычно выполняющие по совместительству функции маршрутизаторов, могут собирать фрагментированные пакеты, чтобы проверять их содержимое целиком. Это позволяет предотвратить атаки определенного рода за счет дополнительного усложнения устройства экрана. Кроме того, для этого требуется, чтобы конкретный экран был единственной точкой соединения сети с внешней сетью, что сокращает возможности по обеспечению избыточности.

Бит DF протокола IPv4 и его аналог в IPv6 могут использоваться для обнаружения транспортной MTU (*path MTU discovery*) (RFC 1191 [78] для IPv4 и RFC 1981 [71] для IPv6). Например, если TCP использует этот прием с IPv4, он отправляет все дейтаграммы с установленным битом DF. Если какой-нибудь промежуточный маршрутизатор возвращает сообщение об ошибке ICMP «Место назначения недоступно, необходима фрагментация, но установлен бит DF», TCP уменьшает количество данных, которые он отправляет в каждой дейтаграмме, и передает их повторно. Обнаружение транспортной MTU не обязательно для IPv4, тогда как реализации IPv6 должны либо поддерживать обнаружение транспортной MTU, либо отсылать пакеты только с минимальной MTU.

■ IPv4 и IPv6 определяют минимальный размер буфера сборки (*minimum reassembly buffer size*) — максимальный размер дейтаграммы, который гарантированно поддерживает любая реализация. Для IPv4 этот размер равен 576 байт, для IPv6 он увеличен до 1500 байт. Например, в случае IPv4 мы не знаем, может ли данный пункт назначения принять дейтаграмму в 577 байт. Поэтому многие приложения IPv4, использующие UDP (DNS, RIP, TFTP, BOOTP, SNMP) предотвращают возможность генерирования приложением IP-дейтаграмм, превышающих этот размер.

■ Для протокола TCP определен максимальный размер сегмента (*MSS, maximum segment size*). MSS указывает собеседнику максимальный объем данных TCP, которые собеседник может отправлять в каждом сегменте. Параметр MSS мы видели в сегментах SYN на рис. 2.5. Цель параметра MSS — сообщить собеседнику действительный размер буфера сборки и попытаться предотвратить фрагментацию. Размер MSS часто устанавливается равным значению MTU интерфейса минус фиксированные размеры заголовков

IP и TCP. В Ethernet при использовании IPv4 это будет 1460, а в Ethernet при использовании IPv6 — 1440 (заголовок TCP для обоих протоколов имеет длину 20 байт, но заголовок IPv4 имеет длину 20 байт, а заголовок IPv6 — 40 байт).

16-разрядное поле MSS ограничивает величину соответствующего параметра на уровне 65 536. Это хорошо для IPv4, поскольку максимальное количество данных TCP в дейтаграмме IPv4 равно 65 495 (65 535 минус 20-байтовый заголовок IPv4 и 20-байтовый заголовок TCP). Но в случае увеличенного объема полезных данных дейтаграммы IPv6 используется другая технология (см. документ RFC 2675 [9]). Прежде всего, максимальное количество данных TCP в дейтаграмме IPv6 без увеличения объема полезных данных равно 65 515 байт (65 535 минус 20-байтовый заголовок IPv6). Следовательно, значение MSS, равное 65 535, считается особым случаем, обозначающим «бесконечность». Это значение используется только вместе с параметром увеличения объема полезных данных, что требует размера MTU, превышающего 65 535. Если TCP использует параметр увеличения объема полезных данных и получает от собеседника объявление размера MSS, равного 65 535 байт, предельный размер дейтаграммы, посылаемой им, будет равен просто величине MTU интерфейса. Если оказывается, что этот размер слишком велик (например, в пути существует канал с меньшим размером MTU), при обнаружении транспортной MTU будет установлено меньшее значение MSS.

- SCRIPT устанавливает параметр фрагментации равным наименьшей транспортной MTU для всех адресов собеседника. Сообщения, объем которых превышает эту величину, разбиваются на более мелкие, которые могут быть отправлены в одной IP-дейтаграмме. Параметр сокета `SCRIPT_MAXSEG` дает пользователю возможность установить меньший предел фрагментации.

Отправка по TCP

Приняв все вышеизложенные термины и определения, посмотрим на рис. 2.15, где показано, что происходит, когда приложение записывает данные в сокет TCP.

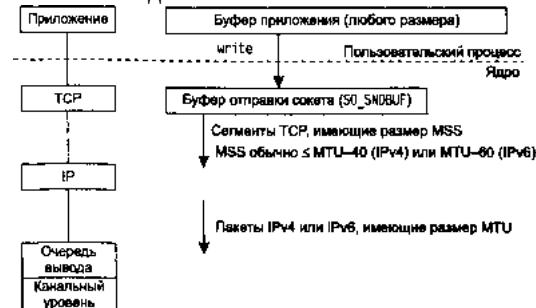


Рис. 2.15. Этапы записи данных в сокет TCP и буферы, используемые при этой записи

У каждого сокета TCP есть буфер отправки, и мы можем изменять размер этого буфера с помощью параметра сокета `SO_SNDBUF` (см. раздел 7.5). Когда приложение вызывает функцию `write`, ядро копирует данные из буфера приложения в буфер отправки сокета. Если для всех данных приложения недостаточно места в буфере сокета (либо буфер приложения больше буфера отправки сокета, либо в буфере отправки сокета уже имеются данные), процесс приостанавливается (переходит в состояние ожидания). Подразумевается, что мы используем обычный блокируемый сокет (о неблокируемых сокетах мы поговорим в главе 15). Ядро возвращает управление из функции `write` только после того, как последний байт в буфере приложения будет скопирован в буфер отправки сокета. Следовательно, успешное возвращение управления из функции `write` в сокет TCP говорит нам лишь о том, что мы можем снова использовать наш буфер приложения. Оно не говорит о том, получил ли собеседник отправленные данные или получило ли их приложение-адресат (более подробно мы рассмотрим это при описании параметра сокета `SO_LINGER` в разделе 7.5).

TCP помещает данные в буфер отправки сокета и отправляет их собеседнику TCP, основываясь на всех правилах передачи данных TCP (главы 19 и 20 [111]). Собеседник TCP должен подтвердить данные, и только когда от него придет сегмент ACK, подтверждающий прием данных, наш TCP сможет удалить подтвержденные данные из буфера отправки сокета. TCP должен хранить копию данных, пока их прием не будет подтвержден адресатом.

TCP отправляет данные IP порциями размером MSS или меньше, добавляя свой заголовок TCP к каждому сегменту. Здесь MSS — это значение, анонсированное собеседником, или 536, если собеседник не

указал значения для MSS. IP добавляет свой заголовок, ищет в таблице маршрутизации IP-адрес назначения (соответствующая запись в таблице маршрутизации задает исходящий интерфейс, то есть интерфейс для исходящих пакетов) и передает дейтаграмму на соответствующий канальный уровень. IP может выполнить фрагментацию перед передачей дейтаграммы, но, как мы отмечали выше, одна из целей параметра MSS — не допустить фрагментации; а более новые реализации также используют обнаружение транспортной MTU. У каждого канального соединения имеется очередь вывода, и если она заполнена, пакет игнорируется, и вверх по стеку протоколов возвращается ошибка: от канального уровня к IP и затем от IP к TCP. TCP учит эту ошибку и попытается отправить сегмент позже. Приложение не информируется об этом временном состоянии.

Отправка по UDP

На рис. 2.16 показано, что происходит, когда приложение записывает данные в сокет UDP.



Рис. 2.16. Отправка данных через сокет UDP

На этот раз буфер отправки сокета изображен пунктирными линиями, поскольку он (буфер) на самом деле не существует. У сокета UDP есть размер буфера отправки (который мы можем изменить с помощью параметра сокета `SO_SNDBUF`, см. раздел 7.5), но это просто верхнее ограничение на размер дейтаграммы UDP, которая может быть записана в сокет. Если приложение записывает дейтаграмму размером больше буфера отправки сокета, возвращается ошибка `EMSGSIZE`. Поскольку протокол UDP не является надежным, ему не нужно хранить копию данных приложения. Ему также не нужно иметь настоящий буфер отправки (данные приложения обычно копируются в буфер ядра по мере их движения вниз по стеку протоколов, но эта копия сбрасывается канальным уровнем после передачи данных).

UDP просто добавляет свой 8-байтовый заголовок и передает дейтаграмму протоколу IP. IPv4 или IPv6 добавляет свой заголовок, определяет исходящий интерфейс, выполняя функцию маршрутизации, и затем либо добавляет дейтаграмму в очередь вывода канального уровня (если размер дейтаграммы не превосходит MTU), либо фрагментирует дейтаграмму и добавляет каждый фрагмент в очередь вывода канального уровня.

Если приложение UDP отправляет большие дейтаграммы (например, 2000-байтовые), существует гораздо большая вероятность фрагментации, чем в случае TCP, поскольку TCP разбивает данные приложения на порции, равные по размеру MSS, а этому параметру нет аналога в UDP.

Успешное возвращение из функции записи в сокет UDP говорит о том, что либо дейтаграмма, либо фрагменты дейтаграммы были добавлены к очереди вывода канального уровня. Если для дейтаграммы или одного из ее фрагментов недостаточно места, приложению в большинстве случаев возвращается сообщение `ENOBUFS`.

ПРИМЕЧАНИЕ

К сожалению, некоторые реализации не возвращают этой ошибки, не предоставив приложению никаких указаний на то, что дейтаграмма была проигнорирована еще до начала передачи.

Отправка по SCRIPT

На рис. 2.17 показан процесс записи данных в сокет SCRIPT.

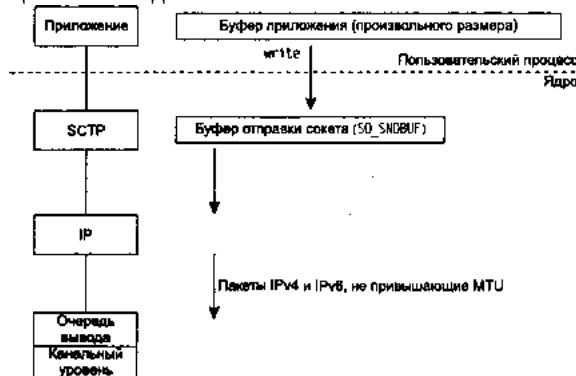


Рис. 2.17. Отправка данных через сокет SCRIPT

Для обеспечения надежности в SCRIPT предусмотрен буфер отправки. Приложение может менять размер этого буфера при помощи параметра сокета SO_SNDBUF (см. раздел 7.5), как и при работе с TCP. Когда приложение вызывает функцию `write`, ядро копирует все данные из буфера приложения в буфер отправки сокета. Если в буфере сокета недостаточно места для размещения всего объема данных приложения (то есть буфер приложения больше буфера сокета или в последнем уже имелись данные), пользовательский процесс приостанавливается. Приостановка производится для блокируемых сокетов. По умолчанию сокеты SCRIPT являются блокируемыми (о неблокируемых сокетах речь пойдет в главе 16). Ядро не возвращает управление процессу до тех пор, пока все байты буфера приложения не будут скопированы в буфер отправки сокета. Успешное возвращение из вызова `write` для сокета SCRIPT означает лишь, что приложение снова может воспользоваться своим буфером. Оно вовсе не означает, что SCRIPT адресата или приложение-адресат получили отправленные данные.

SCRIPT обрабатывает данные, которые находятся в буфере отправки на основании правил передачи SCRIPT (подробнее см. главу 5 [117]). Передающий SCRIPT должен дождаться получения порции SACK, в которой передается кумулятивное уведомление о приеме, чтобы удалить данные из буфера отправки сокета.

2.12. Стандартные службы Интернета

В табл. 2.1 перечислены некоторые стандартные службы, предоставляемые большинством реализаций TCP/IP. Заметьте, что все они поддерживают и TCP, и UDP, и номер порта для обоих протоколов один и тот же.

Таблица 2.1. Стандартные службы TCP/IP, предоставляемые в большинстве реализаций

Имя	Порт TCP	Порт UDP	RFC	Описание
echo	7	7	862	Сервер возвращает то, что посыпает клиент
discard	9	9	863	Сервер игнорирует все данные, присланные клиентом
daytime	13	13	867	Сервер возвращает время и дату в формате, удобном для восприятия человеком TCP-сервер посыпает непрерывный поток символов, пока соединение не будет разорвано клиентом. UDP-сервер посыпает дейтаграмму со случайным количеством символов каждый раз, когда клиент посыпает дейтаграмму
chargen	19	19	864	Сервер возвращает текущее время в виде двоичного 32-разрядного числа. Это число представляет собой количество секунд, прошедших с полуночи 1 января 1900 года (UTC)
time	37	37	868	

Часто эти службы предоставляются демоном `inetd` на узлах Unix (см. раздел 13.5). Стандартные службы делают возможным простейшее тестирование при помощи стандартного клиента Telnet.

Вот, например, тесты для сервера, определяющего время и дату, и для эхо-сервера.

```
aix % telnet freebsd daytime
Trying 12.106.32.254...      вывод клиента Telnet
Connected to freebsd.unpbook.com  вывод клиента Telnet
```

```

Escape character is '^]'.
Mon Jul 28 11:56:22 2003
Connection closed by foreign host.

```

вывод клиента Telnet
вывод сервера времени и даты
вывод клиента Telnet (сервер закрыл соединение)

```

aix % telnet freebsd echo
Trying 12.106.32.254...           вывод клиента Telnet
Connected to freebsd.unpbook.com   вывод клиента Telnet
Escape character is '^]'.
hello, world                      вывод с клавиатуры
hello, world                       эхо-ответ сервера
^]                                ввод с клавиатуры для обращения к клиенту Telnet
telnet> quit                      команда клиенту на завершение соединения
Connection closed. на этот раз соединение завершает клиент

```

В этих двух примерах мы вводим имя узла и название службы (daytime и echo). Соответствие названий служб и номеров портов (см. табл. 2.1) устанавливается в файле /etc/services (см. раздел 11.5).

Заметьте, что когда мы соединяемся с сервером daytime, сервер выполняет активное закрытие. В случае эхо-сервера активное закрытие выполняет клиент. Вспомним рис. 2.4, где показано, что узел, выполняющий активное закрытие, — это узел, проходящий состояние TIME_WAIT.

В современных системах стандартные службы чаще всего отключены по умолчанию, потому что через них могут быть проведены атаки типа «отказ в обслуживании» и другие, связанные с чрезмерным потреблением ресурсов.

2.13. Использование протоколов типичными приложениями Интернета

Таблица 2.2 иллюстрирует использование протоколов типичными приложениями Интернета.

Таблица 2.2. Использование протоколов типичными приложениями Интернета

Приложение	IP	ICMP	UDP	TCP	SCRIPT
ping
traceroute
OSPF (протокол маршрутизации)
RIP (протокол маршрутизации)
BGP (протокол маршрутизации)
BOOTP (протокол bootstrap — протокол дистанционной загрузки и запуска устройств в сети)
DHCP (протокол bootstrap)
NTP (синхронизирующий сетевой протокол)
TFTP (упрощенный протокол передачи файлов)
SNMP (управление сетью)
SMTP (электронная почта)
Telnet (удаленный вход в систему)
FTP (передача файлов)
HTTP (протокол передачи HTML-файлов по сети WWW)
NNTP (сетевой протокол передачи новостей)
DNS (система доменных имен)
NFS (сетевая файловая система)
Sun RPC (удаленный вызов процедур)
DCE RPC (удаленный вызов процедур)
IUA (ISDN поверх IP)
M2UA, M3UA (телефонная связь SS7)
H.248 (управление шлюзом)

• • •
H.323 (IP-телефония)

SIP (IP-телефония)

Первые два приложения, `ping` и `traceroute`, являются диагностическими и используют протокол ICMP, `traceroute` создает свои собственные пакеты UDP и считывает ответы ICMP.

Три популярных протокола маршрутизации демонстрируют многообразие транспортных протоколов, которые используются протоколами маршрутизации. Алгоритм OSPF (Open Shortest Path First — первоочередное открытие кратчайших маршрутов) использует IP непосредственно через символьный сокет, в то время как RIP (Routing Information Protocol — протокол информации о маршрутизации) использует UDP, а BGP (Border Gateway Protocol — протокол граничных шлюзов) использует TCP.

Далее идут пять приложений, основанные на UDP, за ними следуют семь приложений TCP и четыре приложения UDP/TCP. Последние пять приложений относятся к IP-телефонии. Они могут использовать либо только SCRIPT, либо UDP, TCP и SCRIPT по выбору.

2.14. Резюме

UDP является простым, ненадежным протоколом, не ориентированным на установление соединения, в то время как TCP — это сложный, надежный, ориентированный на установление соединения протокол. SCRIPT сочетает особенности обоих протоколов, расширяя возможности TCP. Хотя большинство приложений в Интернете используют протокол TCP (веб-сервисы, Telnet, FTP, электронная почта), существует потребность во всех трех транспортных протоколах. В разделе 22.4 мы рассматриваем причины, по которым иногда вместо TCP выбирается UDP. В разделе 23.12 будут проанализированы ситуации, в которых SCRIPT предпочтительнее TCP.

TCP устанавливает соединения, используя трехэтапное рукопожатие, и разрывает соединение, используя обмен четырьмя пакетами. Когда соединение TCP установлено, оно переходит из состояния CLOSED в состояние ESTABLISHED. При разрыве соединения оно переходит обратно в состояние CLOSED. Всего существует 11 состояний, в которых может находиться соединение TCP, и диаграмма переходов состояний определяет правила перемещения между этими состояниями. Понимание этой диаграммы существенно для диагностики проблем при использовании программы `netstat` и для понимания того, что происходит, когда мы вызываем такие функции, как `connect`, `accept` и `close`.

Состояние TCP TIME_WAIT — неиссякаемый источник путаницы, возникающей у сетевых программистов. Это состояние существует для того, чтобы реализовать разрыв двустороннего соединения TCP (то есть для решения проблем, возникающих в случае потери последнего сегмента ACK), а также чтобы дождаться, когда истечет время жизни в сети старых дублированных сегментов.

SCRIPT устанавливает ассоциацию, выполняя четырехэтапное рукопожатие, и завершает соединение обменом тремя пакетами. При установлении ассоциации SCRIPT происходит переход из состояния CLOSED в состояние ESTABLISHED, а при завершении ассоциации — возврат к состоянию CLOSED. Ассоциация SCRIPT может находиться в восьми состояниях, правила перехода между которыми описываются диаграммой состояний. Благодаря использованию контрольных меток SCRIPT не нуждается в состоянии TIME_WAIT.

Упражнения

1. Мы говорили об IPv4 и IPv6. А что произошло с версией 5 и каковы были версии 0, 1, 2 и 3? (Подсказка: найдите журнал IANA «Internet Protocol». Можете сразу переходить к решению, если вы не можете подключиться к <http://www.iana.org/>.)

2. Где вы будете искать дополнительную информацию о протоколе, которому присвоено название «IP версия 5»?

3. Описывая рис. 2.15, мы отметили, что TCP считает MSS равным 536, если не получает величину параметра MSS от собеседника. Почему используется это значение?

4. Нарисуйте рисунок, аналогичный рис. 2.5, для клиент-серверного приложения времени и даты из главы 1, предполагая, что сервер возвращает 26 байт данных в отдельном сегменте TCP.

5. Допустим, что установлено соединение между узлом в Ethernet, чей TCP объявляет MSS, равный 1460, и узлом в Token-ring, чей TCP объявляет MSS, равный 4096. Ни один из узлов не пытается обнаружить, чему равна транспортная MTU. При просмотре пакетов мы никогда не видим более 1460 байт данных в любом направлении. Почему?

6. Описывая табл. 2.2, мы отметили, что OSPF использует IP непосредственно. Каково значение поля протокола в заголовке IPv4 (см. рис. A.1) для дейтаграмм OSPF?

7. Обсуждая отправку данных по SCript, мы отметили, что отправителю приходится ждать получения кумулятивного уведомления, чтобы удалить данные из буфера сокета. Если еще до получения кумулятивного уведомления принято выборочное уведомление, указывающее, что данные уже доставлены, почему буфер все равно не может быть освобожден?

Часть 2

Элементарные сокеты

Глава 3

Введение в сокеты

3.1. Введение

Эта глава начинается с описания программного интерфейса приложения (API) сокетов. Мы начнем со структур адресов сокетов, которые будут встречаться почти в каждом примере на протяжении всей книги. Эти структуры можно передавать в двух направлениях: от процесса к ядру и от ядра к процессу. Последний случай — пример аргумента, через который передается возвращаемое значение, и далее в книге мы встретимся с другими примерами таких аргументов.

Перевод текстового представления адреса в двоичное значение, входящее в структуру адреса сокета, осуществляется функциями преобразования адресов. В большей части существующего кода IPv4 используются функции `inet_addr` и `inet_ntoa`, но две новых функции `inet_pton` и `inet_ntop` работают и с IPv4, и с IPv6.

Одной из проблем этих функций является то, что они зависят от протокола, так как для них имеет значение тип преобразуемого адреса — IPv4 или IPv6. Мы разработали набор функций, названия которых начинаются с `sock_`, работающих со структурами адресов сокетов независимо от протокола. Эти функции мы и будем использовать, чтобы сделать наш код не зависящим от протокола.

3.2. Структуры адреса сокетов

Большинство функций сокетов используют в качестве аргумента указатель на структуру адреса сокета. Каждый набор протоколов определяет свою собственную структуру адреса сокетов. Имена этих структур начинаются с `sockaddr_` и заканчиваются уникальным суффиксом для каждого набора протоколов.

Структура адреса сокета IPv4

Структура адреса сокета IPv4, обычно называемая структурой адреса сокета Интернета, именуется `sockaddr_in` и определяется в заголовочном файле `<netinet/in.h>`. В листинге 3.1^[1] представлено определение POSIX.

Листинг 3.1. Структура адреса сокета Интернета (IPv4): `sockaddr_in`

```
struct in_addr {  
    in_addr_t s_addr; /* 32-разрядный адрес IPv4 */  
    /* сетевой порядок байтов */  
};  
  
struct sockaddr_in {  
    uint8_t sin_len; /* длина структуры (16) */  
    sa_family_t sin_family; /* AF_INET */  
    in_port_t sin_port; /* 16-разрядный номер порта TCP или UDP */  
    /* сетевой порядок байтов */  
    struct in_addr sin_addr; /* 32-разрядный адрес IPv4 */  
    /* сетевой порядок байтов */  
    char sin_zero[8]; /* не используется */  
};
```

Есть несколько моментов, касающихся структур адреса сокета в целом, которые мы покажем на примере.

- Элемент длины `sin_len` появился в версии 4.3BSD-Reno, когда была добавлена поддержка протоколов OSI (см. рис. 1.6). До этой реализации первым элементом был `sin_family`, который исторически имел тип `unsigned short` (целое без знака). Не все производители поддерживают поле длины для структур адреса сокета, и в POSIX, например, не требуется наличия этого элемента. Типы данных, подобные `uint8_t`, введены в POSIX (см. табл. 3.1). Наличие поля длины упрощает обработку структур адреса сокета с переменной длиной.

- Даже если поле длины присутствует, нам не придется устанавливать и проверять его значение, пока мы не имеем дела с маршрутизирующими сокетами (см. главу 18). Оно используется внутри ядра процедурами, работающими со структурами адресов сокетов из различных семейств протоколов (например, код таблицы маршрутизации).

ПРИМЕЧАНИЕ

Четыре функции, передающие структуру адреса сокета от процесса к ядру, — bind, connect, sendto и sendmsg — используют функцию sockargs в реализациях, ведущих происхождение от Беркли [128, с. 452]. Эта функция копирует структуру адреса сокета из процесса и затем явно присваивает элементу sin_len значение размера структуры, переданной в качестве аргумента этим четырем функциям. Пять функций, передающих структуру адреса сокета от ядра к процессу, — accept, recvfrom, recvmsg, getpeername и getsockname — устанавливают элемент sin_len перед возвращением управления процессу.

К сожалению, обычно не существует простого теста, выполняемого в процессе компиляции и определяющего, задает ли реализация поле длины для своих структур адреса сокета. В нашем коде мы тестируем собственную константу HAVE_SOCKADDR_SA_LEN (см. листинг Г.2), но для того чтобы определить, задавать эту константу или нет, требуется откомпилировать простую тестовую программу, использующую необязательный элемент структуры, и проверить, успешно ли выполнена компиляция. В листинге 3.3 мы увидим, что от реализаций IPv6 требуется задавать SIN6_LEN, если структура адреса сокета имеет поле длины. В некоторых реализациях IPv4 (например, Digital Unix) поле длины предоставляется для приложений, основанных на параметре времени компиляции (например, _SOCKADDR_LEN). Это свойство обеспечивает совместимость с другими, более ранними программами.

- POSIX требует наличия только трех элементов структуры: sin_family, sin_addr и sin_port. POSIX-совместимая реализация может определять дополнительные элементы структуры, и это норма для структуры адреса сокета Интернета. Почти все реализации добавляют элемент sin_zero, так что все структуры адреса сокета имеют размер как минимум 16 байт.

■ Типы элементов sin_addr, sin_family и sin_port мы указываем согласно POSIX. Тип данных in_addr_t соответствует целому числу без знака длиной как минимум 32 бита, in_port_t — целому числу без знака длиной как минимум 16 бит, а sa_family_t — это произвольное целое число без знака. Последнее обычно представляет собой 8-разрядное целое без знака, если реализация поддерживает поле длины, либо 16-разрядное целое без знака, если поле длины не поддерживается. В табл. 3.1 перечислены эти три типа данных POSIX вместе с некоторыми другими типами данных POSIX, с которыми мы встретимся.

Таблица 3.1. Типы данных, требуемые POSIX

Тип данных	Описание	Заголовочный файл
int8_t	8-разрядное целое со знаком	<sys/types.h>
uint8_t	8-разрядное целое без знака	<sys/types.h>
int16_t	16-разрядное целое со знаком	<sys/types.h>
uint16_t	16-разрядное целое без знака	<sys/types.h>
int32_t	32-разрядное целое со знаком	<sys/types.h>
uint32_t	32-разрядное целое без знака	<sys/types.h>
sa_family_t	семейство адресов структуры адреса сокета	<sys/socket.h>
socklen_t	длина структуры адреса сокета, обычно типа uint32_t	<sys/socket.h>
in_addr_t	IPv4-адрес, обычно типа uint32_t	<netinet/in.h>
in_port_t	порт TCP или UDP, обычно типа uint16_t	<netinet/in.h>

■ Вы также встретите типы данных u_char, u_short, u_int и u_long, которые не имеют знака. POSIX определяет их с замечанием, что они устарели. Они предоставляются в целях обратной совместимости.

■ И адрес IPv4, и номер порта TCP и UDP всегда хранятся в структуре в соответствии с порядком байтов, определенным в сети (*сетевой порядок байтов — network byte order*). Об этом нужно помнить при

использовании этих элементов (более подробно о разнице между порядком байтов узла и порядком байтов в сети мы поговорим в разделе 3.4).

■ К 32-разрядному адресу IPv4 можно обратиться двумя путями. Например, если `serv` — это структура адреса сокета Интернета, то `serv.sin_addr` указывает на 32-разрядный адрес IPv4 как на структуру `in_addr`, в то время как `serv.sin_addr.s_addr` указывает на тот же 32-разрядный адрес IPv4 как на значение типа `in_addr_t` (обычно это 32-разрядное целое число без знака). Нужно следить за корректностью обращения к адресам IPv4, особенно при использовании их в качестве аргументов различных функций, потому что компиляторы часто передают структуры не так, как целочисленные переменные.

ПРИМЕЧАНИЕ

Причина того, что `sin_addr` является структурой, а не просто целым числом без знака, носит исторический характер. В более ранних реализациях (например, 4.2BSD) структура `in_addr` определялась как объединение (`union`) различных структур, чтобы сделать возможным доступ к каждому из четырех байтов 32-разрядного IPv4-адреса, а также к обоим входящим в него 16-разрядным значениям. Эта возможность использовалась в адресах классов A, B и C для выборки соответствующих байтов адреса. Но с появлением подсетей и последующим исчезновением различных классов адресов (см. раздел А.4) и введением бесклассовой адресации (`classless addressing`) необходимость в объединении структур отпала. В настоящее время большинство систем отказались от использования объединения и просто определяют `in_addr` как структуру, содержащую один элемент типа `in_addr_t`.

■ Элемент `sin_zero` не используется, но мы всегда устанавливаем его в нуль при заполнении одной из этих структур. Перед заполнением структуры мы всегда обнуляем все ее элементы, а не только `sin_zero`.

ПРИМЕЧАНИЕ

В большинстве случаев при использовании этой структуры не требуется, чтобы элемент `sin_zero` был равен нулю, но, например, при привязке конкретного адреса IPv4 (а не произвольного интерфейса) этот элемент обязательно должен быть нулевым [128, с. 731-732].

■ Структуры адреса сокета используются только на данном узле: сама структура не передается между узлами, хотя определенные поля (например, поля IP-адреса и порта) используются для соединения.

Универсальная структура адреса сокета

Структуры адреса сокета всегда передаются по ссылке при передаче в качестве аргумента для любой функции сокета. Но функции сокета, принимающие один из этих указателей в качестве аргумента, должны работать со структурами адреса сокета из *любого* поддерживаемого семейства протоколов.

Проблема в том, как объявить тип передаваемого указателя. Для ANSI C решение простое: `void*` является указателем на неопределенный (универсальный) тип (*generic pointer type*). Но функции сокетов существовали до появления ANSI C, и в 1982 году было принято решение определить *универсальную* структуру адреса сокета (*generic socket address structure*) в заголовочном файле `<sys/socket.h>`, которая показана в листинге 3.2.

Листинг 3.2. Универсальная структура адреса сокета: `sockaddr`

```
struct sockaddr {
    uint8_t sa_len;
    sa_family_t sa_family; /* семейство адресов: константа AF_XXX */
    char sa_data[14];      /* адрес, специфичный для протокола */
};
```

Функции сокетов определяются таким образом, что их аргументом является указатель на общую структуру адреса сокета, как показано в прототипе функции `bind` (ANSI C):

```
int bind(int, struct sockaddr*, socklen_t);
```

При этом требуется, чтобы для любых вызовов этих функций указатель на структуру адреса сокета, специфичную для протокола, был преобразован в указатель на универсальную структуру адреса сокета. Например:

```
struct sockaddr_in serv; /* структура адреса сокета IPv4 */  
  
/* заполняем serv{} */  
bind(sockfd, (struct sockaddr*)&serv, sizeof(serv));
```

Если мы не выполним преобразование (`struct sockaddr*`), компилятор С генерирует предупреждение в форме "Warning: passing arg 2 of 'bind' from incompatible pointer type" (Передается указатель несовместимого типа). Здесь мы предполагаем, что в системных заголовочных файлах имеется прототип ANSI C для функции `bind`.

С точки зрения разработчика приложений, универсальная структура адреса сокета используется только для преобразования указателей на структуры адресов конкретных протоколов.

ПРИМЕЧАНИЕ

Вспомните, что в нашем заголовочном файле `inpr.h` (см. раздел 1.2) мы определили `SA` как строку "`struct sockaddr`", чтобы сократить код, который мы написали для преобразования этих указателей.

С точки зрения ядра основанием использовать в качестве аргументов указатели на универсальные структуры адреса сокетов является то, что ядро должно получать указательзывающей функции, преобразовывать его в `struct sockaddr`, а затем по значению элемента `sa_family` определять тип структуры. Но разработчику приложений было бы проще работать с указателем `void*`, поскольку это избавило бы его от необходимости выполнять явное преобразование указателя.

Структура адреса сокета IPv6

Структура адреса сокета IPv6 задается при помощи включения заголовочного файла `<netinet/in.h>`, как показано в листинге 3.3.

Листинг 3.3. Структура адреса сокета IPv6: `sockaddr_in6`

```
struct in6_addr {  
    uint8_t s6_addr[16]; /* 128-разрядный адрес IPv6 */  
    /* сетевой порядок байтов */  
};  
  
#define SIN6_LEN /* требуется для проверки во время компиляции */  
  
struct sockaddr_in6 {  
    uint8_t sin_len; /* длина этой структуры (24) */  
    sa_family_t sin6_family; /* AF_INET6 */  
    in_port_t sin6_port; /* номер порта транспортного уровня */  
    /* сетевой порядок байтов */  
    uint32_t sin6_flowinfo; /* приоритет и метка потока */  
    /* сетевой порядок байтов */  
    struct in6_addr sin6_addr; /* IPv6-адрес */  
    /* сетевой порядок байтов */  
    uint32_t sin6_scope_id; /* набор интерфейсов */  
};
```

ПРИМЕЧАНИЕ

Расширения API сокетов для IPv6 описаны в RFC 3493 [36].

Отметим следующие моменты относительно листинга 3.3:

- Константа `SIN6_LEN` должна быть задана, если система поддерживает поле длины для структур адреса сокета.
- Семейством IPv6 является `AF_INET6`, в то время как семейство IPv4 — `AF_INET`.
- Элементы в структуре упорядочены таким образом, что если структура `sockaddr_in6` выровнена по 64 битам, то так же выровнен и 128-разрядный элемент `sin6_addr`. На некоторых 64-разрядных процессорах доступ к данным с 64-разрядными значениями оптимизирован, если данные выровнены так, что их адрес кратен 64.
- Элемент `sin6_flowinfo` разделен на три поля:
 - 20 бит младшего порядка — это метка потока;
 - следующие 12 бит зарезервированы.

Поле метки потока и поле приоритета рассматриваются в описании рис. А.2. Отметим, что использование поля приоритета еще не определено.

- Элемент `sin6_scope_id` определяет контекст, в котором действует контекстный адрес (scoped address). Чаще всего это бывает индекс интерфейса для локальных адресов (см. раздел А.5).

Новая универсальная структура адреса сокета

Новая универсальная структура адреса сокета была определена как часть API сокетов IPv6 с целью преодолеть некоторые недостатки существующей структуры `sockaddr`. В отличие от структуры `sockaddr`, новая структура `sockaddr_storage` достаточно велика для хранения адреса сокета любого типа, поддерживаемого системой. Новая структура задается подключением заголовочного файла `<netinet/in.h>`, часть которого показана в листинге 3.4.

Листинг 3.4. Структура хранения адреса сокета `sockaddr_storage`

```
struct sockaddr_storage {  
    uint8_t ss_len; /* длина этой структуры (зависит от реализации) */  
    sa_family_t ss_family; /* семейство адреса. AF_XXX */  
    /* зависящие от реализации элементы, обеспечивающие:  
     * а) выравнивание, достаточное для выполнения требований по выравниванию всех  
     * типов адресов сокетов, поддерживаемых системой;  
     * б) достаточный объем для хранения адреса сокета любого типа,  
     * поддерживаемого системой. */  
};
```

Тип `sockaddr_storage` — это универсальная структура адреса сокета, отличающаяся от `struct sockaddr` по следующим параметрам:

1. Если к структурам адресов сокетов, поддерживаемым системой, предъявляются требования по выравниванию, структура `sockaddr_storage` выполняет самое жесткое из них.
2. Структура `sockaddr_storage` достаточно велика для размещения любой структуры адреса сокета, поддерживаемой системой.

Заметьте, что поля структуры `sockaddr_storage` непрозрачны для пользователя, за исключением `ss_family` и `ss_len` (если таковые заданы). Структура `sockaddr_storage` должна преобразовываться в структуру адреса соответствующего типа для обращения к содержимому остальных полей.

Сравнение структур адреса сокетов

На рис. 3.1 показано сравнение пяти структур адресов сокетов, с которыми мы встретимся в тексте, предназначенных для IPv4, IPv6, доменного сокета Unix (см. листинг 15.1), канального уровня (см. листинг 18.1) и хранения. Подразумевается, что все структуры адреса сокета содержат 1-байтовое поле длины, поле семейства также занимает 1 байт и длина любого поля, размер которого ограничен снизу, в точности равна этому ограничению.

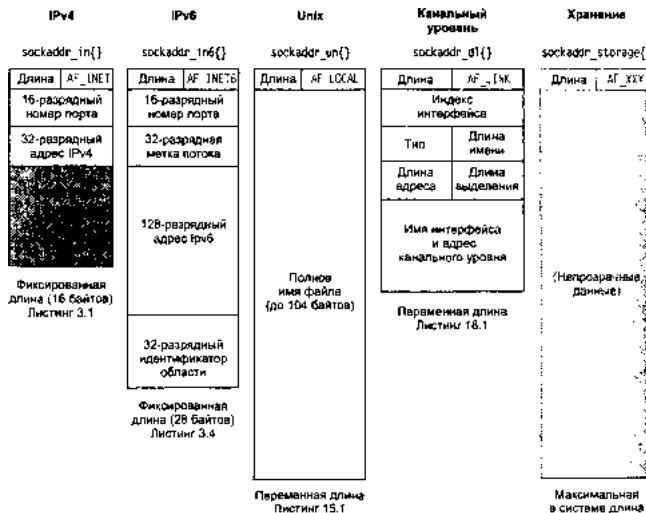


Рис. 3.1. Сравнение различных структур адресов сокетов

Две структуры адреса сокета имеют фиксированную длину, а структура доменного сокета Unix и структура канального уровня — переменную. При обработке структур переменной длины мы передаем функциям сокетов указатель на структуру адреса сокета, а в другом аргументе передаем длину этой структуры. Под каждой структурой фиксированной длины мы показываем ее размер в байтах (для реализации 4.4BSD).

ПРИМЕЧАНИЕ

Сама структура sockaddr_un имеет фиксированную длину, но объем информации в ней — длина полного имени (pathname) — может быть переменным. Передавая указатели на эти структуры, следует соблюдать аккуратность при обработке поля длины — как длины в структуре адреса сокета (если поле длины поддерживается данной реализацией), так и длины данных, передаваемых ядру и принимаемых от него.

Этот рисунок служит также иллюстрацией стиля, которого мы придерживаемся в этой книге: названия структур на рисунках всегда выделяются полужирным шрифтом, а за ними следуют фигурные скобки.

Ранее отмечалось, что в реализации 4.3BSD Reno ко всем структурам адресов сокетов было добавлено поле длины. Если бы поле длины присутствовало в оригинальной реализации сокетов, то не возникло бы необходимости передавать аргумент длины функциям сокетов (третий аргумент функций bind и connect). Вместо этого размер структуры мог бы храниться в поле длины структуры.

3.3. Аргументы типа «значение-результат»

Мы отмечали, что когда структура адреса сокета передается какой-либо из функций сокетов, она всегда передается по ссылке, то есть в качестве аргумента передается указатель на структуру. Длина структуры также передается в качестве аргумента. Но способ, которым передается длина, зависит от того, в каком направлении передается структура: от процесса к ядру или наоборот.

1. Три функции bind, connect и sendto передают структуру адреса сокета от процесса к ядру. Один из аргументов этих функций — указатель на структуру адреса сокета, другой аргумент — это целочисленный размер структуры, как показано в следующем примере:

```
struct sockaddr_in serv;
```

```
/* заполняем serv[] */  
connect(sockfd, (SA*)&serv, sizeof(serv));
```

Поскольку ядру передается и указатель, и размер структуры, на которую он указывает, становится точно известно, какое количество данных нужно скопировать из процесса в ядро. На рис. 3.2 показан этот

сценарий.

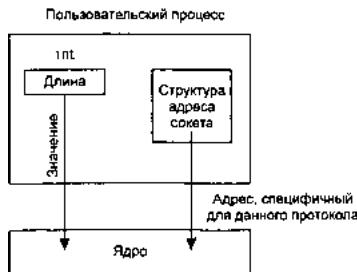


Рис. 3.2. Структура адреса сокета, передаваемая от процесса к ядру

В следующей главе мы увидим, что размер структуры адреса сокета в действительности имеет тип `socklen_t`, а не `int`, но POSIX рекомендует определять `socklen_t` как `uint32_t`.

2. Четыре функции `accept`, `recvfrom`, `getsockname` и `getpeername` передают структуру адреса сокета от ядра к процессу, то есть в направлении, противоположном предыдущему случаю. Этим функциям передается указатель на структуру адреса сокета и указатель на целое число, содержащее размер структуры, как показано в следующем примере:

```
struct sockaddr_un cli; /* домен Unix */
socklen_t len;
len = sizeof(cli);      /* len - это значение */
getpeername(unixfd, (SA*)&cli, &len);
/* значение len могло измениться */
```

Причина замены типа для аргумента «длина» с целочисленного на указатель состоит в том, что «длина» эта является и значением при вызове функции (сообщает ядру размер структуры, так что ядро при заполнении структуры знает, где нужно остановиться), и результатом, когда функция возвращает значение (сообщает процессу, какой объем информации ядро действительно сохранило в этой структуре). Такой тип аргумента называется *аргументом типа «значение-результат»* (*value-result argument*). На рис. 3.3 представлен этот сценарий.

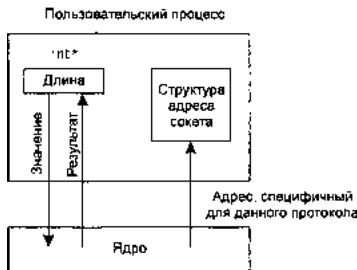


Рис. 3.3. Структура адреса сокета, передаваемая от ядра к процессу

Пример аргументов типа «значение-результат» вы увидите в листинге 4.2.

Если при использовании аргумента типа «значение-результат» для длины структуры структура адреса сокета имеет фиксированную длину (см. рис. 3.1), то значение, возвращаемое ядром, будет всегда равно этому фиксированному размеру: 16 для `sockaddr_in` IPv4 и 24 для `sockaddr_in6` IPv6. Для структуры адреса сокета переменной длины (например, `sockaddr_un` домена Unix) возвращаемое значение может быть меньше максимального размера структуры (вы увидите это в листинге 15.2).

ПРИМЕЧАНИЕ

Мы говорили о структурах адресов сокетов, передаваемых между процессом и ядром. Для такой реализации, как 4.4BSD, где все функции сокетов являются системными вызовами внутри ядра, это верно. Но в некоторых реализациях, особенно в System V, функции сокетов являются лишь библиотечными функциями, которые выполняются как часть обычного пользовательского процесса. То, как эти функции взаимодействуют со стеком протоколов в ядре, относится к деталям реализации, которые обычно нас не волнуют. Тем не менее для простоты изложения мы будем продолжать говорить об этих структурах как о передаваемых между процессом и ядром такими функциями, как `bind` и `connect`. (В разделе B.1 вы увидите, что реализации System V

действительно передают пользовательские структуры адресов сокетов между процессом и ядром, но как часть сообщений потоков STREAMS.)

Существует еще две функции, передающие структуры адресов сокетов: это `recvmsg` и `sendmsg` (см. раздел 14.5). Однако при их вызове поле длины не является отдельным аргументом функции, а передается как одно из полей структуры.

В сетевом программировании наиболее общим примером аргумента типа «значение-результат» может служить длина возвращаемой структуры адреса сокета. Вы встретите и другие аргументы типа «значение-результат»:

- Три средних аргумента функции `select` (раздел 6.3).
- Аргумент «длина» для функции `getsockopt` (см. раздел 7.2).
- Элементы `msg_namelen` и `msg_controllen` структуры `msghdr` при использовании с функцией `recvmsg` (см. раздел 14.5).
- Элемент `ifc_len` структуры `ifconf` (см. листинг 17.1).
- Первый из двух аргументов длины в функции `sysctl` (см. раздел 18.4).

3.4. Функции определения порядка байтов

Рассмотрим 16-разрядное целое число, состоящее из двух байтов. Возможно два способа хранения этих байтов в памяти. Такое расположение, когда первым идет младший байт, называется *прямым порядком байтов* (*little-endian*), а когда первым расположен старший байт — *обратным порядком байтов* (*big-endian*). На рис. 3.4 показаны оба варианта.

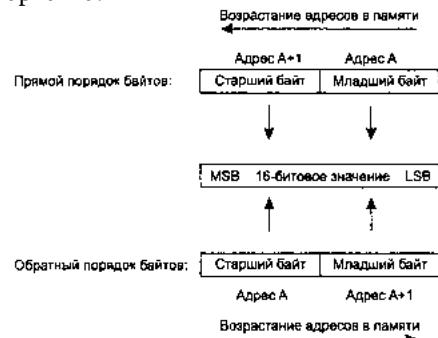


Рис. 3.4. Прямой и обратный порядок байтов для 16-разрядного целого числа

Сверху на этом рисунке изображены адреса, возрастающие справа налево, а снизу — слева направо. Старший бит (*most significant bit, MSB*) является в 16-разрядном числе крайним слева, а младший бит (*least significant bit, LSB*) — крайним справа.

ПРИМЕЧАНИЕ

Термины «прямой порядок байтов» и «обратный порядок байтов» указывают, какой конец многобайтового значения — младший байт или старший — хранится в качестве начального адреса значения.

К сожалению, не существует единого стандарта порядка байтов, и можно встретить системы, использующие оба формата. Способ упорядочивания байтов, используемый в конкретной системе, мы называем *порядком байтов узла* (*host byte order*). Программа, представленная в листинге 3.5, выдает порядок байтов узла.

Листинг 3.5. Программа для определения порядка байтов узла

```
//intro/bytorder.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
```

```

4 {
5 union {
6 short s;
7 char c[sizeof(short)];
8 } un;

9 un.s = 0x0102;
10 printf("%s: ", CPU_VENDOR_OS);
11 if (sizeof(short) == 2) {
12 if (un.c[0] == 1 && un.c[1] == 2)
13 printf("big-endian\n");
14 else if (un.c[0] == 2 && un.c[1] == 1)
15 printf("little-endian\n");
16 else
17 printf("unknown\n");
18 } else
19 printf('sizeof(short) = %d\n", sizeof(short));
20 exit(0);
21 }

```

Мы помещаем двухбайтовое значение 0x0102 в переменную типа `short` (короткое целое) и проверяем значения двух байтов этой переменной: `c[0]` (адрес А на рис. 3.4) и `c[1]` (адрес А + 1 на рис. 3.4), чтобы определить порядок байтов.

Константа `CPU_VENDOR_OS` определяется программой GNU (аббревиатура «GNU» раскрывается рекурсивно — GNU's Not Unix) `autoconf` в процессе конфигурации, необходимой для выполнения программ из этой книги. В этой константе хранится тип центрального процессора, а также сведения о производителе и реализации операционной системы. Ниже представлены некоторые примеры вывода этой программы при запуске ее в различных системах (см. рис. 1.7).

```

freebsd4 % byteorder
i386-unknown-freebsd4.8: little-endian

macosx % byteorder
powerpc-apple-darwin6.6: big-endian

freebsd5 % byteorder
sparc64-unknown-freebsd5.1: big-endian

aix % byteorder
powerpc-ibm-aix5.1.0.0: big-endian

hpx % byteorder
hppa1.1-hp-ux11 11: big-endian

linux % byteorder
i586-pc-linux-gnu: little-endian

solaris % byteorder
sparc-sun-solaris2.9: big-endian

```

Все, что было сказано об определении порядка байтов 16-разрядного целого числа, конечно, справедливо и в отношении 32-разрядного целого.

ПРИМЕЧАНИЕ

Существуют системы, в которых возможен переход от прямого к обратному порядку байтов либо при перезапуске системы (MIPS 2000), либо в любой момент выполнения программы (Intel i860).

Разработчикам сетевых приложений приходится обрабатывать различия в определении порядка байтов, поскольку в сетевых протоколах используется *сетевой порядок байтов* (*network byte order*). Например, в сегменте TCP есть 16-разрядный номер порта и 32-разрядный адрес IPv4. Стеки отправляющего и принимающего протоколов должны согласовывать порядок, в котором передаются байты этих многобайтовых полей. Протоколы Интернета используют обратный порядок байтов.

Теоретически реализация Unix могла бы хранить поля структуры адреса сокета в порядке байтов узла, а затем выполнять необходимые преобразования при перемещении полей в заголовки протоколов и обратно, позволяя нам не беспокоиться об этом. Но исторически и с точки зрения POSIX определяется, что для некоторых полей в структуре адреса сокета порядок байтов всегда должен быть сетевым. Поэтому наша задача — выполнить преобразование из порядка байтов узла в сетевой порядок и обратно. Для этого мы используем следующие четыре функции:

```
#include <netinet/in.h>
```

```
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
```

Обе функции возвращают значение, записанное в сетевом порядке байтов

```
uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);
```

Обе функции возвращают значение, записанное в порядке байтов узла

В названиях этих функций h обозначает узел, n обозначает сеть, s — тип *short*, l — тип *long*. Термины *short* и *long* являются наследием времен реализации 4.2BSD Digital VAX. Следует воспринимать s как 16-разрядное значение (например, номер порта TCP или UDP), а l — как 32-разрядное значение (например, адрес IPv4). В самом деле, в 64-разрядной системе Digital Alpha длинное целое занимает 64 разряда, а функции htonl и ntohl оперируют 32-разрядными значениями (несмотря на то, что используют тип long).

Используя эти функции, мы можем не беспокоиться о реальном порядке байтов на узле и в сети. Для преобразования порядка байтов в конкретном значении следует вызвать соответствующую функцию. В системах с таким же порядком байтов, как в протоколах Интернета (обратным), эти четыре функции обычно определяются как пустой макрос.

Мы еще вернемся к проблеме определения порядка байтов, обсуждая данные, содержащиеся в сетевом пакете, и сравнивая их с полями в заголовках протокола, в разделе 5.18 и упражнении 5.8.

Мы до сих пор не определили термин байт. Его мы будем использовать для обозначения 8 бит, поскольку практически все современные компьютерные системы используют 8-битовые байты. Однако в большинстве стандартов Интернета для обозначения 8 бит используется термин *октет*. Началось это на заре TCP/IP, поскольку большая часть работы выполнялась в системах типа DEC-10, в которых не применялись 8-битовые байты. Еще одно важное соглашение, принятое в стандартах Интернета, связано с порядком битов. Во многих стандартах вы можете увидеть «изображения» пакетов, подобные приведенному ниже (это первые 32 разряда заголовка IPv4 из RFC 791):

0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+			
Version IHL Type of Service	Total Length		
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+			

В этом примере приведены четыре байта в том порядке, в котором они передаются по проводам. Крайний слева бит является наиболее значимым. Однако нумерация начинается с нуля, который соответствует как раз наиболее значимому биту. Вам необходимо получше ознакомиться с этой записью, чтобы не испытывать трудностей при чтении описаний протоколов в RFC.

ПРИМЕЧАНИЕ

Типичной ошибкой среди программистов сетевых приложений начала 80-х, разрабатывающих код на рабочих станциях Sun (Motorola 68000 с обратным порядком байтов), было забыть вызвать одну из указанных четырех функций. На этих рабочих станциях программы работали нормально, но при переходе на машины с прямым порядком байтов они переставали работать.

3.5. Функции управления байтами

Существует две группы функций, работающих с многобайтовыми полями без преобразования данных и без интерпретации их в качестве строк языка С с завершающим нулем. Они необходимы нам при обработке структур адресов сокетов, поскольку такие поля этих структур, как IP-адреса, могут содержать нулевые байты, но при этом не являются строками С. Строки с завершающим нулем обрабатываются функциями языка С, имена которых начинаются с аббревиатуры `str`. Эти функции подключаются с помощью файла `<string.h>`.

Первая группа функций, названия которых начинаются с `b` (от слова «byte» — «байт»), взяты из реализации 4.2BSD и все еще предоставляются практически любой системой, поддерживающей функции сокетов. Вторая группа функций, названия которых начинаются с `mem` (от слова «memory» — память), взяты из стандарта ANSI C и доступны в любой системе, обеспечивающей поддержку библиотеки ANSI C.

Сначала мы представим функции, которые берут начало от реализации Беркли, хотя в книге мы будем использовать только одну из них — `bzero`. (Дело в том, что она имеет только два аргумента и ее проще запомнить, чем функцию `memset` с тремя аргументами, как объяснялось в разделе 1.2.) Две другие функции, `bcopy` и `bcmpl`, могут встретиться вам в существующих приложениях.

```
#include <strings.h>

void bzero(void *dest, size_t nbytes);

void bcopy(const void *src, void *dest, size_t nbytes);

int bcmpl(const void *ptr1, const void *ptr2, size_t nbytes);
Возвращает: 0 в случае равенства, ненулевое значение в случае неравенства
```

ПРИМЕЧАНИЕ

Мы впервые встречаемся со спецификатором `const`. В приведенном примере он служит признаком того, что значения, на которые указывает указатель, то есть `src`, `ptr1` и `ptr2`, не изменяются функцией. Другими словами, область памяти, на которую указывает указатель со спецификатором `const`, считывается функцией, но не изменяется.

Функция `bzero` обнуляет заданное число байтов в указанной области памяти. Мы часто используем эту функцию для инициализации структуры адреса сокета нулевым значением. Функция `bcopy` копирует заданное число байтов из источника в место назначения. Функция `bcmpl` сравнивает две произвольных последовательности байтов и возвращает нулевое значение, если две байтовых строки идентичны, и ненулевое — в противном случае.

Следующие функции являются функциями ANSI C:

```
#include <string.h>

void *memset(void *dest, int c, size_t len);

void *memcpy(void *dest, const void *src, size_t nbytes);

int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);
Возвращает: 0 в случае равенства, значение <0 или >0 в случае неравенства (см. текст)
```

Функция `memset` присваивает заданному числу байтов значение `c`. Функция `memcpy` аналогична функции `bcopy`, но имеет другой порядок двух аргументов. Функция `bcopy` корректно обрабатывает перекрывающиеся поля, в то время как поведение функции `memcpy` не определено, если источник и место назначения перекрываются. В случае перекрывания полей должна использоваться функция ANSI C `memmove` (упражнение 30.3).

ПРИМЕЧАНИЕ

Чтобы запомнить порядок аргументов функции `memcp`, подумайте о том, что он совпадает с порядком аргументов в операторе присваивания (справа — оригинал, слева — копия).

```
dest = src;
```

Последним аргументом этой функции (как и всех ANSI-функций `memXXX`) всегда является длина области памяти.

Функция `memcmp` сравнивает две произвольных последовательности байтов и возвращает нуль, если они идентичны. В противном случае знак возвращаемого значения определяется знаком разности между первыми несовпадающими байтами, на которые указывают `ptr1` и `ptr2`. Предполагается, что сравниваемые байты принадлежат к типу `unsigned char`.

3.6. Функции `inet_aton`, `inet_addr` и `inet_ntoa`

Существует две группы функций преобразования адресов, которые мы рассматриваем в этом и следующем разделах. Они выполняют преобразование адресов Интернета из строк ASCII (удобных для человеческого восприятия) в двоичные значения с сетевым порядком байтов (эти значения хранятся в структурах адресов сокетов).

1. Функции `inet_aton`, `inet_ntoa` и `inet_addr` преобразуют адрес IPv4 из точечно-десятичной записи (например, 206.168.112.96) в 32-разрядное двоичное значение в сетевом порядке байтов. Возможно, вы встретите эти функции в многочисленных существующих программах.

2. Более новые функции `inet_pton` и `inet_ntop` работают и с адресами IPv4, и с адресами IPv6. Эти функции, описываемые в следующем разделе, мы используем в книге.

```
#include <arpa/inet.h>
```

```
int inet_aton(const char *strptr, struct in_addr *addrptr);
```

Возращает: 1, если строка преобразована успешно, 0 в случае ошибки

```
in_addr_t inet_addr(const char *strptr);
```

Возращает: 32-разрядный адрес IPv4 в сетевом порядке байтов: `INADDR_NONE` в случае ошибки

```
char *inet_ntoa(struct in_addr inaddr);
```

Возращает: указатель на строку с адресом в точечно-десятичной записи

Первая из названных функций, `inet_aton`, преобразует строку, на которую указывает `strptr`, в 32-разрядное двоичное число, записанное в сетевом порядке байтов, передаваемое через указатель `addrptr`. При успешном выполнении возвращаемое значение равно 1, иначе возвращается нуль.

ПРИМЕЧАНИЕ

Функция `inet_aton` обладает одним недокументированным свойством: если `addrptr` — пустой указатель (null pointer), функция все равно выполняет проверку допустимости адреса, содержащегося во входной строке, но не сохраняет результата.

Функция `inet_addr` выполняет то же преобразование, возвращая в качестве значения 32-разрядное двоичное число в сетевом порядке байтов. Проблема при использовании этой функции состоит в том, что все 2^{32} возможных двоичных значений являются действительными IP-адресами (от 0.0.0.0 до 255.255.255.255), но в случае возникновения ошибки функция возвращает константу `INADDR_NONE` (обычно представленную двоичным числом, состоящим из 32 бит, установленных в единицу). Это означает, что точечно-десятичная запись 255.255.255.255 (ограниченный адрес для широковещательной передачи IPv4, см. раздел 18.2) не может быть обработана этой функцией, поскольку ее двоичное значение выглядит как указание на сбой при выполнении функции.

ПРИМЕЧАНИЕ

Характерной проблемой, сопровождающей выполнение функции `inet_addr`, может стать то, что, как утверждается в некоторых руководствах, в случае ошибки она возвращает значение `-1` вместо `INADDR_NONE`. С некоторыми компиляторами это может вызвать проблемы при сравнении возвращаемого значения функции (значение без знака) с отрицательной константой.

На сегодняшний день функция `inet_addr` является нерекомендуемой, или устаревшей, и в создаваемом коде вместо нее должна использоваться функция `inet_aton`. Еще лучше использовать более новые функции, описанные в следующем разделе, работающие с IPv4, и с IPv6.

Функция `inet_ntoa` преобразует 32-разрядный двоичный адрес IPv4, хранящийся в сетевом порядке байтов, в точечно-десятичную строку. Стока, на которую указывает возвращаемый функцией указатель, находится в статической памяти. Это означает, что функция не допускает повторного вхождения, то есть не является повторно входимой (reentrant), что мы обсудим в разделе 11.14. Наконец, отметим, что эта функция принимает в качестве аргумента структуру, а не указатель на структуру.

ПРИМЕЧАНИЕ

Функции, принимающие структуры в качестве аргументов, встречаются редко. Более общим способом является передача указателя на структуру.

3.7. Функции `inet_pton` и `inet_ntop`

Эти функции появились с IPv6 и работают как с адресами IPv4, так и с адресами IPv6. Их мы и будем использовать в книге. Символы `r` и `n` обозначают соответственно формат *представления* и *числennyй* формат. Формат представления адреса часто является строкой ASCII, а числennyй формат — это двоичное значение, входящее в структуру адреса сокета. #include <arpa/inet.h>

```
int inet_pton(int family, const char *strptr, void *addrptr);
```

Возвращает: 1 в случае успешного выполнения функции; 0, если входная строка имела неверный формат представления; -1 в случае ошибки

```
const char *inet_ntop(int family, const void *addrptr,
                      char *strptr, size_t len);
```

Возвращает: указатель на результат, если выполнение функции прошло успешно. `NULL` в случае ошибки

Значением аргумента `family` для обеих функций может быть либо `AF_INET`, либо `AF_INET6`. Если `family` не поддерживается, обе функции возвращают ошибку со значением переменной `errno`, равным `EAFNOSUPPORT`.

Первая функция пытается преобразовать строку, на которую указывает `strptr`, сохраняя двоичный результат с помощью указателя `addrptr`. При успешном выполнении ее возвращаемое значение равно 1. Если входная строка находится в неверном формате представления для заданного семейства (`family`), возвращается нуль.

Функция `inet_ntop` выполняет обратное преобразование: из численного формата (`addrptr`) в формат представления (`strptr`). Аргумент `len` — это размер принимающей строки, который передается, чтобы функция не переполнила буфер вызывающего процесса. Чтобы облегчить задание этого размера, в заголовочный файл <netinet/in.h> включаются следующие определения:

```
#define INET_ADDRSTRLEN 16 /* для точечно-десятичной записи IPv4-адреса */
#define INET6_ADDRSTRLEN 46 /* для шестнадцатеричной записи IPv6-адреса */
```

Если аргумент `len` слишком мал для хранения результирующего формата представления вместе с символом конца строки (`terminating null`), возвращается пустой указатель и переменной `errno` присваивается значение `ENOSPC`.

Аргумент `strptr` функции `inet_ntop` не может быть пустым указателем. Вызывающий процесс должен выделить память для хранения преобразованного значения и задать ее размер. При успешном выполнении функции возвращаемым значением является этот указатель.

На рис. 3.5 приведена схема действия пяти функций, описанных в этом и предыдущем разделах.

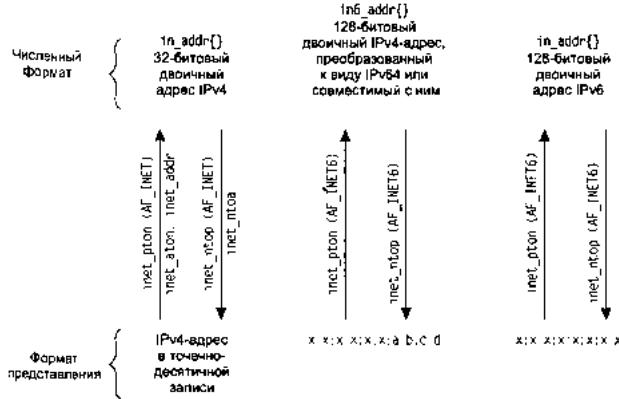


Рис. 3.5. Функции преобразования адресов

Пример

Даже если ваша система еще не поддерживает IPv6, вы можете использовать новые функции, заменив вызовы вида

```
foo.sin_addr.s_addr = inet_addr(cp);
на
inet_pton(AF_INET, cp, &foo.sin_addr);
а также заменив вызовы вида
ptr = inet_ntoa(foo.sin_addr);
на
char str[INET_ADDRSTRLEN];
ptr = inet_ntop(AF_INET, &foo.sin_addr, str, sizeof(str));
```

В листинге 3.6 представлено простое определение функции `inet_pton`, поддерживающей только IPv4, а в листинге 3.7 — версия `inet_ntop`, поддерживающая только IPv4.

Листинг 3.6. Простая версия функции `inet_pton`, поддерживающая только IPv4

```
//libfree/inet_pton_ipv4.c
10 int
11 inet_pton(int family, const char *strptr, void *addrptr)
12 {
13     if (family == AF_INET) {
14         struct in_addr in_val;
15
16         if (inet_aton(strptr, &in_val)) {
17             memcpy(addrptr, &in_val, sizeof(struct in_addr));
18             return (1);
19         }
20     }
21     errno = EAFNOSUPPORT;
22     return (-1);
23 }
```

Листинг 3.7. Простая версия функции `inet_ntop`, поддерживающая только IPv4

```
//libfree/inet_ntop_ipv4.c
8 const char *
9 inet_ntop(int family, const void *addrptr, char *strptr, size_t len)
10 {
11     const u_char *p = (const u_char*)addrptr;
12
13     if (family == AF_INET) {
14         char temp[INET_ADDRSTRLEN];
```

```

14  sprintf(temp, sizeof(temp), "%d.%d.%d.%d",
15  p[0], p[1], p[2], p[3]);
16  if (strlen(temp) >= len) {
17      errno = ENOSPC;
18      return (NULL);
19  }
20  strcpy(strptr, temp);
21  return (strptr);
22 }
23 errno = EAFNOSUPPORT;
24 return (NULL);
25 }
```

3.8. Функция `sock_ntop` и связанные с ней функции

Основная проблема, связанная с функцией `inet_ntop`, состоит в том, что вызывающий процесс должен передать ей указатель на двоичный адрес. Этот адрес обычно содержится в структуре адреса сокета, поэтому вызывающему процессу необходимо знать формат структуры и семейство адресов. Следовательно, чтобы использовать эту функцию, для IPv4 нужно написать код следующего вида:

```

struct sockaddr_in addr;
inet_ntop(AF_INET, &addr.sin_addr, str, sizeof(str));
или для IPv6 такого вида:
struct sockaddr_in6 addr6;
inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
```

Как видите, код становится зависящим от протокола.

Чтобы решить эту проблему, напишем собственную функцию и назовем ее `sock_ntop`. Она получает указатель на структуру адреса сокета, исследует эту структуру и вызывает соответствующую функцию для того, чтобы возвратить формат представления адреса.

```
#include "unp.h"
```

```
char *sock_ntop(const struct sockaddr *sockaddr, socklen_t addrlen);
```

Возращает: непустой указатель, если функция выполнена успешно, NULL в случае ошибки

`sockaddr` указывает на структуру адреса сокета, длина которой равна значению `addrlen`. Функция `sock_ntop` использует свой собственный статический буфер для хранения результата и возвращает указатель на этот буфер.

Формат представления — либо точечно-десятичная форма записи адреса IPv4, либо шестнадцатеричная форма записи адреса IPv6, за которой следует завершающий символ (мы используем точку, как в программе `netstat`), затем десятичный номер порта, а затем завершающий нуль. Следовательно, размер буфера должен быть равен как минимум `INET_ADDRSTRLEN` плюс 6 байт для IPv4 (16 + 6 - 22) либо `INET6_ADDRSTRLEN` плюс 6 байт для IPv6 (46 + 6 - 52).

ПРИМЕЧАНИЕ

Обратите внимание, что при статическом хранении результата функция не допускает повторного вхождения (не является повторно входимой) и не может быть использована несколькими программными потоками (не является безопасной в многопоточной среде — thread-safe). Более подробно мы поговорим об этом в разделе 11.18. Мы допустили такое решение для этой функции, чтобы ее было легче вызывать из простых программ, приведенных в книге.

В листинге 3.8 представлена часть исходного кода, обрабатывающая семейство `AF_INET`.

Листинг 3.8. Наша функция `sock_ntop`

```

//lib/sock_ntop.c
5 char *
6 sock_ntop(const struct sockaddr *sa, socklen_t salen)
```

```

7 {
8 char portstr[7];
9 static char str[128]; /* макс. длина для доменного сокета Unix */

10 switch (sa->sa_family) {
11 case AF_INET: {
12     struct sockaddr_in *sin = (struct sockaddr_in*)sa;

13     if (inet_ntop(AF_INET, &sin->sin_addr. str, sizeof(str)) == NULL)
14         return (NULL);
15     if ( ntohs(sin->sin_port) != 0) {
16         sprintf(portstr, sizeof(portstr), ntohs(sin->sin_port));
17         strncat(str, portstr);
18     }
19     return (str);
20 }

```

Для работы со структурами адресов сокетов мы определяем еще несколько функций, которые упростят переносимость нашего кода между IPv4 и IPv6.

```
#include "unp.h"
```

```
int sock_bind_wild(int sockfd, int family);
```

Возращает: 0 в случае успешного выполнения функции, -1 в случае ошибки

```
int sock_cmp_addr(const struct sockaddr *sockaddr1,
                  const struct sockaddr *sockaddr2, socklen_t addrlen);
```

Возращает: 0, если адреса относятся к одному семейству и совпадают, ненулевое значение в противном случае

```
int sock_cmp_port(const struct sockaddr *sockaddr1,
                  const struct sockaddr *sockaddr2, socklen_t addrlen);
```

Возращает: 0, если адреса относятся к одному семейству и порты совпадают, ненулевое значение в противном случае

```
int sock_get_port(const struct sockaddr *sockaddr, socklen_t addrlen);
```

Возращает: неотрицательный номер порта для адресов IPv4 или IPv6, иначе -1

```
char *sock_ntop_host(const struct sockaddr *sockaddr, socklen_t addrlen);
```

Возращает: непустой указатель в случае успешного выполнения функции, NULL в случае ошибки

```
void sock_set_addr(const struct sockaddr *sockaddr,
                   socklen_t addrlen, void *ptr);
```

```
void sock_set_port(const struct sockaddr *sockaddr,
                   socklen_t addrlen, int port);
```

```
void sock_set_wild(struct sockaddr *sockaddr, socklen_t addrlen);
```

Функция `sock_bind_wild` связывает универсальный адрес и динамически назначаемый порт с сокетом.

Функция `sock_cmp_addr` сравнивает адресные части двух структур адреса сокета, а функция `sock_cmp_port` сравнивает номера их портов. Функция `sock_get_port` возвращает только номер порта, а функция `sock_ntop_host` преобразует к формату представления только ту часть структуры адреса сокета, которая относится к узлу (все, кроме порта, то есть IP-адрес узла). Функция `sock_set_addr` присваивает адресной части структуры значение, указанное аргументом `ptr`, а функция `sock_set_port` задает в структуре адреса сокета только номер порта. Функция `sock_set_wild` задает адресную часть структуры через символы подстановки. Как обычно, мы предоставляем для всех этих функций функции-обертки, которые возвращают значение, отличное от типа `void`, и в наших программах обычно вызываем именно обертки. Мы не приводим в данной книге исходный код для этих функций, так как он свободно доступен (см. предисловие).

3.9. Функции readn, writen и readline

Потоковые сокеты (например, сокеты TCP) демонстрируют с функциями `read` и `write` поведение, отличное от обычного ввода-вывода файлов. Функция `read` или `write` на потоковом сокете может ввести или вывести немного меньше байтов, чем запрашивалось, но это не будет ошибкой. Причиной может быть достижение границы буфера для сокета в ядре. Все, что требуется в этой ситуации — чтобы процесс повторил вызов функции `read` или `write` для ввода или вывода оставшихся байтов. (Некоторые версии Unix ведут себя аналогично при записи в канал (pipe) более 4096 байт.) Этот сценарий всегда возможен на потоковом сокете при выполнении функции `read`, но с функцией `write` он обычно наблюдается, только если сокет неблокируемый. Тем не менее вместо `write` мы всегда вызываем функцию `writen` на тот случай, если в данной реализации возможно возвращение меньшего количества данных, чем мы запрашиваем.

Введем три функции для чтения и записи в потоковый сокет.

```
#include "unp.h"
```

```
ssize_t readn(int filedes, void *buff, size_t nbytes);
ssize_t writen(int filedes, const void *buff, size_t nbytes);
ssize_t readline(int filedes, void *buff, size_t maxlen);
```

Все функции возвращают: количество считанных или записанных байтов, -1 в случае ошибки

В листинге 3.9 представлена функция `readn`, в листинге 3.10 — функция `writen`, а в листинге 3.11 — функция `readline`.

Листинг 3.9. Функция `readn`: считывание n байт из дескриптора

```
//lib/readn.c
```

```
1 #include "unp.h"
```

```
2 ssize_t /* Считывает n байт из дескриптора */
3 readn(int fd, void *vptr, size_t n)
4 {
5     size_t nleft;
6     ssize_t nread;
7     char *ptr;
8
8     ptr = vptr;
9     nleft = n;
10    while (nleft > 0) {
11        if ((nread = read(fd, ptr, nleft)) < 0) {
12            if (errno == EINTR)
13                nread = 0; /* и вызывает снова функцию read() */
14            else
15                return (-1);
16        } else if (nread == 0)
17            break; /* EOF */
18
18    nleft -= nread;
19    ptr += nread;
20 }
21    return (n - nleft); /* возвращает значение >= 0 */
22 }
```

Листинг 3.10. Функция `writen`: запись n байт в дескриптор

```
//lib/writen.c
```

```
1 #include "unp.h"
```

```
2 ssize_t /* Записывает n байт в дескриптор */
3 writen(int fd, const void *vptr, size_t n)
4 {
5     size_t nleft;
6     ssize_t nwritten;
```

```

7 const char *ptr;

8 ptr = vptr;
9 nleft = n;
10 while (nleft > 0) {
11     if ((nwritten = write(fd, ptr, nleft)) <= 0) {
12         if (errno == EINTR)
13             nwritten = 0; /* и снова вызывает функцию write() */
14         else
15             return (-1); /* ошибка */
16     }
17     nleft -= nwritten;
18     ptr += nwritten;
19 }
20 return (n);
21 }
```

Листинг 3.11. Функция readline: считывание следующей строки из дескриптора, по одному байту за один раз

```

//test/readline1.c
1 #include "unp.h"
/* Ужасно медленная версия, приводится только для примера */

2 ssize_t
3 readline(int fd, void *vptr, size_t maxlen)
4 {
5     ssize_t n, rc;
6     char c, *ptr;

7     ptr = vptr;
8     for (n = 1; n < maxlen; n++) {
9         again:
10        if ((rc = read(fd, &c, 1)) == 1) {
11            *ptr++ = c;
12            if (c == '\n')
13                break; /* записан символ новой строки, как в fgets() */
14        } else if (rc == 0) {
15            if (n == 1)
16                return (0); /* EOF, данные не считаны */
17            else
18                break; /* EOF, некоторые данные были считаны */
19        } else {
20            if (errno == EINTR)
21                goto again;
22            return (-1); /* ошибка, errno задается функцией read() */
23        }
24    }

25    *ptr = 0; /* завершаем нулем, как в fgets() */
26    return (n);
27 }
```

Если функция чтения или записи (read или write) возвращает ошибку, то наши функции проверяют, не совпадает ли код ошибки с EINTR (прерывание системного вызова сигналом, см. раздел 5.9). В этом случае прерванная функция вызывается повторно. Мы обрабатываем ошибку в этой функции, чтобы не заставлять процесс снова вызвать read или write, поскольку целью наших функций является предотвращение обработки нехватки данных вызывающим процессом.

В разделе 14.3 мы покажем, что вызов функции `recv` с флагом `MSG_WAITALL` позволяет обойтись без использования отдельной функции `readn`.

Заметим, что наша функция `readline` вызывает системную функцию `read` один раз для каждого байта данных. Это очень неэффективно, поэтому мы и написали в примечании «Ужасно медленно!». Возникает соблазн обратиться к стандартной библиотеке ввода-вывода (`stdio`). Об этом мы поговорим через некоторое время в разделе 14.8, но учтите, что это может привести к определенным проблемам. Буферизация, предоставляемая `stdio`, решает проблемы с производительностью, но при этом создает множество логистических сложностей, которые в свою очередь порождают скрытые ошибки в приложении. Дело в том, что состояние буферов `stdio` недоступно процессу. Рассмотрим, например, строчный протокол взаимодействия клиента и сервера, причем такой, что могут существовать разные независимые реализации клиентов и серверов (достаточно типичное явление; например, множество веб-браузеров и веб-серверов были разработаны независимо в соответствии со спецификацией HTTP). Хороший стиль программирования заключается в том, что эти программы должны не только ожидать от своих собеседников соблюдения того же протокола, но и контролировать трафик на возможность получения непредвиденного трафика. Подобные нарушения протокола должны рассматриваться как ошибки, чтобы программисты имели возможность находить и устранять неполадки в коде, а также обнаруживать попытки взлома систем. Обработка некорректного трафика должна давать приложению возможность продолжать работу. Буферизация `stdio` мешает достижению перечисленных целей, поскольку приложение не может проверить наличие непредвиденных (некорректных) данных в буферах `stdio` в любой конкретный момент.

Существует множество сетевых протоколов, основанных на использовании строк текста: SMTP, HTTP, FTP, finger. Поэтому соблазн работать со строками будет терзать вас достаточно часто. Наш совет: мыслить в терминах буферов, а не строк. Пишите код таким образом, чтобы считывать содержимое буфера, а не отдельные строки. Если же ожидается получение строки, ее всегда можно поискать в считанном буфере.

В листинге 3.12 приведена более быстрая версия функции `readline`, использующая свой собственный буфер (а не буферизацию `stdio`). Основное достоинство этого буфера состоит в его открытости, благодаря чему вызывающий процесс всегда знает, какие именно данные уже принятые. Несмотря на это, использование `readline` все равно может вызвать проблемы, как мы увидим в разделе 6.3. Системные функции типа `select` ничего не знают о внутреннем буфере `readline`, поэтому неаккуратно написанная программа с легкостью может очутиться в состоянии ожидания в вызове `select`, при том, что данные уже будут находиться в буферах `readline`. По этой причине сочетание вызовов `readn` и `readline` не будет работать так, как этого хотелось бы, пока функция `readn` не будет модифицирована с учетом наличия внутреннего буфера.

Листинг 3.12. Улучшенная версия функции `readline`

```
//lib/readline.c
1 #include "unp.h"

2 static int read_cnt;
3 static char *read_ptr;
4 static char read_buf[MAXLINE];

5 static ssize_t
6 my_read(int fd, char *ptr)
7 {

8     if (read_cnt <= 0) {
9         again:
10        if ((read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
11            if (errno == EINTR)
12                goto again;
13            return(-1);
14        } else if (read_cnt == 0)
15            return(0);
16        read_ptr = read_buf;
17    }
18    read_cnt--;
```

```

19 *ptr = *read_ptr++;
20 return(1);
21 }

22 ssize_t
23 readline(int fd, void *vptr, size_t maxlen)
24 {
25     ssize_t n, rc;
26     char c, *ptr;

27     ptr = vptr;
28     for (n = 1; n < maxlen; n++) {
29         if ((rc = my_read(fd, &c)) == 1) {
30             *ptr++ = c;
31             if (c == '\n')
32                 break; /* Записан символ новой строки, как в fgets() */
33         } else if (rc == 0) {
34             *ptr = 0;
35             return(n - 1); /* EOF, считано n-1 байт данных */
36         } else
37             return(-1); /* ошибка, read() задает значение errno */
38     }

39     *ptr = 0; /* завершающий нуль, как в fgets() */
40     return(n);
41 }

42 ssize_t
43 readlinebuf(void **vptrptr)
44 {
45     if (read_cnt)
46         *vptrptr = read_ptr;
47     return(read_cnt);
48 }

```

2-21 Внутренняя функция `my_read` считывает до `MAXLINE` символов за один вызов и затем возвращает их по одному.

29 Единственное изменение самой функции `readline` заключается в том, что теперь она вызывает функцию `my_read` вместо `read`.

42-48 Новая функция `readlinebuf` выдает сведения о состоянии внутреннего буфера, что позволяет вызывающим функциям проверить, нет ли в нем других данных, помимо уже принятой строки.

ПРИМЕЧАНИЕ

К сожалению, использование переменных типа `static` в коде `readline.c` для поддержки информации о состоянии при последовательных вызовах приводит к тому, что функция больше не является безопасной в многопоточной системе (`thread-safe`) и повторно входимой (`reentrant`). Мы обсуждаем это в разделах 11.18 и 26.5. Мы предлагаем версию, безопасную в многопоточной системе, основанную на собственных данных программных потоков, в листинге 26.5.

3.10. Резюме

Структуры адресов сокетов являются неотъемлемой частью каждой сетевой программы. Мы выделяем для них место в памяти, заполняем их и передаем указатели на них различным функциям сокетов. Иногда мы передаем указатель на одну из этих структур функции сокета, и она сама заполняет поля структуры. Мы всегда передаем эти структуры по ссылке (то есть передаем указатель на структуру, а не саму структуру) и

всегда передаем размер структуры в качестве дополнительного аргумента. Когда функция сокета заполняет структуру, длина также передается по ссылке, и ее значение может быть изменено функцией, поэтому мы называем такой аргумент «значение-результат» (value-result).

Структуры адресов сокетов являются самоопределяющимися, поскольку они всегда начинаются с поля `family`, которое идентифицирует семейство адресов, содержащихся в структуре. Более новые реализации, поддерживающие структуры адресов сокетов переменной длины, также содержат поле, которое определяет длину всей структуры.

Две функции, преобразующие IP-адрес из формата представления (который мы записываем в виде последовательности символов ASCII) в численный формат (который входит в структуру адреса сокета) и обратно, называются `inet_pton` и `inet_ntop`. Эти функции являются зависящими от протокола. Более совершенной методикой является работа со структурами адресов сокетов как с непрозрачными (opaque) объектами, когда известны лишь указатель на структуру и ее размер. Мы разработали набор функций `sock_`, которые помогут сделать наши программы не зависящими от протокола. Создание наших не зависящих от протокола средств мы завершим в главе 11 функциями `getaddrinfo` и `getnameinfo`.

Сокеты TCP предоставляют приложению поток байтов, лишенный маркеров записей. Возвращаемое значение функции `read` может быть меньше запрашиваемого, но это не обязательно является ошибкой. Чтобы упростить считывание и запись потока байтов, мы разработали три функции `readn`, `writen` и `readline`, которые и используем в книге. Однако сетевые программы должны быть написаны в расчете на работу с буферами, а не со строками.

Упражнения

1. Почему аргументы типа «значение-результат», такие как длина структуры адреса сокета, должны передаваться по ссылке?

2. Почему и функция `readn`, и функция `writen` копируют указатель `void*` в указатель `char*`?

3. Функции `inet_aton` и `inet_addr` характеризуются традиционно нестрогим отношением к тому, что они принимают в качестве точечно-десятичной записи адреса IPv4: допускаются от одного до четырех десятичных чисел, разделенных точками; также допускается задавать шестнадцатеричное число с помощью начального `0x` или восьмеричное число с помощью начального `0` (выполните команду `telnet 0xe`, чтобы увидеть поведение этих функций). Функция `inet_pton` намного более строга в отношении адреса IPv4 и требует наличия именно четырех чисел, разделенных точками, каждое из которых является десятичным числом от 0 до 255. Функция `inet_pton` не разрешает задавать точечно-десятичный формат записи адреса, если семейство адресов — `AF_INET6`, хотя существует мнение, что это можно было бы разрешить, и тогда возвращаемое значение было бы адресом IPv4, преобразованным к виду IPv6 (см. рис. A.6). Напишите новую функцию `inet_pton_loose`, реализующую такой сценарий: если используется семейство адресов `AF_INET` и функция `inet_pton` возвращает нуль, вызовите функцию `inet_aton` и посмотрите, успешно ли она выполнится. Аналогично, если используется семейство адресов `AF_INET6` и функция `inet_pton` возвращает нуль, вызовите функцию `inet_aton`, и если она выполнится успешно, возвратите адрес IPv4, преобразованный к виду IPv6.

Глава 4

Элементарные сокеты TCP

4.1. Введение

В этой главе описываются элементарные функции сокетов, необходимые для написания полностью работоспособного клиента и сервера TCP. Сначала мы опишем все элементарные функции сокетов, которые будем использовать, а затем в следующей главе создадим клиент и сервер. С этими приложениями мы будем работать на протяжении всей книги, постоянно их совершенствуя (см. табл. 1.3 и 1.4).

Мы также опишем параллельные (concurrent) серверы — типичную технологию Unix для обеспечения параллельной обработки множества клиентов одним сервером. Подключение очередного клиента заставляет сервер выполнить функцию `fork`, порождающую новый серверный процесс для обслуживания этого клиента. Здесь применительно к использованию функции `fork` мы будем рассматривать модель «каждому клиенту — один процесс», а в главе 26 при обсуждении программных потоков расскажем о модели «каждому клиенту — один поток».

На рис. 4.1 представлен типичный сценарий взаимодействия, происходящего между клиентом и сервером. Сначала запускается сервер, затем, спустя некоторое время, запускается клиент, который соединяется с сервером. Предполагается, что клиент посыпает серверу запрос, сервер этот запрос обрабатывает и посыпает клиенту ответ. Так продолжается, пока клиентская сторона не закроет соединение, посыпая при этом серверу признак конца файла. Затем сервер закрывает свой конец соединения и либо завершает работу, либо ждет подключения нового клиента.

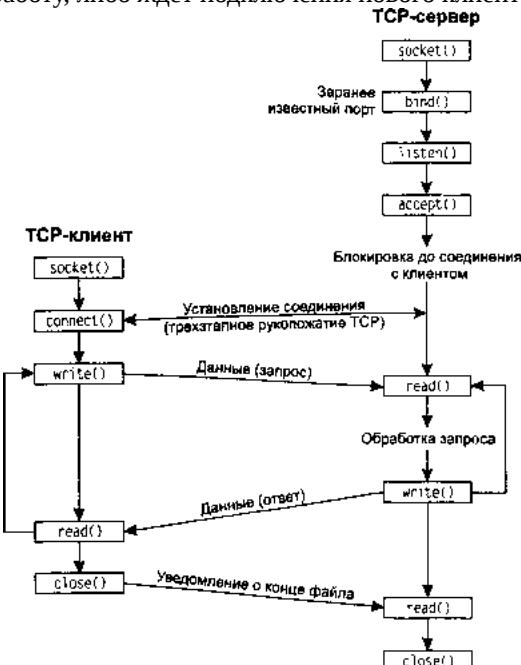


Рис. 4.1. Функции сокетов для элементарного клиент-серверного соединения TCP

4.2. Функция `socket`

Чтобы обеспечить сетевой ввод-вывод, процесс должен начать работу с вызова функции `socket`, задав тип желаемого протокола (TCP с использованием IPv4, UDP с использованием IPv6, доменный сокет Unix и т.д.).

```
#include <sys/socket.h>
```

```
int socket(int family, int type, int protocol);
```

Возвращает: неотрицательный дескриптор, если функция выполнена успешно, -1 в случае ошибки

Константа `family` задает семейство протоколов. Ее возможные значения приведены в табл. 4.1. Часто этот параметр функции `socket` называют «областью» или «доменом» (*domain*), а не семейством. Значения константы `type` (тип) перечислены в табл. 4.2. Аргумент `protocol` должен быть установлен в соответствии с используемым протоколом (табл. 4.3) или должен быть равен нулю для выбора протокола, по умолчанию соответствующего заданному семейству и типу.

Таблица 4.1. Константы протокола (`family`) для функции `socket`

Семейство сокетов (family) Описание

AF_INET	Протоколы IPv4
AF_INET6	Протоколы IPv6
AF_LOCAL	Протоколы доменных сокетов Unix (см. главу 14)
AF_ROUTE	Маршрутизирующие сокеты (см. главу 17)
AF_KEY	Сокет управления ключами

Таблица 4.2. Тип сокета для функции `socket`

Тип (type)	Описание
SOCK_STREAM	Потоковый сокет
SOCK_DGRAM	Сокет дейтаграмм
SOCK_SEQPACKET	Сокет последовательных пакетов
SOCK_RAW	Символьный (неструктурированный) сокет

Таблица 4.3. Возможные значения параметра `protocol`

Protocol	Значение
IPPROTO_TCP	Транспортный протокол TCP
IPPROTO_UDP	Транспортный протокол UDP
IPPROTO_SCTP	Транспортный протокол SCTP

Не все сочетания констант `family` и `type` допустимы. В табл. 4.4 показаны допустимые сочетания, а также протокол, соответствующий каждой паре. Клетки таблицы, содержащие «Да», соответствуют допустимым комбинациям, для которых нет удобных сокращений. Пустая клетка означает, что данное сочетание не поддерживается.

Таблица 4.4. Сочетания констант `family` и `type` для функции `socket`

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP/SCTP	TCP/SCTP	Да		
SOCK_DGRAM	UDP	UDP	Да		
SOCK_SEQPACKET	SCTP	SCTP	Да		
SOCK_RAW	IPv4	IPv6		Да	Да

ПРИМЕЧАНИЕ

В качестве первого аргумента функции `socket` вы также можете встретить константу `PF_XXX`. Подробнее об этом мы расскажем в конце данного раздела.

Кроме того, вам может встретиться название `AF_UNIX` (исторически сложившееся в Unix) вместо `AF_LOCAL` (название из POSIX), и более подробно мы поговорим об этом в главе 14.

Для аргументов `family` и `type` существуют и другие значения. Например, 4.4BSD поддерживает и `AF_NS` (протоколы Xerox NS, часто называемые XNS), и `AF_ISO` (протоколы OSI). Но сегодня очень немногие используют какой-либо из этих протоколов. Аналогично, значение `type` для `SOCK_SEQPACKET`, сокета последовательных пакетов, реализуется и протоколами Xerox NS, и протоколами OSI. Но протокол TCP является потоковым и поддерживает только сокеты `SOCK_STREAM`.

Linux поддерживает новый тип сокетов, SOCK_PACKET, предоставляющий доступ к канальному уровню, аналогично BPF и DLPI на рис. 2.1. Об этом более подробно рассказывается в главе 29.

Сокет управления ключами AF_KEY является новшеством. Аналогично тому, как маршрутизирующий сокет (AF_ROUTE) является интерфейсом к таблице маршрутизации ядра, сокет управления ключами — это интерфейс к таблице ключей ядра. Подробнее об этом рассказывается в главе 19.

При успешном выполнении функция `socket` возвращает неотрицательное целое число, аналогичное дескриптору файла. Мы называем это число *дескриптором сокета* (*socket descriptor*), или `sockfd`. Чтобы получить дескриптор сокета, достаточно указать лишь семейство протоколов (IPv4, IPv6 или Unix) и тип сокета (потоковый, символьный или дейтаграммный). Мы еще не задали ни локальный адрес протокола, ни удаленный адрес протокола.

AF_XXX и PF_XXX

Префикс `AF_` обозначает *семейство адресов* (*address family*), а `PF_` — *семейство протоколов* (*protocol family*). Исторически ставилась такая цель, чтобы отдельно взятое семейство протоколов могло поддерживать множество семейств адресов и значение `PF_` использовалось для создания сокета, а значение `AF_` — в структурах адресов сокетов. Но в действительности семейства протоколов, поддерживающего множество семейств адресов, никогда не существовало, и поэтому в заголовочном файле `<sys/socket.h>` значение `PF_` для протокола задается равным значению `AF_`. Хотя не гарантируется, что это равенство будет всегда справедливо, но при попытке изменить ситуацию для существующих протоколов большая часть написанного кода потеряет работоспособность.

ПРИМЕЧАНИЕ

Просмотр 137 программ с вызовами функции `socket` в реализации BSD/OS 2.1 показывает, что в 143 случаях вызова задается значение `AF_`, и только в 8 случаях — значение `PF_`.

Причина создания аналогичных наборов констант с префиксами `AF_` и `PF_` восходит к 4.1cBSD [69] и к версии функции `socket`, предшествующей описываемой нами версии (которая появилась с 4.2BSD). Версия функции `socket` в 4.1cBSD получала четыре аргумента, одним из которых был указатель на структуру `sockproto`. Первый элемент этой структуры назывался `sp_family`, и его значение было одним из значений `PF_`. Второй элемент, `sp_protocol`, был номером протокола, аналогично третьему аргументу нынешней функции `socket`. Единственный способ задать семейство протоколов заключался в том, чтобы задать эту структуру. Следовательно, в этой системе значения `PF_` использовались как элементы для задания семейства протоколов в структуре `sockproto`. Значения `AF_` играли роль элементов для задания семейства адресов в структурах адресов сокетов. Структура `sockproto` еще присутствует в 4.4BSD [128, с. 626-627], но служит только для внутреннего использования ядром. Начальное определение содержало для элемента `sp_family` комментарий «семейство протоколов», но в исходном коде 4.4BSD он был изменен на «семейство адресов».

Еще большую путаницу в эту ситуацию вносит то, что в Беркли-реализации структура данных ядра, содержащая значение, которое сравнивается с первым аргументом функции `socket` (элемент `dom_family` структуры `domain` [128, с. 187]), сопровождается комментарием, где сказано, что в этой структуре содержится значение `AF_`. Но некоторые структуры `domain` внутри ядра инициализированы с помощью константы `AF_` [128, с. 192], в то время как другие — с помощью `PF_` [128, с. 646], [112, с. 229].

Еще одно историческое замечание. Страница руководства по 4.2BSD от июля 1983 года, посвященная функции `socket`, называет ее первый аргумент `af` и перечисляет его возможные значения как константы `AF_`.

Наконец, отметим, что POSIX задает первый аргумент функции `socket` как значение `PF_`, а значение `AF_` использует для структуры адреса сокета. Но далее в структуре `addrinfo` определяется только одно значение семейства (см. раздел 11.2), предназначенное для использования либо в вызове функции `socket`, либо в структуре адреса сокета!

В целях согласования с существующей практикой программирования мы используем в тексте только константы AF_-, хотя вы можете встретить и значение PF_-, в основном в вызовах функции socket.

4.3. Функция connect

Функция connect используется клиентом TCP для установления соединения с сервером TCP.

```
#include <sys/socket.h>
```

```
int connect(int sockfd, const struct sockaddr *servaddr,  
           socklen_t addrlen);
```

Возвращает: 0 в случае успешного выполнения функции, -1 в случае ошибки

Аргумент sockfd — это дескриптор сокета, возвращенный функцией socket. Второй и третий аргументы — это указатель на структуру адреса сокета и ее размер (см. раздел 3.3). Структура адреса сокета должна содержать IP-адрес и номер порта сервера. Пример применения этой функции был представлен в листинге 1.1.

Клиенту нет необходимости вызывать функцию bind (которую мы описываем в следующем разделе) до вызова функции connect: при необходимости ядро само выберет и динамически назначаемый порт, и IP-адрес отправителя.

В случае сокета TCP функция connect инициирует трехэтапное рукопожатие TCP (см. раздел 2.6). Функция возвращает значение, только если установлено соединение или произошла ошибка. Возможно несколько ошибок:

1. Если клиент TCP не получает ответа на свой сегмент SYN, возвращается сообщение ETIMEDOUT. 4.4BSD, например, отправляет один сегмент SYN, когда вызывается функция connect, второй — 6 с спустя, и еще один — через 24 с [128, с. 828]. Если ответ не получен в течение 75 с, возвращается ошибка.

Некоторые системы позволяют администратору устанавливать значение времени ожидания; см. приложение Е [111].

2. Если на сегмент SYN сервер отвечает сегментом RST, это означает, что ни один процесс на узле сервера не находится в ожидании подключения к указанному нами порту (например, нужный процесс может быть не запущен). Это *устойчивая неисправность (hard error)*, и клиенту возвращается сообщение ECONNREFUSED сразу же по получении им сегмента RST.

RST (от «reset» — сброс) — это сегмент TCP, отправляемый собеседнику при возникновении ошибок. Вот три условия, при которых генерируется RST: сегмент SYN приходит для порта, не имеющего прослушивающего сервера (что мы только что описали); TCP хочет разорвать существующее соединение; TCP получает сегмент для несуществующего соединения (дополнительная информация содержится на с. 246–250 [111]).

3. Если сегмент SYN клиента приводит к получению сообщения ICMP о недоступности получателя от какого-либо промежуточного маршрутизатора, это считается *случайным сбоем (soft error)*. Клиентское ядро сохраняет сообщение об ошибке, но продолжает отправлять сегменты SYN с теми же временными интервалами, что и в первом сценарии. Если же по истечении определенного фиксированного времени (75 с для 4.4BSD) ответ не получен, сохраненная ошибка ICMP возвращается процессу либо как ENETUNREACH, либо как EHOSTUNREACH. Может случиться, что удаленная система будет недоступна по любому маршруту из таблицы маршрутизации локального узла, или что возврат из connect произойдет без всякого ожидания.

ПРИМЕЧАНИЕ

Многие более ранние системы, такие как 4.2BSD, некорректно прерывали попытки установления соединения при получении сообщения ICMP о недоступности получателя. Это было неверно, поскольку данная ошибка ICMP может указывать на временную неисправность. Например, может быть так, что эта ошибка вызвана проблемой маршрутизации, которая исправляется в течение 15 с.

Обратите внимание, что мы не включили ENETUNREACH в табл. А.5 несмотря на то, что сеть получателя действительно может быть недоступна. Недоступность сети считается устаревшей ошибкой, и даже если 4.4BSD получает такое сообщение, приложению возвращается EHOSTUNREACH.

Эти ошибки мы можем наблюдать на примере нашего простого клиента, созданного в листинге 1.1. Сначала мы указываем адрес нашего собственного узла (127.0.0.1), на котором работает сервер времени и даты, и видим обычный вывод:

```
solaris % daytimecpcli 127.0.0.1  
Sun Jul 27 22:01:51 2003
```

Укажем IP-адрес другого компьютера (HP-UX):

```
solaris % daytimecpcli 192.6.38.100  
Sun Jul 27 22:04:59 PDT 2003
```

Затем мы задаем IP-адрес в локальной подсети (192.168.1/24) с несуществующим адресом узла (100).

Когда клиент посыпает запросы ARP (запрашивая аппаратный адрес узла), он не получает никакого ответа:

```
solaris % daytimecpcli 192.168.1.100  
connect error: Connection timed out
```

Мы получаем сообщение об ошибке только по истечении времени выполнения функции connect (которое, как мы говорили, для Solaris 9 составляет 3 мин). Обратите внимание, что наша функция err_sys выдает текстовое сообщение, соответствующее коду ошибки ETIMEDOUT.

В следующем примере мы пытаемся обратиться к локальному маршрутизатору, на котором не запущен сервер времени и даты:

```
solaris % daytimecpcli 192.168.1.5  
connect error: Connection refused
```

Сервер отвечает немедленно, отправляя сегмент RST.

В последнем примере мы пытаемся обратиться к недоступному адресу из сети Интернет. Просмотрев пакеты с помощью программы tcpdump, мы увидим, что маршрутизатор, находящийся на расстоянии шести прыжков от нас, возвращает сообщение ICMP о недоступности узла:

```
solaris % daytimecpcli 192.3.4.5  
connect error: No route to host
```

Как и в случае ошибки ETIMEDOUT, в этом примере функция connect возвращает ошибку EHOSTUNREACH только после ожидания в течение определенного времени.

В терминах диаграммы перехода состояний TCP (см. рис. 2.4) функция connect переходит из состояния CLOSED (состояния, в котором сокет начинает работать при создании с помощью функции socket) в состояние SYN_SENT, а затем, при успешном выполнении, в состояние ESTABLISHED. Если выполнение функции connect окажется неудачным, сокет больше не используется и должен быть закрыт. Мы не можем снова вызвать функцию connect для сокета. В листинге 11.4 вы увидите, что если функция connect выполняется в цикле, проверяя каждый IP-адрес данного узла, пока он не заработает, то каждый раз, когда выполнение функции оказывается неудачным, мы должны закрыть дескриптор сокета с помощью функции close и снова вызвать функцию socket.

4.4. Функция bind

Функция bind связывает сокет с локальным адресом протокола. В случае протоколов Интернета адрес протокола — это комбинация 32-разрядного адреса IPv4 или 128-разрядного адреса IPv6 с 16-разрядным номером порта TCP или UDP.

```
#include <sys/socket.h>
```

```
int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);  
Возвращает: 0 в случае успешного выполнения, -1 в случае ошибки
```

ПРИМЕЧАНИЕ

В руководстве при описании функции bind говорилось: «функция bind присваивает имя неименованному сокету». Использование термина «имя» спорно, обычно оно вызывает ассоциацию с доменными именами (см. главу 11), такими как foo.bar.com. Функция bind не имеет ничего общего с именами. Она задает сокету адрес протокола, а что означает этот адрес — зависит от самого протокола.

Вторым аргументом является указатель на специфичный для протокола адрес, а третий аргумент — это размер структуры адреса. В случае TCP вызов функции `bind` позволяет нам задать номер порта или IP-адрес, а также задать оба эти параметра или вообще не указывать ничего.

■ Серверы связываются со своим заранее известным портом при запуске. Мы видели это в листинге 1.5. Если клиент или сервер TCP не делает этого, ядро выбирает динамически назначаемый порт для сокета либо при вызове функции `connect`, либо при вызове функции `listen`. Клиент TCP обычно позволяет ядру выбирать динамически назначаемый порт, если приложение не требует зарезервированного порта (см. рис. 2.10), но сервер TCP достаточно редко предоставляет ядру право выбора, так как обращение к серверам производится через заранее известные порты.

ПРИМЕЧАНИЕ

Исключением из этого правила являются серверы удаленного вызова процедур RPC (Remote Procedure Call). Обычно они позволяют ядру выбирать динамически назначаемый порт для их прослушиваемого сокета, поскольку затем этот порт регистрируется программой отображения портов RPC. Клиенты должны соединиться с этой программой, чтобы получить номер динамически назначаемого порта до того, как они смогут соединиться с сервером с помощью функции `connect`. Это также относится к серверам RPC, использующим протокол UDP.

■ С помощью функции `bind` процесс может связать конкретный IP-адрес с сокетом. IP-адрес должен соответствовать одному из интерфейсов узла. Так определяется IP-адрес, который будет использоваться для отправляемых через сокет IP-дейтаграмм. При этом для сервера TCP на сокет накладывается ограничение: он может принимать только такие входящие соединения клиента, которые предназначены именно для этого IP-адреса.

Обычно клиент TCP не связывает IP-адрес с сокетом при помощи функции `bind`. Ядро выбирает IP-адрес отправителя в момент подключения клиента к сокету, основываясь на используемом исходяющим интерфейсе, который, в свою очередь, зависит от маршрута, требуемого для обращения к серверу [128, с. 737].

Если сервер TCP не связывает IP-адрес с сокетом, ядро назначает ему IP-адрес (указываемый в исходящих пакетах), который совпадает с адресом получателя сегмента SYN клиента [128, с. 943].

Как мы уже говорили, вызов функции `bind` позволяет нам задать IP-адрес и порт (вместе или по отдельности) либо не задавать никаких аргументов. В табл. 4.5 приведены все возможные значения, которые присваиваются аргументам `sin_addr` и `sin_port` либо `sin6_addr` и `sin6_port` в зависимости от желаемого результата.

Таблица 4.5. Результаты задания IP-адреса и (или) номера порта в функции `bind`

Процесс задает		Результат
IP-адрес	Порт	
Универсальный	0	Ядро выбирает IP-адрес и порт
Универсальный	Ненулевое значение	Ядро выбирает IP-адрес, процесс задает порт
Локальный	0	Процесс задает IP-адрес, ядро выбирает порт
Локальный	Ненулевое значение	Процесс задает IP-адрес и порт

Если мы зададим нулевой номер порта, то при вызове функции `bind` ядро выберет динамически назначаемый порт. Но если мы зададим IP-адрес с помощью символов подстановки (`wildcard`), задается константой `INADDR_ANY`, значение которой обычно нулевое. Это указывает ядру на необходимость выбора IP-адреса. Пример вы видели в листинге 1.5:

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* универсальный */
```

Этот прием работает с IPv4, где IP-адрес является 32-разрядным значением, которое можно представить как простую численную константу (в данном случае 0), но воспользоваться им при работе с IPv6 мы не можем, поскольку 128-разрядный адрес IPv6 хранится в структуре. (В языке C мы не можем

поместить структуру в правой части оператора присваивания.) Эта проблема решается следующим образом:

```
struct sockaddr_in6 serv;
serv.sin6_addr = in6addr_any; /* универсальный */
```

Система выделяет место в памяти и инициализирует переменную `in6addr_any`, присваивая ей значение константы `IN6ADDR_ANY_INIT`. Объявление внешней константы `in6addr_any` содержится в заголовочном файле `<netinet/in.h>`.

Значение `INADDR_ANY` (0) не зависит от порядка байтов, поэтому использование функции `htonl` в действительности не требуется. Но поскольку все константы `INADDR_`, определенные в заголовочном файле `<netinet/in.h>`, задаются в порядке байтов узла, с любой из этих констант следует использовать функцию `htonl`.

Если мы поручаем ядру выбрать для нашего сокета номер динамически назначаемого порта, то функция `bind` не возвращает выбранное значение. В самом деле, она не может возвратить это значение, поскольку второй аргумент функции `bind` имеет спецификатор `const`. Чтобы получить значение динамически назначаемого порта, заданного ядром, потребуется вызвать функцию `getsockname`, которая возвращает локальный адрес протокола.

Типичным примером процесса, связывающего с сокетом конкретный IP-адрес, служит узел, на котором работают веб-серверы нескольких организаций (см. раздел 14.2 [112]). Прежде всего, у каждой организации есть свое собственное доменное имя, например `www.organization.com`. Доменному имени каждой организации сопоставляется некоторый IP-адрес; различным организациям сопоставляются различные адреса, но обычно из одной и той же подсети. Например, если маска подсети 198.69.10, то IP-адресом первой организации может быть 198.69.10.128, следующей — 198.69.10.129, и т.д. Все эти IP-адреса затем становятся псевдонимами, или альтернативными именами (*alias*), одного сетевого интерфейса (например, при использовании параметра `alias` команды `ifconfig` в 4.4BSD). В результате уровень IP будет принимать входящие дейтаграммы, предназначенные для любого из адресов, являющихся псевдонимами. Наконец, для каждой организации запускается по одной копии сервера HTTP, и каждая копия связывается с помощью функции `bind` только с IP-адресом определенной организации.

ПРИМЕЧАНИЕ

В качестве альтернативы можно запустить одиночный сервер, связанный с универсальным адресом. Когда происходит соединение, сервер вызывает функцию `getsockname`, чтобы получить от клиента IP-адрес получателя, который (см. наше обсуждение ранее) может быть равен 198.69.10.128, 198.69.10.129 и т.д. Затем сервер обрабатывает запрос клиента на основе именно этого IP-адреса, к которому было направлено это соединение.

Одним из преимуществ связывания с конкретным IP-адресом является то, что демультиплексирование данного IP-адреса с процессом сервера выполняется ядром.

Следует внимательно относиться к различию интерфейса, на который приходит пакет, и IP-адреса получателя этого пакета. В разделе 8.8 мы поговорим о моделях систем с гибкой привязкой (*weak end system*) и с жесткой привязкой (*strong end system*). Большинство реализаций используют первую модель, то есть считают обычным явлением принятие пакета на интерфейсе, отличном от указанного в IP-адресе получателя. (При этом подразумевается узел с несколькими сетевыми интерфейсами.) При связывании с сокетом конкретного IP-адреса на этом сокете будут приниматься дейтаграммы с заданным IP-адресом получателя, и только они. Никаких ограничений на принимающий интерфейс не накладывается — эти ограничения возникают только в случае, если используется модель системы с жесткой привязкой.

Общей ошибкой выполнения функции `bind` является `EADDRINUSE`, указывающая на то, что адрес уже используется. Более подробно мы поговорим об этом в разделе 7.5, когда будем рассматривать параметры сокетов `SO_REUSEADDR` и `SO_REUSEPORT`.

4.5. Функция `listen`

Функция `listen` вызывается только сервером TCP и выполняет два действия.

1. Когда сокет создается с помощью функции `socket`, считается, что это активный сокет, то есть клиентский сокет, который запустит функцию `connect`. Функция `listen` преобразует неприсоединенный сокет в пассивный сокет, запросы на подключение к которому начинают приниматься ядром. В терминах диаграммы перехода между состояниями TCP (см. рис. 2.4) вызов функции `listen` переводит сокет из состояния `CLOSED` в состояние `LISTEN`.

2. Второй аргумент этой функции задает максимальное число соединений, которые ядро может помещать в очередь этого сокета.

```
#include <sys/socket.h>
```

```
int listen(int sockfd, int backlog);
```

Возвращает: 0 в случае успешного выполнения, -1 в случае ошибки

Эта функция обычно вызывается после функций `socket` и `bind`. Она должна вызываться перед вызовом функции `accept`.

Чтобы уяснить смысл аргумента `backlog`, необходимо понять, что для данного прослушиваемого сокета ядро поддерживает две очереди:

1. *Очередь не полностью установленных соединений (incomplete connection queue)*, содержащую запись для каждого сегмента SYN, пришедшего от клиента, для которого сервер ждет завершения трехэтапного рукопожатия TCP. Эти сокеты находятся в состоянии `SYN_RCVD` (см. рис. 2.4).

2. *Очередь полностью установленных соединений (complete connection queue)*, содержащую запись для каждого клиента, с которым завершилось трехэтапное рукопожатие TCP. Эти сокеты находятся в состоянии `ESTABLISHED` (см. рис. 2.4).

На рис. 4.2 представлены обе эти очереди для прослушиваемого сокета.

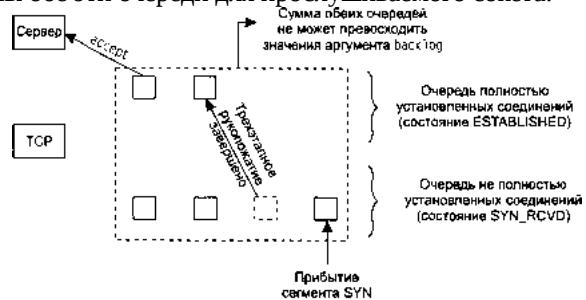


Рис. 4.2. Две очереди, поддерживаемые прослушиваемым сокетом TCP

Когда в очередь не полностью установленных соединений добавляется новая запись, параметры прослушиваемого сокета копируются на создаваемое соединение. Механизм создания соединения полностью автоматизирован, и процесс сервера в нем не участвует. На рис. 4.3 показан обмен пакетами во время установления соединения с использованием этих очередей.



Рис. 4.3. Обмен пакетами в процессе установления соединения с применением очередей

Когда от клиента приходит сегмент SYN, TCP создает новую запись в очереди не полностью установленных соединений, а затем отвечает вторым сегментом трехэтапного рукопожатия, посыпая сегмент SYN вместе с сегментом ACK, подтверждающим прием клиентского сегмента SYN (см. раздел 2.6). Эта запись останется в очереди не полностью установленных соединений, пока не придет третий сегмент трехэтапного рукопожатия (клиентский сегмент ACK для сегмента сервера SYN) или пока не истечет время жизни этой записи. (В реализациях, происходящих от Беркли, время ожидания (тайм-аут) для элементов очереди не полностью установленных соединений равно 75 с.) Если трехэтапное рукопожатие завершается нормально, запись переходит из очереди не полностью установленных соединений в конец очереди полностью установленных соединений. Когда процесс вызывает функцию `accept` (о которой мы поговорим в следующем разделе), ему возвращается первая запись из очереди

полностью установленных соединений, а если очередь пуста, процесс переходит в состояние ожидания до появления записи в ней.

Есть несколько важных моментов, которые нужно учитывать при работе с этими очередями.

■ Аргумент *backlog* функции *listen* исторически задавал максимальное суммарное значение для обеих очередей.

■ Беркли-реализации включают поправочный множитель для аргумента *backlog*, равный 1,5 [111, с. 257], [128, с. 462]. Например, при типичном значении аргумента *backlog* = 5 в таких системах допускается до восьми записей в очередях, как показано в табл. 4.6.

ПРИМЕЧАНИЕ

Формального определения аргумента *backlog* никогда не существовало. В руководстве 4.2BSD сказано, что «он определяет максимальную длину, до которой может вырасти очередь не полностью установленных соединений». Многие руководства и даже POSIX копируют это определение дословно, но в нем не говорится, в каком состоянии должно находиться соединение — в состоянии SYN_RCVD, ESTABLISHED (до вызова *accept*), или же в любом из них. Определение, приведенное выше, относится к реализации Беркли 4.2BSD, и копируется многими другими реализациями.

ПРИМЕЧАНИЕ

Причина возникновения этого множителя теряется в истории [57]. Но если мы рассматриваем *backlog* как способ задания максимального числа установленных соединений, которые ядро помещает в очередь прослушиваемого сокета (об этом вскоре будет рассказано), этот множитель нужен для учета не полностью установленных соединений, находящихся в очереди [8].

■ Не следует задавать нулевое значение аргументу *backlog*, поскольку различные реализации интерпретируют это по-разному (см. табл. 4.6). Некоторые реализации допускают помещение в очередь одного соединения, в то время как в других вообще невозможно помещать соединения в очередь. Если вы не хотите, чтобы клиенты соединялись с вашим прослушиваемым сокетом, просто закройте прослушиваемый сокет.

■ Если трехэтапное рукопожатие завершается нормально (то есть без потерянных сегментов и повторных передач), запись остается в очереди не полностью установленных соединений на время одного периода обращения (round-trip time, RTT), какое бы значение ни имел этот параметр для конкретного соединения между клиентом и сервером. В разделе 14.4 [112] показано, что для одного веб-сервера средний период RTT оказался равен 187 мс. (Чтобы редкие большие числа не искажали картину, здесь использована медиана, а не обычное среднее арифметическое по всем клиентам.)

■ Традиционно в примерах кода всегда используется значение *backlog*, равное 5, поскольку это было максимальное значение, которое поддерживалось в системе 4.2BSD. Это было актуально в 80-х, когда загруженные серверы могли обрабатывать только несколько сотен соединений в день. Но с ростом Сети (WWW), когда серверы обрабатывают миллионы соединений в день, столь малое число стало абсолютно неприемлемым [112, с. 187–192]. Серверам HTTP необходимо намного большее значение аргумента *backlog*, и новые ядра должны поддерживать такие значения.

ПРИМЕЧАНИЕ

В настоящее время многие системы позволяют администраторам изменять максимальное значение аргумента *backlog*.

■ Возникает вопрос: какое значение аргумента *backlog* должно задавать приложение, если значение 5 часто является неадекватным? На этот вопрос нет простого ответа. Серверы HTTP сейчас задают большее значение, но если заданное значение является в исходном коде константой, то для увеличения константы

требуется перекомпиляция сервера. Другой способ — принять некоторое значение по умолчанию и предоставить возможность изменять его с помощью параметра командной строки или переменной окружения. Всегда можно задавать значение больше того, которое поддерживается ядром, так как ядро должно обрезать значение до максимального, не возвращая при этом ошибку [128, с. 456].

Мы приводим простое решение этой проблемы, изменив нашу функцию-обертку для функции `listen`. В листинге 4.1^[1] представлен действующий код. Переменная окружения `LISTENQ` позволяет переопределить значение по умолчанию.

Листинг 4.1. Функция-обертка для функции `listen`, позволяющая переменной окружения переопределить аргумент `backlog`

```
//lib/wrapsoc.c
137 void
138 Listen(int fd, int backlog)
139 {
140     char *ptr;

141     /* может заменить второй аргумент на переменную окружения */
142     if ((ptr = getenv("LISTENQ")) != NULL)
143         backlog = atoi(ptr);

144     if (listen(fd, backlog) < 0)
145         err_sys("listen error");
146 }
```

■ Традиционно в руководствах и книгах утверждалось, что помещение фиксированного числа соединений в очередь позволяет обрабатывать случай загруженного серверного процесса между последовательными вызовами функции `accept`. При этом подразумевается, что из двух очередей больше записей будет содержаться, вероятнее всего, в очереди полностью установленных соединений. Но оказалось, что для действительно загруженных веб-серверов это не так. Причина задания большего значения `backlog` в том, что очередь не полностью установленных соединений растет по мере поступления сегментов SYN от клиентов; элементы очереди находятся в состоянии ожидания завершения трехэтапного рукопожатия.

■ Если очереди заполнены, когда приходит клиентский сегмент SYN, то TCP игнорирует приходящий сегмент SYN [128, с. 930–931] и не посыпает RST. Это происходит потому, что состояние считается временным, и TCP клиента должен еще раз передать свой сегмент SYN, для которого в ближайшее время, вероятно, найдется место в очереди. Если бы TCP сервера послал RST, функция `connect` клиента сразу же возвратила бы ошибку, заставив приложение обработать это условие, вместо того чтобы позволить TCP выполнить повторную передачу. Кроме того, клиент не может увидеть разницу между сегментами RST в ответе на сегмент SYN, означающими, что на данном порте нет сервера либо на данном порте есть сервер, но его очереди заполнены.

ПРИМЕЧАНИЕ

Некоторые реализации отправляют сегмент RST в описанной выше ситуации, что некорректно по изложенным выше причинам. Если вы не пишете клиент специально для работы с подобным сервером, лучше всего игнорировать такую возможность. Ее учет при кодировании клиента снизит его устойчивость и увеличит нагрузку на сеть, если окажется, что порт действительно не прослушивается сервером.

■ Данные, которые приходят после завершения трехэтапного рукопожатия, но до того, как сервер вызывает функцию `accept`, должны помещаться в очередь TCP-сервера, пока не будет заполнен приемный буфер.

В табл. 4.6 показано действительное число установленных в очередь соединений для различных значений аргумента `backlog` в операционных системах, показанных на рис. 1.7. Семь операционных систем помещены в пять колонок, что иллюстрирует многообразие значений аргумента `backlog`.

Таблица 4.6. Действительное количество соединений в очереди для различных значений аргумента backlog

backlog	MacOS 10.2.6	AIX 5.1	Linux 2.4.7	HP-UX 11.11	FreeBSD 4.8	FreeBSD 5.1	Solaris 2.9
0	1	3	1	1		1	
1	2	4	1	2		2	
2	4	5	3	3		4	
3	5	6	4	4		5	
4	7	7	6	5		6	
5	8	8	7	6		8	
6	10	9	9	7		10	
7	И	10	10	8		11	
8	13	11	12	9		13	
9	14	12	13	10		14	
10	16	13	15	11		16	
11	17	14	16	12		17	
12	19	15	18	13		19	
13	20	16	19	14		20	
14	22	17	21	15		22	

Системы AIX, BSD/OX и SunOS реализуют традиционный алгоритм Беркли, хотя последний не допускает значения аргумента *backlog* больше пяти. В системах HP-UX и Solaris 2.6 используется другой поправочный множитель к аргументу *backlog*. Системы Digital Unix, Linux и UnixWare воспринимают этот аргумент буквально, то есть не используют поправочный множитель, а в Solaris 2.5.1 к аргументу *backlog* просто добавляется единица.

ПРИМЕЧАНИЕ

Программа для измерения этих значений представлена в решении упражнения 15.4.

Как мы отмечали, традиционно аргумент *backlog* задавал максимальное значение для суммы обеих очередей. В 1996 году была предпринята новая атака через Интернет, названная SYN flooding (лавинная адресация сегмента SYN). Написанная хакером программа отправляет жертве сегменты SYN с высокой частотой, заполняя очередь не полностью установленных соединений для одного или нескольких портов TCP. (Хакером мы называем атакующего, как сказано в предисловии к [20].) Кроме того, IP-адрес отправителя каждого сегмента SYN задается случайным числом — формируются вымышленные IP-адреса (IP spoofing), что ведет к получению доступа обманным путем. Таким образом, сегмент сервера SYN/ACK уходит в никуда. Это не позволяет серверу узнать реальный IP-адрес хакера. Очередь не полностью установленных соединений заполняется ложными сегментами SYN, в результате чего для подлинных сегментов SYN в ней не хватает места — происходит отказ в обслуживании (denial of service) нормальных клиентов. Существует два типичных способа противостояния этим атакам [8]. Но самое интересное в этом примечании — это еще одно обращение к вопросу о том, что на самом деле означает аргумент *backlog* функции *listen*. Он должен задавать максимальное число установленных соединений для данного сокета, которые ядро помещает в очередь. Ограничение количества установленных соединений имеет целью приостановить получение ядром новых запросов на соединение для данного сокета, когда их не принимает приложение (по любой причине). Если система реализует именно такую интерпретацию, как, например, BSD/OS 3.0, то приложению не нужно задавать большие значения аргумента *backlog* только потому, что сервер обрабатывает множество клиентских запросов (например, занятый веб-сервер), или для защиты от «наводнения» SYN (лавинной адресации сегмента SYN). Ядро обрабатывает множество не полностью установленных соединений вне зависимости от того, являются ли они законными или приходят от хакера. Но даже в такой интерпретации мы видим (см. табл. 4.6), что значения 5 тут явно недостаточно.

4.6. Функция accept

Функция `accept` вызывается сервером TCP для возвращения следующего установленного соединения из начала очереди полностью установленных соединений (см. рис. 4.2). Если очередь полностью установлена пуста, процесс переходит в состояние ожидания (по умолчанию предполагается блокирующий сокет).

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Возвращает: неотрицательный дескриптор в случае успешного выполнения функции, -1 в случае ошибки

Аргументы `cliaddr` и `addrlen` используются для возвращения адреса протокола подключившегося процесса (клиента). Аргумент `addrlen` — это аргумент типа «значение-результат» (см. раздел 3.3). Перед вызовом мы присваиваем целому числу, на которое указывает `*addrlen`, размер структуры адреса сокета, на которую указывает аргумент `cliaddr`, и по завершении функции это целое число содержит действительное число байтов, помещенных ядром в структуру адреса сокета.

Если выполнение функции `accept` прошло успешно, она возвращает новый дескриптор, автоматически созданный ядром. Этот дескриптор используется для обращения к соединению TCP с конкретным клиентом. При описании функции `accept` мы называем ее первый аргумент *прослушиваемым сокетом* (*listening socket*) (дескриптор, созданный функцией `socket` и затем используемый в качестве аргумента для функций `bind` и `listen`), а значение, возвращаемое этой функцией, мы называем *присоединенным сокетом* (*connected socket*). Сервер обычно создает только один прослушиваемый сокет, который существует в течение всего времени жизни сервера. Затем ядро создает по одному присоединенному сокету для каждого клиентского соединения, принятого с помощью функции `accept` (для которого завершено трехэтапное рукопожатие TCP). Когда сервер заканчивает предоставление сервиса данному клиенту, сокет закрывается.

Эта функция возвращает до трех значений: целое число, которое является либо дескриптором сокета, либо кодом ошибки, а также адрес протокола клиентского процесса (через указатель `cliaddr`) и размер адреса (через указатель `addrlen`). Если нам не нужно, чтобы был возвращен адрес протокола клиента, следует сделать указатели `cliaddr` и `addrlen` пустыми указателями.

В листинге 1.5 показаны эти моменты. Присоединенный сокет закрывается при каждом прохождении цикла, но прослушиваемый сокет остается открытим в течение времени жизни сервера. Мы также видим, что второй и третий аргументы функции `accept` являются пустыми указателями, поскольку нам не нужно идентифицировать клиент.

Пример: аргументы типа «значение-результат»

В листинге 4.2 представлен измененный код из листинга 1.5 (вывод IP-адреса и номера порта клиента), обрабатывающий аргумент типа «значение-результат» функции `accept`.

Листинг 4.2. Сервер определения времени и даты, сообщающий IP-адрес и номер порта клиента

```
//intro/daytimetcpsrv1.c
1 #include "unp.h"
2 #include <time.h>
3 int
4 main(int argc, char **argv)
5 {
6     int listenfd, connfd;
7     socklen_t len;
8     struct sockaddr_in servaddr, cliaddr;
9     char buff[MAXLINE];
10    time_t ticks;
11    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
12    bzero(&servaddr, sizeof(servaddr));
13    servaddr.sin_family = AF_INET;
14    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```

15 servaddr.sin_port = htons(13); /* сервер времени и даты */
16 Bind(listenfd, (SA*)&servaddr, sizeof(servaddr));
17 Listen(listenfd, LISTENQ);

18 for (;;) {
19     len = sizeof(cliaddr);
20     connfd = Accept(listenfd, (SA*)&cliaddr, &len);
21     printf("connection from %s, port %d\n",
22            Inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)));
23     ntohs(cliaddr.sin_port));

24     ticks = time(NULL);
25     sprintf(buff, sizeof(buff), "% 24s\r\n", ctime(&ticks));
26     Write(connfd, buff, strlen(buff));

27     Close(connfd);
28 }
29 }
```

Новые объявления

7-8 Мы определяем две новых переменных: `len`, которая будет переменной типа «значение-результат», и `cliaddr`, которая будет содержать адрес протокола клиента.

Принятие соединения и вывод адреса клиента

19-23 Мы инициализируем переменную `len`, присвоив ей значение, равное размеру структуры адреса сокета, и передаем указатель на структуру `cliaddr` и указатель на `len` в качестве второго и третьего аргументов функции `accept`. Мы вызываем функцию `inet_ntop` (см. раздел 3.7) для преобразования 32-битового IP-адреса в структуре адреса сокета в строку ASCII (точечно-десятичную запись), а затем вызываем функцию `ntohs` (см. раздел 3.4) для преобразования сетевого порядка байтов в 16-битовом номере порта в порядок байтов узла.

ПРИМЕЧАНИЕ

При вызове функции `sock_ntop` вместо `inet_ntop` наш сервер станет меньше зависеть от протокола, однако он все равно зависит от IPv4. Мы покажем версию этого сервера, не зависящего от протокола, в листинге 11.7.

Если мы запустим наш новый сервер, а затем запустим клиент на том же узле, то дважды соединившись с сервером, мы получим от клиента следующий вывод:

```
solaris % daytimetcpccli 127.0.0.1
Thu Sep 11 12:44:00 2003
solaris % daytimetcpccli 192.168.1.20
Thu Sep 11 12:44:09 2003
```

Сначала мы задаем IP-адрес сервера как адрес закольцовки на себя (loopback address) (127.0.0.1), а затем как его собственный IP-адрес (192.168.1.20). Вот соответствующий вывод сервера:

```
solaris # daytimetcpsrv1
connection from 127.0.0.1, port 43388
connection from 192.168.1.20, port 43389
```

Обратите внимание на то, что происходит с IP-адресом клиента. Поскольку наш клиент времени и даты (см. листинг 1.1) не вызывает функцию `bind`, как сказано в разделе 4.4, ядро выбирает IP-адрес отправителя, основанный на используемом исходящем интерфейсе. В первом случае ядро задает IP-адрес

равным адресу закольцовки, во втором случае — равным IP-адресу интерфейса Ethernet. Кроме того, мы видим, что динамически назначаемый порт, выбранный ядром Solaris, — это 33 188, а затем 33 189 (см. рис. 2.10).

Наконец, заметьте, что приглашение интерпретатора команд изменилось на знак # — это приглашение к вводу команды для привилегированного пользователя. Наш сервер должен обладать правами привилегированного пользователя, чтобы с помощью функции bind связать зарезервированный порт 13. Если у нас нет прав привилегированного пользователя, вызов функции bind оказывается неудачным:

```
solaris % daytimecpsrv1
bind error: Permission denied
```

4.7. Функции fork и exec

Прежде чем рассматривать создание параллельного сервера (что мы сделаем в следующем разделе), необходимо описать функцию Unix fork. Эта функция является единственным способом создания нового процесса в Unix.

```
#include <unistd.h>
```

```
pid_t fork(void);
```

Возвращает: 0 в дочернем процессе, идентификатор дочернего процесса в родительском процессе, -1 в случае ошибки

Если вы никогда не встречались с этой функцией, трудным для понимания может оказаться то, что она вызывается *один раз*, а возвращает *два значения*. Одно значение эта функция возвращает в вызывающем процессе (который называется родительским процессом) — этим значением является идентификатор созданного процесса (который называется дочерним процессом). Второе значение (нуль) она возвращает в дочернем процессе. Следовательно, по возвращаемому значению можно определить, является ли данный процесс родительским или дочерним.

Причина того, что функция fork возвращает в дочернем процессе нуль, а не идентификатор родительского процесса, заключается в том, что у дочернего процесса есть только один родитель, и дочерний процесс всегда может получить идентификатор родительского, вызвав функцию getpid. У родителя же может быть любое количество дочерних процессов, и способа получить их идентификаторы не существует. Если родительскому процессу требуется отслеживать идентификаторы своих дочерних процессов, он должен записывать возвращаемые значения функции fork.

Все дескрипторы, открытые в родительском процессе перед вызовом функции fork, становятся доступными дочерним процессам. Вы увидите, как это свойство используется сетевыми серверами: родительский процесс вызывает функцию accept, а затем функцию fork. Затем присоединенный сокет совместно используется родительским и дочерним процессами. Обычно дочерний процесс использует присоединенный сокет для чтения и записи, а родительский процесс только закрывает присоединенный сокет.

Существует два типичных случая применения функции fork:

1. Процесс создает свои копии таким образом, что каждая из них может обрабатывать одно задание. Это типичная ситуация для сетевых серверов. Далее в тексте вы увидите множество подобных примеров.

2. Процесс хочет запустить другую программу. Поскольку единственный способ создать новый процесс — это вызвать функцию fork, процесс сначала вызывает функцию fork, чтобы создать свою копию, а затем одна из копий (обычно дочерний процесс) вызывает функцию exec (ее описание следует за описанием функции fork), чтобы заменить себя новой программой. Этот сценарий типичен для таких программ, как интерпретаторы командной строки.

Единственный способ запустить в Unix на выполнение какой-либо файл — вызвать функцию exec. (Мы будем часто использовать общее выражение «функция exec», когда неважно, какая из шести функций семейства exec вызывается.) Функция exec заменяет копию текущего процесса новым программным файлом, причем в новой программе обычно запускается функция main. Идентификатор процесса при этом не изменяется. Процесс, вызывающий функцию exec, мы будем называть *вызывающим процессом*, а выполняемую при этом программу — *новой программой*.

ПРИМЕЧАНИЕ

В старых описаниях и книгах новая программа ошибочно называется «новым процессом». Это неверно, поскольку новый процесс не создается.

Различие между шестью функциями exec заключается в том, что они допускают различные способы задания аргументов:

- выполняемый программный файл может быть задан или *именем файла (filename)*, или *полным именем (pathname)*;
- аргументы новой программы либо перечисляются один за другим, либо на них имеется ссылка через массив указателей;
- новой программе либо передается окружение вызывающего процесса, либо задается новое окружение.

```
#include <unistd.h>
```

```
int execl(const char * pathname, const char * arg0, ... /* (char*)0 */ );
int execv(const char * pathname, char *const argv[]);
int execle(const char * pathname, const char * arg0 ... /* (char*)0,
    char *const envp[] */ );
int execve(const char * pathname, char *const argv[], char *const envp[]);
int execlp(const char * filename, const char * arg0, .... /* (char*)0 */ );
int execvp(const char * filename, char *const argv[]);
```

Все шесть функций возвращают: -1 в случае ошибки, если же функция выполнена успешно, то ничего не возвращается

Эти функции возвращают вызывающему процессу значение -1, только если происходит ошибка. Иначе управление передается в начало новой программы, обычно функции main.

Отношения между этими шестью функциями показаны на рис. 4.4. Обычно только функция execve является системным вызовом внутри ядра, а остальные представляют собой библиотечные функции, вызывающие execve.



Рис. 4.4. Отношения между шестью функциями exec

Отметим различия между этими функциями:

1. Три верхних функции (см. рис. 4.4) принимают каждую строку как отдельный аргумент, причем перечень аргументов завершается пустым указателем (так как их количество может быть различным). У трех нижних функций имеется массив argv, содержащий указатели на строки. Этот массив должен содержать пустой указатель, определяющий конец массива, поскольку размер массива не задается.

2. Две функции в левой колонке получают аргумент filename. Он преобразуется в pathname с использованием текущей переменной окружения PATH. Если аргумент filename функций execvp или execv содержит косую черту (/) в любом месте строки, переменная PATH не используется. Четыре функции в двух правых колонках получают полностью определенный аргумент pathname.

3. Четыре функции в двух левых колонках не получают явного списка переменных окружения. Вместо этого с помощью текущего значения внешней переменной environ создается список переменных окружения, который передается новой программе. Две функции в правой колонке получают точный список переменных окружения. Массив указателей envp должен быть завершен пустым указателем.

Дескрипторы, открытые в процессе перед вызовом функции exec, обычно остаются открытыми во время ее выполнения. Мы говорим «обычно», поскольку это свойство может быть отключено при использовании функции fcntl для установки флага дескриптора FD_CLOEXEC. Это нужно серверу inetd, о котором пойдет речь в разделе 13.5.

4.8. Параллельные серверы

Сервер, представленный в листинге 4.2, является *последовательным (итеративным) сервером*. Для такого простого сервера, как сервер времени и даты, это допустимо. Но когда обработка запроса клиента

занимает больше времени, мы не можем связывать один сервер с одним клиентом, поскольку нам хотелось бы обрабатывать множество клиентов одновременно. Простейшим способом написать параллельный сервер под Unix является вызов функции `fork`, порождающей дочерний процесс для каждого клиента. В листинге 4.3 представлена общая схема типичного параллельного сервера.

Листинг 4.3. Типичный параллельный сервер

```
pid_t pid;
int listenfd, connfd;

listenfd = Socket( ... );

/* записываем в sockaddr_in{} параметры заранее известного порта сервера */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for (;;) {
    connfd = Accept(listenfd, ...); /* вероятно, блокировка */

    if ((pid = Fork()) == 0) {
        Close(listenfd); /* дочерний процесс закрывает
                           прослушиваемый сокет */
        doit(connfd); /* обработка запроса */
        Close(connfd); /* с этим клиентом закончено */
        exit(0); /* дочерний процесс завершен */
    }

    Close(connfd); /* родительский процесс закрывает
                     присоединенный сокет */
}

}
```

Когда соединение установлено, функция `accept` возвращает управление, сервер вызывает функцию `fork` и затем дочерний процесс занимается обслуживанием клиента (по присоединенному сокету `connfd`), а родительский процесс ждет другого соединения (на прослушиваемом сокете `listenfd`). Родительский процесс закрывает присоединенный сокет, поскольку новый клиент обрабатывается дочерним процессом.

Мы предполагаем, что функция `doit` в листинге 4.3 выполняет все, что требуется для обслуживания клиента. Когда эта функция возвращает управление, мы явно закрываем присоединенный сокет с помощью функции `close` в дочернем процессе. Делать это не обязательно, так как в следующей строке вызывается `exit`, а прекращение процесса подразумевает, в частности, закрытие ядром всех открытых дескрипторов. Включать явный вызов функции `close` или нет — дело вкуса программиста.

В разделе 2.6 мы сказали, что вызов функции `close` на сокете TCP вызывает отправку сегмента FIN, за которой следует обычная последовательность прекращения соединения TCP. Почему же функция `close(connfd)` из листинга 4.3, вызванная родительским процессом, не завершает соединение с клиентом? Чтобы понять происходящее, мы должны учитывать, что у каждого файла и сокета есть счетчик ссылок (`reference count`). Для счетчика ссылок поддерживается своя запись в таблице файла [110, с. 57–60]. Эта запись содержит значения счетчика дескрипторов, открытых в настоящий момент, которые соответствуют этому файлу или сокету. В листинге 4.3 после завершения функции `socket` запись в таблице файлов, связанная с `listenfd`, содержит значение счетчика ссылок, равное 1. Но после завершения функции `fork` дескрипторы дублируются (для совместного использования и родительским, и дочерним процессом), поэтому записи в таблице файла, ассоциированные с этими сокетами, теперь содержат значение 2. Следовательно, когда родительский процесс закрывает `connfd`, счетчик ссылок уменьшается с 2 до 1. Но фактического закрытия дескриптора не произойдет, пока счетчик ссылок не станет равен 0. Это случится несколько позже, когда дочерний процесс закроет `connfd`.

Рассмотрим пример, иллюстрирующий листинг 4.3. Прежде всего, на рис. 4.5 показано состояние клиента и сервера в тот момент, когда сервер блокируется при вызове функции `accept` и от клиента приходит запрос на соединение.



Рис. 4.5. Состояние соединения клиент-сервер перед завершением вызванной функции accept

Сразу же после завершения функции accept мы получаем сценарий, изображенный на рис. 4.6. Соединение принимается ядром и создается новый сокет — connfd. Это присоединенный сокет, и теперь данные могут считываться и записываться по этому соединению.

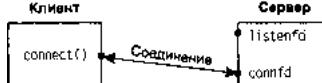


Рис. 4.6. Состояние соединения клиент-сервер после завершения функции accept

Следующим действием параллельного сервера является вызов функции fork. На рис. 4.7 показано состояние соединения после вызова функции fork.

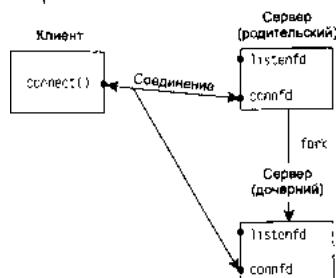


Рис. 4.7. Состояние соединения клиент-сервер после вызова функции fork

Обратите внимание, что оба дескриптора listenfd и connfd совместно используются родительским и дочерним процессами.

Далее родительский процесс закрывает присоединенный сокет, а дочерний процесс закрывает прослушиваемый сокет. Это показано на рис. 4.8.

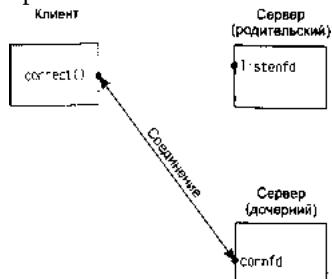


Рис. 4.8. Состояние соединения клиент-сервер после закрытия родительским и дочерним процессами соответствующих сокетов

Это и есть требуемое конечное состояние сокетов. Дочерний процесс управляет соединением с клиентом, а родительский процесс может снова вызвать функцию accept на прослушиваемом сокете, чтобы обрабатывать следующее клиентское соединение.

4.9. Функция close

Обычная функция Unix close также используется для закрытия сокета и завершения соединения TCP.

```
#include <unistd.h>
```

```
int close(int sockfd);
```

По умолчанию функция close помечает сокет TCP как закрытый и немедленно возвращает управление процессу. Дескриптор сокета больше не используется процессом и не может быть передан в качестве аргумента функции read или write. Но TCP попытается отправить данные, которые уже установлены в очередь, и после их отправки осуществит нормальную последовательность завершения соединения TCP (см. раздел 2.5).

В разделе 7.5 рассказывается о параметре сокета SO_LINGER, который позволяет нам изменять последовательность закрытия сокета TCP. В этом разделе мы также назовем действия, благодаря которым приложение TCP может получить гарантию того, что приложение-собеседник получило данные, поставленные в очередь на отправку, но еще не отправленные.

Счетчик ссылок дескриптора

В конце раздела 4.8 мы отметили, что когда родительский процесс на нашем параллельном сервере закрывает присоединенный сокет с помощью функции `close`, счетчик ссылок дескриптора уменьшается лишь на единицу. Поскольку счетчик ссылок при этом все еще оставался больше нуля, вызов функции `close` не инициировал последовательность завершения TCP-соединения, состоящую из четырех пакетов. Нам нужно, чтобы наш параллельный сервер с присоединенным сокетом, разделяемым между родительским и дочерним процессами, работал именно по этому принципу.

Если мы хотим отправить сегмент FIN по соединению TCP, вместо функции `close` должна использоваться функция `shutdown` (см. раздел 6.6). Причины мы рассмотрим в разделе 6.5.

Необходимо также знать, что происходит с нашим параллельным сервером, если родительский процесс не вызывает функцию `close` для каждого присоединенного сокета, возвращаемого функцией `accept`. Прежде всего, родительский процесс в какой-то момент израсходует все дескрипторы, поскольку обычно число дескрипторов, которые могут быть открыты процессом, ограничено. Но что более важно, ни одно из клиентских соединений не будет завершено. Когда дочерний процесс закрывает присоединенный сокет, его счетчик ссылок уменьшается с 2 до 1 и остается равным 1, поскольку родительский процесс не закрывает присоединенный сокет с помощью функции `close`. Это помешает выполнить последовательность завершения соединения TCP, и соединение останется открытым.

4.10. Функции `getsockname` и `getpeername`

Эти две функции возвращают либо локальный (функция `getsockname`), либо удаленный (функция `getpeername`) адрес протокола, связанный с сокетом.

```
#include <sys/socket.h>
```

```
int getsockname(int sockfd, struct sockaddr *localaddr,  
    socklen_t *addrlen);  
int getpeername(int sockfd, struct sockaddr *peeraddr,  
    socklen_t *addrlen);
```

Обратите внимание, что последний аргумент обеих функций относится к типу «значение-результат», то есть обе функции будут заполнять структуру адреса сокета, на которую указывает аргумент `localaddr` или `peeraddr`.

ПРИМЕЧАНИЕ

Обсуждая функцию `bind`, мы отметили, что термин «имя» используется некорректно. Эти две функции возвращают адрес протокола, связанный с одним из концов сетевого соединения, что для протоколов IPv4 и IPv6 является сочетанием IP-адреса и номера порта. Эти функции также не имеют ничего общего с доменными именами (глава 11).

Функции `getsockname` и `getpeername` необходимы нам по следующим соображениям:

- После успешного выполнения функции `connect` и возвращения управления в клиентский процесс TCP, который не вызывает функцию `bind`, функция `getsockname` возвращает IP-адрес и номер локального порта, присвоенные соединению ядром.

- После вызова функции `bind` с номером порта 0 (что является указанием ядру на необходимость выбрать номер локального порта) функция `getsockname` возвращает номер локального порта, который был задан.

- Функцию `getsockname` можно вызвать, чтобы получить семейство адресов сокета, как это показано в листинге 4.4.

- Сервер TCP, который с помощью функции `bind` связывается с универсальным IP-адресом (см. листинг 1.5), как только устанавливается соединение с клиентом (функция `accept` успешно выполнена), может вызвать функцию `getsockname`, чтобы получить локальный IP-адрес соединения. Аргумент `sockfd` (дескриптор сокета) в этом вызове должен содержать дескриптор присоединенного, а не прослушиваемого сокета.

■ Когда сервер запускается с помощью функции exec процессом, вызывающим функцию accept, он может идентифицировать клиента только одним способом - вызвать функцию getpeername. Это происходит, когда функция inetd (см. раздел 13.5) вызывает функции fork и exec для создания сервера TCP. Этот сценарий представлен на рис. 4.9. Функция inetd вызывает функцию accept (верхняя левая рамка), после чего возвращаются два значения: дескриптор присоединенного сокета connfd (это возвращаемое значение функции), а также IP-адрес и номер порта клиента, отмеченные на рисунке небольшой рамкой с подписью «адрес собеседника» (структура адреса сокета Интернета). Далее вызывается функция fork и создается дочерний процесс функции inetd. Поскольку дочерний процесс запускается с копией содержимого памяти родительского процесса, структура адреса сокета доступна дочернему процессу, как и дескриптор присоединенного сокета (так как дескрипторы совместно используются родительским и дочерним процессами). Но когда дочерний процесс с помощью функции exec запускает выполнение реального сервера (скажем, сервера Telnet), содержимое памяти дочернего процесса заменяется новым программным файлом для сервера Telnet (то есть структура адреса сокета, содержащая адрес собеседника, теряется). Однако во время выполнения функции exec дескриптор присоединенного сокета остается открытым. Один из первых вызовов функции, который выполняет сервер Telnet, — это вызов функции getpeername для получения IP-адреса и номера порта клиента.

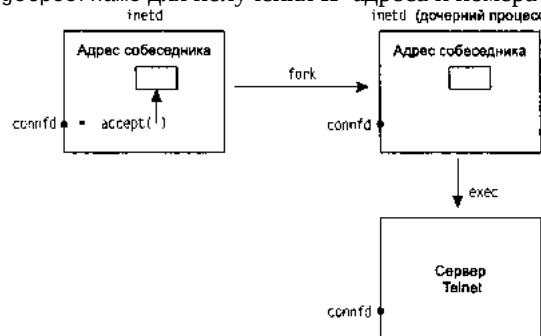


Рис. 4.9. Порождение сервера демоном inetd

Очевидно, что в приведенном примере сервер Telnet при запуске должен знать значение функции connfd. Этого можно достичь двумя способами. Во-первых, процесс,зывающий функцию exec, может отформатировать номер дескриптора как символьную строку и передать ее в виде аргумента командной строки программы, выполняемой с помощью функции exec. Во-вторых, можно заключить соглашение относительно определенных дескрипторов: некоторый дескриптор всегда присваивается присоединенному сокету перед вызовом функции exec. Последний случай соответствует действию функции inetd — она всегда присваивает дескрипторы 0, 1 и 2 присоединенным сокетам.

Пример: получение семейства адресов сокета

Функция sockfd_to_family, представленная в листинге 4.4, возвращает семейство адресов сокета.

Листинг 4.4. Возвращаемое семейство адресов сокета

```

//lib/sockfd_to_family.c
1 #include "unp.h"

2 int
3 sockfd_to_family(int sockfd)
4 {
5     union {
6         struct sockaddr sa;
7         char data[MAXSOCKADDR];
8     } un;
9     socklen_t len;

10    len = MAXSOCKADDR;
11    if (getsockname(sockfd, (SA*)un.data, &len) < 0)
12        return (-1);

```

```
13 return (un.sa.sa_family);
14 }
```

Выделение пространства для наибольшей структуры адреса сокета

5-8 Поскольку мы не знаем, какой тип структуры адреса сокета нужно будет разместить в памяти, мы используем в нашем заголовочном файле `inpr.h` константу `MAXSOCKADDR`, которая представляет собой размер наибольшей структуры адреса сокета в байтах. Мы определяем массив типа `char` соответствующего размера в объединении, включающем универсальную структуру адреса сокета.

Вызов функции `getsockname`

10-13 Мы вызываем функцию `getsockname` и возвращаем семейство адресов.

Поскольку POSIX позволяет вызывать функцию `getsockname` на неприсоединенном сокете, эта функция должна работать для любого дескриптора открытого сокета.

4.11. Резюме

Все клиенты и серверы начинают работу с вызова функции `socket`, возвращающей дескриптор сокета. Затем клиенты вызывают функцию `connect`, в то время как серверы вызывают функции `bind`, `listen` и `accept`. Сокеты обычно закрываются с помощью стандартной функции `close`, хотя в разделе 6.6 вы увидите другой способ закрытия, реализуемый с помощью функции `shutdown`. Мы также проверим влияние параметра сокета `SO_LINGER` (см. раздел 7.5).

Большинство серверов TCP являются параллельными. При этом для каждого клиентского соединения, которым управляет сервер, вызывается функция `fork`. Вы увидите, что большинство серверов UDP являются последовательными. Хотя обе эти модели успешно использовались на протяжении ряда лет, имеются и другие возможности создания серверов с использованием программных потоков и процессов, которые мы рассмотрим в главе 30.

Упражнения

1. В разделе 4.4 мы утверждали, что константы `INADDR_`, определенные в заголовочном файле `<netinet/in.h>`, расположены в порядке байтов узла. Каким образом мы можем это определить?
2. Измените листинг 1.1 так, чтобы вызвать функцию `getsockname` после успешного завершения функции `connect`. Выведите локальный IP-адрес и локальный порт, присвоенный сокету TCP, используя функцию `sock_ntop`. В каком диапазоне (см. рис. 2.10) будут находиться динамически назначаемые порты вашей системы?
3. Предположим, что на параллельном сервере после вызова функции `fork` запускается дочерний процесс, который завершает обслуживание клиента перед тем, как результат выполнения функции `fork` возвращается родительскому процессу. Что происходит при этих двух вызовах функции `close` в листинге 4.3?
4. В листинге 4.2 сначала измените порт сервера с 13 на 9999 (так, чтобы для запуска программы вам не потребовались права привилегированного пользователя). Удалите вызов функции `listen`. Что происходит?
5. Продолжайте предыдущее упражнение. Удалите вызов функции `bind`, но оставьте вызов функции `listen`. Что происходит?

Глава 5

Пример TCP-соединения клиент-сервер

5.1. Введение

Напишем простой пример пары клиент-сервер, используя элементарные функции из предыдущей главы. Наш простой пример — это эхо-сервер, функционирующий следующим образом:

1. Клиент считывает строку текста из стандартного потока ввода и отправляет ее серверу.
2. Сервер считывает строку из сети и отсылает эту строку обратно клиенту.
3. Клиент считывает отраженную строку и помещает ее в свой стандартный поток вывода.

На рис. 5.1 изображена пара клиент-сервер вместе с функциями, используемыми для ввода и вывода.

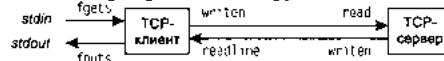


Рис. 5.1. Простой эхо-клиент и эхо-сервер

Между клиентом и сервером мы показали две стрелки, но на самом деле это одно двустороннее соединение TCP. Функции fgets и fputs имеются в стандартной библиотеке ввода-вывода, а функции written и readline приведены в разделе 3.9.

Мы разрабатываем нашу собственную реализацию эхо-сервера, однако большинство реализаций TCP/IP предоставляют готовый эхо-сервер, работающий как с TCP, так и с UDP (см. раздел 2.12). С нашим собственным клиентом мы также будем использовать и готовый сервер.

Соединение клиент-сервер, отражающее вводимые строки, является корректным и в то же время простым примером сетевого приложения. На этом примере можно проиллюстрировать все основные действия, необходимые для реализации соединения клиент-сервер. Все, что вам нужно сделать, чтобы применить его к вашему приложению, — это изменить операции, которые выполняет сервер с принимаемыми от клиентов данными.

С помощью этого примера мы можем не только проанализировать запуск нашего клиента и сервера в нормальном режиме (ввести строку и посмотреть, как она отражается), но и исследовать множество «границных условий»: выяснить, что происходит в момент запуска клиента и сервера; что происходит, когда клиент нормальным образом завершает работу; что происходит с клиентом, если процесс сервера завершается до завершения клиента или если возникает сбой на узле сервера, и т.д. Рассмотрев эти сценарии мы сможем понять, что происходит на уровне сети и как это представляется для API сокетов, и научиться писать приложения так, чтобы они умели обрабатывать подобные ситуации.

Во всех рассматриваемых далее примерах присутствуют зависящие от протоколов жестко заданные (hard coded) константы, такие как адреса и порты. Это обусловлено двумя причинами. Во-первых, нам необходимо точно понимать, что нужно хранить в структурах адресов, относящихся к конкретным протоколам. Во-вторых, мы еще не рассмотрели библиотечные функции, которые сделали бы наши программы более переносимыми. Эти функции рассматриваются в главе 11.

В последующих главах код клиента и сервера будет претерпевать многочисленные изменения, по мере того как вы будете больше узнавать о сетевом программировании (см. табл. 1.3 и 1.4).

5.2. Эхо-сервер TCP: функция main

Наши клиент и сервер TCP используют функции, показанные на рис. 4.1. Программа параллельного сервера представлена в листинге 5.1^[1].

Листинг 5.1. Эхо-сервер TCP (улучшенный в листинге 5.9)

```
//tcpcliserv/tcpserv01.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int listenfd, connfd;
6     pid_t childpid;
```

```

7  socklen_t clilen;
8  struct sockaddr_in cliaddr, servaddr;
9
10 listenfd = Socket(AF_INET, SOCK_STREAM, 0);
11
12 bzero(&servaddr, sizeof(servaddr));
13 servaddr.sin_family = AF_INET;
14 servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15 servaddr.sin_port = htons(SERV_PORT);
16
17 Bind(listenfd, (SA*)&servaddr, sizeof(servaddr));
18
19 Listen(listenfd, LISTENQ);
20
21 for (;;) {
22     clilen = sizeof(cliaddr);
23     connfd = Accept(listenfd, (SA*)&cliaddr, &clilen);
24
25     if ((childpid = Fork()) == 0) { /* дочерний процесс */
26         Close(listenfd); /* закрываем прослушиваемый сокет */
27         str_echo(connfd); /* обрабатываем запрос */
28         exit(0);
29     }
30     Close(connfd); /* родительский процесс закрывает
31                     присоединенный сокет */
32 }
33 }
```

Создание сокета, связывание с известным портом сервера

9-15 Создается сокет TCP. В структуру адреса сокета Интернета записывается универсальный адрес (INADDR_ANY) и номер заранее известного порта сервера (SERV_PORT, который определен как 9877 в нашем заголовочном файле `inpr.h`). В результате связывания с универсальным адресом системе сообщается, что мы примем соединение, предназначенное для любого локального интерфейса в том случае, если система имеет несколько сетевых интерфейсов. Наш выбор номера порта TCP основан на рис. 2.10. Он должен быть больше 1023 (нам не нужен зарезервированный порт), больше 5000 (чтобы не допустить конфликта с динамически назначаемыми портами, которые выделяются многими реализациями, происходящими от Беркли), меньше 49 152 (чтобы избежать конфликта с «правильным» диапазоном динамически назначаемых портов) и не должен конфликтовать ни с одним зарегистрированным портом. Сокет преобразуется в прослушиваемый при помощи функции `listen`.

Ожидание завершения клиентского соединения

17-18 Сервер блокируется в вызове функции `accept`, ожидая подключения клиента.

Параллельный сервер

19-24 Для каждого клиента функция `fork` порождает дочерний процесс, и дочерний процесс обслуживает запрос этого клиента. Как мы говорили в разделе 4.8, дочерний процесс закрывает прослушиваемый сокет, а родительский процесс закрывает присоединенный сокет. Затем дочерний процесс вызывает функцию `str_echo` (см. листинг 5.2) для обработки запроса клиента.

5.3. Эхо-сервер TCP: функция str_echo

Функция `str_echo`, показанная в листинге 5.2, выполняет серверную обработку запроса клиента: считывание строк от клиента и отражение их обратно клиенту.

Листинг 5.2. Функция str_echo: отраженные строки на сокете

```
//lib/str_echo.c
1 #include "unp.h"

2 void
3 str_echo(int sockfd)
4 {
5     ssize_t n;
6     char buf[MAXLINE];

7     for (;;) {
8         if ((n = read(sockfd, buf, MAXLINE)) > 0)
9             return; /* соединение закрыто с другого конца */

10    Writen(sockfd, line, n);
11 }
12 }
```

Чтение строки и ее отражение

7-11 Функция `read` считывает очередную строку из сокета, после чего строка отражается обратно клиенту с помощью функции `written`. Если клиент закрывает соединение (нормальный сценарий), то при получении клиентского сегмента FIN функция дочернего процесса `read` возвращает нуль. После этого происходит возврат из функции `str_echo` и далее завершается дочерний процесс, приведенный в листинге 5.1.

5.4. Эхо-клиент TCP: функция main

В листинге 5.3 показана функция `main` TCP-клиента.

Листинг 5.3. Эхо-клиент TCP

```
16 exit(0);
17 }
```

Создание сокета, заполнение структуры его адреса

9-13 Создается сокет TCP и структура адреса сокета заполняется IP-адресом сервера и номером порта. IP-адрес сервера мы берем из командной строки, а известный номер порта сервера (SERV_PORT) — из нашего заголовочного файла `unp.h`.

Соединение с сервером

14-15 Функция `connect` устанавливает соединение с сервером. Затем функция `str_cli` (см. листинг 5.4) выполняет все необходимые действия со стороны клиента.

5.5. Эхо-клиент TCP: функция str_cli

Эта функция, показанная в листинге 5.4, обеспечивает отправку запроса клиента и прием ответа сервера в цикле. Функция считывает строку текста из стандартного потока ввода, отправляет ее серверу и считывает отраженный ответ сервера, после чего помещает отраженную строку в стандартный поток вывода.

Листинг 5.4. Функция `str_cli`: цикл формирования запроса клиента

```
//lib/str_cli.c
1 #include "unp.h"

2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     char sendline[MAXLINE], recvline[MAXLINE];

6     while (Fgets(sendline, MAXLINE, fp) != NULL) {

7         Writen(sockfd, sendline, strlen(sendline));

8         if (Readline(sockfd, recvline, MAXLINE) == 0)
9             err_quit("str_cli: server terminated prematurely");

10        Fputs(recvline, stdout);
11    }
12 }
```

Считывание строки, отправка серверу

6-7 Функция `fgets` считывает строку текста, а функция `writen` отправляет эту строку серверу.

Считывание отраженной сервером строки, запись в стандартный поток вывода

8-10 Функция `readline` принимает отраженную сервером строку, а функция `fputs` записывает ее в стандартный поток вывода.

Возврат в функцию main

11-12 Цикл завершается, когда функция fgets возвращает пустой указатель, что означает достижение конца файла или обнаружение ошибки. Наша функция-обертка Fgets проверяет наличие ошибки, и если ошибка действительно произошла, прерывает выполнение программы. Таким образом, функция Fgets возвращает пустой указатель только при достижении конца файла.

5.6. Нормальный запуск

Наш небольшой пример использования TCP (около 150 строк кода для двух функций main, str_echo, str_cli, readline и writen) позволяет понять, как запускаются и завершаются клиент и сервер и, что наиболее важно, как развиваются события, если произошел сбой на узле клиента или в клиентском процессе, потеряна связь в сети и т.д. Только при понимании этих «границных условий» и их взаимодействия с протоколами TCP/IP мы сможем обеспечить устойчивость клиентов и серверов, которые смогут справляться с подобными ситуациями.

Сначала мы запускаем сервер в фоновом режиме на узле linux.

```
linux % tcpserv01 &
[1] 17870
```

Когда сервер запускается, он вызывает функции socket, bind, listen и accept, а затем блокируется в вызове функции accept. (Мы еще не запустили клиент.) Перед тем, как запустить клиент, мы запускаем программу netstat, чтобы проверить состояние прослушиваемого сокета сервера.

```
linux % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp        0      0 *:9877          *:*      LISTEN
```

Здесь мы показываем только первую строку вывода и интересующую нас строку. Эта команда показывает состояние *всех* сокетов в системе, поэтому вывод может быть большим. Для просмотра прослушиваемых сокетов следует указать параметр -a.

Результат совпадает с нашими ожиданиями. Сокет находится в состоянии LISTEN, локальный IP-адрес задан с помощью символа подстановки (то есть является универсальным) и указан локальный порт 9877. Функция netstat выводит звездочку для нулевого IP-адреса (INADDR_ANY, универсальный адрес) или для нулевого порта.

Затем на том же узле мы запускаем клиент, задав IP-адрес сервера 127.0.0.1. Мы могли бы задать здесь и нормальный адрес сервера (его IP-адрес в сети).

```
linux % tcpcli01 127.0.0.1
```

Клиент вызывает функции socket и connect, последняя осуществляет трехэтапное рукопожатие TCP. Когда рукопожатие TCP завершается, функция connect возвращает управление процессу-клиенту, а функция accept — процессу-серверу. Соединение установлено. Затем выполняются следующие шаги:

1. Клиент вызывает функцию str_cli, которая блокируется в вызове функции fgets, поскольку мы еще ничего не ввели.

2. Когда функция accept возвращает управление процессу-серверу, последний вызывает функцию fork, а дочерний процесс вызывает функцию str_echo. Та вызывает функцию read, блокируемую в ожидании получения данных от клиента.

3. Родительский процесс сервера снова вызывает функцию accept и блокируется в ожидании подключения следующего клиента.

У нас имеется три процесса, и все они находятся в состоянии ожидания (блокированы): клиент, родительский процесс сервера и дочерний процесс сервера.

ПРИМЕЧАНИЕ

Мы специально поставили первым пунктом (после завершения трехэтапного рукопожатия) вызов функции str_cli, происходящий на стороне клиента, а затем уже перечислили действия на стороне сервера. Причину объясняет рис. 2.5: функция connect возвращает управление, когда клиент получает второй сегмент рукопожатия. Однако функция accept не возвращает управление до тех пор, пока сервер не получит третий сегмент рукопожатия, то есть пока не пройдет половина периода RTT после завершения функции connect.

Мы намеренно запускаем и клиент, и сервер на одном узле — так проще всего экспериментировать с клиент-серверными приложениями. Поскольку клиент и сервер запущены на одном узле, функция netstat отображает теперь две дополнительные строки вывода, соответствующие соединению TCP:

```
linux % netstat -a
Proto Recv-Q Send-Q Local Address      Foreign Address State
tcp        0      0 localhost:9877    localhost:42758 ESTABLISHED
tcp        0      0 localhost:42758    localhost:42758 ESTABLISHED
tcp        0      0 *:9877            *:*          LISTEN
```

Первая из строк состояния ESTABLISHED соответствует дочернему сокету сервера, поскольку локальным портом является порт 9877. Вторая строка ESTABLISHED — это клиентский сокет, поскольку локальный порт — порт 42758. Если мы запускаем клиент и сервер на разных узлах, на узле клиента будет отображаться только клиентский сокет, а на узле сервера — два серверных сокета.

Для проверки состояний процессов и отношений между ними можно также использовать команду ps:

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
PID  PPID TT      STAT COMMAND      WCHAN
22038 22036 pts/6 S      -bash      wait4
17870 22038 pts/6 S      ./tcperv01 wait_for_connect
19315 17870 pts/6 S      ./tcperv01 tcp_data_wait
19314 22038 pts/6 S      ./tcpcli01 127.0.0.1 read_chan
```

Мы вызвали ps с несколько необычным набором аргументов для того, чтобы получить всю необходимую для дальнейшего обсуждения информацию. Мы запустили клиент и сервер из одного окна (pts/6, что означает псевдотерминал 6). В колонках PID и PPID показаны отношения между родительским и дочерним процессами. Можно точно сказать, что первая строка tcperv01 соответствует родительскому процессу, а вторая строка tcperv01 — дочернему, поскольку PPID дочернего процесса — это PID родительского. Кроме того, PPID родительского процесса совпадает с PID интерпретатора команд (bash).

Колонка STAT для всех трех сетевых процессов отмечена символом S. Это означает, что процессы находятся в состоянии ожидания (sleeping). Если процесс находится в состоянии ожидания, колонка WCHAN сообщает нам о том, чем он занят. В Linux значение wait_for_connect выводится, если процесс блокируется функцией accept или connect, значение tcp_data_wait — если процесс блокируется при вводе или выводе через сокет, а read_chan — если процесс блокируется при терминальном вводе-выводе. Так что для наших трех сетевых процессов значения WCHAN выглядят вполне осмысленно.

5.7. Нормальное завершение

На этом этапе соединение установлено, и все, что бы мы ни вводили на стороне клиента, отражается обратно.

```
linux % tcpcli01 127.0.0.1 эту строку мы показывали раньше
hello, world наш ввод
hello, world отраженная сервером строка
good bye
good bye
^D Ctrl+D - наш завершающий символ для обозначения конца файла
```

Мы вводим две строки, каждая из них отражается, затем мы вводим символ конца файла (EOF) **Ctrl+d**, который завершает работу клиента. Если мы сразу же выполним команду netstat, то увидим следующее:

```
linux % netstat -a | grep 9877
tcp 0 0 *:9877            *:*
tcp 0 0 local host:42758  localhost:9877
```

Клиентская часть соединения (локальный порт 42758) входит в состояние TIME_WAIT (см. раздел 2.6), и прослушивающий сервер все еще ждет подключения другого клиента. (В этот раз мы передаем вывод netstat программе grep, чтобы вывести только строки с заранее известным портом нашего сервера. Но при этом также удаляется строка заголовка.)

Перечислим этапы нормального завершения работы нашего клиента и сервера.

1. Когда мы набираем символ EOF, функция fgets возвращает пустой указатель, и функция str_cli возвращает управление (см. листинг 5.4).
2. Когда функция str_cli возвращает управление клиентской функции main (см. листинг 5.3), последняя завершает работу, вызывая функцию exit.

3. При завершении процесса выполняется закрытие всех открытых дескрипторов, так что клиентский сокет закрывается ядром. При этом серверу посыпается сегмент FIN, на который TCP сервера отвечает сегментом ACK. Это первая половина последовательности завершения работы соединения TCP. На этом этапе сокет сервера находится в состоянии CLOSE_WAIT, а клиентский сокет — в состоянии FIN_WAIT_2 (см. рис. 2.4 и 2.5).

4. Когда TCP сервера получает сегмент FIN, дочерний процесс сервера находится в состоянии ожидания в вызове функции `read` (см. листинг 5.2), а затем функция `read` возвращает нуль. Это заставляет функцию `str_echo` вернуть управление функции `main` дочернего процесса сервера.

5. Дочерний процесс сервера завершается с помощью вызова функции `exit` (см. листинг 5.1).

6. Все открытые дескрипторы в дочернем процессе сервера закрываются. Закрытие присоединенного сокета дочерним процессом вызывает отправку двух последних сегментов завершения соединения TCP: FIN от сервера клиенту и ACK от клиента (см. рис. 2.5). На этом этапе соединение полностью завершается. Клиентский сокет входит в состояние TIME_WAIT.

7. Другая часть завершения процесса относится к сигналу `SIGCHLD`. Он отправляется родительскому процессу, когда завершается дочерний процесс. Это происходит и в нашем примере, но мы не перехватываем данный сигнал в коде, и по умолчанию он игнорируется. Дочерний процесс входит в состояние зомби (zombie). Мы можем проверить это с помощью команды `ps`.

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
PID  PPID TT      STAT COMMAND          WCHAN
22038 22036 pts/6 S      -bash      read_chan
17870 22038 pts/6 S      ./tcpserver01 wait_for_connect
19315 17870 pts/6 Z      [tcpserver01 <defu do_exit]
```

Теперь дочерний процесс находится в состоянии Z (зомби).

Процессы-зомби нужно своевременно удалять, а это требует работы с сигналами Unix. Поэтому в следующем разделе мы сделаем обзор управления сигналами, а затем продолжим рассмотрение нашего примера.

5.8. Обработка сигналов POSIX

Сигнал — это уведомление процесса о том, что произошло некое событие. Иногда сигналы называют *программными прерываниями* (*software interrupts*). Подразумевается, что процесс не знает заранее о том, когда придет сигнал.

Сигналы могут посыпаться в следующих направлениях:

- одним процессом другому процессу (или самому себе);
- ядром процессу.

Сигнал `SIGCHLD`, упомянутый в конце предыдущего раздела, ядро посыпает родительскому процессу при завершении дочернего.

Для каждого сигнала существует определенное действие (*action* или *disposition* — *характер*). Действие, соответствующее сигналу, задается с помощью вызова функции `sigaction` (ее описание следует далее) и может быть выбрано тремя способами:

1. Мы можем предоставить функцию, которая вызывается при перехвате определенного сигнала. Эта функция называется *обработчиком сигнала* (*signal handler*), а действие называется *перехватыванием сигнала* (*catching*). Сигналы `SIGKILL` и `SIGSTOP` перехватить нельзя. Наша функция вызывается с одним целочисленным аргументом, который является номером сигнала, и ничего не возвращает. Следовательно, прототип этой функции имеет вид:

```
void handler(int signo);
```

Для большинства сигналов вызов функции `sigaction` и задание функции, вызываемой при получении сигнала, — это все, что требуется для обработки сигнала. Но дальше вы увидите, что для перехватывания некоторых сигналов, в частности `SIGIO`, `SIGPOLL` и `SIGURG`, требуются дополнительные действия со стороны процесса.

2. Мы можем *игнорировать* сигнал, если действие задать как `SIG_IGN`. Сигналы `SIGKILL` и `SIGSTOP` не могут быть проигнорированы.

3. Мы можем установить действие для сигнала *по умолчанию*, задав его как `SIG_DFL`. Действие сигнала по умолчанию обычно заключается в завершении процесса по получении сигнала, а некоторые сигналы генерируют копию области памяти процесса в его текущем каталоге (так называемый *дамп* — *core dump*). Есть несколько сигналов, для которых действием по умолчанию является игнорирование. Например,

`SIGCHLD` и `SIGURG` (посыдается по получении внеполосных данных, см. главу 24) — это два сигнала, игнорируемых по умолчанию, с которыми мы встретимся в тексте.

Функция signal

Согласно POSIX, чтобы определить действие для сигнала, нужно вызывать функцию `sigaction`. Однако это достаточно сложно, поскольку один аргумент этой функции — это структура, для которой необходимо выделение памяти и заполнение. Поэтому проще задать действие сигнала с помощью функции `signal`. Первый ее аргумент — это имя сигнала, а второй — либо указатель на функцию, либо одна из констант `SIG_IGN` и `SIG_DFL`. Но функция `signal` существовала еще до появления POSIX.1, и ее различные реализации имеют разную семантику сигналов с целью обеспечения обратной совместимости. В то же время POSIX четко диктует семантику при вызове функции `sigaction`. Это обеспечивает простой интерфейс с соблюдением семантики POSIX. Мы включили эту функцию в нашу собственную библиотеку вместе функциями `err_xxx` и функциями-обертками, которые мы используем для построения всех наших программ. Она представлена в листинге 5.5. Функция-обертка `Signal` здесь не показана, потому что ее вид не зависит от того, какую именно функцию `signal` она должна вызывать.

Листинг 5.5. Функция `signal`, вызывающая функцию POSIX `sigaction`

```
//lib/signal.c
1 #include "unp.h"

2 Sigfunc*
3 signal(int signo, Sigfunc *func)
4 {
5     struct sigaction act, oact;

6     act.sa_handler = func;
7     sigemptyset(&act.sa_mask);
8     act.sa_flags = 0;
9     if (signo == SIGALRM) {
10 #ifdef SA_INTERRUPT
11     act.sa_flags |= SA_INTERRUPT; /* SunOS 4.x */
12 #endif
13 } else {
14 #ifdef SA_RESTART
15     act.sa_flags |= SA_RESTART; /* SVR4, 44BSD */
16 #endif
17 }
18 if (sigaction(signo, &act, &oact) < 0)
19     return (SIG_ERR);
20     return (oact.sa_handler);
21 }
```

Упрощение прототипа функции при использовании `typedef`

2-3 Обычный прототип для функции `signal` усложняется наличием вложенных скобок:

```
void (*signal(int signo, void (*func)(int)))(int);
```

Чтобы упростить эту запись, мы определяем тип `Sigfunc` в нашем заголовочном файле `unp.h` следующим образом:

```
typedef void Sigfunc(int);
```

указывая тем самым, что обработчики сигналов — это функции с целочисленным аргументом, ничего не возвращающие (`void`). Тогда прототип функции выглядит следующим образом:

```
Sigfunc *signal(int signo, Sigfunc *func);
```

Указатель на функцию, являющуюся обработчиком сигнала, — это второй аргумент функции и в то же время возвращаемое функцией значение.

Установка обработчика

6 Элемент `sa_handler` структуры `sigaction` устанавливается равным аргументу `func` функции `signal`.

Установка маски сигнала для обработчика

7 POSIX позволяет нам задавать набор сигналов, которые будут блокированы при вызове обработчика сигналов. Любой блокируемый сигнал не может быть доставлен процессу. Мы устанавливаем элемент `sa_mask` равным пустому набору. Это означает, что во время работы обработчика дополнительные сигналы не блокируются. POSIX гарантирует, что перехватываемый сигнал всегда блокирован, пока выполняется его обработчик.

Установка флага `SA_RESTART`

8-17 Флаг `SA_RESTART` не является обязательным, и если он установлен, то системный вызов, прерываемый этим сигналом, будет автоматически снова выполнен ядром. (В продолжении нашего примера мы более подробно поговорим о прерванных системных вызовах.) Если перехватываемый сигнал не является сигналом `SIGALRM`, мы задаем флаг `SA_RESTART`, если таковой определен. (Причина, по которой сигнал `SIGALRM` обрабатывается отдельно, состоит в том, что обычно цель его генерации - ввести ограничение по времени в операцию ввода-вывода, как показано в листинге 14.2. В этом случае мы хотим, чтобы блокированный системный вызов был прерван сигналом.) Более ранние системы, особенно SunOS 4.x, автоматически перезапускают прерванный системный вызов по умолчанию и затем определяют флаг `SA_INTERRUPT`. Если этот флаг задан, мы устанавливаем его при перехвате сигнала `SIGALRM`.

Вызов функции `sigaction`

18-20 Мы вызываем функцию `sigaction`, а затем возвращаем старое действие сигнала как результат функции `signal`.

В книге мы везде используем функцию `signal` из листинга 5.5.

Семантика сигналов POSIX

Сведем воедино следующие моменты, относящиеся к обработке сигналов в системе, совместимой с POSIX.

- Однажды установленный обработчик сигналов остается установленным (в более ранних системах обработчик сигналов удалялся каждый раз по выполнении).
- На время выполнения функции — обработчика сигнала доставляемый сигнал блокируется. Более того, любые дополнительные сигналы, заданные в наборе сигналов `sa_mask`, переданном функции `sigaction` при установке обработчика, также блокируются. В листинге 5.5 мы устанавливаем `sa_mask` равным пустому набору, что означает, что никакие сигналы, кроме перехватываемого, не блокируются.
- Если сигнал генерируется один или несколько раз, пока он блокирован, то обычно после разблокирования он доставляется только один раз, то есть по умолчанию сигналы Unix не устанавливаются в очередь. Пример мы рассмотрим в следующем разделе. Стандарт POSIX реального времени 1003.1b определяет набор надежных сигналов, которые помещаются в очередь, но в этой книге мы их не используем.
- Существует возможность выборочного блокирования и разблокирования набора сигналов с помощью функции `sigprocmask`. Это позволяет нам защитить критическую область кода, не допуская перехватывания определенных сигналов во время ее выполнения.

5.9. Обработка сигнала `SIGCHLD`

Назначение состояния зомби — сохранить информацию о дочернем процессе, чтобы родительский процесс мог ее впоследствии получить. Эта информация включает идентификатор дочернего процесса, статус завершения и данные об использовании ресурсов (время процессора, память и т.д.). Если у завершающегося процесса есть дочерний процесс в зомбированном состоянии, идентификатору родительского процесса всех зомбированных дочерних процессов присваивается значение 1 (процесс `init`), что позволяет унаследовать дочерние процессы и сбросить их (то есть процесс `init` будет ждать (`wait`) их завершения, благодаря чему будут удалены зомби). Некоторые системы Unix в столбце COMMAND выводят для зомбированных процессов значение `<defunct>`.

Обработка зомбированных процессов

Очевидно, что нам не хотелось бы оставлять процессы в виде зомби. Они занимают место в ядре, и в конце концов у нас может не остаться идентификаторов для нормальных процессов. Когда мы выполняем функцию `fork` для дочерних процессов, необходимо с помощью функции `wait` дождаться их завершения, чтобы они не превратились в зомби. Для этого мы устанавливаем обработчик сигналов для перехватывания сигнала `SIGCHLD` и внутри обработчика вызываем функцию `wait`. (Функции `wait` и `waitpid` мы опишем в разделе 5.10.) Обработчик сигналов мы устанавливаем с помощью вызова функции

`Signal(SIGCHLD, sig_chld);`

в листинге 5.1, после вызова функции `listen`. (Необходимо сделать это до вызова функции `fork` для первого дочернего процесса, причем только один раз.) Затем мы определяем обработчик сигнала — функцию `sig_chld`, представленную в листинге 5.6.

Листинг 5.6. Версия обработчика сигнала `SIGCHLD`,зывающая функцию `wait` (усовершенствованная версия находится в листинге 5.8)

```
//tcpcliserv/sigchldwait.c
1 #include "unp.h"

2 void
3 sig_chld(int signo)
4 {
5     pid_t pid;
6     int stat;

7     pid = wait(&stat);
8     printf("child terminated\n", pid);
9     return;
10 }
```

ВНИМАНИЕ

В обработчике сигналов не рекомендуется вызов стандартных функций ввода-вывода, таких как `printf`, по причинам, изложенным в разделе 11.18. В данном случае мы вызываем функцию `printf` как средство диагностики, чтобы увидеть, когда завершается дочерний процесс.

В системах System V и Unix 98 дочерний процесс не становится зомби, если процесс задает действие `SIG_IGN` для `SIGCHLD`. К сожалению, это верно только для System V и Unix 98. В POSIX прямо сказано, что такое поведение этим стандартом не предусмотрено. Переносимый способ обработки зомби состоит в том, чтобы перехватывать сигнал `SIGCHLD` и вызывать функцию `wait` или `waitpid`.

Если мы откомпилируем в Solaris 9 программу, представленную в листинге 5.1, вызывая функцию `Signal` с нашим обработчиком `sig_chld`, и будем использовать функцию `signal` из системной библиотеки (вместо нашей версии, показанной в листинге 5.5), то получим следующее:

```
solaris % tcpser02 & запускаем сервер в фоновом режиме
[2] 16939
solaris % tcpcli01 127.0.0.1 затем клиент
hi there набираем эту строку
```

```
hi there и она отражается сервером
^D      вводим символ конца файла
child 16942 terminated функция printf из обработчика сигнала выводит эту строку
accept error: Interrupted system call но функция main преждевременно прекращает выполнение
Последовательность шагов в этом примере такова:
```

1. Мы завершаем работу клиента, вводя символ EOF. TCP клиента посыпает сегмент FIN серверу, и сервер отвечает сегментом ACK.

2. Получение сегмента FIN доставляет EOF ожидающей функции readline дочернего процесса. Дочерний процесс завершается.

3. Родительский процесс блокирован в вызове функции accept, когда доставляется сигнал SIGCHLD. Функция sig_chld (наш обработчик сигнала) выполняется, функция wait получает PID дочернего процесса и статус завершения, после чего из обработчика сигнала вызывается функция printf. Обработчик сигнала возвращает управление.

4. Поскольку сигнал был перехвачен родительским процессом, в то время как родительский процесс был блокирован в *медленном* (см. ниже) системном вызове (функция accept), ядро заставляет функцию accept возвратить ошибку EINTR (прерванный системный вызов). Родительский процесс не обрабатывает эту ошибку корректно (см. листинг 5.1), поэтому функция main преждевременно завершается.

Цель данного примера — показать, что при написании сетевых программ, перехватывающих сигналы, необходимо получать информацию о прерванных системных вызовах и обрабатывать их. В этом специфичном для Solaris 2.5 примере функция signal из стандартной библиотеки C не осуществляет автоматический перезапуск прерванного вызова, то есть флаг SA_RESTART, установленный нами в листинге 5.5, не устанавливается функцией signal из системной библиотеки. Некоторые другие системы автоматически перезапускают прерванный системный вызов. Если мы запустим тот же пример в 4.4BSD, используя ее библиотечную версию функции signal, ядро перезапустит прерванный системный вызов и функция accept не возвратит ошибки. Одна из причин, по которой мы определяем нашу собственную версию функции signal и используем ее далее, — решение этой потенциальной проблемы, возникающей в различных операционных системах (см. листинг 5.5).

Кроме того, мы всегда программируем явную функцию return для наших обработчиков сигналов (см. листинг 5.6), даже если функция ничего не возвращает (void), чтобы этот оператор напоминал нам о возможности прерывания системного вызова при возврате из обработчика.

Обработка прерванных системных вызовов

Термином *медленный системный вызов* (*slow system call*), введенным при описании функции accept, мы будем обозначать любой системный вызов, который может быть заблокирован навсегда. Такой системный вызов может никогда не завершиться. В эту категорию попадает большинство сетевых функций. Например, нет никакой гарантии, что вызов функции accept сервером когда-нибудь будет завершен, если нет клиентов, которые соединяются с сервером. Аналогично, вызов нашим сервером функции read (из readline) в листинге 5.2 никогда не возвратит управление, если клиент никогда не пошлет серверу строку для отражения. Другие примеры медленных системных вызовов — чтение и запись в случае программных каналов и терминальных устройств. Важным исключением является дисковый ввод-вывод, который обычно завершается возвращением управления вызвавшему процессу (в предположении, что не происходит фатальных аппаратных ошибок).

Основное применяемое здесь правило связано с тем, что когда процесс, блокированный в медленном системном вызове, перехватывает сигнал, а затем обработчик сигналов завершает работу, системный вызов может возвратить ошибку EINTR. Некоторые ядра автоматически перезапускают некоторые прерванные системные вызовы. Для обеспечения переносимости программ, перехватывающих сигналы (большинство параллельных серверов перехватывает сигналы SIGCHLD), следует учесть, что медленный системный вызов может возвратить ошибку EINTR. Проблемы переносимости связаны с написанными выше словами «могут» и «некоторые» и тем фактом, что поддержка флага POSIX SA_RESTART не является обязательной. Даже если реализация поддерживает флаг SA_RESTART, не все прерванные системные вызовы могут автоматически перезапуститься. Например, большинство реализаций, происходящих от Беркли, никогда не перезапускают функцию select, а некоторые из этих реализаций никогда не перезапускают функции accept и recvfrom.

Чтобы обработать прерванный вызов функции accept, мы изменяем вызов функции accept, приведенной в листинге 5.1, в начале цикла for следующим образом:

```
for (;;) {
    clilen = sizeof(cliaddr);
    if ((connfd = accept(listenfd, (SA*)&cliaddr, &clilen)) < 0) {
        if (errno == EINTR)
            continue; /* назад в for() */
        else
            err_sys("accept error");
    }
}
```

Обратите внимание, что мы вызываем функцию accept, а не функцию-обертку Accept, поскольку мы должны обработать неудачное выполнение функции самостоятельно.

В этой части кода мы сами перезапускаем прерванный системный вызов. Это допустимо для функции accept и таких функций, как read, write, select и open. Но есть функция, которую мы не можем перезапустить самостоятельно, — это функция connect. Если она возвращает ошибку EINTR, мы не можем снова вызвать ее, поскольку в этом случае немедленно возвратится еще одна ошибка. Когда функция connect прерывается перехваченным сигналом и не перезапускается автоматически, нужно вызывать функцию select, чтобы дождаться завершения соединения (см. раздел 16.3).

5.10. Функции wait и waitpid

В листинге 5.7 мы вызываем функцию wait для обработки завершенного дочернего процесса.

```
#include <sys/wait.h>
```

```
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

Обе функции возвращают ID процесса в случае успешного выполнения, -1 в случае ошибки

Обе функции, и wait, и waitpid, возвращают два значения. Возвращаемое значение каждой из этих функций — это идентификатор завершенного дочернего процесса, а через указатель statloc передается статус завершения дочернего процесса (целое число). Для проверки статуса завершения можно вызвать три макроса, которые сообщают нам, что произошло с дочерним процессом: дочерний процесс завершен normally, уничтожен сигналом или только приостановлен программой управления заданиями (job-control). Дополнительные макросы позволяют получить состояние выхода дочернего процесса, а также значение сигнала, уничтожившего или остановившего процесс. В листинге 15.8 мы используем макроопределения WIFEXITED и WEXITSTATUS.

Если у процесса, вызывающего функцию wait, нет завершенных дочерних процессов, но есть один или несколько выполняющихся, функция wait блокируется до тех пор, пока первый из дочерних процессов не завершится.

Функция waitpid предоставляет более гибкие возможности выбора ожидаемого процесса и его блокирования. Прежде всего, в аргументе pid задается идентификатор процесса, который мы будем ожидать. Значение -1 говорит о том, что нужно дождаться завершения первого дочернего процесса. (Существуют и другие значения идентификаторов процесса, но здесь они нам не понадобятся.) Аргумент options позволяет задавать дополнительные параметры. Наиболее общепротребительным является параметр WNOHANG: он сообщает ядру, что не нужно выполнять блокирование, если нет завершенных дочерних процессов.

Различия между функциями wait и waitpid

Теперь мы проиллюстрируем разницу между функциями wait и waitpid, используемыми для сброса завершенных дочерних процессов. Для этого мы изменим код нашего клиента TCP так, как показано в листинге 5.7. Клиент устанавливает пять соединений с сервером, а затем использует первое из них (sockfd[0]) в вызове функции str_cli. Несколько соединений мы устанавливаем для того, чтобы породить от параллельного сервера множество дочерних процессов, как показано на рис. 5.2.

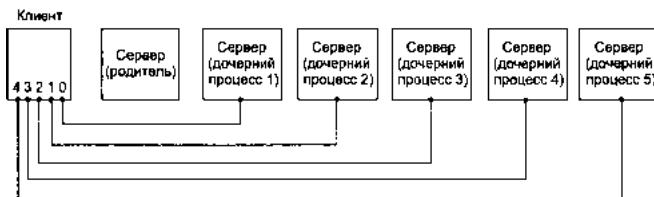


Рис. 5.2. Клиент, установивший пять соединений с одним и тем же параллельным сервером

Листинг 5.7. Клиент TCP, устанавливающий пять соединений с сервером

```
//tcpcliserv/tcpcli04.c
```

```
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int i, sockfd[5];
6     struct sockaddr_in servaddr;

7     if (argc != 2)
8         err_quit("usage: tcpcli <Ipaddress>");

9     for (i = 0; i < 5; i++) {
10        sockfd[i] = Socket(AF_INET, SOCK_STREAM, 0);

11        bzero(&servaddr, sizeof(servaddr));
12        servaddr.sin_family = AF_INET;
13        servaddr.sin_port = htons(SERV_PORT);
14        Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

15        Connect(sockfd[i], (SA*)&servaddr, sizeof(servaddr));
16    }

17    str_cli(stdin, sockfd[0]); /* эта функция выполняет все необходимые
                               действия для формирования запроса клиента */

18    exit(0);
19 }
```

Когда клиент завершает работу, все открытые дескрипторы автоматически закрываются ядром (мы не вызываем функцию `close`, а пользуемся только функцией `exit`) и все пять соединений завершаются приблизительно в одно и то же время. Это вызывает отправку пяти сегментов FIN, по одному на каждое соединение, что, в свою очередь, вызывает примерно одновременное завершение всех пяти дочерних процессов. Это приводит к доставке пяти сигналов SIGCHLD практически в один и тот же момент, что показано на рис. 5.3.

Доставка множества экземпляров одного и того же сигнала вызывает проблему, к рассмотрению которой мы и приступим.

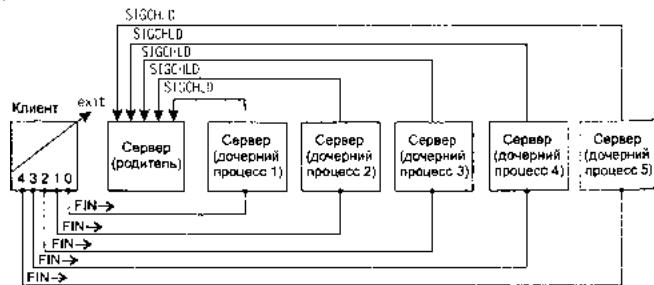


Рис. 5.3. Клиент завершает работу, закрывая все пять соединений и завершая все пять дочерних процессов

Сначала мы запускаем сервер в фоновом режиме, а затем — новый клиент. Наш сервер, показанный в листинге 5.1, несколько модифицирован — теперь в нем вызывается функция `signal` для установки обработчика сигнала `SIGCHLD`, приведенного в листинге 5.6.

```
linux % tcperv03 &
[1] 20419
linux % tcpcli04 206.62.226.35
hello мы набираем эту строку
hello и она отражается сервером
^D мы набираем символ конца файла
child 20426 terminated выводится сервером
```

Первое, что мы можем заметить, — данные выводит только одна функция `printf`, хотя мы предполагаем, что все пять дочерних процессов должны завершиться. Если мы выполним программу `ps`, то увидим, что другие четыре дочерних процессы все еще существуют как зомби.

```
PID TTY TIME CMD
20419 pts/6 00:00:00 tcperv03
20421 pts/6 00:00:00 tcperv03 <defunct>
20422 pts/6 00:00:00 tcperv03 <defunct>
20423 pts/6 00:00:00 tcperv03 <defunct>
```

Установки обработчика сигнала и вызова функции `wait` из этого обработчика недостаточно для предупреждения появления зомби. Проблема состоит в том, что все пять сигналов генерируются до того, как выполняется обработчик сигнала, и вызывается он только один раз, поскольку сигналы Unix обычно не помещаются в очередь. Более того, эта проблема является недетерминированной. В приведенном примере с клиентом и сервером на одном и том же узле обработчик сигнала выполняется один раз, оставляя четыре зомби. Но если мы запустим клиент и сервер на разных узлах, то обработчик сигналов, скорее всего, выполнится дважды: один раз в результате генерации первого сигнала, а поскольку другие четыре сигнала приходят во время выполнения обработчика, он вызывается повторно только один раз. При этом остаются три зомби. Но иногда в зависимости от точного времени получения сегментов FIN на узле сервера обработчик сигналов может выполниться три или даже четыре раза.

Правильным решением будет вызывать функцию `waitpid` вместо `wait`. В листинге 5.8 представлена версия нашей функции `sigchld`, корректно обрабатывающая сигнал `SIGCHLD`. Эта версия работает, потому что мы вызываем функцию `waitpid` в цикле, получая состояние любого из дочерних процессов, которые завершились. Необходимо задать параметр `WNOHANG`: это указывает функции `waitpid`, что не нужно блокироваться, если существуют выполняемые дочерние процессы, которые еще не завершились. В листинге 5.6 мы не могли вызвать функцию `wait` в цикле, поскольку нет возможности предотвратить блокирование функции `wait` при наличии выполняемых дочерних процессов, которые еще не завершились.

В листинге 5.9 показана окончательная версия нашего сервера. Он корректно обрабатывает возвращение ошибки `EINTR` из функции `accept` и устанавливает обработчик сигнала (листинг 5.8), который вызывает функцию `waitpid` для всех завершенных дочерних процессов.

Листинг 5.8. Окончательная (корректная) версия функции `sig_chld`,зывающая функцию `waitpid`

```
//tcpcliserv/sigchldwaitpid.c
```

```
1 #include "unp.h"

2 void
3 sig_chld(int signo)
4 {
5     pid_t pid;
6     int stat;

7     while ((pid = waitpid(-1, &stat, WNOHANG)) >0)
8         printf("child %d terminated\n", pid);
9     return;
10 }
```

Листинг 5.9. Окончательная (корректная) версия TCP-сервера, обрабатывающего ошибку `EINTR` функции `accept`

```
//tcpcliserv/tcperv04.c
```

```

1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int listenfd, connfd;
6     pid_t childpid;
7     socklen_t clilen;
8     struct sockaddr_in cliaddr, servaddr;
9     void sig_chld(int);

10    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

11    bzero(&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14    servaddr.sin_port = htons(SERV_PORT);

15    Bind(listenfd, (SA*)&servaddr, sizeof(servaddr));

16    Listen(listenfd, LISTENQ);
17    Signal(SIGCHLD, sig_chld); /* нужно вызвать waitpid() */

18    for (;;) {
19        clilen = sizeof(cliaddr);
20        if ((connfd = accept(listenfd, (SA*)&cliaddr, &clilen)) < 0) {
21            if (errno == EINTR)
22                continue; /* назад к for() */
23            else
24                err_sys("accept error");
25        }
26        if ((childpid = Fork()) == 0) { /* дочерний процесс */
27            Close(listenfd); /* закрываем прослушиваемый сокет */
28            str_echo(connfd); /* обрабатываем запрос */
29            exit(0);
30        }
31        Close(connfd); /* родитель закрывает присоединенный сокет */
32    }
33 }

```

Целью этого раздела было продемонстрировать три сценария, которые могут встретиться в сетевом программировании.

1. При выполнении функции `fork`, порождающей дочерние процессы, следует перехватывать сигнал `SIGCHLD`.

2. При перехватывании сигналов мы должны обрабатывать прерванные системные вызовы.

3. Обработчик сигналов `SIGCHLD` должен быть создан корректно с использованием функции `waitpid`, чтобы не допустить появления зомби.

Окончательная версия нашего сервера TCP (см. листинг 5.9) вместе с обработчиком сигналов `SIGCHLD` в листинге 5.8 обрабатывает все три сценария.

5.11. Прерывание соединения перед завершением функции `accept`

Существует другое условие, аналогичное прерванному системному вызову, пример которого был описан в предыдущем разделе. Оно может привести к возвращению функцией `accept` нефатальной ошибки, в случае чего следует заново вызвать функцию `accept`. Последовательность пакетов, показанная на рис. 5.4, встречается на загруженных серверах (эта последовательность типична для загруженных веб-серверов).

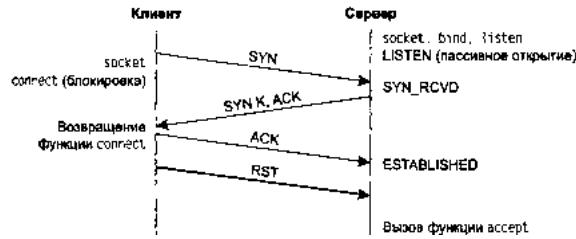


Рис. 5.4. Получение сегмента RST для состояния соединения ESTABLISHED перед вызовом функции accept

Трехэтапное рукопожатие TCP завершается, устанавливается соединение, а затем TCP клиента посыпает сегмент RST. На стороне сервера соединение ставится в очередь в ожидании вызова функции accept, и в это время сервер получает сегмент RST. Спустя некоторое время процесс сервера вызывает функцию accept.

К сожалению, принцип обработки прерванного соединения зависит от реализации. Реализации, происходящие от Беркли, обрабатывают прерванное соединение полностью внутри ядра, и сервер никогда не узнает об этом. Большинство реализаций SVR4, однако, возвращают процессу ошибку, и эта ошибка зависит от реализации. При этом переменная errno принимает значение EPROTO (ошибка протокола), хотя в POSIX указано, что должна возвращаться ошибка ECONNABORTED (прерывание соединения). POSIX определяет эту ошибку иначе, так как ошибка EPROTO возвращается еще и в том случае, когда в подсистеме потоков происходят какие-либо фатальные события, имеющие отношение к протоколу. Возвращение той же ошибки для нефатального прерывания установленного соединения клиентом приводит к тому, что сервер не знает, вызывать снова функцию accept или нет. В случае ошибки ECONNABORTED сервер может игнорировать ошибку и снова вызывать функцию accept.

ПРИМЕЧАНИЕ

Этот сценарий очень просто имитировать. Запустите сервер, который должен вызывать функции socket, bind и listen, а затем перед вызовом функции accept переведите сервер на короткое время в состояние ожидания. Пока процесс сервера находится в состоянии ожидания, запустите клиент, который вызовет функции socket и connect. Как только функция connect завершится, установите параметр сокета SOLINGER, чтобы сгенерировать сегмент RST (который мы описываем в разделе 7.5 и демонстрируем в листинге 16.14), и завершите процессы.

ПРИМЕЧАНИЕ

В [128] описана обработка этой ошибки в Беркли-ядрах (Berkeley-derived kernels), которые никогда не передают ее процесссу. Обработка RST с вызовом функции tcp_close представлена в [128, с. 964]. Эта функция вызывает функцию in_pcdbdetach [128, с. 897], которая, в свою очередь, вызывает функцию soffree [128, с. 719]. Функция soffree [128, с. 473] обнаруживает, что сокет все еще находится в очереди полностью установленных соединений прослушиваемого сокета. Она удаляет этот сокет из очереди и освобождает сокет. Когда сервер, наконец, вызовет функцию accept, он не сможет узнать, что установленное соединение было удалено из очереди.

Мы вернемся к подобным прерванным соединениям в разделе 16.6 и покажем, какие проблемы они могут порождать совместно с функцией select и прослушиваемым сокетом в нормальном режиме блокирования.

5.12. Завершение процесса сервера

Теперь мы запустим соединение клиент-сервер и уничтожим дочерний процесс сервера. Это симулирует сбой процесса сервера, благодаря чему мы сможем выяснить, что происходит с клиентом в подобных ситуациях. (Следует точно различать сбой процесса сервера, который мы рассмотрим здесь, и сбой на самом узле сервера, о котором речь пойдет в разделе 5.14.) События развиваются так:

1. Мы запускаем сервер и клиент на разных узлах и вводим на стороне клиента одну строку, чтобы проверить, все ли в порядке. Стока отражается дочерним процессом сервера.

2. Мы находим идентификатор дочернего процесса сервера и уничтожаем его с помощью программы `kill`. Одним из этапов завершения процесса является закрытие всех открытых дескрипторов в дочернем процессе. Это вызывает отправку сегмента FIN клиенту, и TCP клиента отвечает сегментом ACK. Это первая половина завершения соединения TCP.

3. Родительскому процессу сервера посыпается сигнал `SIGCHLD`, и он корректно обрабатывается (см. листинг 5.9).

4. С клиентом ничего не происходит. TCP клиента получает от TCP сервера сегмент FIN и отвечает сегментом ACK, но проблема состоит в том, что клиентский процесс блокирован в вызове функции `fgets` в ожидании строки от терминала.

5. Запуск программы `netstat` на этом шаге из другого окна на стороне клиента показывает состояние клиентского сокета:

```
linux % netstat -a | grep 9877
tcp 0 0 *:9877          *:*          LISTEN
tcp 0 0 localhost:9877  localhost:9877 FIN_WAIT2
tcp 1 0 localhost.43604 localhost:9877 CLOSE_WAIT
```

Как видите, согласно рис. 2.4, осуществлялась половина последовательности завершения соединения TCP.

6. Мы можем снова ввести строку на стороне клиента. Вот что происходит на стороне клиента (начиная с шага 1):

```
linux % tcpcli01 127.0.0.1 запускаем клиент
hello первая строка, которую мы ввели
hello она корректно отражается
теперь мы уничтожаем (kill) дочерний процесс
сервера на узле сервера
another line затем мы вводим следующую строку на стороне клиента
str_cli: server terminated prematurely
```

Когда мы вводим следующую строку, функция `str_cli` вызывает функцию `writen`, и TCP клиента отправляет данные серверу. TCP это допускает, поскольку получение сегмента FIN протоколом TCP клиента указывает только на то, что процесс сервера закрыл свой конец соединения и больше не будет отправлять данные. Получение сегмента FIN не сообщает протоколу TCP клиента, что процесс сервера завершился (хотя в данном случае он завершился). Мы вернемся к этому вопросу в разделе 6.6, когда будем говорить о половинном закрытии TCP.

Когда TCP сервера получает данные от клиента, он отвечает сегментом RST, поскольку процесс, у которого был открытый сокет, завершился. Мы можем проверить, что этот сегмент RST отправлен, просмотрев пакеты с помощью программы `tcpdump`.

7. Однако процесс клиента не увидит сегмента RST, поскольку он вызывает функцию `readline` сразу же после вызова функции `writen`, и `readline` сразу же возвращает 0 (признак конца файла) по причине того, что на шаге 2 был получен сегмент FIN. Наш клиент не предполагает получать признак конца файла на этом этапе (см. листинг 5.3), поэтому он завершает работу, сообщая об ошибке `Server terminated prematurely` (Сервер завершил работу преждевременно).

ПРИМЕЧАНИЕ

Этапы описанной последовательности также зависят от синхронизации времени. Вызов `readline` на стороне клиента может произойти до получения им пакета RST от сервера, но может произойти и после. Если `readline` вызывается до получения RST, происходит то, что мы описали выше (клиент считывает символ конца файла). Если же первым будет получен пакет RST, функция `readline` возвратит ошибку ECONNRESET (соединение сброшено собеседником).

8. Когда клиент завершает работу (вызывая функцию `err_quit` в листинге 5.4), все его открытые дескрипторы закрываются.

Проблема заключается в том, что клиент блокируется в вызове функции `fgets`, когда сегмент FIN приходит на сокет. Клиент в действительности работает с двумя дескрипторами — дескриптором сокета и

дескриптором ввода пользователя, и поэтому он должен блокироваться при вводе из любого источника (сейчас в функции `str_cli` он блокируется при вводе только из одного источника). Обеспечить подобное блокирование — это одно из назначений функций `select` и `poll`, о которых рассказывается в главе 6. Когда в разделе 6.4 мы перепишем функцию `str_cli`, то как только мы уничтожим с помощью программы `kill` дочерний процесс сервера, клиенту будет отправлено уведомление о полученном сегменте FIN.

5.13. Сигнал SIGPIPE

Что происходит, если клиент игнорирует возвращение ошибки из функции `readline` и отсылает следующие данные серверу? Это может произойти, если, например, клиенту нужно выполнить две операции по отправке данных серверу перед считыванием данных от него, причем первая операция отправки данных вызывает RST.

Применяется следующее правило: когда процесс производит запись в сокет, получивший сегмент RST, процессу посыпается сигнал `SIGPIPE`. По умолчанию действием этого сигнала является завершение процесса, так что процесс должен перехватить сигнал, чтобы не произошло непроизвольного завершения.

Если процесс либо перехватывает сигнал и возвращается из обработчика сигнала, либо игнорирует сигнал, то операция записи возвращает ошибку `EPIPE`.

ПРИМЕЧАНИЕ

Часто задаваемым вопросом (FAQ) в Usenet является такой: как получить этот сигнал при первой, а не при второй операции записи? Это невозможно. Как следует из приведенных выше рассуждений, первая операция записи выявляет сегмент RST, а вторая — сигнал. Если запись в сокет, получивший сегмент FIN, допускается, то запись в сокет, получивший сегмент RST, является ошибочной.

Чтобы увидеть, что происходит с сигналом `SIGPIPE`, изменим код нашего клиента так, как показано в листинге 5.10.

Листинг 5.10. Функция `str_cli`, дважды вызывающая функцию `writen`

```
//tcpcliserv/str_cli11.c
1 #include "unp.h"

2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     char sendline[MAXLINE], recvline[MAXLINE];

6     while (Fgets(sendline, MAXLINE, fp) != NULL) {

7         Writen(sockfd, sendline, 1);
8         sleep(1);
9         Writen(sockfd, sendline + 1, strlen(sendline) - 1);
10        if (Readline(sockfd, recvline, MAXLINE) == 0)
11            err_quit("str_cli, server terminated prematurely");

12        Fputs(recvline, stdout);
13    }
14 }
```

7-9 Все изменения, которые мы внесли, — это повторный вызов функции `writen`: сначала в сокет записывается первый байт данных, за этим следует пауза в 1 с и далее идет запись остатка строки. Наша цель — выявить сегмент RST при первом вызове функции `writen` и генерировать сигнал `SIGPIPE` при втором вызове.

Если мы запустим клиент на нашем узле Linux, мы получим:

```
linux % tcpcli11 127.0.0.1
hi there    мы вводим эту строку
```

```
hi there      и она отражается сервером  
здесь       мы завершаем дочерний процесс сервера  
bye        затем мы вводим эту строку  
Broken pipe это сообщение выводится интерпретатором
```

Мы запускаем клиент, вводим одну строку, видим, что строка отражена корректно, и затем завершаем дочерний процесс сервера на узле сервера. Затем мы вводим другую строку (`bye`), но ничего не отражается, а интерпретатор сообщает нам о том, что процесс получил сигнал `SIGPIPE`. Некоторые интерпретаторы не выводят никаких сообщений, если процесс завершает работу без дампа памяти, но в нашем примере использовался интерпретатор `bash`, который берет на себя эту работу.

Рекомендуемый способ обработки сигнала `SIGPIPE` зависит от того, что приложение собирается делать, когда получает этот сигнал. Если ничего особенного делать не нужно, проще всего установить действие `SIG_IGN`, предполагая, что последующие операции вывода перехватят ошибку `EPIPE` и завершатся. Если при появлении сигнала необходимо проделать специальные действия (возможно, запись в системный журнал), то сигнал следует перехватить и выполнить требуемые действия в обработчике сигнала. Однако отдавайте себе отчет в том, что если используется множество сокетов, то при доставке сигнала мы не получаем информации о том, на каком сокете произошла ошибка. Если нам нужно знать, какая именно операция `write` вызвала ошибку, следует либо игнорировать сигнал, либо вернуть управление из обработчика сигнала и обработать ошибку `EPIPE` из функции `write`.

5.14. Сбой на узле сервера

В следующем примере мы проследим за тем, что происходит в случае сбоя на узле сервера. Чтобы мы могли имитировать эту ситуацию, клиент и сервер должны работать на разных узлах. Мы запускаем сервер, запускаем клиент, вводим строку на стороне клиента для проверки работоспособности соединения, отсоединяем узел сервера от сети и вводим еще одну строку на стороне клиента. Этот сценарий охватывает также ситуацию, в которой узел сервера становится недоступен во время отправки данных клиентом (например, после того как соединение установлено, выключается некий промежуточный маршрутизатор).

События развиваются следующим образом:

1. Когда происходит сбой на узле сервера, по существующим сетевым соединениям от сервера не отправляется никакой информации. Мы считаем, что на узле происходит именно сбой, а не завершение работы компьютера оператором (что мы рассмотрим в разделе 5.16).

2. Мы вводим строку на стороне клиента, она записывается с помощью функции `writen` (см. листинг 5.3) и отправляется протоколом TCP клиента как сегмент данных. Затем клиент блокируется в вызове функции `readline` в ожидании отраженного ответа.

3. Если мы понаблюдаем за сетью с помощью программы `tcpdump`, то увидим, что TCP клиента последовательно осуществляет повторные передачи сегмента данных, пытаясь получить сегмент ACK от сервера. В разделе 25.11 [128] показан типичный образец повторных передач TCP: реализации, происходящие от Беркли, делают попытки передачи сегмента данных 12 раз, ожидая около 9 мин перед прекращением попыток. Когда TCP клиента наконец прекращает попытки ретрансляции (считая, что узел сервера за это время не перезагружался или что он все еще недоступен, если на узле сервера сбоя не было, но он был недоступен по сети), клиентскому процессу возвращается ошибка. Поскольку клиент блокирован в вызове функции `readline`, она и возвращает эту ошибку. Если на узле сервера произошел сбой, и на все сегменты данных клиента не было ответа, будет возвращена ошибка `ETIMEDOUT`. Но если некий промежуточный маршрутизатор определил, что узел сервера был недоступен, и ответил сообщением ICMP о недоступности получателя, клиент получит либо ошибку `ENHOSTUNREACH`, либо ошибку `ENETUNREACH`.

Хотя наш клиент в конце концов обнаруживает, что собеседник выключен или недоступен, бывает, что нужно определить это раньше, чем пройдут условленные девять минут. В таком случае следует поместить тайм-аут в вызов функции `readline`, о чем рассказывается в разделе 14.2.

В описанном сценарии сбой на узле сервера можно обнаружить, только послав данные на этот узел. Если мы хотим обнаружить сбой на узле сервера, не посыпая *данные*, требуется другая технология. Мы рассмотрим параметр сокета `SO_KEEPALIVE` в разделе 7.5.

5.15. Сбой и перезагрузка на узле сервера

В этом сценарии мы устанавливаем соединение между клиентом и сервером и затем считаем, что на узле сервера происходит сбой, после чего узел перезагружается. В предыдущем разделе узел сервера был выключен, когда мы отправляли ему данные. Здесь же перед отправкой данных серверу узел сервера перезагрузится. Простейший способ имитировать такую ситуацию — установить соединение, отсоединить сервер от сети, выключить узел сервера и перезагрузить его, а затем снова присоединить узел сервера к сети. Мы не хотим, чтобы клиент знал о завершении работы сервера (о такой ситуации речь пойдет в разделе 5.16).

Как было сказано в предыдущем разделе, если клиент не посыпает данные серверу, то он не узнает о произошедшем на узле сервера сбое. (При этом считается, что мы не используем параметр сокета `SO_KEEPALIVE`.) События развиваются следующим образом:

1. Мы запускаем сервер, затем — клиент, и вводим строку для проверки установленного соединения. Получаем ответ сервера.
2. Узел сервера выходит из строя и перезагружается.
3. Мы вводим строку на стороне клиента, которая посыпается как сегмент данных TCP на узел сервера.
4. Когда узел сервера перезагружается после сбоя, его TCP теряет информацию о существовавших до сбоя соединениях. Следовательно, TCP сервера отвечает на полученный от клиента сегмент данных, посыпая RST.
5. Наш клиент блокирован в вызове функции `readline`, когда приходит сегмент RST, заставляющий функцию `readline` возвратить ошибку `ECONNRESET`.

Если для нашего клиента важно диагностировать выход из строя узла сервера, даже если клиент активно не посыпает данные, то требуется другая технология (с использованием параметра сокета `SO_KEEPALIVE` или некоторых функций, проверяющих наличие связи в клиент-серверном соединении).

5.16. Выключение узла сервера

В двух предыдущих разделах рассматривался выход из строя узла сервера или недоступность узла сервера в сети. Теперь мы рассмотрим, что происходит, если узел сервера выключается оператором в то время, когда на этом узле выполняется наш серверный процесс.

Когда система Unix выключается, процесс `init` обычно посыпает всем процессам сигнал `SIGTERM` (мы можем перехватить этот сигнал), ждет в течение некоторого фиксированного времени (часто от 5 до 20 с), а затем посыпает сигнал `SIGKILL` (который мы перехватить не можем) всем еще выполняемым процессам. Это дает всем выполняемым процессам короткое время для завершения работы. Если мы не завершили выполнение процесса, это сделает сигнал `SIGKILL`. При завершении процесса закрываются все открытые дескрипторы, а затем мы проходим ту же последовательность шагов, что описывалась в разделе 5.12. Там же было отмечено, что в нашем клиенте следует использовать функцию `select` или `poll`, чтобы клиент определил завершение процесса сервера, как только оно произойдет.

5.17. Итоговый пример TCP

Прежде чем клиент и сервер TCP смогут взаимодействовать друг с другом, каждый из них должен определить пару сокетов для соединения: локальный IP-адрес, локальный порт, удаленный IP-адрес, удаленный порт. На рис. 5.5 мы схематически изображаем эти значения черными кружками. На этом рисунке ситуация представлена с точки зрения клиента. Удаленный IP-адрес и удаленный порт должны быть заданы клиентом при вызове функции `connect`. Два локальных значения обычно выбираются ядром тоже при вызове функции `connect`. У клиента есть выбор: он может задать только одно из локальных значений или оба, вызвав функцию `bind` перед вызовом функции `connect`, однако второй подход используется редко.

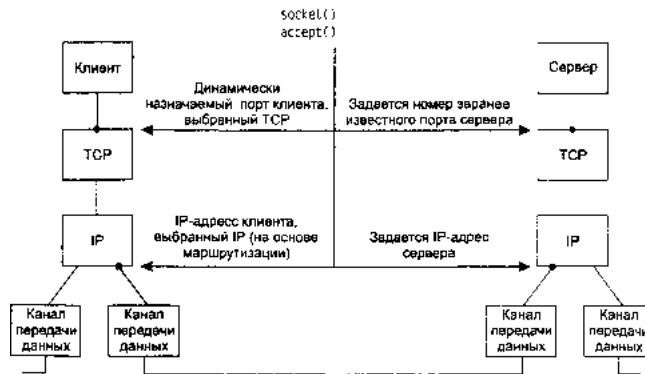


Рис. 5.5. TCP-соединение клиент-сервер с точки зрения клиента

Как мы отмечали в разделе 4.10, клиент может получить два локальных значения, выбранных ядром, вызвав функцию `getsockname` после установления соединения.

На рис. 5.6 показаны те же четыре значения, но с точки зрения сервера.



Рис. 5.6. TCP-соединение клиент-сервер с точки зрения сервера

Локальный порт (заранее известный порт сервера) задается функцией `bind`. Обычно сервер также задает в этом вызове универсальный IP-адрес, хотя может и ограничиться получением соединений, предназначенных для одного определенного локального интерфейса путем связывания с IP-адресом, записанным без символов подстановки (то есть не универсального). Если сервер связывается с универсальным IP-адресом на узле с несколькими сетевыми интерфейсами, он может определить локальный IP-адрес (указываемый как адрес отправителя в исходящих пакетах) при помощи вызова функции `getsockname` после установления соединения (см. раздел 4.10). Два значения удаленного адреса возвращаются серверу при вызове функции `accept`. Как мы отмечали в разделе 4.10, если сервером,зывающим функцию `accept`, выполняется с помощью функции `exec` другая программа, то эта программа может вызвать функцию `getpeername`, чтобы при необходимости определить IP-адрес и порт клиента.

5.18. Формат данных

В нашем примере сервер никогда не исследует запрос, который он получает от клиента. Сервер лишь читает все данные, включая символ перевода строки, и отправляет их обратно клиенту, отслеживая только разделитель строк. Это исключение, а не правило, так как обычно необходимо принимать во внимание формат данных, которыми обмениваются клиент и сервер.

Пример: передача текстовых строк между клиентом и сервером

Изменим наш сервер так, чтобы он, по-прежнему принимая текстовую строку от клиента, предполагал, что строка содержит два целых числа, разделенных пробелом, и возвращал сумму этих чисел. Функции `main` наших клиента и сервера остаются прежними, как и функция `str_cli`. Меняется только функция `str_echo`, что мы показываем в листинге 5.11.

Листинг 5.11. Функция `str_echo`, суммирующая два числа

```
//tcpcliserv/str_echo08.c
```

```

1 #include "unp.h"

2 void
3 str_echo(int sockfd)
4 {
5     long arg1, arg2;
6     ssize_t n;
7     char line[MAXLINE];

8     for (;;) {
9         if ((n = Readline(sockfd, line, MAXLINE)) == 0)
10             return; /* соединение закрывается удаленным концом */

11         if (sscanf(line, "%ld%ld", &arg1, &arg2) == 2)
12             snprintf(line, sizeof(line), "%ld\n", arg1 + arg2);
13         else
14             snprintf(line, sizeof(line), "input error\n");
15         n = strlen(line);
16         Writen(sockfd, line, n);
17     }
18 }
```

11-14 Мы вызываем функцию `sscanf`, чтобы преобразовать два аргумента из текстовых строк в целые числа типа `long`, а затем функцию `snprintf` для преобразования результата в текстовую строку.

Эти клиент и сервер работают корректно вне зависимости от порядка байтов на их узлах.

Пример: передача двоичных структур между клиентом и сервером

Теперь мы изменим код клиента и сервера, чтобы передавать через сокет не текстовые строки, а двоичные значения. Мы увидим, что клиент и сервер работают некорректно, когда они запущены на узлах с различным порядком байтов или на узлах с разными размерами целого типа `long` (см. табл. 1.5).

Функции `main` наших клиента и сервера не изменяются. Мы определяем одну структуру для двух аргументов, другую структуру для результата и помещаем оба определения в наш заголовочный файл `sum.h`, представленный в листинге 5.12. В листинге 5.13 показана функция `str_cli`.

Листинг 5.12. Заголовочный файл `sum.h`

```
//tcpcliserv/sum.h
1 struct args {
2     long arg1;
3     long arg2;
4 };
```

```
5 struct result {
6     long sum;
7 };
```

Листинг 5.13. Функция `str_cli`, отправляющая два двоичных целых числа серверу

```
//tcpcliserv/str_cli09.c
1 #include "unp.h"
2 #include "sum.h"

3 void
4 str_cli(FILE *fp, int sockfd)
5 {
6     char sendline[MAXLINE];
7     struct args args;
8     struct result result;
```

```

9 while (Fgets(sendline, MAXLINE, fp) != NULL) {
10    if (sscanf(sendline, "%ld%ld", &args.arg1, &args.arg2) != 2) {
11        printf("invalid input, %s", sendline);
12        continue;
13    }
14    Writen(sockfd, &args, sizeof(args));
15    if (Readn(sockfd, &result, sizeof(result)) == 0)
16        err_quit("str_cli: server terminated prematurely");
17    printf("%ld\n", result.sum);
18 }
19 }
```

10-14 Функция `sscanf` преобразует два аргумента из текстовых строк в двоичные. Мы вызываем функцию `writen` для отправки структуры серверу.

15-17 Мы вызываем функцию `readn` для чтения ответа и выводим результат с помощью функции `printf`.

В листинге 5.14 показана наша функция `str_echo`.

Листинг 5.14. Функция `str_echo`, складывающая два двоичных целых числа

```

//tcpcliserv/str_echo09.c
1 #include "unp.h"
2 #include "sum.h"

3 void
4 str_echo(int sockfd)
5 {
6     ssize_t n;
7     struct args args;
8     struct result result;

9     for (;;) {
10        if ((n = Readn(sockfd, &args, sizeof(args))) == 0)
11            return; /* соединение закрыто удаленным концом */

12        result.sum = args.arg1 + args.arg2;
13        Writen(sockfd, &result, sizeof(result));
14    }
15 }
```

9-14 Мы считываем аргументы при помощи вызова функции `readn`, вычисляем и запоминаем сумму и вызываем функцию `writen` для отправки результирующей структуры обратно.

Если мы запустим клиент и сервер на двух машинах с аналогичной архитектурой, например на двух компьютерах SPARC, все будет работать нормально:

```

solaris % tcpcli09 12.106.32.254
11 22 мы вводим эти числа
33 а это ответ сервера
-11 -44
-55
```

Но если клиент и сервер работают на машинах с разными архитектурами, например, сервер в системе FreeBSD на SPARC, в которой используется обратный порядок байтов (big-endian), а клиент — в системе Linux на Intel с прямым порядком байтов (little-endian), результат будет неверным:

```

linux % tcpcli09 206.168.112.96
1 2      мы вводим эти числа
3      и сервер дает правильный ответ
-22 -77  потом мы вводим эти числа
-16777314 и сервер дает неверный ответ
```

Проблема заключается в том, что два двоичных числа передаются клиентом через сокет в формате с прямым порядком байтов, а сервер интерпретирует их как целые числа, записанные с обратным порядком байтов. Мы видим, что это допустимо для положительных целых чисел, но для отрицательных такой подход не срабатывает (см. упражнение 5.8). Действительно, в подобной ситуации могут возникнуть три проблемы:

1. Различные реализации хранят двоичные числа в различных форматах. Наиболее характерный пример — прямой и обратный порядок байтов, описанный в разделе 3.4.
2. Различные реализации могут хранить один и тот же тип данных языка С по-разному. Например, большинство 32-разрядных систем Unix используют 32 бита для типа `long`, но 64-разрядные системы обычно используют 64 бита для того же типа данных (см. табл. 1.5). Нет никакой гарантии, что типы `short`, `int` или `long` имеют какой-либо определенный размер.
3. Различные реализации по-разному упаковывают структуры в зависимости от числа битов, используемых для различных типов данных, и ограничений по выравниванию для данного компьютера. Следовательно, неразумно передавать через сокет двоичные структуры.

Есть два общих решения проблемы, связанной с различными форматами данных:

1. Передавайте все численные данные как текстовые строки. Это то, что мы делали в листинге 5.11. При этом предполагается, что у обоих узлов один и тот же набор символов.
2. Явно определяйте двоичные форматы поддерживаемых типов данных (число битов и порядок байтов) и передавайте все данные между клиентом и сервером в этом формате. Пакеты удаленного вызова процедур (Remote Procedure Call, RPC) обычно используют именно эту технологию. В RFC 1832 [109] описывается *стандарт представления внешних данных* (External Data Representation, XDR), используемый с пакетом Sun RPC.

5.19. Резюме

Первая версия наших эхо-клиента и эхо-сервера содержала около 150 строк (включая функции `readline` и `writen`), но многие ее детали пришлось модифицировать. Первой проблемой, с которой мы столкнулись, было превращение дочерних процессов в зомби, и для обработки этой ситуации мы перехватывали сигнал `SIGCHLD`. Затем наш обработчик сигнала вызывал функцию `waitpid`, и мы показали, что должны вызывать именно эту функцию вместо более старой функции `wait`, поскольку сигналы Unix не помещаются в очередь. В результате мы рассмотрели некоторые подробности обработки сигналов POSIX, аза дополнительной информацией по этой теме вы можете обратиться к [110, глава 10].

Следующая проблема, с которой мы столкнулись, состояла в том, что клиент не получал уведомления о завершении процесса сервера. Мы видели, что TCP нашего клиента получал уведомление, но оно не доходило до клиентского процесса, поскольку тот был блокирован в ожидании ввода пользователя. В главе 6 для обработки этого сценария мы будем использовать функции `select` или `poll`, позволяющие ожидать готовности любого из множества дескрипторов вместо блокирования при обращении к одному дескриптору.

Мы также обнаружили, что если узел сервера выходит из строя, мы не можем определить это до тех пор, пока клиент не пошлет серверу какие-либо данные. Некоторые приложения должны узнавать об этом факте раньше, о чем мы поговорим далее, когда в разделе 7.5 будем рассматривать параметр сокета `SO_KEEPALIVE`.

В нашем простом примере происходил обмен текстовыми строками, и поскольку от сервера не требовалось просматривать отражаемые им строки, все работало нормально. Передача численных данных между клиентом и сервером может привести к ряду новых проблем, что и было продемонстрировано.

Упражнения

1. Создайте сервер TCP на основе листингов 5.1 и 5.2 и клиент TCP на основе листингов 5.3 и 5.4. Запустите сервер, затем запустите клиент. Введите несколько строк, чтобы проверить, что клиент и сервер работают. Завершите работу клиента, введя символ конца файла, и заметьте время. Используйте программу `netstat` на узле клиента для проверки того, что клиентский конец соединения проходит состояние `TIME_WAIT`. Запускайте `netstat` примерно каждые 5 с, чтобы посмотреть, когда закончится состояние `TIME_WAIT`. Каково время MSL для вашей реализации?

2. Что происходит с нашим соединением клиент-сервер, если мы запускаем клиент и подключаем к стандартному потоку ввода двоичный файл?

3. В чем разница между нашим соединением клиент-сервер и использованием клиента Telnet для взаимодействия с нашим эхо-сервером?

4. В нашем примере в разделе 5.12 мы проверили, что первые два сегмента завершения соединения (сегмент FIN от сервера, на который затем клиент отвечает сегментом ACK) отправляются, при просмотре состояний сокета с помощью программы netstat. Происходит ли обмен двумя последними сегментами (FIN от клиента, на который затем сервер отвечает сегментом ACK)? Если да, то когда? Если нет, то почему?

5. Что произойдет с примером, рассмотренным в разделе 5.14, если между шагами 2 и 3 мы перезапустим сервер на узле сервера?

6. Чтобы проверить, что происходит с сигналом SIGPIPE в разделе 5.13, измените листинг 5.3 следующим образом. Напишите обработчик сигнала для SIGPIPE, который будет просто выводить сообщение и возвращать управление. Установите этот обработчик сигнала перед вызовом функции connect. Измените номер порта сервера на 13 (порт сервера времени и даты). Когда соединение установится, с помощью функции sleep войдите в состояние ожидания на 2 с, с помощью функции write запишите несколько байтов в сокет, проведите в состоянии ожидания (sleep) еще 2 с и с помощью функции write запишите еще несколько байтов. Запустите программу. Что происходит?

7. Что произойдет на рис. 5.5, если IP-адрес узла сервера, заданный клиентом при вызове функции connect, является IP-адресом, связанным с крайним правым канальным уровнем на стороне сервера, а не IP-адресом, связанным с крайним левым канальным уровнем?

8. В нашем примере эхо-сервера, осуществляющего сложение двух целых чисел (см. листинг 5.14), когда клиент и сервер принадлежат системам с различным порядком байтов, для небольших положительных чисел получается правильный ответ, но для небольших отрицательных чисел ответ неверен. Почему? (*Подсказка:* нарисуйте схему обмена значениями через сокет, аналогичную рис. 3.4.)

9. В нашем примере в листинге 5.13 и 5.14 можем ли мы решить проблему, связанную с различным порядком байтов на стороне клиента и на стороне сервера, если клиент преобразует два аргумента в сетевой порядок байтов, используя функцию htonl, а сервер затем вызывает функцию ntohl для каждого аргумента перед сложением и выполняет аналогичное преобразование результата?

10. Что произойдет в листинге 5.13 и 5.14, если в качестве узла клиента используется компьютер SPARC, где данные типа long занимают 32 бита, а в качестве узла сервера — Digital Alpha, где данные типа long занимают 64 бита? Изменится ли что-либо, если клиент и сервер поменяются местами?

11. На рис. 5.5 указано, что IP-адрес клиента выбирается IP на основе маршрутизации. Что это значит?

Глава 6

Мультиплексирование ввода-вывода: функции `select` и `poll`

6.1. Введение

В разделе 5.12 мы видели, что наш TCP-клиент обрабатывает два входных потока одновременно: стандартный поток ввода и сокет TCP. Проблема, с которой мы столкнулись, состояла в том, что пока клиент был блокирован в вызове функции `fgets` (чтение из стандартного потока ввода), процесс сервера мог быть уничтожен. TCP сервера корректно отправляет сегмент FIN протоколу TCP клиента, но поскольку процесс клиента блокирован при чтении из стандартного потока ввода, он не получит признак конца файла, пока не считает данные из сокета (возможно, значительно позже). Нам нужна возможность сообщить ядру, что мы хотим получить уведомления о том, что выполняется одно или несколько условий для ввода-вывода (например, присутствуют данные для считывания или дескриптор готов к записи новых данных). Эта возможность называется *мультиплексированием* (multiplexing) ввода-вывода и обеспечивается функциями `select` и `poll`. Мы рассмотрим также более новый вариант функции `select`, входящей в стандарт POSIX, называемый `pselect`.

ПРИМЕЧАНИЕ

В некоторых системах предоставляются более мощные средства ожидания событий. Одним из механизмов является устройство опроса (poll device), которое по-разному реализуется разными производителями. Этот механизм описывается в главе 14.

Мультиплексирование ввода-вывода обычно используется сетевыми приложениями в следующих случаях:

- Когда клиент обрабатывает множество дескрипторов (обычно интерактивный ввод и сетевой сокет), должно использоваться мультиплексирование ввода-вывода. Это сценарий, который мы только что рассмотрели.
 - Возможно, хотя это и редкий случай, что клиент одновременно обрабатывает множество сокетов. Такой пример мы приведем в разделе 16.5 при использовании функции `select` в контексте веб-клиента.
 - Если сервер TCP обрабатывает и прослушиваемый сокет, и присоединенные сокеты, обычно используется мультиплексирование ввода-вывода, как это показано в разделе 6.8.
 - Если сервер работает и с TCP, и с UDP, обычно также используется мультиплексирование ввода-вывода. Такой пример мы приводим в разделе 8.15.
 - Если сервер обрабатывает несколько служб и, возможно, несколько протоколов (например, демон `inetd`, который описан в разделе 12.5), обычно используется мультиплексирование ввода-вывода.

Область применения мультиплексирования ввода-вывода не ограничивается только сетевым программированием. Любому нетривиальному приложению часто приходится использовать эту технологию.

6.2. Модели ввода-вывода

Прежде чем начать описание функций `select` и `poll`, мы должны вернуться назад и уяснить основные различия между пятью моделями ввода-вывода, доступными нам в Unix:

- блокируемый ввод-вывод;
- неблокируемый ввод-вывод;
- мультиплексирование ввода-вывода (функции `select` и `poll`);
- ввод-вывод, управляемый сигналом (сигнал `SIGIO`);
- асинхронный ввод-вывод (функции POSIX `aio_`).

Возможно, вы захотите пропустить этот раздел при первом прочтении, а затем вернуться к нему по мере знакомства с различными моделями ввода-вывода, подробно рассматриваемыми в дальнейших главах.

Как вы увидите в примерах этого раздела, обычно различаются две фазы операции ввода:

1. Ожидание готовности данных.

2. Копирование данных от ядра процессу.

Первый шаг операции ввода на сокете обычно включает ожидание прихода данных по сети. Когда пакет приходит, он копируется в буфер внутри ядра. Второй шаг — копирование этих данных из буфера ядра в буфер приложения.

Модель блокируемого ввода-вывода

Наиболее распространенной моделью ввода-вывода является модель блокируемого ввода-вывода, которую мы использовали для всех предыдущих примеров. По умолчанию все сокеты являются блокируемыми. Используя в наших примерах сокет дейтаграмм, мы получаем сценарий, показанный на рис. 6.1.

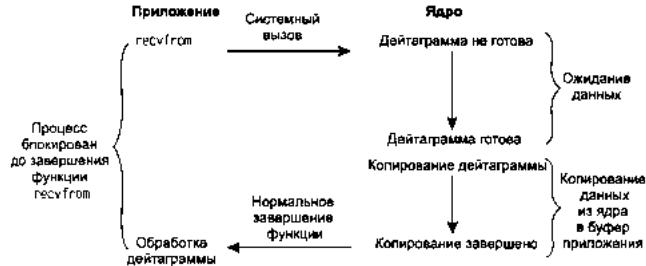


Рис. 6.1. Модель блокируемого ввода-вывода

В этом примере вместо TCP мы используем UDP, поскольку в случае UDP признак готовности данных очень прост: получена вся дейтаграмма или нет. В случае TCP он становится сложнее, поскольку приходится учитывать дополнительные переменные, например минимальный объем данных в сокете (low water-mark).

В примерах этого раздела мы говорим о функции `recvfrom` как о системном вызове, поскольку делаем различие между нашим приложением и ядром. Вне зависимости от того, как реализована функция `recvfrom` (как системный вызов в ядре, происходящем от Беркли, или как функция, активизирующая системный вызов `getmsg` в ядре System V), она обычно выполняет переключение между работой в режиме приложения и работой в режиме ядра, за которым через определенный промежуток времени следует возвращение в режим приложения.

На рис. 6.1 процесс вызывает функцию `recvfrom`, и системный вызов не возвращает управление, пока дейтаграмма не придет и не будет скопирована в буфер приложения либо пока не произойдет ошибка. Наиболее типичная ошибка — это прерывание системного вызова сигналом, о чём рассказывалось в разделе 5.9. Процесс блокирован в течение всего времени с момента, когда он вызывает функцию `recvfrom`, до момента, когда эта функция завершается. Когда функция `recvfrom` выполняется normally, наше приложение обрабатывает дейтаграмму.

Модель неблокируемого ввода-вывода

Когда мы определяем сокет как неблокируемый, мы тем самым сообщаем ядру следующее: «когда запрашиваемая нами операция ввода-вывода не может быть завершена без перевода процесса в состояние ожидания, следует не переводить процесс в состояние ожидания, а возвратить ошибку». Неблокируемый ввод-вывод мы описываем подробно в главе 16, а на рис. 6.2 лишь демонстрируем его свойства.

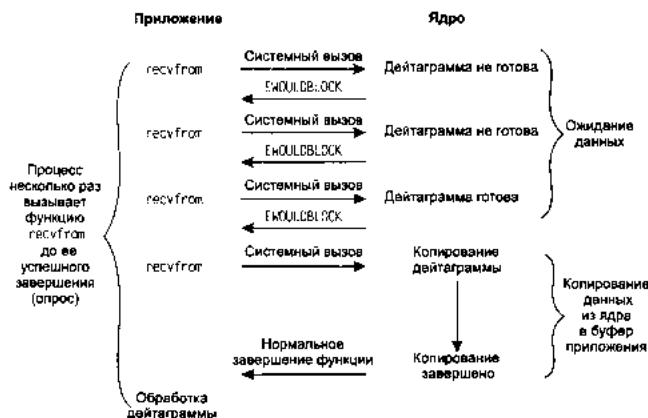


Рис. 6.2. Модель неблокируемого ввода-вывода

В первых трех случаях вызова функции recvfrom для возвращения нет, поэтому ядро немедленно возвращает ошибку EWOULDBLOCK. Когда мы в четвертый раз вызываем функцию recvfrom, дейтаграмма готова, поэтому она копируется в буфер приложения и функция recvfrom успешно завершается. Затем мы обрабатываем данные.

Такой процесс, когда приложение находится в цикле и вызывает функцию recvfrom на неблокируемом дескрипторе, называется *опросом (polling)*. Приложение последовательно опрашивает ядро, чтобы увидеть, что какая-то операция может быть выполнена. Часто это пустая траты времени процессора, но такая модель все же иногда используется, обычно в специализированных системах.

Модель мультиплексирования ввода-вывода

В случае мультиплексирования ввода-вывода мы вызываем функцию select или poll, и блокирование происходит в одном из этих двух системных вызовов, а не в действительном системном вызове ввода-вывода. На рис. 6.3 обобщается модель мультиплексирования ввода-вывода.

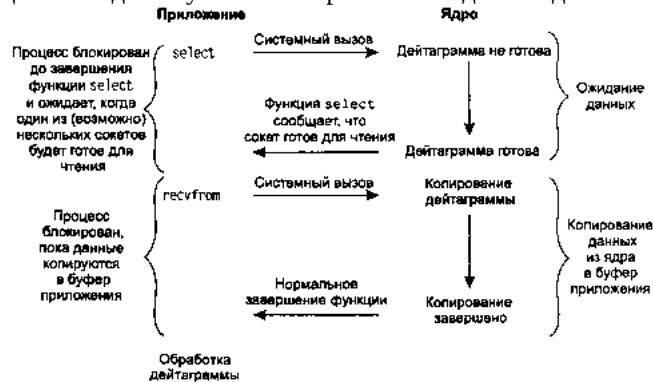


Рис. 6.3. Модель мультиплексирования ввода-вывода

Процесс блокируется в вызове функции select, ожидая, когда дейтаграммный сокет будет готов для чтения. Когда функция select возвращает сообщение, что сокет готов для чтения, процесс вызывает функцию recvfrom, чтобы скопировать дейтаграмму в буфер приложения.

Сравнивая рис. 6.3 и 6.1, мы не найдем в модели мультиплексирования ввода-вывода каких-либо преимуществ, более того, она даже обладает незначительным недостатком, поскольку использование функции select требует двух системных вызовов вместо одного. Но преимущество использования функции select, которое мы увидим далее в этой главе, состоит в том, что мы сможем ожидать готовности не одного дескриптора, а нескольких.

ПРИМЕЧАНИЕ

Разновидностью данного способа мультиплексирования является многопоточное программирование с блокируемым вводом-выводом. Отличие состоит в том, что вместо вызова `select` с блокированием программа использует несколько потоков (по одному на каждый дескриптор), которые могут блокироваться в вызовах типа `recvfrom`.

Модель ввода-вывода, управляемого сигналом

Мы можем сообщить ядру, что необходимо уведомить процесс о готовности дескриптора с помощью сигнала `SIGIO`. Такая модель имеет название *ввод-вывод, управляемый сигналом* (*signal-driven I/O*). Она представлена в обобщенном виде на рис. 6.4.

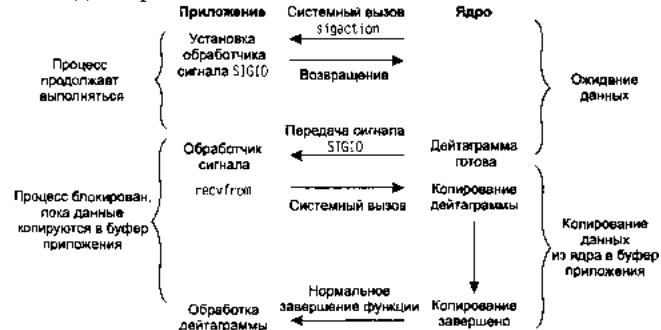


Рис. 6.4. Модель управляемого сигналом ввода-вывода

Сначала мы включаем на сокете управляемый сигналом ввод-вывод (об этом рассказывается в разделе 22.2) и устанавливаем обработчик сигнала при помощи системного вызова `sigaction`. Возвращение из этого системного вызова происходит незамедлительно, и наш процесс продолжается (он не блокирован). Когда дейтаграмма готова для чтения, для нашего процесса генерируется сигнал `SIGIO`. Мы можем либо прочитать дейтаграмму из обработчика сигнала с помощью вызова функции `recvfrom` и затем уведомить главный цикл о том, что данные готовы для обработки (см. раздел 22.3), либо уведомить основной цикл и позволить ему прочитать дейтаграмму.

Независимо от способа обработки сигнала эта модель имеет то преимущество, что во время ожидания дейтаграммы не происходит блокирования. Основной цикл может продолжать выполнение, ожидая уведомления от обработчика сигнала о том, что данные готовы для обработки либо дейтаграмма готова для чтения.

Модель асинхронного ввода-вывода

Асинхронный ввод-вывод был введен в редакции стандарта POSIX.1g 1993 г. (расширения реального времени). Мы сообщаем ядру, что нужно начать операцию и уведомить нас о том, когда вся операция (включая копирование данных из ядра в наш буфер) завершится. Мы не обсуждаем эту модель в этой книге, поскольку она еще не получила достаточного распространения. Ее основное отличие от модели ввода-вывода, управляемого сигналом, заключается в том, что при использовании сигналов ядро сообщает нам, когда операция ввода-вывода может быть инициирована, а в случае асинхронного ввода-вывода — когда операция завершается. Пример этой модели приведен на рис. 6.5.

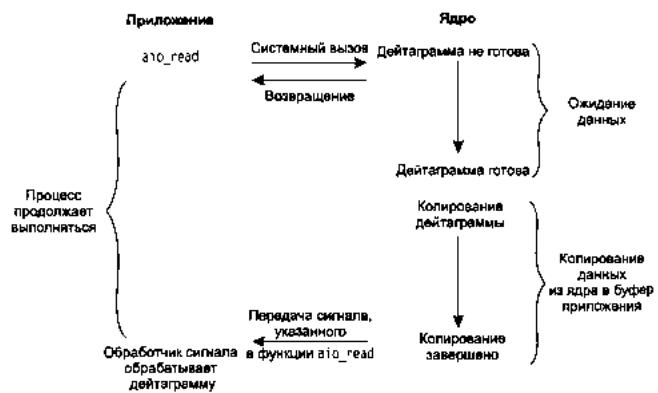


Рис. 6.5. Модель асинхронного ввода-вывода

Мы вызываем функцию aio_read (функции асинхронного ввода-вывода POSIX начинаются с aio_ или lio_) и передаем ядру дескриптор, указатель на буфер, размер буфера (те же три аргумента, что и для функции read), смещение файла (аналогично функции lseek), а также указываем, как уведомить нас, когда операция полностью завершится. Этот системный вызов завершается немедленно, и наш процесс не блокируется в ожидании завершения ввода-вывода. В этом примере предполагается, что мы указали ядру генерировать некий сигнал, когда операция завершится. Сигнал не генерируется до тех пор, пока данные не скопированы в наш буфер приложения, что отличает эту модель от модели ввода-вывода, управляемого сигналом ввода-вывода.

ПРИМЕЧАНИЕ

На момент написания книги только некоторые системы поддерживали асинхронный ввод-вывод стандарта POSIX. Например, мы не уверены, что какие-либо системы поддерживают его для сокетов. Мы используем его только как пример для сравнения с моделью управляемого сигналом ввода-вывода.

Сравнение моделей ввода-вывода

На рис. 6.6 сравниваются пять различных моделей ввода-вывода. Здесь видно главное отличие четырех первых моделей в первой фазе, поскольку вторая фаза у них одна и та же: процесс блокируется в вызове функции recvfrom на то время, пока данные копируются из ядра в буфер вызывающего процесса. Асинхронный ввод-вывод отличается от первых четырех моделей в обеих фазах.

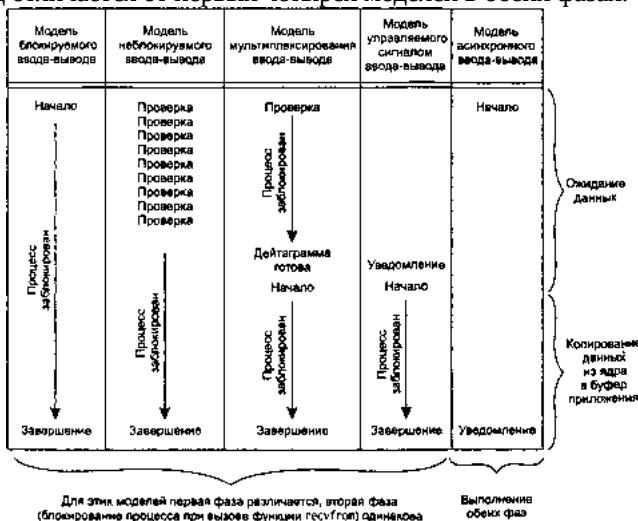


Рис. 6.6. Сравнение моделей ввода-вывода

Сравнение синхронного и асинхронного ввода-вывода

POSIX дает следующие определения этих терминов:

- Операция синхронного ввода-вывода блокирует запрашивающий процесс до тех пор, пока операция ввода-вывода не завершится.

- Операция асинхронного ввода-вывода не вызывает блокирования запрашивающего процесса.

Используя эти определения, можно сказать, что первые четыре модели ввода-вывода — блокируемая, неблокируемая, модель мультиплексирования ввода-вывода и модель управляемого сигналом ввода-вывода — являются синхронными, поскольку фактическая операция ввода-вывода (функция `recvfrom`) блокирует процесс. Только модель асинхронного ввода-вывода соответствует определению асинхронного ввода-вывода.

6.3. Функция `select`

Эта функция позволяет процессу сообщить ядру, что необходимо подождать, пока не произойдет одно из некоторого множества событий, и вывести процесс из состояния ожидания, только когда произойдет одно или несколько таких событий или когда пройдет заданное количество времени.

Например, мы можем вызвать функцию `select` и сообщить ядру, что возвращать управление нужно только когда наступит любое из следующих событий:

- любой дескриптор из набора {1, 4, 5} готов для чтения;
- любой дескриптор из набора {2, 7} готов для записи;
- любой дескриптор из набора {1, 4} вызывает исключение, требующее обработки;
- истекает 10,2 с.

Таким образом, мы сообщаем ядру, какие дескрипторы нас интересуют (готовые для чтения, готовые для записи или требующие обработки исключения) и как долго нужно ждать. Интересующие нас дескрипторы не ограничиваются сокетами: любой дескриптор можно проверить с помощью функции `select`.

ПРИМЕЧАНИЕ

Беркли-реализации всегда допускали мультиплексирование ввода-вывода с любыми дескрипторами. Система SVR3 ограничивала мультиплексирование ввода-вывода дескрипторами, которые являлись устройствами STREAMS (см. главу 31), но это ограничение было снято в SVR4.

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfdp1, fd_set *readset, fd_set *writeset,
           fd_set *exceptset, const struct timeval *timeout);
Возвращает: положительное число - счетчик готовых дескрипторов, 0 в случае тайм-аута, -1 в
случае ошибки
```

Описание этой функции мы начнем с последнего аргумента, который сообщает ядру, сколько следует ждать, пока один из заданных дескрипторов не будет готов. Структура `timeval` задает число секунд и микросекунд:

```
struct timeval {
    long tv_sec; /* секунды */
    long tv_usec; /* микросекунды */
};
```

С помощью этого аргумента можно реализовать три сценария:

1. Ждать вечно: завершать работу, только когда один из заданных дескрипторов готов для ввода-вывода. Для этого нужно определить аргумент `timeout` как пустой указатель.

2. Ждать в течение определенного времени: завершение будет происходить, когда один из заданных дескрипторов готов для ввода-вывода, но период ожидания ограничивается количеством секунд и микросекунд, заданным в структуре `timeval`, на которую указывает аргумент `timeout`.

3. Не ждать вообще: завершение происходит сразу же после проверки дескрипторов. Это называется *опросом (polling)*. Аргумент `timeout` должен указывать на структуру `timeval`, а значение таймера (число секунд и микросекунд, заданных этой структурой) должно быть нулевым.

Ожидание в первых двух случаях обычно прерывается, когда процесс перехватывает сигнал и возвращается из обработчика сигнала.

ПРИМЕЧАНИЕ

Ядра реализаций, происходящих от Беркли, никогда автоматически не перезапускают функцию `select` [128, с. 527], в то время как ядра SVR4 перезапускают, если задан флаг `SA_RESTART` при установке обработчика сигнала. Это значит, что в целях переносимости мы должны быть готовы к тому, что функция `select` вернёт ошибку `EINTR`, если мы перехватываем сигналы.

Хотя структура `timeval` позволяет нам задавать значение с точностью до микросекунд, реальная точность, поддерживаемая ядром, часто значительно ниже. Например, многие ядра Unix округляют значение тайм-аута до числа, кратного 10 мс. Присутствует также и некоторая скрытая задержка: между истечением времени таймера и моментом, когда ядро запустит данный процесс, проходит некоторое время.

ПРИМЕЧАНИЕ

В некоторых системах при задании поля `tv_sec` более 100 млн с функция `select` завершается с кодом ошибки `EINVAL`. Это, конечно, достаточно большое число (более трех лет), но факт остается фактом: структура `timeval` может содержать значения, не поддерживаемые функцией `select`.

Спецификатор `const` аргумента `timeout` означает, что данный аргумент не изменяется функцией `select` при ее возвращении. Например, если мы зададим предел времени, равный 10 с, и функция `select` вернет управление до истечения этого времени с одним или несколькими готовыми дескрипторами или ошибкой `EINTR`, то структура `timeval` не изменится, то есть при завершении функции значение тайм-аута не станет равно числу секунд, оставшихся от исходных 10. Чтобы узнать количество неизрасходованных секунд, следует определить системное время до вызова функции `select`, а когда она завершится, определить его еще раз и вычесть первое значение из второго. Устойчивая программа должна учитывать тот факт, что системное время может периодически корректироваться администратором или демоном типа `ntpd`.

ПРИМЕЧАНИЕ

В современных системах Linux структура `timeval` изменяема. Следовательно, в целях переносимости будем считать, что структура `timeval` по возвращении становится неопределенной, и будем инициализировать ее перед каждым вызовом функции `select`. В POSIX указывается спецификатор `const`.

Три средних аргумента, `readset`, `writeset` и `exceptset`, определяют дескрипторы, которые ядро должно проверить на возможность чтения и записи и на наличие исключений (exceptions). В настоящее время поддерживается только два исключения:

1. На сокет приходят внеполосные данные. Более подробно мы опишем этот случай в главе 24.
2. Присутствие информации об управлении состоянием (control status information), которая должна быть считана с управляющего (master side) псевдотерминала, помещенного в режим пакетной обработки. Псевдотерминалы в данном томе не рассматриваются.

Проблема в том, как задать одно или несколько значений дескрипторов для каждого из трех аргументов. Функция `select` использует наборы дескрипторов, обычно это массив целых чисел, где каждый бит в каждом числе соответствует дескриптору. Например, при использовании 32-разрядных

целых чисел первый элемент массива (целое число) соответствует дескрипторам от 0 до 31, второй элемент — дескрипторам от 32 до 63, и т.д. Детали реализации не влияют на приложение и скрыты в типе данных `fd_set` и следующих четырех макросах:

```
void FD_ZERO(fd_set *fdset); /* сбрасываем все биты в fdset */
void FD_SET(int fd, fd_set *fdset); /* устанавливаем бит для fd в fdset */
void FD_CLR(int fd, fd_set *fdset); /* сбрасываем бит для fd в fdset */
int FD_ISSET(int fd, fd_set *fdset); /* установлен ли бит для fd в fdset? */
```

Мы размещаем в памяти набор дескрипторов типа `fd_set`, с помощью этих макросов устанавливаем и проверяем биты в наборе, а также можем присвоить его (как значение) другому набору дескрипторов с помощью оператора присваивания языка С.

ПРИМЕЧАНИЕ

Описываемый нами массив целых чисел, использующий по одному биту для каждого дескриптора, — это только один из возможных способов реализации функции `select`. Тем не менее является обычной практикой ссылаться на отдельные дескрипторы в наборе дескрипторов как на биты, например так: «установить бит для прослушиваемого дескриптора в наборе для чтения».

В разделе 6.10 мы увидим, что функция `poll` использует совершенно другое представление: массив структур переменной длины, по одной структуре для каждого дескриптора.

Например, чтобы определить переменную типа `fd_set` и затем установить биты для дескрипторов 1, 4 и 5, мы пишем:

```
fd_set rset;

FD_ZERO(&rset); /* инициализируем набор все биты сброшены */
FD_SET(1, &rset); /* устанавливаем бит для fd 1 */
FD_SET(4, &rset); /* устанавливаем бит для fd 4 */
FD_SET(5, &rset); /* устанавливаем бит для fd 5 */
```

Важно инициализировать набор, так как если набор будет создан в виде автоматической переменной и не проинициализирован, результат может оказаться непредсказуемым.

Любой из трех средних аргументов функции `select` — `readset`, `writeset` или `exceptset` — может быть задан как пустой указатель, если нас не интересует определяемое им условие. На самом деле, если все три указателя пустые, мы просто получаем таймер большей точности, чем обычная функция Unix `sleep` (позволяющая задавать время с точностью до секунды). Функция `poll` обеспечивает аналогичную функциональность. На рис. C.9 и C.10 [110] показана функция `sleep_us`, реализованная с помощью функций `select` и `poll`, которая позволяет устанавливать время ожидания с точностью до микросекунд.

Аргумент `maxfdp1` задает число проверяемых дескрипторов. Его значение на единицу больше максимального номера проверяемого дескриптора (поэтому мы назвали его `maxfdp1`). Проверяются дескрипторы 0, 1, 2 и далее до `maxfdp1 - 1` включительно.

Константа `FD_SETSIZE`, определяемая при подключении заголовочного файла `<sys/select.h>`, является максимальным числом дескрипторов для типа данных `fd_set`. Ее значение часто равно 1024, но такое количество дескрипторов используется очень немногими программами. Аргумент `maxfdp1` заставляет нас вычислять наибольший интересующий нас дескриптор и затем сообщать ядру его значение. Например, в предыдущем коде, который включает дескрипторы 1, 4 и 5, значение аргумента `maxfdp1` равно 6. Причина, по которой это 6, а не 5, в том, что мы задаем количество дескрипторов, а не наибольшее значение, а нумерация дескрипторов начинается с нуля.

ПРИМЕЧАНИЕ

Зачем нужно было включать этот аргумент и вычислять его значение? Причина в том, что он повышает эффективность работы ядра. Хотя каждый набор типа `fd_set` может содержать множество дескрипторов (обычно до 1024), реальное количество дескрипторов, используемое типичным процессом, значительно меньше. Эффективность возрастает за счет того, что не

копируются ненужные части набора дескрипторов между ядром и процессом и не требуется проверять биты, которые всегда являются нулевыми (см. раздел 16.13 [128]).

Функция `select` изменяет наборы дескрипторов, на которые указывают аргументы `readset`, `writeset` и `exceptset`. Эти три аргумента являются аргументами типа «значение-результат». Когда мы вызываем функцию, мы указываем интересующие нас дескрипторы, а по ее завершении результат показывает нам, какие дескрипторы готовы. Проверить определенный дескриптор из структуры `fd_set` после завершения вызова можно с помощью макроса `FD_ISSET`. Для дескриптора, не готового для чтения или записи, соответствующий бит в наборе дескрипторов будет сброшен. Поэтому мы устанавливаем все интересующие нас биты во всех наборах дескрипторов каждый раз, когда вызываем функцию `select`.

ПРИМЕЧАНИЕ

Две наиболее общих ошибки программирования при использовании функции `select` — это забыть добавить единицу к наибольшему номеру дескриптора и забыть, что наборы дескрипторов имеют тип «значение-результат». Вторая ошибка приводит к тому, что функция `select` вызывается с нулевым битом в наборе дескрипторов, когда мы думаем, что он установлен в единицу.

Возвращаемое этой функцией значение указывает общее число готовых дескрипторов во всех наборах дескрипторов. Если значение таймера истекает до того, как какой-нибудь из дескрипторов оказывается готов, возвращается нулевое значение. Возвращаемое значение -1 указывает на ошибку (которая может произойти, если, например, выполнение функции прервано перехваченным сигналом).

ПРИМЕЧАНИЕ

В ранних реализациях SVR4 функция `select` содержала ошибку: если один и тот же бит находился в нескольких наборах дескрипторов — допустим, дескриптор был готов и для чтения, и для записи, — он учитывался только один раз. В современных реализациях эта ошибка исправлена.

При каких условиях дескриптор становится готовым?

Мы говорили об ожидании готовности дескриптора для ввода-вывода (чтения или записи) или возникновения исключительной ситуации, требующей обработки (внеполосные данные). В то время как готовность к чтению и записи очевидна для файловых дескрипторов, в случае дескрипторов сокетов следует более внимательно изучить те условия, при которых функция `select` сообщает, что сокет готов (см. рис. 16.52 [128]).

1. Сокет готов для чтения, если выполнено хотя бы одно из следующих условий:

1) число байтов данных в приемном буфере сокета больше или равно текущему значению минимального количества данных (*low water-mark*) для приемного буфера сокета. Операция считывания данных из сокета не блокируется и возвратит значение, большее нуля (то есть данные, готовые для чтения). Мы можем задать значение минимального количества данных (*low-water mark*) с помощью параметра сокета `SO_RCVLOWAT`. По умолчанию для сокетов TCP и UDP это значение равно 1;

2) на противоположном конце соединение закрывается (нами получен сегмент FIN). Операция считывания данных из сокета не блокируется и возвратит нуль (то есть признак конца файла);

3) сокет является прослушиваемым, и число установленных соединений ненулевое. Функция `accept` на прослушиваемом сокете в таком случае обычно не блокируется, хотя в разделе 16.6 мы описываем ситуацию, в которой функция `accept` может заблокироваться несмотря на наличие установленных соединений;

4) ошибка сокета, ожидающая обработки. Операция чтения на сокете не блокируется и возвратит ошибку (-1) со значением переменной `errno`, указывающим на конкретное условие ошибки. Эти ошибки, ожидающие обработки, можно также получить, вызвав функцию `getsockopt` с параметром `SO_ERROR`, после чего состояние ошибки будет сброшено.

2. Сокет готов для записи, если выполнено одно из следующих условий:

1) количество байтов доступного пространства в буфере отправки сокета больше или равно текущему значению минимального количества данных для буфера отправки сокета и либо сокет является присоединенным, либо сокету не требуется соединения (например, сокет UDP). Это значит, что если мы отключим блокировку для сокета (см. главу 16), операция записи не блокирует процесс и возвратит положительное значение (например, число байтов, принятых транспортным уровнем). Устанавливать минимальное количество данных мы можем с помощью параметра сокета SO_SNDBUF. По умолчанию это значение равно 2048 для сокетов TCP и UDP;

2) получатель, которому отправляются данные, закрывает соединение. Операция записи в сокет генерирует сигнал SIGPIPE (см. раздел 5.12);

3) ошибка сокета, ожидающая обработки. Операция записи в сокет не блокируется и возвратит ошибку (-1) со значением переменной errno, указывающей на конкретное условие ошибки. Эти ошибки, ожидающие обработки, можно также получить ибросить, вызвав функцию getsockopt с параметром сокета SO_ERROR.

3. Исключительная ситуация, требующая обработки, может возникнуть на сокете в том случае, если приняты внеполосные данные либо если отметка вне- полосных данных в принимаемом потоке еще не достигнута. (Внеполосные данные описываются в главе 24.)

ПРИМЕЧАНИЕ

Наши определения «готов для чтения» и «готов для записи» взяты непосредственно из макроопределений ядра soreadable и sowritable (которые описываются в [128, с. 530-531]). Аналогично, наше определение «исключительной ситуации» взято из функции soo_select, которая описана там же.

Обратите внимание, что когда происходит ошибка на сокете, функция select отмечает его готовым как для чтения, так и для записи.

Значения минимального количества данных (low-water mark) для приема и отправки позволяют приложению контролировать, сколько данных должно быть доступно для чтения или сколько места должно быть доступно для записи перед тем, как функция select сообщит, что сокет готов для чтения или записи. Например, если мы знаем, что наше приложение не может сделать ничего полезного, пока не будет получено как минимум 64 байт данных, мы можем установить значение минимального количества данных равным 64, чтобы функция select не вывела нас из состояния ожидания, если для чтения готово менее 64 байт.

Пока значение минимального количества данных для отправки в сокете UDP меньше, чем буфер отправки сокета (а такое отношение между ними всегда устанавливается по умолчанию), сокет UDP всегда готов для записи, поскольку соединения не требуется.

В табл. 6.1 суммируются описанные выше условия, при которых сокет становится готовым для вызова функции select.

Таблица 6.1. Условия, при которых функция select сообщает, что сокет готов для чтения или для записи либо, что необходима обработка исключительной ситуации

Условие	Сокет готов для чтения	Сокет готов для записи	Исключительная ситуация
Данные для чтения	•		
Считывающая половина соединения закрыта	•		
Для прослушиваемого сокета готово новое соединение	•		
Пространство, доступное для записи		•	
Записывающая половина соединения закрыта		•	
Ошибка, ожидающая обработки	•	•	
Внеполосные данные TCP			•

Максимальное число дескрипторов для функции select

Ранее мы сказали, что большинство приложений не используют много дескрипторов. Например, редко можно найти приложение, использующее сотни дескрипторов. Но такие приложения существуют, и часто они используют функцию `select` для мультиплексирования дескрипторов. Когда функция `select` была создана, операционные системы обычно имели ограничение на максимальное число дескрипторов для каждого процесса (этот предел в реализации 4.2BSD составлял 31), и функция `select` просто использовала тот же предел. Но современные версии Unix допускают неограниченное число дескрипторов для каждого процесса (часто оно ограничивается только количеством памяти и административными правилами), поэтому возникает вопрос: как же теперь работает функция `select`?

Многие реализации имеют объявления, аналогичные приведенному ниже, которое взято из заголовочного файла 4.4BSD `<sys/types.h>`:

```
/*
Значение FD_SETSIZE может быть определено пользователем,
но заданное здесь по умолчанию
является достаточным в большинстве случаев.
*/
#ifndef FD_SETSIZE
#define FD_SETSIZE 256
#endif
```

Исходя из этого комментария, можно подумать, что если перед подключением этого заголовочного файла присвоить `FD_SETSIZE` значение, превышающее 256, то увеличится размер наборов дескрипторов, используемых функцией `select`. К сожалению, обычно это не действует.

ПРИМЕЧАНИЕ

Чтобы понять, в чем дело, обратите внимание, что на рис. 16.53 [128] объявляются три набора дескрипторов внутри ядра, а в качестве верхнего предела используется определенное в ядре значение `FD_SETSIZE`. Единственный способ увеличить размер наборов дескрипторов — это увеличить значение `FD_SETSIZE` и затем перекомпилировать ядро. Изменения значения без перекомпиляции ядра недостаточно.

Некоторые производители изменяют свои реализации функции `select`, с тем чтобы позволить процессу задавать значение `FD_SETSIZE`, превышающее значение по умолчанию. BSD/OS также изменила реализацию ядра, чтобы допустить большие наборы дескрипторов, кроме того, в ней добавлено четыре новых макроопределения `FD_xxx` для динамического размещения больших наборов дескрипторов в памяти и для работы с ними. Однако с точки зрения переносимости не стоит злоупотреблять использованием больших наборов дескрипторов.

6.4. Функция str_cli (продолжение)

Теперь мы можем переписать нашу функцию `str_cli`, представленную в разделе 5.5 (на этот раз используя функцию `select`), таким образом, чтобы мы получали уведомление, как только завершится процесс сервера. Проблема с предыдущей версией состояла в том, что процесс мог оказаться заблокированным в вызове функции `fgets`, когда что-то происходило на сокете. Наша новая версия этой функции вместо этого блокируется в вызове функции `select`, ожидая готовности для чтения либо стандартного потока ввода, либо сокета. На рис. 6.7 показаны различные условия, обрабатываемые с помощью вызова функции `select`.

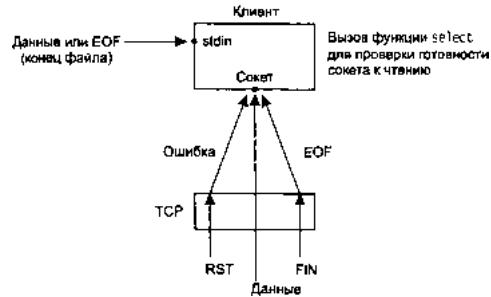


Рис. 6.7. Условия, обрабатываемые функцией select в вызове функции str_cli

Сокет обрабатывает три условия:

1. Если протокол TCP собеседника отправляет данные, сокет становится готовым для чтения, и функция read возвращает положительное значение (то есть число байтов данных).
2. Если протокол TCP собеседника отправляет сегмент FIN (процесс завершается), сокет становится готовым для чтения, и функция read возвращает нуль (признак конца файла).
3. Если TCP собеседника отправляет RST (узел вышел из строя и перезагрузился), сокет становится готовым для чтения, и функция read возвращает -1, а переменная errno содержит код соответствующей ошибки.

В листинге 6.1^[1] представлен исходный код этой версии функции.

Листинг 6.1. Реализация функции str_cli с использованием функции select (усовершенствованный вариант находится в листинге 6.2)

```
//select/strcli select01.c
1 #include "unp.h"

2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     int maxfdp1;
6     fd_set rset;
7     char sendline[MAXLINE], recvline[MAXLINE];

8     FD_ZERO(&rset);
9     for (;;) {
10        FD_SET(fileno(fp), &rset);
11        FD_SET(sockfd, &rset);
12        maxfdp1 = max(fileno(fp), sockfd) + 1;
13        Select(maxfdp1, &rset, NULL, NULL, NULL);

14        if (FD_ISSET(sockfd, &rset)) { /* сокет готов для чтения */
15            if (Readline(sockfd, recvline, MAXLINE) == 0)
16                err_quit("str_cli: server terminated prematurely");
17            Fputs(recvline, stdout);
18        }
19        if (FD_ISSET(fileno(fp), &rset)) { /* входное устройство готово для
           чтения */
20            if (Fgets(sendline, MAXLINE, fp) == NULL)
21                return; /* все сделано */
22            Writen(sockfd, sendline, strlen(sendline));
23        }
24    }
25 }
```

Вызов функции select

8-13 Нам нужен только один набор дескрипторов — для проверки готовности сокета для чтения. Этот набор дескрипторов инициализируется макросом `FD_ZERO`, после чего с помощью макроса `FD_SET` устанавливаются два бита: бит, соответствующий указателю файла `fp` стандартного потока ввода-вывода, и бит, соответствующий дескриптору сокета `sockfd`. Функция `fileno` преобразует указатель файла стандартного потока ввода-вывода в соответствующий ему дескриптор. Функция `select` (а также `poll`) работает только с дескрипторами.

Функция `select` вызывается после определения максимального из двух дескрипторов. В этом вызове указатель на набор дескрипторов для записи и указатель на набор дескрипторов с исключениями являются пустыми. Последний аргумент (ограничение по времени) также является пустым указателем, поскольку мы хотим, чтобы процесс был блокирован, пока не будут готовы данные для чтения.

Обработка сокета, готового для чтения

14-18 Если по завершении функции `select` сокет готов для чтения, отраженная строка считывается функцией `readline` и выводится функцией `fputs`.

Обработка ввода, допускающего возможность чтения

19-23 Если стандартный поток ввода готов для чтения, строка считывается функцией `fgets` и записывается в сокет с помощью функции `writen`.

Обратите внимание, что используются те же четыре функции ввода-вывода, что и в листинге 5.4: `fgets`, `writen`, `readline` и `fputs`, но порядок их следования внутри функции `str_cli` изменился. Раньше выполнение функции `str_cli` определялось функцией `fgets`, а теперь ее место заняла `select`. С помощью всего нескольких дополнительных строк кода (сравните листинги 6.1 и 5.4) мы значительно увеличили устойчивость клиента.

6.5. Пакетный ввод

К сожалению, наша функция `str_cli` все еще не вполне корректна. Сначала вернемся к ее исходной версии, приведенной в листинге 5.4. Эта функция работает в режиме остановки и ожидания (*stop-and-wait mode*), что удобно для интерактивного использования: функция отправляет строку серверу и затем ждет его ответа. Время ожидания складывается из одного периода обращения (RTT) и времени обработки сервером (которое близко к нулю в случае простого эхо-сервера). Следовательно, мы можем предположить, сколько времени займет отражение данного числа строк, если мы знаем время обращения (RTT) между клиентом и сервером.

Измерить RTT позволяет утилита `ping`. Если мы измерим с ее помощью время обращения к `connix.com` с нашего узла `solaris`, то средний период RTT после 30 измерений будет равен 175 мс. В [111, с. 89] показано, что это справедливо для дейтаграммы IP длиной 84 байт. Если мы возьмем первые 2000 строк файла `termcap Solaris 2.5`, то итоговый размер файла будет равен 98 349 байт, то есть в среднем 49 байт на строку. Если мы добавим размеры заголовка IP (20 байт) и заголовка TCP (20 байт), то средний сегмент TCP будет составлять 89 байт, почти как размер пакета утилиты `ping`. Следовательно, мы можем предположить, что общее время составит около 350 с для 2000 строк ($2000 \times 0,175$ с). Если мы запустим наш эхо-клиент TCP из главы 5, действительное время получится около 354 с, что очень близко к нашей оценке.

Если считать, что сеть между клиентом и сервером является двусторонним каналом, когда запросы идут от клиента серверу, а ответы в обратном направлении, то получится изображенный на рис. 6.8 режим остановки и ожидания.

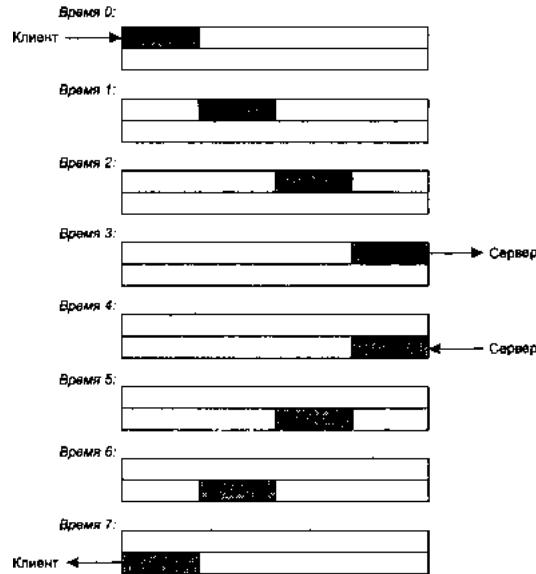


Рис. 6.8. Временная диаграмма режима остановки и ожидания: интерактивный ввод

Запрос отправляется клиентом в нулевой момент времени, и мы предполагаем, что время обращения RTT равно 8 условным единицам. Ответ, отправленный в момент времени 4, доходит до клиента в момент времени 7. Мы также считаем, что время обработки сервером нулевое и что размер запроса равен размеру ответа. Мы показываем только пакеты данных между клиентом и сервером, игнорируя подтверждения TCP, которые также передаются по сети.

Но поскольку между отправкой пакета и его приходом на другой конец канала существует задержка и канал является двусторонним, в этом примере мы используем только восьмую часть вместимости канала. Режим остановки и ожидания удобен для интерактивного ввода, но поскольку наш клиент считывает данные из стандартного потока ввода и записывает в стандартный поток вывода, а перенаправление ввода и вывода выполнить в интерпретаторе команд крайне просто, мы легко можем запустить наш клиент в пакетном режиме. Однако когда мы перенаправляем ввод и вывод, получающийся файл вывода всегда меньше файла ввода (а для эхо-сервера требуется их идентичность).

Чтобы понять происходящее, обратите внимание, что в пакетном режиме мы отправляем запросы так быстро, как их может принять сеть. Сервер обрабатывает их и отправляет обратно ответы с той же скоростью. Это приводит к тому, что в момент времени 7 канал целиком заполнен, как показано на рис. 6.9.

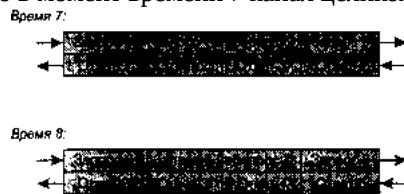


Рис. 6.9. Заполнение канала между клиентом и сервером: пакетный режим

Предполагается, что после отправки первого запроса мы немедленно посыпаем другой запрос и т.д. Также предполагается, что мы можем отправлять запросы с той скоростью, с какой сеть способна их принимать, и обрабатывать ответы так быстро, как сеть их поставляет.

ПРИМЕЧАНИЕ

Существуют различные нюансы, имеющие отношение к передаче большого количества данных TCP (bulk data flow), которые мы здесь игнорируем. К ним относится алгоритм медленного запуска (slow start algorithm), ограничивающий скорость, с которой данные отправляются на новое или незанятое соединение, и возвращаемые сегменты ACK. Все эти вопросы рассматриваются в главе 20 [111].

Чтобы увидеть, в чем заключается проблема с нашей функцией `str_cli`, представленной в листинге 6.1, будем считать, что файл ввода содержит только девять строк. Последняя строка отправляется в момент времени 8, как показано на рис. 6.9. Но мы не можем закрыть соединение после записи этого запроса, поскольку в канале еще есть другие запросы и ответы. Причина возникновения проблемы кроется в нашем способе обработки конца файла при вводе, когда процесс возвращается в функцию `main`, которая затем завершается. Но в пакетном режиме конец файла при вводе не означает, что мы закончили читать из сокета — в нем могут оставаться запросы к серверу или ответы от сервера.

Нам нужен способ закрыть одну половину соединения TCP. Другими словами, мы хотим отправить серверу сегмент FIN, тем самым сообщая ему, что закончили отправку данных, но оставляем дескриптор сокета открытым для чтения. Это делается с помощью функции `shutdown`, которая описывается в следующем разделе.

Вообще говоря, буферизация ввода-вывода для повышения производительности приводит к усложнению сетевых приложений (от чего пострадала и программа в листинге 6.1). Рассмотрим пример, в котором из стандартного потока ввода считывается несколько строк текста. Функция `select` передаст управление строке 20, в которой функция `fgets` считает доступные данные в буфер библиотеки `stdio`. Однако эта функция возвратит приложению только одну строку, а все остальные так и останутся в буфере. Считанная строка будет отправлена серверу, после чего будет снова вызвана функция `select`, которая будет ждать появления новых данных в стандартном потоке ввода несмотря на наличие еще не обработанных строк в буфере `stdio`. Причина в том, что `select` ничего не знает о буферах `stdio` и сообщает о доступности дескриптора для чтения с точки зрения системного вызова `read`, а не библиотечного вызова `fgets`. По этой причине использование `fgets` и `select` в одной программе считается опасным и требует особой осторожности.

Та же проблема связана с вызовом `readline` в листинге 6.1. Теперь данные скрываются от функции `select` уже не в буфере `stdio`, а в буфере `readline`. Вспомните, что в разделе 3.9 мы создали функцию, проверявшую состояние буфера `readline`. Мы могли бы воспользоваться ею перед вызовом `select`, чтобы проверить, нет ли в буфере `readline` данных, ожидающих обработки. Наша программа усложнится еще больше, если мы допустим, что буфер `readline` может содержать лишь часть строки (то есть нам придется дожидаться считывания этой строки целиком).

Проблемы буферизации мы постараемся решить в усовершенствованной версии `str_cli` в разделе 6.7.

6.6. Функция `shutdown`

Обычный способ завершить сетевое соединение — вызвать функцию `close`. Но у функции `close` есть два ограничения, которых лишена функция `shutdown`:

1. Функция `close` последовательно уменьшает счетчик ссылок дескриптора и закрывает сокет, только если счетчик доходит до нуля. Мы рассматривали это в разделе 4.8. Используя функцию `shutdown`, мы можем инициировать обычную последовательность завершения соединения TCP (четыре сегмента, начинающихся с FIN, на рис. 2.5) независимо от значения счетчика ссылок.

2. Функция `close` завершает оба направления передачи данных — и чтение, и запись. Поскольку соединение TCP является двусторонним, возможны ситуации, когда нам понадобится сообщить другому концу соединения, что мы закончили отправку, даже если на том конце соединения имеются данные для отправки нам. Это случай, рассмотренный в предыдущем разделе при описании работы нашей функции `str_cli` в пакетном режиме. На рис. 6.10 показаны типичные вызовы функций в этом сценарии.

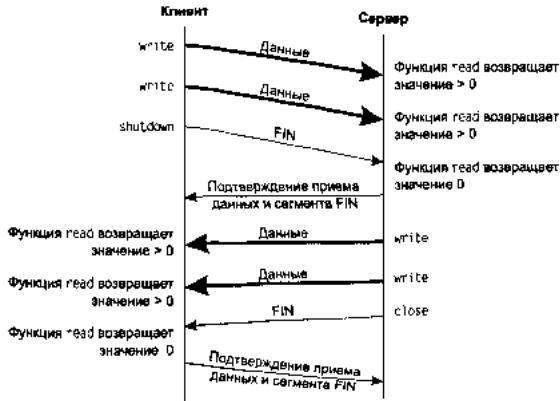


Рис. 6.10. Вызов функции `shutdown` для закрытия половины соединения TCP
`#include <sys/socket.h>`

```
int shutdown(int sockfd, int howto);
```

Возвращает: 0 в случае успешного выполнения, -1 в случае ошибки

Действие функции зависит от значения аргумента `howto`.

- `SHUT_RD`. Закрывается считающая половина соединения: из сокета больше нельзя считывать данные, и все данные, находящиеся в данный момент в буфере приема сокета, сбрасываются. Процесс больше не может выполнять функции чтения из сокета. Любые данные для сокета TCP, полученные после вызова функции `shutdown` с этим аргументом, подтверждаются и «молча» игнорируются.

ПРИМЕЧАНИЕ

По умолчанию все, что записывается в маршрутизирующий сокет (см. главу 17), возвращается как возможный ввод на все маршрутизирующие сокеты узла. Некоторые программы вызывают функцию `shutdown` со вторым аргументом `SHUT_RD`, чтобы предотвратить получение подобной копии. Другой способ избежать копирования — отключить параметр сокета `SO_USELOOPBACK`.

- `SHUT_WR`. Закрывается записывающая половина соединения. В случае TCP это называется *половинным закрытием* (см. раздел 18.5 [111]). Все данные, находящиеся в данный момент в буфере отправки сокета, будут отправлены, а затем будет выполнена обычная последовательность действий по завершению соединения TCP. Как мы отмечали ранее, закрытие записывающей половины соединения выполняется независимо от того, является ли значение в счетчике ссылок дескриптора сокета положительным или нет. Процесс теряет возможность записывать данные в сокет.

- `SHUT_RDWR`. Закрываются и читающая, и записывающая половины соединения. Это эквивалентно двум вызовам функции `shutdown`: сначала с аргументом `SHUT_RD`, затем — с аргументом `SHUT_WR`.

В табл. 7.4 приведены все возможные сценарии, доступные процессу при вызове функций `shutdown` и `close`. Действие функции `close` зависит от значения параметра сокета `SO_LINGER`.

ПРИМЕЧАНИЕ

Три константы `SHUT_xxx` определяются в спецификации POSIX. Типичные значения аргумента `howto`, с которыми вы встретитесь, — это 0 (закрытие читающей половины), 1 (закрытие записывающей половины) и 2 (закрытие обеих половин).

6.7. Функция `str_cli` (еще раз)

В листинге 6.2 представлена наша обновленная (и корректная) функция `str_cli`. В этой версии используются функции `select` и `shutdown`. Первая уведомляет нас о том, когда сервер закрывает свой конец

соединения, а вторая позволяет корректно обрабатывать пакетный ввод. Эта версия избавлена от ориентации на строки. Вместо этого она работает с буферами, что позволяет полностью избавиться от проблем, описанных в конце раздела 6.5.

Листинг 6.2. функция str_cli, использующая функцию select, которая корректно обрабатывает конец файла

```
//select/strcliselect02.c
1 #include "unp.h"

2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     int maxfdp1, stdineof;
6     fd_set rset;
7     char buf[MAXLINE];
8     int n;

9     stdineof = 0;
10    FD_ZERO(&rset);
11    for (;;) {
12        if (stdineof == 0)
13            FD_SET(fileno(fp), &rset);
14        FD_SET(sockfd, &rset);
15        maxfdp1 = max(fileno(fp), sockfd) + 1;
16        Select(maxfdp1, &rset, NULL, NULL, NULL);

17        if (FD_ISSET(sockfd, &rset)) { /* сокет готов для чтения */
18            if ((n = Read(sockfd, buf, MAXLINE)) == 0) {
19                if (stdineof == 1)
20                    return; /* нормальное завершение */
21                else
22                    err_quit("str_cli: server terminated prematurely");
23            }
24            Write(fileno(stdout), buf, n);
25        }

26        if (FD_ISSET(fileno(fp), &rset)) { /* есть данные на входе */
27            if ((n = Read(fileno(fp), buf, MAXLINE)) == 0) {
28                stdineof = 1;
29                Shutdown(sockfd, SHUT_WR); /* отправка сегмента FIN */
30                FD_CLR(fileno(fp), &rset);
31                continue;
32            }
33            Writen(sockfd, buf, n);
34        }
35    }
36 }
```

5-8 stdineof — это новый флаг, инициализируемый нулем. Пока этот флаг равен нулю, мы будем проверять готовность стандартного потока ввода к чтению с помощью функции select.

16-24 Если мы считываем на сокете признак конца файла, когда нам уже встретился ранее признак конца файла в стандартном потоке ввода, это является нормальным завершением и функция возвращает управление. Но если конец файла в стандартном потоке ввода еще не встречался, это означает, что процесс сервера завершился преждевременно. В новой версии мы вызываем функции read и write и работаем с буферами, а не со строками, благодаря чему функция select действует именно так, как мы рассчитывали.

25-33 Когда нам встречается признак конца файла на стандартном устройстве ввода, наш новый флаг `stdineof` устанавливается в единицу и мы вызываем функцию `shutdown` со вторым аргументом `SHUT_WR` для отправки сегмента FIN.

Если мы измерим время работы нашего клиента TCP, использующего функцию `str_cli`, показанную в листинге 6.2, с тем же файлом из 2000 строк, это время составит 12,3 с, что почти в 30 раз быстрее, чем при использовании версии этой функции, работающей в режиме остановки и ожидания.

Мы еще не завершили написание нашей функции `str_cli`: в разделе 15.2 мы разработаем ее версию с использованием неблокируемого ввода-вывода, а в разделе 23.3 — версию, работающую с программными потоками.

6.8. Эхо-сервер TCP (продолжение)

Вернемся к нашему эхо-серверу TCP из разделов 5.2 и 5.3. Перепишем сервер как одиночный процесс, который будет использовать функцию `select` для обработки любого числа клиентов, вместо того чтобы порождать с помощью функции `fork` по одному дочернему процессу для каждого клиента. Перед тем как представить этот код, взглянем на структуры данных, используемые для отслеживания клиентов. На рис. 6.11 показано состояние сервера до того, как первый клиент установил соединение.

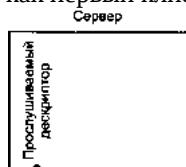


Рис. 6.11. Сервер TCP до того, как первый клиент установил соединение

У сервера имеется одиничный прослушиваемый дескриптор, показанный на рисунке точкой.

Сервер обслуживает только набор дескрипторов для чтения, который мы показываем на рис. 6.12. Предполагается, что сервер запускается в приоритетном (foreground) режиме, а дескрипторы 0, 1 и 2 соответствуют стандартным потокам ввода, вывода и ошибок. Следовательно, первым доступным для прослушиваемого сокета дескриптором является дескриптор 3. Массив целых чисел `client` содержит дескрипторы присоединенного сокета для каждого клиента. Все элементы этого массива инициализированы значением -1.

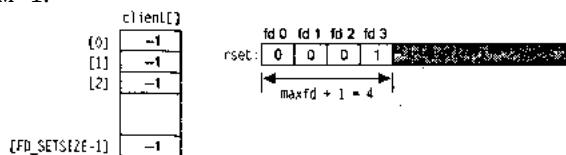


Рис. 6.12. Структуры данных для сервера TCP с одним прослушиваемым сокетом

Единственная ненулевая запись в наборе дескрипторов — это запись для прослушиваемого сокета, и поэтому первый аргумент функции `select` будет равен 4.

Когда первый клиент устанавливает соединение с нашим сервером, прослушиваемый дескриптор становится доступным для чтения и сервер вызывает функцию `accept`. Новый присоединенный дескриптор, возвращаемый функцией `accept`, будет иметь номер 4, если выполняются приведенные выше предположения. На рис. 6.13 показано соединение клиента с сервером.

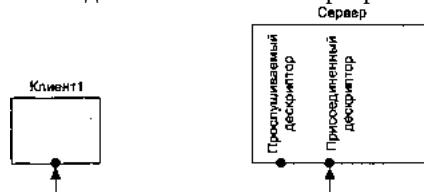


Рис. 6.13. Сервер TCP после того как первый клиент установил соединение

Теперь наш сервер должен запомнить новый присоединенный сокет в своем массиве `client`, и присоединенный сокет должен быть добавлен в набор дескрипторов. Изменившиеся структуры данных показаны на рис. 6.14.

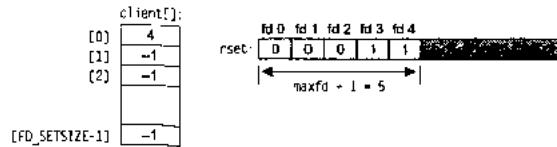


Рис. 6.14. Структуры данных после того как установлено соединение с первым клиентом

Через некоторое время второй клиент устанавливает соединение, и мы получаем сценарий, показанный на рис. 6.15.

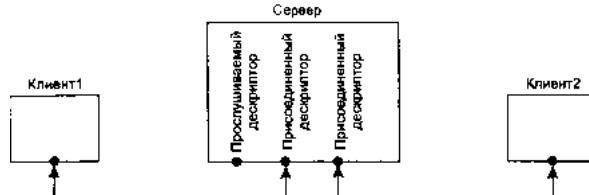


Рис. 6.15. Сервер TCP после того как установлено соединение со вторым клиентом

Новый присоединенный сокет (который имеет номер 5) должен быть размещен в памяти, в результате чего структуры данных меняются так, как показано на рис. 6.16.

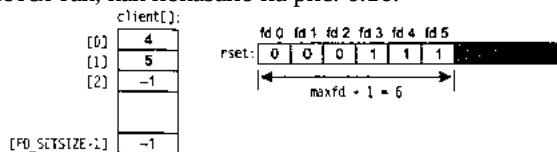


Рис. 6.16. Структуры данных после того как установлено соединение со вторым клиентом

Далее мы предположим, что первый клиент завершает свое соединение. TCP-клиент отправляет сегмент FIN, превращая тем самым дескриптор номер 4 на стороне сервера в готовый для чтения. Когда наш сервер считывает этот присоединенный сокет, функция `readline` возвращает нуль. Затем мы закрываем сокет, и соответственно изменяются наши структуры данных. Значение `client[0]` устанавливается в -1, а дескриптор 4 в наборе дескрипторов устанавливается в нуль. Это показано на рис. 6.17. Обратите внимание, что значение переменной `maxfd` не изменяется.

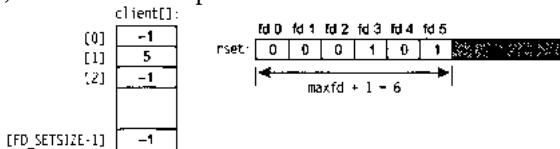


Рис. 6.17. Структуры данных после того как первый клиент разрывает соединение

Итак, по мере того как приходят клиенты, мы записываем дескриптор их присоединенного сокета в первый свободный элемент массива `client` (то есть в первый элемент со значением -1). Следует также добавить присоединенный сокет в набор дескрипторов для чтения. Переменная `maxi` — это наибольший используемый в данный момент индекс в массиве `client`, а переменная `maxfd` (плюс один) — это текущее значение первого аргумента функции `select`. Единственным ограничением на количество обслуживаемых сервером клиентов является минимальное из двух значений: `FD_SETSIZE` и максимального числа дескрипторов, которое допускается для данного процесса ядром (о чем мы говорили в конце раздела 6.3).

В листинге 6.3 показана первая половина этой версии сервера.

Листинг 6.3. Сервер TCP, использующий одиночный процесс и функцию `select`: инициализация

```
//tcpcliserv/tcpserselect01.c
```

```
1 #include "unp.h"
```

```
2 int
3 main(int argc, char **argv)
4 {
5     int i, maxi, maxfd, listenfd, connfd, sockfd;
6     int nready, client[FD_SETSIZE],
7     ssize_t n;
8     fd_set rset, allset;
9     char buf[MAXLINE];
```

```

10  socklen_t clilen;
11  struct sockaddr_in cliaddr, servaddr;
12
13  listenfd = Socket(AF_INET, SOCK_STREAM, 0);
14
15  bzero(&servaddr, sizeof(servaddr));
16  servaddr.sin_family = AF_INET;
17  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
18  servaddr.sin_port = htons(SERV_PORT);
19
20  Bind(listenfd, (SA*)&servaddr, sizeof(servaddr));
21
22  Listen(listenfd, LISTENQ);
23
24  maxfd = listenfd; /* инициализация */
25  maxi = -1; /* индекс в массиве client[] */
26  for (i = 0; i < FD_SETSIZE; i++)
27    client[i] = -1; /* -1 означает свободный элемент */
28  FD_ZERO(&allset);
29  FD_SET(listenfd, &allset);

```

Создание прослушиваемого сокета и инициализация функции select

12-24 Этапы создания прослушиваемого сокета те же, что и раньше: вызов функций `socket`, `bind` и `listen`. Мы инициализируем структуры данных при том условии, что единственный дескриптор, который мы с помощью функции `select` выберем, изначально является прослушиваемым сокетом.

Вторая половина функции `main` показана в листинге 6.4.

Листинг 6.4. Сервер TCP, использующий одиночный процесс и функцию `select`: цикл

```

//tcpcliserv/tcpserveiselect01.c
25  for (;;) {
26    rset = allset; /* присваивание значения структуре */
27    nready = Select(maxfd + 1, &rset, NULL, NULL, NULL);
28
29    if (FD_ISSET(listenfd, &rset)) { /* соединение с новым клиентом */
30      clilen = sizeof(cliaddr);
31      connfd = Accept(listenfd, (SA*)&cliaddr, &clilen);
32
33      for (i = 0; i < FD_SETSIZE; i++)
34        if (client[i] < 0) {
35          client[i] = connfd; /* сохраняем дескриптор */
36          break;
37        }
38      if (i == FD_SETSIZE)
39        err_quit("too many clients");
40
41      FD_SET(connfd, &allset); /* добавление нового дескриптора */
42      if (connfd > maxfd)
43        maxfd = connfd; /* для функции select */
44      if (i > maxi)
45        maxi = i; /* максимальный индекс в массиве client[] */
46
47      if (--nready <= 0)
48        continue; /* больше нет дескрипторов, готовых для чтения */
49    }
50    for (i = 0; i <= maxi; i++) /* проверяем все клиенты на наличие

```

```

        данных */
47     if ((sockfd - client[i]) < 0)
48         continue;
49     if (FD_ISSET(sockfd, &rset)) {
50         if ((n = Read(sockfd, buf, MAXLINE)) == 0) {
51             /* соединение закрыто клиентом */
52             Close(sockfd);
53             FD_CLR(sockfd, &allset);
54             client[i] = -1;
55         } else
56             Writen(sockfd, line, n);

57     if (--nready <= 0)
58         break; /* больше нет дескрипторов, готовых для чтения */
59 }
60 }
61 }
62 }

```

Блокирование в функции select

26-27 Функция `select` ждет, пока не будет установлено новое клиентское соединение или на существующем соединении не прибудут данные, сегмент FIN или сегмент RST.

Принятие новых соединений с помощью функции accept

28-45 Если прослушиваемый сокет готов для чтения, новое соединение установлено. Мы вызываем функцию `accept` и соответствующим образом обновляем наши структуры данных. Для записи присоединенного сокета мы используем первый незадействованный элемент массива `client`. Число готовых дескрипторов уменьшается, и если оно равно нулю, мы можем не выполнять следующий цикл `for`. Это позволяет нам использовать значение, возвращаемое функцией `select`, чтобы избежать проверки не готовых дескрипторов.

Проверка существующих соединений

46-60 Каждое существующее клиентское соединение проверяется на предмет того, содержится ли его дескриптор в наборе дескрипторов, возвращаемом функцией `select`. Если да, то из этого дескриптора считывается строка, присланная клиентом, и отражается обратно клиенту. Если клиент закрывает соединение, функция `read` возвращает нуль и мы обновляем структуры соответствующим образом.

Мы не уменьшаем значение переменной `maxi`, но могли бы проверять возможность сделать это каждый раз, когда клиент закрывает свое соединение.

Этот сервер сложнее, чем сервер, показанный в листингах 5.1 и 5.2, но он позволяет избежать затрат на создание нового процесса для каждого клиента, что является хорошим примером использования функции `select`. Тем не менее в разделе 15.6 мы опишем проблему, связанную с этим сервером, которая, однако, легко устраняется, если сделать прослушиваемый сокет неблокируемым, а затем проверить и проигнорировать несколько ошибок из функции `accept`.

Атака типа «отказ в обслуживании»

К сожалению, функционирование только что описанного сервера вызывает проблемы. Посмотрим, что произойдет, если некий клиент-злоумышленник соединится с сервером, отправит 1 байт данных (отличный от разделителя строк) и войдет в состояние ожидания. Сервер вызовет функцию `readline`, которая

прочитает одиночный байт данных от клиента и заблокируется в следующем вызове функции `read`, ожидая следующих данных от клиента. Сервер блокируется (вернее, «подвешивается») этим клиентом и не может предоставить обслуживание никаким другим клиентам (ни новым клиентским соединениям, ни данным существующих клиентов), пока упомянутый клиент-злоумышленник не отправит символ перевода строки или не завершит свой процесс.

Дело в том, что обрабатывая множество клиентов, сервер *никогда* не должен блокироваться в вызове функции, относящейся к одному клиенту. В противном можно «подвесить» сервер, что приведет к отказу в обслуживании для всех остальных клиентов. Это называется атакой типа «отказ в обслуживании» (DoS attack — Denial of Service). Такая атака воздействует на сервер, делая невозможным обслуживание нормальных клиентов. Обезопасить себя от подобных атак позволяют следующие решения: использовать неблокируемый ввод-вывод (см. главу 16), предоставлять каждому клиенту обслуживание отдельным потоком (например, для каждого клиента порождать процесс или поток) или установить тайм-аут для ввода-вывода (см. раздел 14.2).

6.9. Функция pselect

Функция `pselect` была введена в POSIX и в настоящий момент поддерживается множеством версий Unix.

```
#include <sys/select.h>
#include <signal.h>
#include <time.h>

int pselect(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
    const struct timespec *timeout, const sigset_t *sigmask);
```

Возвращает: количество готовых дескрипторов, 0 в случае тайм-аута, -1 в случае ошибки

Функция `pselect` имеет два отличия от обычной функции `select`:

1. Функция `pselect` использует структуру `timespec`, нововведение стандарта реального времени POSIX, вместо структуры `timeval`.

```
struct timespec {
    time_t tv_sec; /* секунды */
    long tv_nsec; /* наносекунды */
};
```

Эти структуры отличаются вторыми элементами: элемент `tv_nsec` новой структуры задает наносекунды, в то время как элемент `tv_usec` прежней структуры задает микросекунды.

2. В функции `pselect` добавляется шестой аргумент — указатель на маску сигналов. Это позволяет программе отключить доставку ряда сигналов, проверить какие-либо глобальные переменные, установленные обработчиками этих отключенных сигналов, а затем вызвать функцию `pselect`, сообщив ей, что нужно переустановить маску сигналов.

В отношении второго пункта рассмотрим следующий пример (описанный на с. 308–309 [110]). Обработчик сигнала нашей программы для сигнала `SIGINT` просто устанавливает глобальную переменную `intr_flag` и возвращает управление. Если наш процесс блокирован в вызове функции `select`, возвращение из обработчика сигнала заставляет функцию завершить работу, присвоив `errno` значение `EINTR`. Код вызова `select` выглядит следующим образом:

```
if (intr_flag)
    handle_intr(); /* обработка этого сигнала */
if ((nready = select(...)) < 0) {
    if (errno == EINTR) {
        if (intr_flag)
            handle_intr();
    }
    ...
}
```

Проблема заключается в том, что если сигнал придет в промежутке между проверкой переменной `intr_flag` и вызовом функции `select`, он будет потерян в том случае, если функция `select` заблокирует процесс навсегда. С помощью функции `pselect` мы можем переписать этот пример так, чтобы он работал более надежно:

```

sigset_t newmask, oldmask, zeromask;

sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* блокирование сигнала SIGINT */
if (intr_flag)
    handle_intr(); /* обработка этого сигнала */
if ((nready = pselect(..., &zeromask)) < 0) {
    if (errno == EINTR) {
        if (intr_flag)
            handle_intr();
    }
    ...
}

```

Перед проверкой переменной `intr_flag` мы блокируем сигнал `SIGINT`. Когда вызывается функция `pselect`, она заменяет маску сигналов процесса пустым набором (`zeromask`), а затем проверяет дескрипторы, возможно, переходя в состояние ожидания. Но когда функция `pselect` возвращает управление, маске сигналов процесса присваивается то значение, которое предшествовало вызову функции `pselect` (то есть сигнал `SIGINT` блокируется).

Мы поговорим о функции `pselect` более подробно и приведем ее пример в разделе 20.5. Функцию `pselect` мы используем в листинге 20.3, а в листинге 20.4 показываем простую, хотя и не вполне корректную реализацию этой функции.

ПРИМЕЧАНИЕ

Есть одно незначительное различие между функциями `select` и `pselect`. Первый элемент структуры `timeval` является целым числом типа `long` со знаком, в то время как первый элемент структуры `timspec` имеет тип `time_t`. Число типа `long` со знаком в первой функции также должно было относиться к типу `time_t`, но мы не меняли его тип, чтобы не разрушать существующего кода. Однако в новой функции это можно было бы сделать.

6.10. Функция poll

Функция `poll` появилась впервые в SVR3, и изначально ее применение ограничивалось потоковыми устройствами (STREAMS devices) (см. главу 31). В SVR4 это ограничение было снято, что позволило функции `poll` работать с любыми дескрипторами. Функция `poll` предоставляет функциональность, аналогичную функции `select`, но позволяет получать дополнительную информацию при работе с потоковыми устройствами.

```
#include <poll.h>
```

```
int poll(struct pollfd *fdarray, unsigned long nfds, int timeout);
```

Возвращает: количество готовых дескрипторов, 0 в случае тайм-аута, -1 в случае ошибки

Первый аргумент — это указатель на первый элемент массива структур. Каждый элемент массива — это структура `pollfd`, задающая условия, проверяемые для данного дескриптора `fd`.

```

struct pollfd {
    int fd;          /* дескриптор, который нужно проверить */
    short events;   /* события на дескрипторе, которые нас интересуют */
    short revents;  /* события, произошедшие на дескрипторе fd */
};
```

Проверяемые условия задаются элементом `events`, и состояние этого дескриптора функция возвращает в соответствующем элементе `revents`. (Наличие двух переменных для каждого дескриптора, одна из которых — значение, а вторая — результат, дает возможность обойтись без аргументов типа «значение-результат». Вспомните, что три средних аргумента функции `select` имеют тип «значение-

результат».) Каждый из двух элементов состоит из одного или более битов, задающих определенное условие. В табл. 6.2 перечислены константы, используемые для задания флага events и для проверки флага revents.

Таблица 6.2. Различные значения флагов events и revents для функции poll

Константа	На входе (events)	На выходе (revents)	Описание
POLLIN	•	•	Можно считывать обычные или приоритетные данные
POLLRDNORM	•	•	Можно считывать обычные данные
POLLRDBAND	•	•	Можно считывать приоритетные данные
POLLPRI	•	•	Можно считывать данные с высоким приоритетом
POLLOUT	•	•	Можно записывать обычные данные
POLLWRNORM	•	•	Можно записывать обычные данные
POLLWRBAND	•	•	Можно записывать приоритетные данные
POLLERR		•	Произошла ошибка
POLLHUP		•	Произошел разрыв соединения
POLLNVAL		•	Дескриптор не соответствует открытому файлу

Мы разделили эту таблицу на три части: первые четыре константы относятся ко вводу, следующие три — к выводу, а последние три — к ошибкам. Обратите внимание, что последние три константы не могут устанавливаться в элементе events, но всегда возвращаются в revents, когда выполняется соответствующее условие.

Существует три класса данных, различаемых функцией poll: *обычные*, *приоритетные* и *данные с высоким приоритетом*. Эти термины берут начало в реализациях, основанных на потоках (см. рис. 31.5).

ПРИМЕЧАНИЕ

Константа POLLIN может быть задана путем логического сложения констант POLLRDNORM и POLLRDBAND. Константа POLLIN существовала еще в реализациях SVR3, которые предшествовали полосам приоритета в SVR4, то есть эта константа существует в целях обратной совместимости. Аналогично, константа POLLOUT эквивалентна POLLWRNORM, и первая из них предшествовала второй.

Для сокетов TCP и UDP при описанных условиях функция poll возвращает указанный флаг revent. К сожалению, в определении функции poll стандарта POSIX имеется множество слабых мест (неоднозначностей):

- Все регулярные данные TCP и все данные UDP считаются обычными.
- Внеполосные данные TCP (см. главу 24) считаются приоритетными.
- Когдачитывающая половина соединения TCP закрывается (например, если получен сегмент FIN), это также считается равнозначным обычным данным, и последующая операция чтения возвратит нуль.
- Наличие ошибки для соединения TCP может расцениваться либо как обычные данные, либо как ошибка (POLLERR). В любом случае последующая функция read возвращает -1, что сопровождается установкой переменной errno в соответствующее значение. Это происходит при получении RST или истечении таймера.
- Информация о доступности нового соединения на прослушиваемом сокете может считаться либо обычными, либо приоритетными данными. В большинстве реализаций эта информация рассматривается как обычные данные.

Число элементов в массиве структур задается аргументом nfds.

ПРИМЕЧАНИЕ

Исторически этот аргумент имел тип long без знака, что является некоторым излишеством. Достаточно будет типа int без знака. В Unix 98 для этого аргумента определяется новый тип — nfds_t.

Аргумент timeout определяет, как долго функция находится в ожидании перед завершением. Положительным значением задается количество миллисекунд — время ожидания. В табл. 6.3 показаны возможные значения аргумента timeout.

Таблица 6.3. Значения аргумента timeout для функции poll

Значение аргумента timeout Описание

INFTIM	Ждать вечно
0	Возвращать управление немедленно, без блокирования
>0	Ждать в течение указанного числа миллисекунд

Константа INFTIM определена как отрицательное значение. Если таймер в данной системе не обеспечивает точность порядка миллисекунд, значение округляется в большую сторону до ближайшего поддерживаемого значения.

ПРИМЕЧАНИЕ

POSIX требует, чтобы константа INFTIM была определена в заголовочном файле <poll.h>, но многие системы все еще определяют ее в заголовочном файле <sys/stropts.h>.

Как и в случае функции select, любой тайм-аут, установленный для функции poll, ограничивается снизу разрешающей способностью часов в конкретной реализации (обычно 10 мс).

Функция poll возвращает -1, если произошла ошибка, 0 — если нет готовых дескрипторов до истечения времени таймера, иначе возвращается число дескрипторов с ненулевым элементом revents.

Если нас больше не интересует конкретный дескриптор, достаточно установить элемент fd структуры pollfd равным отрицательному значению. В этом случае элемент events будет проигнорирован, а элемент revents при возвращении функции будет сброшен в нуль.

Вспомните наши рассуждения в конце раздела 6.3 относительно константы FD_SETSIZE и максимального числа дескрипторов в наборе в сравнении с максимальным числом дескрипторов для процесса. У нас не возникает подобных проблем с функцией poll, поскольку вызывающий процесс отвечает за размещение массива структур pollfd в памяти и за последующее сообщение ядру числа элементов в массиве. Не существует типа данных фиксированного размера, аналогичного fd_set, о котором знает ядро.

ПРИМЕЧАНИЕ

POSIX требует наличия и функции select, и функции poll. Но если сравнивать их с точки зрения переносимости, то функцию select в настоящее время поддерживает больше систем, чем функцию poll. POSIX определяет также функцию pselect — усовершенствованную версию функции select, которая обеспечивает возможность блокирования сигналов и предоставляет лучшую разрешающую способность по времени, а для функции poll ничего подобного в POSIX нет.

6.11. Эхо-сервер TCP (еще раз)

Теперь мы изменим наш эхо-сервер TCP из раздела 6.8, используя вместо функции select функцию poll. В предыдущей версии сервера, работая с функцией select, мы должны были выделять массив client вместе с набором дескрипторов rset (см. рис. 6.12). С помощью функции poll мы разместим в памяти массив структур pollfd. В нем же мы будем хранить и информацию о клиенте, не создавая для нее другой массив. Элемент fd этого массива мы обрабатываем тем же способом, которым обрабатывали массив

client (см. рис. 6.12): значение -1 говорит о том, что элемент не используется, а любое другое значение является номером дескриптора. Вспомните из предыдущего раздела, что любой элемент в массиве структур pollfd, передаваемый функции poll с отрицательным значением элемента fd, просто игнорируется.

В листинге 6.5 показана первая часть кода нашего сервера.

Листинг 6.5. Первая часть сервера TCP, использующего функцию poll

```
//tcpcliserv/tcpservpoll01.c
1 #include "unp.h"
2 #include <limits.h> /* для OPEN_MAX */

3 int
4 main(int argc, char **argv)
5 {
6     int i, maxi, listenfd, connfd, sockfd;
7     int nready;
8     ssize_t n;
9     char buf[MAXLINE];
10    socklen_t clilen;
11    struct pollfd client[OPEN_MAX];
12    struct sockaddr_in cliaddr, servaddr;

13    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

14    bzero(&servaddr, sizeof(servaddr));
15    servaddr.sin_family = AF_INET;
16    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
17    servaddr.sin_port = htons(SERV_PORT);

18    Bind(listenfd, (SA*)&servaddr, sizeof(servaddr));

19    Listen(listenfd, LISTENQ);

20    client[0].fd = listenfd;
21    client[0].events = POLLRDNORM;
22    for (i = 1; i < OPEN_MAX; i++)
23        client[i].fd = -1; /* -1 означает, что элемент свободен */
24    maxi = 0; /* максимальный индекс массива client[] */
```

Размещение массива структур pollfd в памяти

11 Мы объявляем массив структур pollfd размером OPEN_MAX. Не существует простого способа определить максимальное число дескрипторов, которые могут быть открыты процессом. Мы снова столкнемся с этой проблемой в листинге 13.1. Один из способов ее решения — вызывать функцию POSIX sysconf с аргументом _SC_OPEN_MAX [110, с. 42-44], а затем динамически выделять в памяти место для массива соответствующего размера. Однако функция sysconf может возвратить некое «неопределенное» значение, и в этом случае нам придется задавать ограничение самим. Здесь мы используем только константу OPEN_MAX стандарта POSIX.

Инициализация

20-24 Мы используем первый элемент в массиве client для прослушиваемого сокета и присваиваем дескрипторам для оставшихся элементов -1. Мы также задаем в качестве аргумента функции poll событие POLLRDNORM, чтобы получить уведомление от этой функции в том случае, когда новое соединение будет готово к приему. Переменная maxi содержит максимальный индекс массива client, используемый в настоящий момент.

Вторая часть нашей функции приведена в листинге 6.6.

Листинг 6.6. Вторая часть сервера TCP, использующего функцию poll

```
//tcpcliserv/tcpservpoll01.c
25  for (;;) {
26      nready = Poll(client, maxi + 1, INFTIM);
27
28      if (client[0].revents & POLLRDNORM) { /* новое соединение
29          с клиентом */
30          clilen = sizeof(cliaddr);
31          connfd = Accept(listenfd, (SA*)&cliaddr, &clilen);
32
33          for (i = 1; i < OPEN_MAX; i++)
34              if (client[i].fd < 0) {
35                  client[i].fd = connfd; /* сохраняем дескриптор */
36                  break;
37              }
38          if (i == OPEN_MAX)
39              err_quit("too many clients");
39
40          client[i].events = POLLRDNORM;
41          if (i > maxi)
42              maxi = i; /* максимальный индекс в массиве client[] */
43
44          if (--nready <= 0)
45              continue; /* больше нет дескрипторов, готовых для чтения */
46      }
47      for (i = 1; i <= maxi; i++) /* проверяем все клиенты на наличие
48          данных */
49      if ((sockfd = client[i].fd) < 0)
50          continue;
51      if (client[i].revents & (POLLRDNORM | POLLERR)) {
52          if ((n = Read(sockfd, buf, MAXLINE)) < 0) {
53              if (errno == ECONNRESET) {
54                  /* соединение переустановлено клиентом */
55                  Close(sockfd);
56                  client[i].fd = -1;
57              } else
58                  err_sys("readline error");
59          } else if (n == 0) {
60              /* соединение закрыто клиентом */
61              Close(sockfd);
62              client[i].fd = -1;
63          } else
64              Writen(sockfd, line, n);
65
66          if (--nready <= 0)
67              break; /* больше нет дескрипторов, готовых для чтения */
68      }
69  }
```

Вызов функции poll, проверка нового соединения

26-42 Мы вызываем функцию `poll` для ожидания нового соединения либо данных на существующем соединении. Когда новое соединение принято, мы находим первый свободный элемент в массиве `client` — это первый элемент с отрицательным дескриптором. Обратите внимание, что мы начинаем поиск с индекса 1, поскольку элемент `client[0]` используется для прослушиваемого сокета. Когда свободный элемент найден, мы сохраняем дескриптор и устанавливаем событие `POLLRDNORM`.

Проверка данных на существующем соединении

43-63 Два события, которые нас интересуют, — это `POLLRDNORM` и `POLLERR`. Второй флаг в элементе `event` мы не устанавливали, поскольку этот флаг возвращается всегда, если соответствующее условие выполнено. Причина, по которой мы проверяем событие `POLLERR`, в том, что некоторые реализации возвращают это событие, когда приходит сегмент RST, другие же в такой ситуации возвращают событие `POLLRDNORM`. В любом случае мы вызываем функцию `read`, и если произошла ошибка, эта функция возвратит ее. Когда существующее соединение завершается клиентом, мы просто присваиваем элементу `fd` значение -1.

6.12. Резюме

В Unix существует пять различных моделей ввода-вывода:

- блокируемый ввод-вывод;
- неблокируемый ввод-вывод;
- мультиплексирование ввода-вывода;
- управляемый сигналом ввод-вывод;
- асинхронный ввод-вывод.

По умолчанию используется блокируемый ввод-вывод, и этот вариант встречается наиболее часто. Неблокируемый ввод-вывод и управляемый сигналом ввод-вывод мы рассмотрим в последующих главах. В этой главе мы рассмотрели мультиплексирование ввода-вывода. Асинхронный ввод-вывод определяется в стандарте POSIX, но поддерживающих его реализаций не так много.

Наиболее часто используемой функцией для мультиплексирования ввода-вывода является функция `select`. Мы сообщаем этой функции, какие дескрипторы нас интересуют (для чтения, записи или условия ошибки), а также передаем ей максимальное время ожидания и максимальное число дескрипторов (увеличенное на единицу). Большинство вызовов функции `select` определяют количество дескрипторов, готовых для чтения, и, как мы отметили, единственное условие исключения при работе с сокетами — это прибытие внеполосных данных (см. главу 21). Поскольку функция `select` позволяет ограничить время блокирования функции, мы используем это свойство в листинге 14.3 для ограничения по времени операции ввода.

Используя эхо-клиент в пакетном режиме с помощью функции `select`, мы выяснили, что даже если обнаружен признак конца файла, данные все еще могут находиться в канале на пути к серверу или от сервера. Обработка этого сценария требует применения функции `shutdown`, которая позволяет воспользоваться таким свойством TCP, как возможность половинного закрытия соединения (half-close feature).

POSIX определяет функцию `pselect` (повышающую точность таймера с микросекунд до наносекунд) которой передается новый аргумент — указатель на набор сигналов. Это позволяет избежать ситуации гонок (race condition) при перехвате сигналов, о которой мы поговорим более подробно в разделе 20.5.

Функция `poll` из System V предоставляет функциональность, аналогичную функции `select`. Кроме того, она обеспечивает дополнительную информацию при работе с потоковыми устройствами. POSIX требует наличия и функции `select`, и функции `poll`, но первая распространена шире.

Упражнения

1. Мы говорили, что набор дескрипторов можно присвоить другому набору дескрипторов, используя оператор присваивания языка C. Как это сделать, если набор дескрипторов является массивом целых чисел? (Подсказка: посмотрите на свой системный заголовочный файл `<sys/select.h>` или `<sys/types.h>`.)

2. Описывая в разделе 6.3 условия, при которых функция `select` сообщает, что дескриптор готов для записи, мы указали, что сокет должен быть неблокируемым, для того чтобы операция записи возвратила положительное значение. Почему?

3. Что произойдет с программой из листинга 6.1, если мы поставим слово `else` перед `if` в строке 19?

4. В листинге 6.3 добавьте необходимый код, чтобы позволить серверу использовать максимальное число дескрипторов, допустимое ядром (*Подсказка:* изучите функцию `setrlimit`.)

5. Посмотрите, что происходит, если в качестве второго аргумента функции `shutdown` передается `SHUT_RD`. Возьмите за основу код клиента TCP, представленный в листинге 5.3, и выполните следующие изменения: вместо номера порта `SERV_PORT` задайте порт 19 (служба `chargen`, см. табл. 2.1), а также замените вызов функции `str_cli` вызовом функции `pause`. Запустите программу, задав IP-адрес локального узла, на котором выполняется сервер `chargen`. Просмотрите пакеты с помощью такой программы, как, например, `tcpdump` (см. раздел B.5). Что происходит?

6. Почему приложение должно вызывать функцию `shutdown` с аргументом `SHUT_RDWR`, вместо того чтобы просто вызвать функцию `close`?

7. Что происходит в листинге 6.4, когда клиент отправляет RST для завершения соединения?

8. Перепишите код, показанный в листинге 6.5, чтобы вызывать функцию `sysconf` для определения максимального числа дескрипторов и размещения соответствующего массива `client` в памяти.

Глава 7

Параметры сокетов

7.1. Введение

Существуют различные способы получения и установки параметров сокетов:

- функции `getsockopt` и `setsockopt`;
- функция `fcntl`;
- функция `ioctl`.

Эту главу мы начнем с описания функций `getsockopt` и `setsockopt`. Далее мы приведем пример, в котором выводятся заданные по умолчанию значения параметров, а затем дадим подробное описание всех параметров сокетов. Мы разделили описание параметров на следующие категории: общие, IPv4, IPv6, TCP и SCTP. При первом прочтении главы можно пропустить подробное описание параметров и при необходимости прочесть отдельные разделы, на которые даны ссылки. Отдельные параметры подробно описываются в дальнейших главах, например параметры многоадресной передачи IPv4 и IPv6 мы обсуждаем в разделе 19.5.

Мы также рассмотрим функцию `fcntl`, поскольку она реализует предусмотренные стандартом POSIX возможности отключить для сокета блокировку ввода-вывода, включить управление сигналами, а также установить владельца сокета. Функцию `ioctl` мы опишем в главе 17.

7.2. Функции `getsockopt` и `setsockopt`

Эти две функции применяются только к сокетам.

```
#include <sys/socket.h>
```

```
int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

Обе функции возвращают 0 в случае успешного завершения, -1 в случае ошибки

Переменная `sockfd` должна ссылаться на открытый дескриптор сокета. Переменная `level` определяет, каким кодом должен интерпретироваться параметр: общими программами обработки сокетов или зависящими от протокола программами (например, IPv4, IPv6, TCP или SCTP).

`optval` — это указатель на переменную, из которой извлекается новое значение параметра с помощью функции `setsockopt` или в которой сохраняется текущее значение параметра с помощью функции `getsockopt`. Размер этой переменной задается последним аргументом. Для функции `setsockopt` тип этого аргумента — *значение*, а для функции `getsockopt` — «*значение-результат*».

В табл. 7.1 и 7.2 сведены параметры, которые могут запрашиваться функцией `getsockopt` или устанавливаться функцией `setsockopt`. В колонке «Тип данных» приводится тип данных того, на что указывает указатель `optval` для каждого параметра. Две фигурные скобки мы используем, чтобы обозначить структуру, например `linger{}` обозначает `struct linger`.

Таблица 7.1. Параметры сокетов для функций `getsockopt` и `setsockopt`

level	optname	get	set	Описание	Флаг	Тип данных
SOL_SOCKET	SO_BROADCAST	•	•	Позволяет посыпать широковещательные дейтаграммы	•	int
	SO_DEBUG	•	•	Разрешает отладку	•	int
	SO_DONTROUTE	•	•	Обходит таблицу маршрутизации	•	int
	SO_ERROR	•		Получает ошибку, ожидающую обработки, и возвращает значение		int

		параметра в исходное состояние	
	SO_KEEPALIVE	• • Проверяет, находится ли соединение в рабочем состоянии	• int
	SO_LINGER	• • Задерживает закрытие сокета, если имеются данные для отправки	linger{}
	SO_OOBINLINE	• • Оставляет полученные внеполосные данные вместе с обычными данными (inline)	• int
	SO_RCVBUF	• • Размер приемного буфера	int
	SO_SNDBUF	• • Размер буфера отправки	int
	SO_RCVLOWAT	• • Минимальное количество данных для приемного буфера сокета	int
	SO SNDLOWAT	• • Минимальное количество данных для буфера отправки сокета	int
	SO_RCVTIMEO	• • Тайм-аут при получении	timeval{}
	SO_SNDTIMEO	• • Тайм-аут при отправке	timeval{}
	SO_REUSEADDR	• • Допускает повторное использование локального адреса	• int
	SO_REUSEPORT	• • Допускает повторное использование локального адреса	• int
	SO_TYPE	• Возвращает тип сокета	int
	SO_USELOOPBACK	• • Маршрутизирующий сокет получает копию того, что он отправляет	• int
IPPROTO_IP	IP_HDRINCL	• • Включается IP-заголовок	• int
	IP_OPTIONS	• • В заголовке IPv4 устанавливаются параметры IP	см. текст
	IP_RECVSTADDR	• • Возвращает IP-адрес получателя	• int
	IP_RECVIF	• • Возвращает индекс интерфейса, на котором принимается дейтаграмма UDP	• int
	IP_TOS	• • Тип сервиса и приоритет	int
	IP_TTL	• • Время жизни	int
	IP_MULTICAST_IF	• • Задает интерфейс для in_addr{}	

		исходящих дейтаграмм	
	IP_MULTICAST_TTL	• • Задает TTL для исходящих дейтаграмм	u_char
	IP_MULTICAST_LOOP	Разрешает или отменяет отправку копии дейтаграммы на тот узел, откуда она была послана (loopback)	u_char
	IP_ADD_MEMBERSHIP	Включение в группу многоадресной передачи	ip_mreq{}
	IP_DROP_MEMBERSHIP	Отключение от группы многоадресной передачи	ip_mreq{}
	IP_{BLOCK, UNBLOCK}_SOURCE	Блокирование и разблокирование источника многоадресной передачи	ip_mreq_source{}
	IP_{ADD, DROP}_SOURCE_MEMBERSHIP	Присоединение или отключение от многоадресной передачи от источника (source-specific)	ip_mreq_source{}
IPPROTO_ICMPV6	ICMP6_FILTER	Указывает тип сообщения ICMPv6, которое передается процессу	icmp6_filter{}
IPPROTO_IPV6	IPV6_ADDRFORM	Меняет формат адреса сокета	int
	IPV6_CHECKSUM	Отступ поля контрольной суммы для символьных (неструктурированных) сокетов	int
	IPV6_DONTFRAG	Не фрагментировать, а сбрасывать большие пакеты	int
	IPV6_NEXTHOP	Задает следующий транзитный адрес	sockaddr{}
	IPV6_PATHMTU	Получение текущей маршрутной МТУ	ip6_mtuinfo{}
	IPV6_RECVDSTOPTS	Получение параметров адресата	int
	IPV6_RECVHOPLIMIT	Получение ограничения на количество транзитных узлов при направленной передаче	int
	IPV6_RECVHOPOPTS	Получение параметров прыжков	int
	IPV6_RECVPATHMTU	Получение	int

		маршрутной MTU	
IPV6_RECVPKTINFO	• .	Получение информации о пакетах	• int
IPV6_RECVRTHDR	• .	Получение маршрута от источника	• int
IPV6_RECVTCLASS	• .	Получение класса трафика	• int
IPV6_UNICAST_HOPS	• .	Предел количества транзитных узлов, задаваемый по умолчанию	int
IPV6_USE_MIN_MTU	• .	Использовать минимальную MTU	• int
IPV6_V60NLY	• .	Отключить совместимость с IPv4	• int
IPV6_XXX	• .	Вспомогательные данные	см. текст
IPV6_MULTICAST_IF	• .	Задает интерфейс для исходящих дейтаграмм	u_int
IPV6_MULTICAST_HOPS	• .	Задает предельное количество транзитных узлов для исходящих широковещательных сообщений	int
IPV6_MULTICAST_LOOP	• .	Разрешает или отменяет отправку копии дейтаграммы на тот узел, откуда она была послана (loopback)	u_int
IPV6_LEAVE_GROUP	• .	Выход из группы многоадресной передачи	ipv6_mreq{}
MCAST_JOIN_GROUP	• .	Присоединение к группе многоадресной передачи	group_req{}
MCAST_LEAVE_GROUP	• .	Выход из группы многоадресной передачи	group_source_r
MCAST_BLOCK_SOURCE	• .	Блокирование источника многоадресной передачи	group_source_r
IPPROTO_IP или IPPROTO_IPV6		Разблокирование источника многоадресной передачи	
MCAST_UNBLOCK_SOURCE	• .	Присоединение к группе многоадресной передачи от источника	group_source_r
MCAST_JOIN_SOURCE_GROUP	• .	Выход из группы многоадресной передачи от источника	group_source_r
MCAST_LEAVE_SOURCE_GROUP	• .	Присоединение к группе многоадресной передачи от источника	group_source_r

Таблица 7.2. Параметры сокетов транспортного уровня

Level	optname	get	set	Описание	Флаг	Тип данных
IPPROTO_TCP	TCP_MAXSEG	•	•	Максимальный размер сегмента TCP		int
	TCP_NODELAY	•	•	Отключает алгоритм Нагла	•	int
IPPROTO_SCTP	SCTP_ADAPTION_LAYER	•	•	Указание на уровень адаптации		sctp_setadaption
	SCTP_ASSOCINFO	+ •	•	Получение и задание сведений об ассоциации		sctp_assocparamms{}
	SCTP_AUTOCLOSE	•	•	Автоматическое закрытие		int
	SCTP_DEFAULT_SEND_PARAM	•	•	Параметры отправки по умолчанию		sctp_sndrcvinfo{}
	SCTP_DISABLE_FRAGMENTS	•	•	Фрагментация SCTP	•	int
	SCTP_EVENTS	•	•	Уведомление об интересующих событиях		sctp_event_subscribe{}
	SCTP_GET_PEER_ADDR_INFO	+ •		Получение состояния адреса собеседника		sctp_paddrinfo{}
	SCTP_I_WANT_MAPPED_V4_ADDR	•	•	Отображение адресов IPv4	•	int
	SCTP_INITMSG	•	•	Параметры пакета INIT по умолчанию		sctp_initmsg{}
	SCTP_MAXBURST	•	•	Максимальный размер набора пакетов		int
	SCTP_MAXSEG	•	•	Максимальный размер фрагментации		int
	SCTP_NODELAY	•	•	Отключение алгоритма Нагла	•	int
	SCTP_PEER_ADDR_PARAMS	+ •	•	Параметры адреса собеседника		sctp_paddrparams{}
	SCTP_PRIMARY_ADDR	+ •	•	Основной адрес назначения		sctp_setprim{}
	SCTP_RTOINFO	+ •	•	Информация RTO		sctp_rtinfo{}
	SCTP_SET_PEER_PRIMARY_ADDR	•	•	Основной адрес назначения собеседника		sctp_setpeerprim{}

SCTP_STATUS	+	Получение сведений о статусе ассоциации	sctp_status{}
-------------	---	--------------------------------------------------	---------------

Существует два основных типа параметров: двоичные параметры, включающие или отключающие определенное свойство (флаги), и параметры, получающие и возвращающие значения параметров, которые мы можем либо задавать, либо проверять. В колонке «Флаг» указывается, относится ли параметр к флагам. Для флагов при вызове функции `getsockopt` аргумент `*optval` является целым числом. Возвращаемое значение `*optval` нулевое, если параметр отключен, и ненулевое, если параметр включен. Аналогично, функция `setsockopt` требует ненулевого значения `*optval` для включения параметра, и нулевого значения — для его выключения. Если в колонке «Флаг» не содержится символа «*», то параметр используется для передачи значения заданного типа между пользовательским процессом и системой.

В последующих разделах этой главы приводятся дополнительные подробности о параметрах сокетов.

7.3. Проверка наличия параметра и получение значения по умолчанию

Напишем программу, которая проверяет, поддерживается ли большинство параметров, представленных в табл. 7.1 и 7.2, и если да, то выводит их значения, заданные по умолчанию. В листинге 7.1^[1] содержатся объявления нашей программы.

Листинг 7.1. Объявления для нашей программы, проверяющей параметры сокетов

```
//sockopt/checkopts.c
1 #include "unp.h"
2 #include <netinet/tcp.h> /* определения констант TCP_XXX */

3 union val {
4     int i_val;
5     long l_val;
6     struct linger linger_val;
7     struct timeval timeval_val;
8 } val;

9 static char *sock_str_flag(union val*, int);
10 static char *sock_str_int(union val*, int);
11 static char *sock_str_linger(union val*, int);
12 static char *sock_str_timeval(union val*, int);

13 struct sock_opts {
14     const char *opt_str;
15     int opt_level;
16     int opt_name;
17     char *(*opt_val_str)(union val*, int);
18 } sock_opts[] = {
19 { "SO_BROADCAST",      SOL_SOCKET,    SO_BROADCAST,    sock_str_flag },
20 { "SO_DEBUG",          SOL_SOCKET,    SO_DEBUG,        sock_str_flag },
21 { "SO_DONTROUTE",      SOL_SOCKET,    SO_DONTROUTE,   sock_str_flag },
22 { "SO_ERROR",          SOL_SOCKET,    SO_ERROR,        sock_str_int },
23 { "SO_KEEPALIVE",      SOL_SOCKET,    SO_KEEPALIVE,   sock_str_flag },
24 { "SO_LINGER",          SOL_SOCKET,    SO_LINGER,       sock_str_linger },
25 { "SO_OOBINLINE",      SOL_SOCKET,    SO_OOBINLINE,   sock_str_flag },
26 { "SO_RCVBUF",          SOL_SOCKET,    SO_RCVBUF,      sock_str_int },
27 { "SO_SNDBUF",          SOL_SOCKET,    SO_SNDBUF,      sock_str_int },
28 { "SO_RCVLOWAT",        SOL_SOCKET,    SO_RCVLOWAT,   sock_str_int },
29 { "SO SNDLOWAT",        SOL_SOCKET,    SO SNDLOWAT,   sock_str_int },
30 { "SO_RCVTIMEO",        SOL_SOCKET,    SO_RCVTIMEO,   sock_str_timeval },
31 { "SO SNDTIMEO",        SOL_SOCKET,    SO SNDTIMEO,   sock_str_timeval },
```

```

32 { "SO_REUSEADDR",      SOL_SOCKET,    SO_REUSEADDR,    sock_str_flag },
33 #ifdef SO_REUSEPORT
34 { "SO_REUSEPORT",      SOL_SOCKET,    SO_REUSEPORT,    sock_str_flag },
35 #else
36 { "SO_REUSEPORT",      0,              0,    NULL },
37 #endif
38 { "SO_TYPE",           SOL_SOCKET,    SO_TYPE,        sock_str_int },
39 { "SO_USELOOPBACK",    SOL_SOCKET,    SO_USELOOPBACK, sock_str_flag },
40 { "IP_TOS",             IPPROTO_IP,   IP_TOS,        sock_str_int },
41 { "IP_TTL",             IPPROTO_IP,   IP_TTL,        sock_str_int },
42 { "IPV6_DONTFRAG",    IPPROTO_IPV6, IPV6_DONTFRAG, sock_str_flag },
43 { "IPV6_UNICAST_HOPS", IPPROTO_IPV6, IPV6_UNICAST_HOPS, sock_str_int },
44 { "IPV6_V6ONLY",       IPPROTO_IPV6, IPV6_V6ONLY,   sock_str_flag },
45 { "TCP_MAXSEG",        IPPROTO_TCP,   TCP_MAXSEG,   sock_str_int },
46 { "TCP_NODELAY",       IPPROTO_TCP,   TCP_NODELAY,  sock_str_flag },
47 { "SCTP_AUTOCLOSE",   IPPROTO_SCTP, SCTP_AUTOCLOSE, sock_str_int },
48 { "SCTP_MAXBURST",   IPPROTO_SCTP, SCTP_MAXBURST, sock_str_int },
49 { "SCTP_MAXSEG",       IPPROTO_SCTP, SCTP_MAXSEG,  sock_str_int },
50 { "SCTP_NODELAY",     IPPROTO_SCTP, SCTP_NODELAY, sock_str_flag },
51 { NULL,                  0,              0,    NULL }
52 };

```

Объявление объединения возможных значений

3-9 Наше объединение `val` содержит по одному элементу для каждого возможного возвращаемого значения из функции `getsockopt`.

Задание прототипов функций

10-13 Мы определяем прототипы для четырех функций, которые вызываются для вывода значения данного параметра сокета.

Задание структуры и инициализация массива

14-46 Наша структура `sock_opts` содержит всю информацию, которая необходима, чтобы вызвать функцию `getsockopt` для каждого из параметров сокета и вывести его текущее значение. Последний элемент, `opt_val_str`, является указателем на одну из четырех функций, которые выводят значение параметра. Мы размещаем в памяти и инициализируем массив этих структур, каждый элемент которого соответствует одному параметру сокета.

ПРИМЕЧАНИЕ

Не все реализации поддерживают полный набор параметров сокетов. Чтобы определить, поддерживается ли данный параметр, следует использовать `#ifdef` или `#if defined`, как показано для параметра `SO_REUSEPORT`. Для полноты картины требуется обработать подобным образом все параметры, но в книге мы пренебрегаем этим, потому что `#ifdef` только удлиняет показанный код и не влияет на суть дела.

В листинге 7.2 показана наша функция `main`.

Листинг 7.2. Функция `main` для проверки параметров сокетов

```
//sockopt/checkopts.c
```

```
53 int
```

```

54 main(int argc, char **argv)
55 {
56     int fd;
57     socklen_t len;
58     struct sock_opts *ptr;
59
60     for (ptr = sock_opts; ptr->opt_str != NULL; ptr++) {
61         printf("%s: ptr->opt_str");
62         if (ptr->opt_val_str == NULL)
63             printf("(undefined)\n");
64         else {
65             switch(ptr->opt_level) {
66                 case SOL_SOCKET:
67                 case IPPROTO_IP:
68                 case IPPROTO_TCP:
69                     fd = Socket(AF_INET, SOCK_STREAM, 0);
70                     break;
71                 case IPPROTO_IPV6:
72                     fd = Socket(AF_INET6, SOCK_STREAM, 0);
73                     break;
74             #endif
75             #ifdef IPPROTO_SCTP
76                 case IPPROTO_SCTP:
77                     fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
78                     break;
79             #endif
80             default:
81                 err_quit("Can't create fd for level %d\n", ptr->opt_level);
82         }
83
84         len = sizeof(val);
85         if (getsockopt(fd, ptr->opt_level, ptr->opt_name,
86                         &val, &len) == -1) {
87             err_ret("getsockopt error");
88         } else {
89             printf("default = %s\n", (*ptr->opt_val_str)(&val, len));
90         }
91         close(fd);
92     }
93     exit(0);
94 }
```

Перебор всех параметров

59-63 Мы перебираем все элементы нашего массива. Если указатель opt_val_str пустой, то параметр не определен реализацией (что, как мы показали, возможно для SO_REUSEPORT).

Создание сокета

63-82 Мы создаем сокет, на котором проверяем действие параметров. Для проверки параметров сокета и уровней IPv4 и TCP мы используем сокет IPv4 TCP. Для проверки параметров сокетов уровня IPv6 мы используем сокет IPv6 TCP, а для проверки параметров SCTP — сокет IPv4 SCTP.

Вызов функции *getsockopt*

83-87 Мы вызываем функцию *getsockopt*, но не завершаем ее выполнение, если возвращается ошибка. Многие реализации определяют имена некоторых параметров сокетов, даже если не поддерживают эти параметры. Неподдерживаемые параметры выдают ошибку ENOPROTOOPT.

Вывод значения параметра по умолчанию

88-89 Если функция *getsockopt* успешно завершается, мы вызываем нашу функцию для преобразования значения параметра в строку и выводим эту строку.

В листинге 7.1 мы показали четыре прототипа функций, по одному для каждого типа возвращаемого значения параметра. В листинге 7.3 показана одна из этих функций, *sock_str_flag*, которая выводит значение параметра, являющегося флагом. Другие три функции аналогичны этой.

Листинг 7.3. Функция *sock_str_flag*: преобразование флага в строку

```
//sockopt/checkopts.c
95 static char strres[128];

96 static char *
97 sock_str_flag(union val *ptr, int len)
98 {
99     if (len != sizeof(int))
100     snprintf(strres, sizeof(strres), "size (%d) not sizeof(int)", len);
101 else
102     snprintf(strres, sizeof(strres),
103         "%s", (ptr->i_val == 0) ? "off" : "on");
104 return(strres);
105 }
```

99-104 Вспомните, что последний аргумент функции *getsockopt* — это аргумент типа «значение-результат». Первое, что мы проверяем, — это то, что размер значения, возвращаемого функцией *getsockopt*, совпадает с предполагаемым. В зависимости от того, является ли значение флага нулевым или нет, возвращается строка *off* или *on*.

Выполнение этой программы под FreeBSD 4.8 с пакетами обновлений KAME SCTP дает следующий вывод:

```
freebsd % checkopts
SO_BROADCAST: default = off
SO_DEBUG: default = off
SO_DONTROUTE: default = off
SO_ERROR: default = 0
SO_KEEPALIVE: default = off
SO_LINGER: default = l_onoff = 0, l_linger = 0
SO_OOBINLINE: default = off
SO_RCVBUF: default = 57344
SO_SNDBUF: default = 32768
SO_RCVLOWAT: default = 1
SO_SNDLOWAT: default = 2048
SO_RCVTIMEO: default = 0 sec, 0 usec
SO SNDTIMEO: default = 0 sec, 0 usec
SO_REUSEADDR: default = off
SO_REUSEPORT: default = off
SO_TYPE: default = 1
SO USELOOPBACK: default = off
IP_TOS: default = 0
IP_TTL: default = 64
IPV6_DONTFRAG: default = off
```

```
IPV6_UNICAST_HOPS: default = -1
IPV6_V6ONLY: default = off
TCP_MAXSEG: default = 512
TCP_NODELAY: default = off
SCTP_AUTOCLOSE: default = 0
SCTP_MAXBURST: default = 4
SCTP_MAXSEG: default = 1408
SCTP_NODELAY: default = off
```

Значение 1, возвращаемое для параметра `SO_TYPE`, для этой реализации соответствует `SOCK_STREAM`.

7.4. Состояния сокетов

Для некоторых параметров сокетов время их установки или получения зависит некоторым образом от состояния сокета. Далее мы обсудим эту зависимость для тех параметров, к которым это относится.

Следующие параметры сокетов наследуются присоединенным сокетом TCP от прослушиваемого сокета [128, с. 462-463]: `SO_DEBUG`, `SO_DONTROUTE`, `SO_KEEPALIVE`, `SO_LINGER`, `SO_OOBINLINE`, `SO_RCVBUF`, `SO_RCVLOWAT`, `SO_SNDBUF`, `SO SNDLOWAT`, `TCP_MAXSEG` и `TCP_NODELAY`. Это важно для TCP, поскольку присоединенный сокет не возвращается серверу функцией `accept`, пока трехэтапное рукопожатие не завершится на уровне TCP. Если при завершении трехэтапного рукопожатия мы хотим убедиться, что один из этих параметров установлен для присоединенного сокета, нам следует установить этот параметр еще для прослушиваемого сокета.

7.5. Общие параметры сокетов

Мы начнем с обсуждения общих параметров сокетов. Эти параметры не зависят от протокола (то есть они управляются не зависящим от протокола кодом внутри ядра, а не отдельным модулем протокола, такого как IPv4), но некоторые из них применяются только к определенным типам сокетов. Например, несмотря на то что параметр сокета `SO_BROADCAST` называется общим, он применяется только к дейтаграммным сокетам.

Параметр сокета `SO_BROADCAST`

Этот параметр управляет возможностью отправки широковещательных сообщений. Широковещательная передача поддерживается только для сокетов дейтаграмм и только в сетях, поддерживающих концепцию широковещательных сообщений (Ethernet, Token Ring и т.д.). Широковещательная передача в сетях типа «точка-точка» или по ориентированному на установление соединения транспортному протоколу типа SCTP или TCP, неосуществима. Более подробно о широковещательной передаче мы поговорим в главе 18.

Поскольку перед отправкой широковещательной дейтаграммы приложение должно установить этот параметр сокета, оно не сможет отправить широковещательное сообщение, если это не предполагалось заранее. Например, приложение UDP может принять IP-адрес получателя в качестве аргумента командной строки, но оно может и не предполагать, что пользователь вводит широковещательный адрес. Проверку того, является ли данный адрес широковещательным, осуществляет не приложение, а ядро: если адрес получателя является широковещательным адресом и данный параметр сокета не установлен, возвратится ошибка `EACCESS` [128, с. 233].

Параметр сокета `SO_DEBUG`

Этот параметр поддерживается только протоколом TCP. При подключении к сокету TCP ядро отслеживает подробную информацию обо всех пакетах, отправленных или полученных протоколом TCP для сокета. Они хранятся в кольцевом буфере внутри ядра, который можно проверить с помощью программы `trpt`. В [128, с. 916-920] приводится более подробная информация и пример использования этого параметра.

Параметр сокета SO_DONTROUTE

Этот параметр указывает, что исходящие пакеты должны миновать обычные механизмы маршрутизации соответствующего протокола. Например, в IPv4 пакет направляется на соответствующий локальный интерфейс, который задается адресом получателя, а именно сетевым адресом и маской подсети. Если локальный интерфейс не может быть определен по адресу получателя (например, получателем не является другой конец соединения типа «точка-точка» или он не находится в той же сети), возвращается ошибка ENETUNREACH.

Эквивалент этого параметра можно также применять к индивидуальным дейтаграммам, используя флаг MSG_DONTROUTE с функциями send, sendto или sendmsg.

Этот параметр часто используется демонами маршрутизации (routed и gated) для того, чтобы миновать таблицу маршрутизации (в случае, если таблица маршрутизации неверна) и заставить пакет отправиться на определенный интерфейс.

Параметр сокета SO_ERROR

Когда на сокете происходит ошибка, модуль протокола в ядре, происходящем от Беркли, присваивает переменной so_error для этого сокета одно из стандартных значений Unix `Exxx`. Это так называемая *ошибка, требующая обработки* (*pending error*) для данного сокета. Процесс может быть немедленно оповещен об ошибке одним из двух способов:

1. Если процесс блокируется в вызове функции `select` (см. раздел 6.3), ожидая готовности данного сокета к чтению или записи, функция `select` возвращает управление и уведомляет процесс о соответствующем состоянии готовности.

2. Если процесс использует управляемый сигналом ввод-вывод (см. главу 25), для него или для группы таких процессов генерируется сигнал `SIGIO`.

Процесс может получить значение переменной `so_error`, указав параметр сокета `SO_ERROR`. Целое значение, возвращаемое функцией `getsockopt`, является кодом ошибки, требующей обработки. Затем значение переменной `so_error` сбрасывается ядром в 0 [128, с. 547].

Если процесс вызывает функцию `read` и возвращаемых данных нет, а значение `so_error` ненулевое, то функция `read` возвращает -1 с `errno`, которой присвоено значение переменной `so_error` [128, с. 516]. Это значение `so_error` затем сбрасывается в 0. Если в очереди для сокета есть данные, эти данные возвращаются функцией `read` вместо кода ошибки. Если значение `so_error` ненулевое, то при вызове процессом функции `write` возвращается -1 с `errno`, равной значению переменной `so_error` [128, с. 495], а значение `so_error` сбрасывается в 0.

ПРИМЕЧАНИЕ

В коде, показанном на с. 495 [128], есть ошибка: `so_error` не сбрасывается в 0. Она была выявлена в реализации BSD/OS. Всегда, когда для сокета возвращается ошибка, требующая обработки, `so_error` должна быть сброшена в 0.

Здесь вы впервые встречаетесь с параметром сокета, который можно получить, но нельзя установить.

Параметр сокета SO_KEEPALIVE

Когда параметр `SO_KEEPALIVE` установлен для сокета TCP и в течение двух часов не происходит обмена данными по сокету в любом направлении, TCP автоматически посыпает собеседнику проверочное сообщение (keepalive probe). Это сообщение — сегмент TCP, на который собеседник должен ответить. Далее события могут развиваться по одному из трех сценариев.

1. Собеседник отвечает, присыпая ожидаемый сегмент ACK. Приложение не получает уведомления (поскольку все в порядке). TCP снова отправит одно проверочное сообщение еще через два часа отсутствия активности в этом соединении.

2. Собеседник отвечает, присыпая сегмент RST, который сообщает локальному TCP, что узел собеседника вышел из строя и перезагрузился. Ошибка сокета, требующая обработки, устанавливается

равной ECONNRESET и сокет закрывается.

3. На проверочное сообщение не приходит ответ от собеседника. Код TCP, происходящий от Беркли, отправляет восемь дополнительных проверочных сообщений с интервалом в 75 с, пытаясь выявить ошибку. TCP прекратит попытки, если ответа не последует в течение 11 мин и 15 с после отправки первого сообщения.

ПРИМЕЧАНИЕ

HP-UX обрабатывает проверочные сообщения так же, как и обычные данные, то есть второе сообщение отсылается по истечении периода повторной передачи, после чего для каждого последующего пакета интервал ожидания удваивается, пока не будет достигнут максимальный интервал (по умолчанию — 10 мин).

Если на все проверочные сообщения TCP не приходит ответа, то ошибка сокета, требующая обработки, устанавливается в ETIMEDOUT и сокет закрывается. Но если сокет получает ошибку ICMP (Internet Control Message Protocol — протокол управляющих сообщений Интернета) в ответ на одно из проверочных сообщений, то возвращается одна из соответствующих ошибок (см. табл. А.5 и А.6), но сокет также закрывается. Типичная ошибка ICMP в этом сценарии — Host unreachable (Узел недоступен) — указывает на то, что узел собеседника не вышел из строя, а только является недоступным. При этом ошибка, ожидающая обработки, устанавливается в EHOSTUNREACH. Это может произойти из-за отказа сети или при выходе удаленного узла из строя и обнаружении этого последним маршрутизатором.

В главе 23 [111] и на с. 828–831 [128] содержатся дополнительные подробности об этом параметре.

Без сомнения, наиболее типичный вопрос, касающийся этого параметра, состоит в том, могут ли изменяться временные параметры (обычно нас интересует возможность сокращения двухчасовой задержки). В разделе 7.9 мы описываем новый параметр TCP_KEEPALIVE, но он не реализован достаточно широко. В приложении Е [111] обсуждается изменение временных параметров для различных ядер. Необходимо учитывать, что большинство ядер обрабатывают эти параметры глобально, и поэтому сокращение времени ожидания, например с 2 час до 15 мин, повлияет на все сокеты узла, для которых включен параметр SO_KEEPALIVE.

Назначение этого параметра — обнаружение сбоя на узле собеседника. Если процесс собеседника выходит из строя, его TCP отправит через соединение сегмент FIN, который мы сможем легко обнаружить с помощью функции select (поэтому мы использовали функцию select в разделе 6.4). Также нужно понимать, что если на проверочное сообщение не приходит ответа (сценарий 3), то это не обязательно означает, что на узле сервера произошел сбой и существует вероятность, что TCP закроет действующее соединение. Если, например, промежуточный маршрутизатор вышел из строя на 15 мин, то эти 15 мин полностью перекрывают период отправки проверочных сообщений от нашего узла, равный 11 мин и 15 с. Поэтому правильнее было бы назвать эту функцию не проверкой жизнеспособности (keep-alive), а контрольным выстрелом (make-dead), поскольку она может завершать еще открытые соединения.

Этот параметр обычно используется серверами, хотя его могут использовать и клиенты. Серверы используют его, поскольку большую часть своего времени они проводят в блокированном состоянии, ожидая ввода по соединению TCP, то есть в ожидании запроса клиента. Но если узел клиента выходит из строя, процесс сервера никогда не узнает об этом и сервер будет продолжать ждать ввода данных, которые никогда не придут. Это называется *наполовину открытым соединением* (*half-open connection*). Данный параметр позволяет обнаружить наполовину открытые соединения и завершить их.

Некоторые серверы, особенно серверы FTP, предоставляют приложению тайм-аут, часто до нескольких минут. Это выполняется самим приложением, обычно при вызове функции read, когда считывается следующая команда клиента. Этот тайм-аут не связан с данным параметром сокета.

ПРИМЕЧАНИЕ

В SCTP имеется механизм проверки пульса (heartbeat), аналогичный механизму проверочных сообщений (keep-alive) TCP. Этот механизм настраивается при помощи элементов параметра сокета SCTP_SET_PEER_ADDR_PARAMS, который будет описан далее, а не при

помощи параметра SO_KEEPALIVE. Последний полностью игнорируется сокетом SCTP и не мешает работе механизма проверки пульса.

В табл. 7.3 суммируются различные методы, применяемые для обнаружения того, что происходит на другом конце соединения TCP. Когда мы говорим «использование функции select для проверки готовности к чтению», мы имеем в виду вызов функции select для проверки, готов ли сокет для чтения.

Таблица 7.3. Методы определения различных условий TCP

Сценарий	Процесс собеседника выходит из строя	Узел собеседника выходит из строя	Узел собеседника недоступен
Наш TCP активно посыпает данные	TCP собеседника посыпает сегмент FIN, что мы можем сразу же обнаружить, используя функцию select для проверки готовности к чтению. Если TCP посыпает второй сегмент, TCP собеседника посыпает в ответ сегмент RST. Если TCP посыпает еще один сегмент, наш TCP посыпает сигнал SIGPIPE	По истечении времени ожидания TCP возвращается ошибка ETIMEDOUT	По истечении времени ожидания TCP возвращается ошибка ETIMEDOUT
Наш TCP активно принимает данные	TCP собеседника посыпает сегмент FIN, который мы прочитаем как признак конца файла (возможно, преждевременный)	Мы больше не получаем никаких данных	Мы больше не получаем никаких данных
Соединение неактивно, посыпается пробный пакет	TCP собеседника посыпает сегмент FIN, который мы можем сразу же обнаружить, используя функцию select для проверки готовности к чтению	По истечении двух часов отсутствия активности отсылается 9 сообщений для проверки наличия связи с собеседником, а затем возвращается ошибка ETIMEDOUT	По истечении двух часов отсутствия активности отсылается 9 сообщений для проверки наличия связи с собеседником, а затем возвращается ошибка ETIMEDOUT
Соединение неактивно, не посыпается	TCP собеседника посыпает сегмент FIN, который мы можем сразу же обнаружить, используя функцию select для проверки проверочное готовности к чтению сообщение	Ничего не происходит	Ничего не происходит

Параметр сокета SO_LINGER

Этот параметр определяет, как работает функция close для протоколов, ориентированных на установление соединения (например, TCP и SCTP, но не UDP). По умолчанию функция close возвращает управление немедленно, но если в отправляющем буфере сокета остаются какие-либо данные, система попытается доставить данные собеседнику.

Параметр сокета SO_LINGER позволяет нам изменять поведение по умолчанию. Для этого необходимо, чтобы между пользовательским процессом и ядром была передана следующая структура, определяемая в заголовочном файле <sys/socket.h>:

```
struct linger {
    int l_onoff; /* 0=off, ненулевое значение=on */ int l_linger;
                /* время ожидания, в POSIX измеряется в секундах */
};
```

Вызов функции setsockopt приводит к одному из трех следующих сценариев в зависимости от значений двух элементов структуры linger.

1. Если l_onoff имеет нулевое значение, параметр выключается. Значение l_linger игнорируется и применяется ранее рассмотренный заданный по умолчанию сценарий TCP: функция close завершается немедленно.

2. Если значение `l_onoff` ненулевое, а `l_linger` равно нулю, TCP сбрасывает соединение, когда оно закрывается [128, с. 1019–1020], то есть TCP игнорирует все данные, остающиеся в буфере отправки сокета, и отправляет собеседнику сегмент RST, а не обычную последовательность завершения соединения, состоящую из четырех пакетов (см. раздел 2.5). Пример мы покажем в листинге 16.14. Тогда не наступает состояние TCP `TIME_WAIT`, но из-за этого возникает возможность создания другого воплощения (*incarnation*) этого соединения в течение 2MSL секунд (удвоенное максимальное время жизни сегмента). Оставшиеся старые дублированные сегменты из только что завершенного соединения могут быть доставлены новому воплощению, что приведет к ошибкам (см. раздел 2.6).

При указанных выше значениях `l_onoff` и `l_linger` SCTP также выполняет аварийное закрытие сокета, отправляя собеседнику пакет ABORT (см. раздел 9.2 [117]).

ПРИМЕЧАНИЕ

Отдельные выступления в Usenet звучат в защиту использования этой возможности, поскольку она позволяет избежать состояния `TIME_WAIT` и снова запустить прослушивающий сервер, даже если соединения все еще используются с известным портом сервера. Так не нужно делать, поскольку это может привести к искажению данных, как показано в RFC 1337 [11]. Вместо этого перед вызовом функции `bind` на стороне сервера всегда нужно использовать параметр сокета `SO_REUSEADDR`, как показано далее. Состояние `TIME_WAIT` — наш друг, так как оно предназначено для того, чтобы помочь нам дождаться, когда истечет время жизни в сети старых дублированных сегментов. Вместо того, чтобы пытаться избежать этого состояния, следует понять его назначение (см. раздел 2.6).

Тем не менее в некоторых обстоятельствах использование аварийного закрытия может быть оправдано. Одним из примеров является сервер терминалов RS-232, который может навечно зависнуть в состоянии `CLOSE_WAIT`, пытаясь доставить данные на забытый порт. Если же он получит сегмент RST, он сможет сбросить накопившиеся данные и заново инициализировать порт.

3. Если оба значения — `l_onoff` и `l_linger` — ненулевые, то при закрытии сокета ядро будет ждать (*linger*) [128, с. 472]. То есть если в буфере отправки сокета еще имеются какие-либо данные, процесс входит в состояние ожидания до тех пор, пока либо все данные не будут отправлены и подтверждены другим концом TCP, либо не истечет время ожидания. Если сокет был установлен как неблокируемый (см. главу 16), он не будет ждать завершения выполнения функции `close`, даже если время задержки ненулевое. При использовании этого свойства параметра `SO_LINGER` приложению важно проверить значение, возвращаемое функцией `close`. Если время ожидания истечет до того, как оставшиеся данные будут отправлены и подтверждены, функция `close` вернет ошибку `EWOULDBLOCK` и все данные, оставшиеся в буфере отправки сокета, будут сброшены.

Теперь нам нужно точно определить, когда завершается функция `close` на сокете в различных сценариях, которые мы рассмотрели. Предполагается, что клиент записывает данные в сокет и вызывает функцию `close`. На рис. 7.1 показана ситуация по умолчанию.

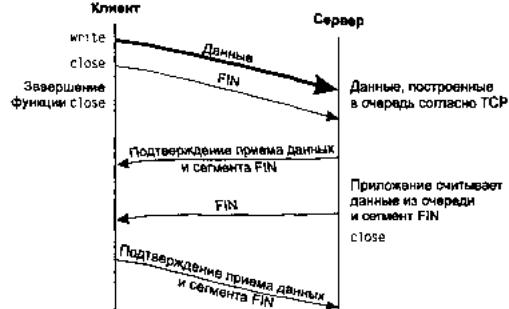


Рис. 7.1. Действие функции `close`, заданное по умолчанию: немедленное завершение

Мы предполагаем, что когда приходят данные клиента, сервер временно занят. Поэтому данные добавляются в приемный буфер сокета его протоколом TCP. Аналогично, следующий сегмент (сегмент FIN клиента) также добавляется к приемному буферу сокета (каким бы образом реализация ни сохраняла сегмент FIN). Но по умолчанию клиентская функция `close` сразу же завершается. Как мы показываем в

в этом сценарии, клиентская функция `close` может завершиться перед тем, как сервер прочитает оставшиеся данные в приемном буфере его сокета. Если узел сервера выйдет из строя перед тем, как приложение-сервер считает оставшиеся данные, клиентское приложение никогда об этом не узнает.

Клиент может установить параметр сокета `SO_LINGER`, задав некоторое положительное время задержки. Когда это происходит, клиентская функция `close` не завершается до тех пор, пока все данные клиента и его сегмент FIN не будут подтверждены протоколом TCP сервера. Мы показываем это на рис. 7.2.

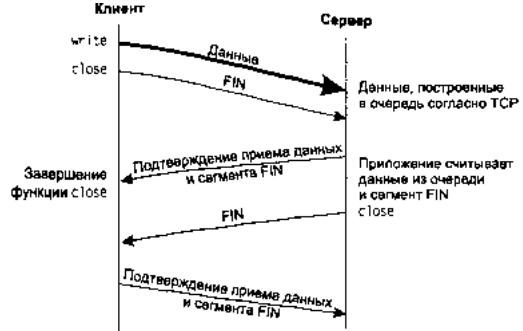


Рис. 7.2. Закрытие сокета с параметром `SO_LINGER` и положительным `l_linger`

Но у нас остается та же проблема, что и на рис. 7.1: если на узле сервера происходит сбой до того, как приложение-сервер считает оставшиеся данные, клиентское приложение никогда не узнает об этом. Еще худший вариант развития событий показан на рис. 7.3, где значение `SO_LINGER` было установлено слишком маленьким.

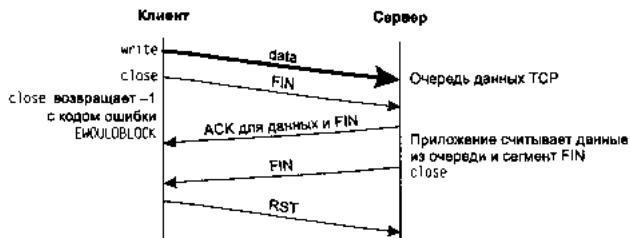


Рис. 7.3. Закрытие сокета с параметром `SO_LINGER` при малом положительном `l_linger`

Основным принципом взаимодействия является то, что успешное завершение функции `close` с установленным параметром сокета `SO_LINGER` говорит нам лишь о том, что данные, которые мы отправили (и наш сегмент FIN) подтверждены протоколом TCP собеседника. Но это не говорит нам, прочитало ли данные приложение собеседника. Если мы не установим параметр сокета `SO_LINGER`, мы не будем знать, подтвердил ли другой конец TCP отправленные ему данные.

Чтобы узнать, что сервер прочитал данные клиента, клиент может вызывать функцию `shutdown` (со вторым аргументом `SHUT_WR`) вместо функции `close` и ждать, когда собеседник закроет с помощью функции `close` свой конец соединения. Этот сценарий показан на рис. 7.4.

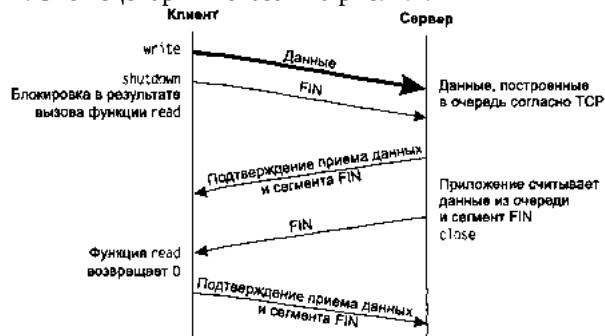


Рис. 7.4. Использование функции `shutdown` для проверки того, что собеседник получил наши данные

Сравнивая этот рисунок с рис. 7.1 и 7.2, мы видим, что когда мы закрываем наш конец соединения, то в зависимости от вызванной функции (`close` или `shutdown`) и от того, установлен или нет параметр сокета `SO_LINGER`, завершение может произойти в один из трех различных моментов времени:

1. Функция `close` завершается немедленно, без всякого ожидания (сценарий, заданный по умолчанию, см. рис. 7.1).

2. Функция `close` задерживается до тех пор, пока не будет получен сегмент ACK, подтверждающий получение сервером сегмента FIN от клиента (см. рис. 7.2).

3. Функция `shutdown`, за которой следует функция `read`, ждет, когда мы получим сегмент FIN собеседника (в данном случае сервера) (см. рис. 7.2).

Другой способ узнать, что приложение-собеседник прочитало наши данные, — использовать *подтверждение на уровне приложения*, или *ACK приложения*. Например, клиент отправляет данные серверу и затем вызывает функцию `read` для одного байта данных:

```
char ack;
```

```
Write(sockfd, data, nbytes); /* данные от клиента к серверу */
n = Read(sockfd, &ack, 1); /* ожидание подтверждения на уровне приложения */
```

Сервер читает данные от клиента и затем отправляет ему 1-байтовый сегмент — подтверждение на уровне приложения:

```
nbytes = Read(sockfd, buff, sizeof(buff)); /* данные от клиента */
/* сервер проверяет, верное ли количество данных он получил от клиента */
Write(sockfd, 1); /* сегмент ACK сервера возвращается клиенту */
```

Таким образом, мы получаем гарантию, что на момент завершения функции `read` на стороне клиента процесс сервера прочитал данные, которые мы отправили. (При этом предполагается, что либо сервер знает, сколько данных отправляет клиент, либо существует некоторый заданный приложением маркер конца записи, который мы здесь не показываем.) В данном случае сегмент ACK на уровне приложения представляет собой нулевой байт, но вообще содержимое этого сегмента можно использовать для передачи от сервера к клиенту сообщений о других условиях. На рис. 7.5 показан возможный обмен пакетами.

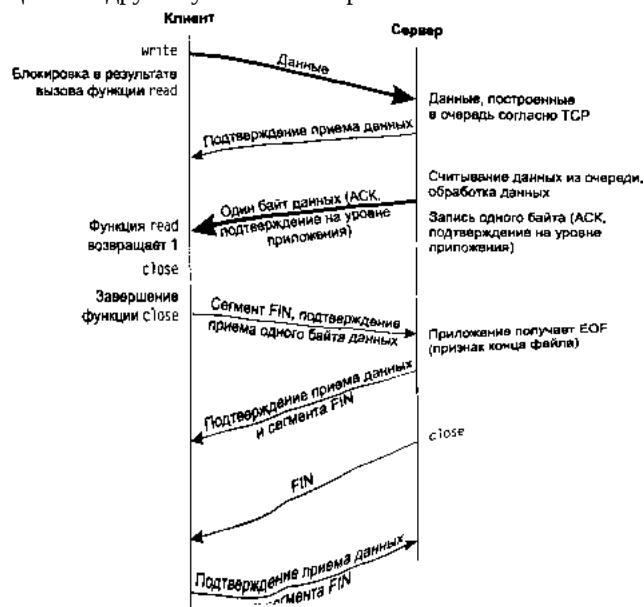


Рис. 7.5. ACK приложения

В табл. 7.4 описаны два возможных вызова функции `shutdown` и три возможных вызова функции `close`, а также их влияние на сокет TCP.

Таблица 7.4. Итоговая таблица сценариев функции `shutdown` и параметров сокета `SO_LINGER`

Функция	Описание
<code>shutdown</code> , SHUT_RD	Через сокет больше нельзя принимать данные; процесс может по-прежнему отправлять данные через этот сокет; приемный буфер сокета сбрасывается; все данные, получаемые в дальнейшем, игнорируются протоколом TCP (см. упражнение 6.5); не влияет на буфер отправки сокета
<code>shutdown</code> ,	Через сокет больше нельзя отправлять данные; процесс может по-прежнему получать данные

SHUT_WR	через этот сокет; содержимое буфера отправки сокета отсылается на другой конец соединения, затем выполняется обычная последовательность действий по завершению соединения TCP (FIN); не влияет на приемный буфер сокета
close, l_onoff = 0 (по умолчанию)	Через сокет больше нельзя отправлять и получать данные; содержимое буфера отправки сокета отсылается на другой конец соединения. Если счетчик ссылок дескриптора становится нулевым, то следом за отправкой данных из буфера отправки сокета выполняется нормальная последовательность завершения соединения TCP (FIN), данные из приемного буфера сокета сбрасываются
close, l_onoff = 1 l_linger = 0	Через сокет больше нельзя отправлять и получать данные. Если счетчик ссылок дескриптора становится нулевым, то на другой конец соединения посылается сегмент RST, соединение переходит в состояние в CLOSED (минута состояния TIME_WAIT), данные из буфера отправки и приемного буфера сокета сбрасываются
close, l_onoff = 1 l_linger = 0	Через сокет больше нельзя отправлять и получать данные; содержимое буфера отправки сокета отсылается на другой конец соединения. Если счетчик ссылок дескриптора становится нулевым, то следом за отправкой данных из буфера отправки сокета выполняется нормальная последовательность завершения соединения TCP (FIN), данные из приемного буфера сокета сбрасываются, и если время задержки истекает, прежде чем оставшиеся в буфере данные будут посланы и будет подтвержден их прием, функция close вернет ошибку EWOULDBLOCK

Параметр сокета SO_OOBINLINE

Когда установлен этот параметр, внеполосные данные помещаются в очередь нормального ввода (то есть вместе с обычными данными (inline)). Когда это происходит, флаг MSG_OOB не может быть использован для чтения полученных внеполосных данных. Более подробно внеполосные данные мы рассмотрим в главе 24.

Параметры сокета SO_RCVBUF и SO_SNDBUF

У каждого сокета имеется буфер отправки и приемный буфер (буфер приема). Мы изобразили действие буферов отправки TCP, UDP и SCTP на рис. 2.15, 2.16 и 2.17.

Приемные буферы используются в TCP, UDP и SCTP для хранения полученных данных, пока они не будут считаны приложением. В случае TCP доступное пространство в приемном буфере сокета — это окно, размер которого TCP сообщает другому концу соединения. Приемный буфер сокета TCP не может переполниться, поскольку собеседнику не разрешается отправлять данные, размер которых превышает размер окна. Так действует управление передачей TCP, и если собеседник игнорирует объявленное окно и отправляет данные, превышающие его размер, принимающий TCP игнорирует эти данные. Однако в случае UDP дейтаграмма, не подходящая для приемного буфера сокета, игнорируется. Вспомните, что в UDP отсутствует управление потоком: более быстрый отправитель легко переполнит буфер медленного получателя, заставляя UDP получателя игнорировать дейтаграммы, как мы покажем в разделе 8.13. Более того, быстрый отправитель может переполнить даже собственный сетевой интерфейс, так что дейтаграммы будут сбрасываться еще до отправки их с исходного узла.

Указанные в заголовке раздела параметры позволяют нам изменять размеры буферов, заданные по умолчанию. Значения по умолчанию сильно отличаются в зависимости от реализации. Более ранние реализации, происходящие от Беркли, по умолчанию имели размеры буферов отправки и приема 4096 байт, а более новые системы используют буферы больших размеров, от 8192 до 61 440 байт. Размер буфера отправки UDP по умолчанию часто составляет около 9000 байт, а если узел поддерживает NFS, то размер приемного буфера UDP увеличивается до 40 000 байт.

При установке размера приемного буфера сокета TCP важен порядок вызова функций, поскольку в данном случае учитывается параметр масштабирования окна TCP (см. раздел 2.5). При установлении соединения обе стороны обмениваются сегментами SYN, в которых может содержаться этот параметр. Для клиента это означает, что параметр сокета SO_RCVBUF должен быть установлен перед вызовом функции connect. Для сервера это означает, что данный параметр должен быть установлен для прослушиваемого сокета перед вызовом функции listen. Установка этого параметра для присоединенного сокета никак не

повлияет на параметр масштабирования окна, поскольку функция `accept` не возвращает управление процессу, пока не завершится трехэтапное рукопожатие TCP. Поэтому данный параметр должен быть установлен для прослушиваемого сокета. (Размеры буферов сокета всегда наследуются от прослушиваемого сокета создаваемым присоединенным сокетом [128, с. 462-463]).

Размеры буферов сокета TCP должны быть как минимум вчетверо больше MSS (максимальный размер сегмента) для соединения. Если мы имеем дело с направленной передачей данных, такой как передача файла в одном направлении, то говоря «размеры буферов сокета», мы подразумеваем буфер отправки сокета на отправляющем узле или приемный буфер сокета на принимающем узле. В случае двусторонней передачи данных мы имеем в виду оба размера буферов на обоих узлах. С типичным размером буфера 8192 байт или больше и типичным MSS, равным 512 или 1460 байт, это требование обычно выполняется. Проблемы были замечены в сетях с большими MTU (максимальная единица передачи), которые предоставляют MSS больше обычного (например, в сетях ATM с MTU, равной 9188).

ПРИМЕЧАНИЕ

Значение минимального множителя (4) обусловлено принципом работы алгоритма быстрого восстановления TCP. Отправитель использует три двойных подтверждения, чтобы обнаружить утерянный пакет (RFC 2581 [4]). Получатель отправляет двойное подтверждение для каждого сегмента, принятого после того, который был пропущен. Если размер окна меньше четырех сегментов, трех двойных подтверждений не будет и алгоритм быстрого восстановления не сработает.

Размеры буфера сокета TCP должны быть также четное число раз кратны размеру MSS для соединения. Некоторые реализации выполняют это требование для приложения, округляя размеры в сторону большего размера буфера сокета после установления соединения [128, с. 902]. Это другая причина, по которой следует задавать эти два параметра сокета перед установлением соединения. Например, если использовать размеры, заданные по умолчанию в 4.4BSD (8192 байт), и считать, что используется Ethernet с размером MSS, равным 1460 байт, то при установлении соединения размеры обоих буферов сокета будут округляться до 8760 байт (6×1460). Это требование не жесткое, лишнее место в буфере просто не будет использоваться.

Другое соображение относительно установки размеров буфера сокета связано с производительностью. На рис. 7.6 показано соединение TCP между двумя конечными точками (которое мы называем *каналом*) с вместимостью, допускающей передачу восьми сегментов.



Рис. 7.6. Соединение TCP (канал), вмещающее восемь сегментов

Мы показываем четыре сегмента данных вверху и четыре сегмента ACK внизу. Даже если в канале только четыре сегмента данных, у клиента должен быть буфер отправки, вмещающий минимум восемь сегментов, потому что TCP клиента должен хранить копию каждого сегмента, пока не получен сегмент ACK от сервера.

ПРИМЕЧАНИЕ

Здесь мы игнорируем некоторые подробности. Прежде всего, алгоритм медленного запуска TCP ограничивает скорость, с которой сегменты начинают отправляться по соединению, которое до этого было неактивным. Далее, TCP часто подтверждает каждый второй сегмент, а не каждый сегмент, как мы это показываем. Все эти подробности описаны в главах 20 и 24 [111].

Нам необходимо понять принцип функционирования двустороннего канала и узнать, что такое его вместимость и как она влияет на размеры буферов сокетов на обоих концах соединения. Вместимость канала характеризуется произведением пропускной способности на задержку (bandwidth-delay product). Мы будем вычислять ее, умножая пропускную способность канала (в битах в секунду) на период обращения (RTT, round-trip time) (в секундах) и преобразуя результат из битов в байты. RTT легко измеряется с помощью утилиты `ping`. Пропускная способность — это значение, соответствующее наиболее медленной

связи между двумя конечными точками; предполагается, что это значение каким-то образом определено. Например, линия T1 (1 536 000 бит/с) с RTT 60 мс дает произведение пропускной способности на задержку, равное 11 520 байт. Если размеры буфера сокета меньше указанного, канал не будет заполнен и производительность окажется ниже предполагаемой. Большие буферы сокетов требуются, когда повышается пропускная способность (например, для линии Т3, где она равна 45 Мбит/с) или когда увеличивается RTT (например, спутниковые каналы связи с RTT около 500 мс). Когда произведение пропускной способности на задержку превосходит максимальный нормальный размер окна TCP (65 535 байт), обоим концам соединения требуются также параметры TCP для канала с повышенной пропускной способностью (*long fat pipe*), о которых мы упоминали в разделе 2.6.

ПРИМЕЧАНИЕ

В большинстве реализаций размеры буферов отправки и приема ограничиваются некоторым предельным значением. В более ранних реализациях, происходящих от Беркли, верхний предел был около 52 000 байт, но в новых реализациях предел по умолчанию равен 256 000 байт или больше, и обычно администратор имеет возможность увеличивать его. К сожалению, не существует простого способа, с помощью которого приложение могло бы узнать этот предел. POSIX определяет функцию `fpathconf`, поддерживаемую большинством реализаций, а в качестве второго аргумента этой функции должна использоваться константа `_PC_SOCK_MAXBUF`. Приложение может также попытаться установить желаемый размер буфера сокета, а если попытка окажется неудачной, сократить размер вдвое и вызвать функцию снова. Наконец, приложение должно убедиться, что оно не уменьшает размер буфера по умолчанию, задавая свое собственное значение. В первую очередь следует вызвать `getsockopt` для определения значения, установленного по умолчанию, которое вполне может оказаться достаточным.

Параметры сокета SO_RCVLOWAT и SO SNDLOWAT

Каждый сокет характеризуется также минимальным количеством данных (*low-water mark*) для буферов приема и отправки. Эти значения используются функцией `select`, как мы показали в разделе 6.3. Указанные параметры сокета позволяют нам изменять эти два значения.

Минимальное количество данных — это количество данных, которые должны находиться в приемном буфере сокета, чтобы функция `select` возвратила ответ «Сокет готов для чтения». По умолчанию это значение равно 1 для сокетов TCP и UDP. Минимальный объем для буфера отправки — это количество свободного пространства, которое должно быть в буфере отправки сокета, чтобы функция `select` возвратила «Сокет готов для записи». Для сокетов TCP по умолчанию оно обычно равно 2048. С UDP это значение используется так, как мы показали в разделе 6.3, но поскольку число байтов доступного пространства в буфере отправки для сокета UDP никогда не изменяется (поскольку UDP не хранит копиидейтограмм, отправленных приложением), сокет UDP всегда готов для записи, пока размер буфера отправки сокета UDP больше минимального объема. Вспомните рис. 2.16: UDP не имеет настоящего буфера отправки, у него есть только параметр размера буфера отправки.

Параметры сокета SO_RCVTIMEO и SO SNDTIMEO

Эти два параметра сокета позволяют нам устанавливать тайм-аут при получении и отправке через сокет. Обратите внимание, что аргумент двух функций `sockopt` — это указатель на структуру `timeval`, ту же, которую использует функция `select` (раздел 6.3). Это позволяет использовать для задания тайм-аута секунды и миллисекунды. Отключение тайм-аута осуществляется установкой его значения в 0 секунд и 0 миллисекунд. Оба тайм-аута по умолчанию отключены.

Тайм-аут приема влияет на пять функций ввода: `read`, `readv`, `recv`, `recvfrom` и `recvmsg`. Тайм-аут отправки влияет на пять функций вывода: `write`, `writev`, `send`, `sendto` и `sendmsg`. Более подробно о тайм-аутах сокета мы поговорим в разделе 14.2.

ПРИМЕЧАНИЕ

Эти два параметра сокета и концепция тайм-аута сокетов вообще были добавлены в реализации 4.3BSD Reno.

В реализациях, происходящих от Беркли, указанные параметры инициализируют таймер отсутствия активности, а не абсолютный таймер системного вызова чтения или записи. На с. 496 и 516 [128] об этом рассказывается более подробно.

Параметры сокета SO_REUSEADDR и SO_REUSEPORT

Параметр сокета SO_REUSEADDR служит для четырех целей.

1. Параметр SO_REUSEADDR позволяет прослушивающему серверу запуститься и с помощью функции bind связаться со своим заранее известным портом, даже если существуют ранее установленные соединения, использующие этот порт в качестве своего локального порта. Эта ситуация обычно возникает следующим образом:

- 1) запускается прослушивающий сервер;
- 2) от клиента приходит запрос на соединение, и для обработки этого клиента генерируется дочерний процесс;
- 3) прослушивающий сервер завершает работу, но дочерний процесс продолжает обслуживание клиента на существующем соединении;
- 4) прослушивающий сервер перезапускается.

По умолчанию, когда прослушивающий сервер перезапускается при помощи вызова функций socket, bind и listen, вызов функции bind оказывается неудачным, потому что прослушивающий сервер пытается связаться с портом, который является частью существующего соединения (обрабатываемого ранее созданным дочерним процессом). Но если сервер устанавливает параметр сокета SO_REUSEADDR между вызовами функций socket и bind, последняя выполнится успешно. Все серверы TCP должны задавать этот параметр сокета, чтобы позволить перезапускать сервер в подобной ситуации.

ПРИМЕЧАНИЕ

Этот сценарий вызывает больше всего вопросов в Usenet.

2. Параметр SO_REUSEADDR позволяет множеству экземпляров одного и того же сервера запускаться на одном и том же порте, если все экземпляры связываются с различными локальными IP-адресами. Это типичная ситуация для узла, на котором размещаются несколько серверов HTTP, использующих технологию альтернативных IP-адресов, или псевдонимов (IP alias technique) (см. раздел А.4). Допустим, первичный IP-адрес локального узла — 198.69.10.2, но он имеет два альтернативных адреса — 198.69.10.128 и 198.69.10.129. Запускаются три сервера HTTP. Первый сервер с помощью функции bind связывается с локальным IP-адресом 198.69.10.128 и локальным портом 80 (заранее известный порт HTTP). Второй сервер с помощью функции bind связывается с локальным IP-адресом 198.69.10.129 и локальным портом 80. Но второй вызов функции bind не будет успешным, пока не будет установлен параметр SO_REUSEADDR перед обращением к ней. Третий сервер вызовет функцию bind с универсальным адресом в качестве локального IP-адреса и локальным портом 80. И снова требуется параметр SO_REUSEADDR, для того чтобы последний вызов оказался успешным. Если считать, что установлен параметр SO_REUSEADDR и запущены три сервера, то входящие запросы TCP на соединение с IP-адресом получателя 198.69.10.128 и портом получателя 80 доставляются на второй сервер, входящие запросы на соединение с IP-адресом получателя 198.69.10.129 и портом получателя 80 — на третий сервер, а все остальные входящие запросы TCP на соединение с портом получателя 80 доставляются на первый сервер. Этот сервер обрабатывает запросы, адресованные на 198.69.10.2, в дополнение к другим альтернативным IP-адресам, для которых этот узел может быть сконфигурирован. Символ подстановки означает в данном случае «все, для чего не нашлось более точного совпадения». Заметим, что этот сценарий, допускающий множество серверов для данной службы, обрабатывается автоматически, если сервер всегда устанавливает параметр сокета SO_REUSEADDR (как мы рекомендуем).

TCP не дает нам возможности запустить множество серверов, которые с помощью функции bind связываются с одним и тем же IP-адресом и одним и тем же портом: это случай *полностью дублированного связывания* (*completely duplicate binding*). То есть мы не можем запустить один сервер, связывающийся с

адресом 198.69.10.2 и портом 80, и другой сервер, также связывающийся с адресом 198.69.10.2 и портом 80, даже если для второго сервера мы установим параметр `SO_REUSEADDR`.

По соображениям безопасности некоторые операционные системы запрещают связывать несколько серверов с адресом подстановки, то есть описанный выше сценарий не работает даже с использованием параметра `SO_REUSEADDR`. В такой системе сервер, связываемый с адресом подстановки, должен запускаться последним. Таким образом предотвращается привязка сервера злоумышленника к IP-адресу и порту, которые уже обрабатываются системной службой. Особенно это важно для службы NFS, которая обычно не использует выделенный порт.

3. Параметр `SO_REUSEADDR` позволяет одиночному процессу связывать один и тот же порт с множеством сокетов, так как при каждом связывании задается уникальный IP-адрес. Это обычное явление для серверов UDP, так как им необходимо знать IP-адрес получателя запросов клиента в системах, не поддерживающих параметр сокета `IP_RECVSTADDR`. Эта технология обычно не применяется с серверами TCP, поскольку сервер TCP всегда может определить IP-адрес получателя при помощи вызова функции `getsockname`, после того как соединение установлено. Однако на многоинтерфейсном узле сервер TCP, работающий с частью адресов локального узла, мог бы воспользоваться этой функцией.

4. Параметр `SO_REUSEADDR` допускает *полностью дублированное связывание*: связывание с помощью функции `bind` с IP-адресом и портом, когда тот же IP-адрес и тот же порт уже связаны с другим сокетом. Обычно это свойство доступно только в системах с поддержкой многоадресной передачи без поддержки параметра сокета `SO_REUSEPORT` (который мы опишем чуть ниже), и только для сокетов UDP (многоадресная передача не работает с TCP).

Это свойство применяется при многоадресной передаче для многократного выполнения одного и того же приложения на одном и том же узле. Когда приходит дейтаграмма UDP для одного из многократно связанных сокетов, действует следующее правило: если дейтаграмма предназначена либо для широковещательного адреса, либо для адреса многоадресной передачи, то одна копия дейтаграммы доставляется каждому сокету. Но если дейтаграмма предназначена для адреса направленной передачи, то дейтаграмма доставляется только на один сокет. Какой сокет получит дейтаграмму, если в случае направленной передачи существует множество сокетов, соответствующих дейтаграмме, — зависит от реализации. На с. 777-779 [128] об этом свойстве рассказывается более подробно. О широковещательной и многоадресной передаче мы поговорим соответственно в главах 20 и 21.

В упражнениях 7.5 и 7.6 показаны примеры использования этого параметра сокета.

Вместо того чтобы перегружать параметр `SO_REUSEADDR` семантикой многоадресной передачи, допускающей полностью дублированное связывание, в 4.4BSD был введен новый параметр сокета `SO_REUSEPORT`, обладающий следующей семантикой:

1. Этот параметр допускает полностью дублированное связывание, но только если каждый сокет, который хочет связаться с тем же IP-адресом и портом, задает этот параметр сокета.

2. Параметр `SO_REUSEADDR` считается эквивалентным параметру `SO_REUSEPORT`, если связываемый IP-адрес является адресом многоадресной передачи [128, с. 731].

Проблема с этим параметром сокета заключается в том, что не все системы его поддерживают. В системах без поддержки этого параметра, но с поддержкой многоадресной передачи его функции выполняет параметр `SO_REUSEADDR`, допускающий полностью дублированное связывание, когда оно имеет смысл (то есть когда имеется сервер UDP, который может быть запущен много раз на одном и том же узле в одно и то же время, предполагающий получать либо широковещательные дейтаграммы, либо дейтаграммы многоадресной передачи).

Обобщить обсуждение этих параметров сокета можно с помощью следующих рекомендаций:

1. Устанавливайте параметр `SO_REUSEADDR` перед вызовом функции `bind` на всех серверах TCP.

2. При создании приложения многоадресной передачи, которое может быть запущено несколько раз на одном и том же узле в одно и то же время, устанавливайте параметр `SO_REUSEADDR` и связывайтесь с адресом многоадресной передачи, используемым в качестве локального IP-адреса.

Более подробно об этих параметрах сокета рассказывается в главе 22 [128].

Существует потенциальная проблема безопасности, связанная с использованием параметра `SO_REUSEADDR`. Если существует сокет, связанный, скажем, с универсальным адресом и портом 5555, то, задав параметр `SO_REUSEADDR`, мы можем связать этот порт с другим IP-адресом, например с основным (primary) IP-адресом узла. Любые приходящие дейтаграммы, предназначенные для порта 5555 и IP-адреса, который мы связали с нашим сокетом, доставляются на наш сокет, а не на другой сокет, связанный с универсальным адресом. Это могут быть сегменты SYN TCP или дейтаграммы UDP. (В упражнении 11.9 показано это свойство для UDP.) Для большинства известных служб, таких как HTTP, FTP и Telnet, это не

составляет проблемы, поскольку все эти серверы связываются с зарезервированным портом. Следовательно, любой процесс, запущенный позже и пытающийся связаться с конкретным экземпляром этого порта (то есть пытающийся завладеть портом), требует прав привилегированного пользователя. Однако NFS (Network File System — сетевая файловая система) может вызвать проблемы, поскольку ее стандартный порт (2049) не зарезервирован.

ПРИМЕЧАНИЕ

Одна из сопутствующих проблем API сокетов в том, что установка пары сокетов выполняется с помощью двух вызовов функций (`bind` и `connect`) вместо одного. В [122] предлагается одиночная функция, разрешающая эту проблему:

```
int bind_connect_listen(int sockfd,
    const struct sockaddr *laddr, int laddrlen,
    const struct sockaddr *faddr, int faddrlen,
    int listen);
```

Аргумент `laddr` задает локальный IP-адрес и локальный порт, аргумент `faddr` — удаленный IP-адрес и удаленный порт, аргумент `listen` задает клиент (0) или сервер (значение ненулевое; то же, что и аргумент `backlog` функции `listen`). В таком случае функция `bind` могла бы быть библиотечной функцией, вызывающей эту функцию с пустым указателем `faddr` и нулевым `faddrlen`, а функция `connect` — библиотечной функцией, вызывающей эту функцию с пустым указателем `laddr` и нулевым `laddrlen`. Существует несколько приложений, особенно FTP, которым необходимо задавать и локальную пару, и удаленную пару, которые могут вызывать `bind_connect_listen` непосредственно. При наличии подобной функции отпадает необходимость в параметре `SO_REUSEADDR`, в отличие от серверов UDP, которым явно необходимо допускать полностью дублированное связывание с одним и тем же IP-адресом и портом. Другое преимущество этой новой функции в том, что сервер TCP может ограничить себя обслуживанием запросов на соединения, приходящих от одного определенного IP-адреса и порта. Это определяется в RFC 793 [96], но невозможно с существующими API сокетов.

Параметр сокета SO_TYPE

Этот параметр возвращает тип сокета. Возвращаемое целое число — константа `SOCK_STREAM` или `SOCK_DGRAM`. Этот параметр обычно используется процессом, наследующим сокет при запуске.

Параметр сокета SO_USELOOPBACK

Этот параметр применяется только к маршрутизирующими сокетам (`AF_ROUTE`). По умолчанию он включен на этих сокетах (единственный из параметров `SO_xxx`, по умолчанию включенный). В этом случае сокет получает копию всего, что отправляется на сокет.

ПРИМЕЧАНИЕ

Другой способ отключить получение этих копий — вызвать функцию `shutdown` со вторым аргументом `SHUT_RD`.

7.6. Параметры сокетов IPv4

Эти параметры сокетов обрабатываются IPv4 и для них аргумент `level` равен `IPPROTO_IP`. Обсуждение пяти параметров сокетов многоадресной передачи мы отложим до раздела 19.5.

Параметр сокета IP_HRDINCL

Если этот параметр задан для символьного сокета IP (см. главу 28), нам следует создать наш собственный заголовок IP для всех дейтаграмм, которые мы отправляем через символьный сокет. Обычно ядро создает заголовок IP для дейтаграмм, отправляемых через символьный сокет, но существует ряд приложений (в частности, `traceroute`), создающих свой собственный заголовок IP, заменяющий значения, которые IP поместил бы в определенные поля заголовка.

Когда установлен этот параметр, мы создаем полный заголовок IP со следующими исключениями:

- IP всегда сам вычисляет и записывает контрольную сумму заголовка IP.
- Если мы устанавливаем поле идентификации IP в 0, ядро устанавливает это поле самостоятельно.
- Если IP-адрес отправителя (source address) — `INADDR_ANY`, IP устанавливает его равным основному IP-адресу исходящего интерфейса.

■ Как устанавливать параметры IP, зависит от реализации. Некоторые реализации добавляют любые параметры IP, установленные с использованием параметра сокета `IP_OPTIONS`, к создаваемому нами заголовку, в то время как другие требуют, чтобы мы сами добавили в заголовок все необходимые параметры IP.

■ Некоторые поля должны располагаться в порядке байтов узла, тогда как другие — в сетевом порядке байтов. Это тоже зависит от реализации, из-за чего программы, работающие с символьными сокетами с параметром `IP_HDRINCL`, становятся не такими переносимыми, как хотелось бы.

Пример использования этого параметра показан в разделе 29.7. Дополнительная информация об этом параметре представлена в [128, с. 1056–1057].

Параметр сокета `IP_OPTIONS`

Установка этого параметра позволяет нам задавать параметры IP в заголовке IPv4. Это требует точного знания формата параметров IP в заголовке IP. Мы рассмотрим этот параметр в контексте маршрутизации от отправителя IPv4 в разделе 27.3.

Параметр сокета `IP_RECVDSTADDR`

Этот параметр сокета заставляет функцию `recvmsg` возвращать IP-адрес получателя в получаемой дейтаграмме UDP в качестве вспомогательных данных. Пример использования этого параметра мы приводим в разделе 22.2.

Параметр сокета `IP_RECVIF`

Этот параметр сокета заставляет функцию `recvmsg` возвращать индекс интерфейса, на котором принимается дейтаграмма UDP, в качестве вспомогательных данных. Пример использования этого параметра мы приводим в разделе 22.2.

Параметр сокета `IP_TOS`

Этот параметр позволяет нам устанавливать поле *тип службы (тип сервиса)* (TOS, type-of-service) (рис. А.1) в заголовке IP для сокета TCP или UDP. Если мы вызываем для этого сокета функцию `getsockopt`, возвращается текущее значение, которое будет помещено в поля DSCP и ECN заголовка IP (по умолчанию значение нулевое). Не существует способа извлечь это значение из полученной дейтаграммы IP.

Приложение может установить DSCP равным одному из значений, о которых существует договоренность с провайдером. Каждому значению соответствует определенный тип обслуживания, например IP-телефонии требуется низкая задержка, а передачи больших объемов данных требуют повышенной пропускной способности. Документ RFC 2474 [82] определяет архитектуру diffserv, которая обеспечивает лишь ограниченную обратную совместимость с историческим определением поля TOS (RFC 1349 [5]). Приложения, устанавливающие параметр `IP_TOS` равным одной из констант, определенных в файле `<netinet/ip.h>` (например, `IPTOS_LOWDELAY` или `IPTOS_THROUGHPUT`), должны вместо этого использовать различные значения DSCP. Архитектура diffserv сохраняет только два значения (6 и 7, что соответствует константам `IPTOS_PREC_NETCONTROL` и `IPTOS_PREC_INTERNETCONTROL`), так что только те приложения, которые используют именно эти константы, будут работать в сетях diffserv.

Документ RFC 3168 [100] определяет поле ECN. Приложениям рекомендуется предоставлять установку этого поля ядру и сбрасывать в нуль два младших бита значения, заданного IP_TOS.

Параметр сокета IP_TTL

С помощью этого параметра мы можем устанавливать и получать заданное по умолчанию значение TTL (time-to-live field — поле времени жизни, рис. A.1), которое система будет использовать для данного сокета. (TTL для многоадресной передачи устанавливается при помощи параметра сокета IP_MULTICAST_TTL, который описывается в разделе 21.6.) В системе 4.4BSD, например, значение TTL по умолчанию для сокетов TCP и UDP равно 64 (оно определяется в RFC 1700), а для символьных сокетов — 255. Как и в случае поля TOS, вызов функции getsockopt возвращает значение поля по умолчанию, которое система будет использовать в исходящих дейтаграммах, и не существует способа определить это значение по полученной дейтаграмме. Мы устанавливаем этот параметр сокета в нашей программе traceroute в листинге 28.15.

7.7. Параметр сокета ICMPv6

Единственный параметр сокета, обрабатываемый ICMPv6, имеет аргумент level, равный IPPROTO_ICMPV6.

Параметр сокета ICMP6_FILTER

Этот параметр позволяет нам получать и устанавливать структуру icmp6_filter, которая определяет, какие из 256 возможных типов сообщений ICMPv6 передаются для обработки на символьный сокет. Мы обсудим этот параметр в разделе 28.4.

7.8. Параметры сокетов IPv6

Эти параметры сокетов обрабатываются IPv6 и имеют аргумент level, равный IPPROTO_IPV6. Мы отложим обсуждение пяти параметров сокетов многоадресной передачи до раздела 21.6. Отметим, что многие из этих параметров используют *вспомогательные данные* с функцией recvmsg, и мы покажем это в разделе 14.6. Все параметры сокетов IPv6 определены в RFC 3493 [36] и RFC 3542 [114].

Параметр сокета IPV6_CHECKSUM

Этот параметр сокета задает байтовое смещение поля контрольной суммы внутри данных пользователя. Если значение неотрицательное, ядро, во-первых, вычисляет и хранит контрольную сумму для всех исходящих пакетов и, во-вторых, проверяет полученную контрольную сумму на вводе, игнорируя пакеты с неверной контрольной суммой. Этот параметр влияет на символьные сокеты IPv6, отличные от символьных сокетов ICMPv6. (Ядро всегда вычисляет и хранит контрольную сумму для символьных сокетов ICMPv6.) Если задано значение -1 (значение по умолчанию), ядро не будет вычислять и хранить контрольную сумму для исходящих пакетов на этом символьном сокете и не будет проверять контрольную сумму для получаемых пакетов.

ПРИМЕЧАНИЕ

Все протоколы, использующие IPv6, должны иметь контрольную сумму в своих собственных заголовках. Эти контрольные суммы включают псевдозаголовок (RFC 2460 [27]), куда входит IPv6-адрес отправителя (что отличает IPv6 от всех остальных протоколов, которые обычно реализуются с использованием символьного сокета IPv4). Ядро не заставляет приложение, использующее символьный сокет, выбирать адрес отправителя, но делает это самостоятельно и затем вычисляет и сохраняет контрольную сумму, включающую псевдозаголовок IPv6.

Параметр сокета IPV6_DONTFRAG

Установка этого параметра запрещает автоматическое включение заголовка фрагментации для UDP и символьных сокетов. При этом исходящие пакеты, размер которых превышает MTU исходящего интерфейса, просто сбрасываются. Системный вызов ошибку не возвращает, так как пакет может быть сброшен и на промежуточном маршрутизаторе, если он превысит MTU одной из промежуточных линий. Чтобы получать уведомления об изменении маршрутной MTU, приложению следует включить параметр сокета IPV6_RECVPATHMTU (см. раздел 22.9).

Параметр сокета IPV6_NECHTHOP

Этот параметр задает адрес следующего транзитного узла для дейтаграммы в виде структуры адреса сокета. Подробнее о нем рассказывается в разделе 22.8.

Параметр сокета IPV6_PATHMTU

Этот параметр может быть только получен, но не установлен. При его считывании система возвращает текущее значение маршрутной MTU, определенное соответствующим методом (см. раздел 22.9).

Параметр сокета IPV6_RECVDSTOPTS

Установка этого параметра означает, что любые полученные IPv6-параметры получателя должны быть возвращены в качестве вспомогательных данных функцией `recvmsg`. По умолчанию параметр отключен. Мы опишем функции, используемые для создания и обработки этих параметров, в разделе 27.5.

Параметр сокета IPV6_RECVHOPLIMIT

Установка этого параметра определяет, что полученное поле предельного количества транзитных узлов (hop limit field) должно быть возвращено в качестве вспомогательных данных функцией `recvmsg`. По умолчанию параметр отключен. Мы опишем функции, используемые для создания и обработки этого параметра, в разделе 22.8.

ПРИМЕЧАНИЕ

В IPv4 не существует способа определить значение получаемого поля TTL.

Параметр сокета IPV6_RECVHOPOPTS

Установка этого параметра означает, что любые полученные параметры транзитных узлов (hop-by-hop options) IPv6 должны быть возвращены в качестве вспомогательных данных функцией `recvmsg`. По умолчанию параметр отключен. Мы опишем функции, используемые для создания и обработки этого параметра, в разделе 27.5.

Параметр сокета IPV6_RECVPATHMTU

Установка этого параметра означает, что маршрутная MTU должна быть возвращена в качестве вспомогательных данных функцией `recvmsg`, при условии, что ее значение изменилось. Параметр будет описан в разделе 22.9.

Параметр сокета IPV6_RECVPKTINFO

Установка этого параметра означает, что два фрагмента информации о полученной дейтаграмме IPv6 — IPv6-адрес получателя и индекс принимающего интерфейса — должны быть возвращены в качестве вспомогательных данных функцией `recvmsg`. Мы опишем этот параметр в разделе 22.8.

Параметр сокета IPV6_RECVRTHDR

Установка этого параметра означает, что получаемый заголовок маршрутизации IPv6 должен быть возвращен в качестве вспомогательных данных функцией `recvmsg`. По умолчанию этот параметр отключен. Мы опишем функции, которые используются для создания и обработки заголовка маршрутизации IPv6, в разделе 27.6.

Параметр сокета IPV6_RECVTCLASS

Установка этого параметра означает, что функция `recvmsg` должна вернуть сведения о классе трафика полученного сообщения (то есть содержимое полей DSCP и ECN) в качестве внешних данных. По умолчанию параметр отключен. Подробнее о нем будет рассказано в разделе 22.8.

Параметр сокета IPV6_UNICAST_HOPS

Этот параметр аналогичен параметру сокета IPv4 `IP_TTL`. Он определяет предельное количество транзитных узлов, заданное по умолчанию для исходящих дейтаграмм, отправляемых через этот сокет. При получении значения этого параметра сокета возвращается предельное количество транзитных узлов, которое ядро будет использовать для сокета. Чтобы определить действительное значение предельного количества транзитных узлов из полученной дейтаграммы IPv6, требуется использовать параметр сокета `IPV6_RECVHOPLIMIT`. Мы устанавливаем этот параметр сокета в нашей программе `traceroute` в листинге 28.15.

Параметр сокета IPV6_USE_MIN_MTU

Установка этого параметра равным 1 указывает на то, что определять маршрутную MTU не следует, а пакеты должны отправляться с минимальным значением MTU для IPv6, что предотвращает их фрагментацию. Если же значение параметра равно 0, определение маршрутной MTU выполняется для всех адресов назначения. Значение -1 (установленное по умолчанию) указывает на необходимость определения маршрутной MTU для направленной передачи, но для многоадресной передачи в этом случае используется минимально возможная MTU. Подробнее об этом параметре рассказывается в разделе 22.9.

Параметр сокета IPV6_V6ONLY

Включение этого параметра для сокета семейства `AF_INET6` ограничивает его использование исключительно протоколом IPv6. По умолчанию параметр отключен, хотя в некоторых системах существует возможность включить его по умолчанию. Взаимодействие по IPv4 и IPv6 через сокеты `AF_INET6` будет описано в разделах 12.2 и 12.3.

Параметры сокета IPV6_XXX

Большинство параметров IPv6, предназначенных для изменения содержимого заголовка, предполагают, что приложение использует сокет UDP и взаимодействует с ядром при помощи функций `recvmsg` и `sendmsg`. Сокет TCP получает и устанавливает значения параметров при помощи специальных функций `getsockopt` и `setsockopt`. Параметр сокета TCP совпадает с типом вспомогательных данных для UDP, а в буфере после вызова функций `getsockopt` и `setsockopt` оказывается та же информация, что и во вспомогательных данных. Подробнее см. раздел 27.7.

7.9. Параметры сокетов TCP

Для сокетов TCP предусмотрены два специальных параметра. Для них необходимо указывать level IPPROTO_TCP.

Параметр сокета TCP_MAXSEG

Этот параметр сокета позволяет нам получать или устанавливать *максимальный размер сегмента* (*maximum segment size, MSS*) для соединения TCP. Возвращаемое значение — это количество данных, которые наш TCP будет отправлять на другой конец соединения. Часто это значение равно MSS, анонсируемому другим концом соединения в его сегменте SYN, если наш TCP не выбирает меньшее значение, чем объявленный MSS собеседника. Если это значение получено до того, как сокет присоединился, возвращаемым значением будет значение по умолчанию, которое используется в том случае, когда параметр MSS не получен с другого конца соединения. Также помните о том, что значение меньше возвращенного действительно может использоваться для соединения, если, например, задействуется параметр отметки времени (timestamp), поскольку в каждом сегменте он занимает 12 байт области, отведенной под параметры TCP.

Максимальное количество данных, которые TCP отправляет в каждом сегменте, также может изменяться во время существования соединения, если TCP поддерживает определение транспортной MTU. Если маршрут к собеседнику изменяется, это значение может увеличиваться или уменьшаться.

В табл. 7.2 мы отметили, что этот параметр сокета может быть также установлен приложением. Это возможно не во всех системах: изначально параметр был доступен только для чтения. 4.4BSD позволяет приложению только лишь уменьшать это значение, но мы не можем его увеличивать [128, с. 1023]. Поскольку этот параметр управляет количеством данных, которое TCP посыпает в каждом сегменте, имеет смысл запретить приложению увеличивать значение. После установления соединения это значение задается величиной MSS, которую объявил собеседник, и мы не можем превысить его. Однако наш TCP всегда может отправить меньше данных, чем было анонсировано собеседником.

Параметр сокета TCP_NODELAY

Если этот параметр установлен, он отключает алгоритм Нагла (*Nagle algorithm*) (см. раздел 19.4 [111] и с. 858–859 [128]). По умолчанию этот алгоритм включен.

Назначение алгоритма Нагла — сократить число небольших пакетов в глобальной сети. Согласно этому алгоритму, если у данного соединения имеются неподтвержденные (outstanding) данные (то есть данные, которые отправил наш TCP и подтверждения которых он ждет), то небольшие пакеты не будут отправляться через соединение до тех пор, пока существующие данные не будут подтверждены. Под «небольшим» пакетом понимается любой пакет, меньший MSS. TCP будет по возможности всегда отправлять пакеты нормального размера. Таким образом, назначение алгоритма Нагла — не допустить, чтобы у соединения было множество небольших пакетов, ожидающих подтверждения.

Два типичных генератора небольших пакетов — клиенты Rlogin и Telnet, поскольку обычно они посыпают каждое нажатие клавиши в отдельном пакете. В быстрой локальной сети мы обычно не замечаем действия алгоритма Нагла с этими клиентами, потому что время, требуемое для подтверждения небольшого пакета, составляет несколько миллисекунд — намного меньше, чем промежуток между вводом двух последовательных символов. Но в глобальной сети, где для подтверждения небольшого пакета может потребоваться секунда, мы можем заметить задержку в отражении символов, и эта задержка часто увеличивается при включении алгоритма Нагла.

Рассмотрим следующий пример. Мы вводим строку из шести символов hello! либо клиенту Rlogin, либо клиенту Telnet, промежуток между вводом символов составляет точно 250 мс. Время обращения к серверу (RTT) составляет 600 мс и сервер немедленно отправляет обратно отражение символа. Мы считаем, что сегмент ACK, подтверждающий получение клиентского символа, отправляется обратно клиенту с отражением символа, а сегменты ACK, которые клиент отправляет для подтверждения приема отраженного сервером символа, мы игнорируем. (Мы поговорим о задержанных сегментах ACK далее.) Считая, что алгоритм Нагла отключен, получаем 12 пакетов, изображенных на рис. 7.7.

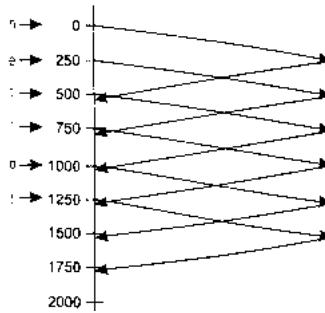


Рис. 7.7. Шесть символов, отраженных сервером при отключенном алгоритме Нагла

Каждый символ отправляется в индивидуальном пакете: сегменты данных слева направо, а сегменты ACK справа налево.

Но если алгоритм Нагла включен (по умолчанию), у нас имеется 8 пакетов, показанных на рис. 7.8. Первый символ посыпается как пакет, но следующие два символа не отправляются, поскольку у соединения есть небольшой пакет, ожидающий подтверждения. Эти пакеты отправляются, когда прошло 600 мс, то есть когда прибывает сегмент ACK, подтверждающий прием первого пакета, вместе с отражением первого символа. Пока второй пакет не будет подтвержден сегментом ACK в момент времени 1200, не будет отправлен ни один небольшой пакет.

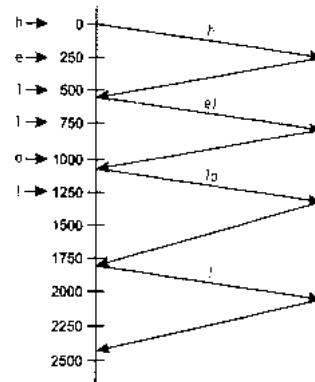


Рис. 7.8. Пакеты, отправляемые при включенном алгоритме Нагла

Алгоритм Нагла часто взаимодействует с другим алгоритмом TCP: алгоритмом задержанного сегмента ACK (*delayed ACK*). Этот алгоритм заставляет TCP не отправлять сегмент ACK сразу же при получении данных — вместо этого TCP ждет в течение небольшого количества времени (типичное значение 50-200 мс) и только после этого отправляет сегмент ACK. Здесь делается расчет на то, что в течение этого непродолжительного времени появятся данные для отправки собеседнику, и сегмент ACK может быть вложен в пакет с этими данными. Таким образом можно будет сэкономить на одном сегменте TCP. Это обычный случай с клиентами Rlogin и Telnet, поэтому сегмент ACK клиентского символа вкладывается в отражение символа сервером.

Проблема возникает с другими клиентами, серверы которых не генерируют трафика в обратном направлении, в который может быть вложен сегмент ACK. Эти клиенты могут обнаруживать значительные задержки, поскольку TCP клиента не будет посылать никаких данных серверу, пока не истечет время таймера для задержанных сегментов ACK сервера. Таким клиентам нужен способ отключения алгоритма Нагла. Осуществить это позволяет параметр `TCP_NODELAY`.

Другой тип клиента, для которого нежелательно использование алгоритма Нагла и задержанных ACK TCP, — это клиент, отправляющий одиночный логический запрос своему серверу небольшими порциями. Например, будем считать, что клиент отправляет своему серверу 400-байтовый запрос, состоящий из 4 байт, задающих тип запроса, за которыми следуют 396 байт данных. Если клиент выполняет функцию `write`, отправляя 4 байт, и затем функцию `write`, отправляя остальные 396 байт, вторая часть не будет отправлена со стороны клиента, пока TCP сервера не подтвердит получение первых 4 байт. Кроме того, поскольку сервер не может работать с 4 байтами данных, пока не получит оставшиеся 396 байт, TCP сервера задержит сегмент ACK, подтверждающий получение 4 байт данных (то есть не будет данных от сервера клиенту, в которые можно вложить сегмент ACK). Есть три способа решить проблему с таким клиентом.

1. Использовать функцию `writev` (раздел 14.4) вместо двух вызовов функции `write`. Один вызов функции `writev` приводит к отправке только одного сегмента TCP в нашем примере. Это предпочтительное решение.

2. Скопировать 4 байт и 396 байт данных в один буфер и вызывать один раз функцию `write` для этого буфера.

3. Установить параметр сокета `TCP_NODELAY` и продолжать вызывать функцию `write` дважды. Это наименее желательное решение.

Упражнения 7.8 и 7.9 продолжают этот пример.

7.10. Параметры сокетов SCTP

Относительно большое количество параметров, определенных для сокетов SCTP (17 на момент написания этой книги), дают возможность разработчику приложения более точно контролировать его поведение. Параметр `level` для сокетов SCTP должен принимать значение `IPPROTO_SCTP`.

Несколько параметров, используемых для получения сведений об SCTP, требуют передачи данных ядру (например, идентификатора ассоциации или адреса собеседника). Не все реализации `getsockopt` поддерживают передачу данных в обе стороны. Интерфейс сокетов SCTP определяет функцию `sctp_opt_info` (раздел 9.11), которая устраниет эту проблему. В некоторых системах, где `getsockopt` поддерживает передачу данных в ядро, функция `sctp_opt_info` является не более, чем оболочкой для `getsockopt`. В других системах она может вызывать функцию `ioctl` или какую-либо иную, возможно, созданную специально для данного случая. Мы рекомендуем получать параметры сокетов SCTP при помощи `sctp_opt_info`, так как в этом случае обеспечивается максимальная переносимость. В табл. 7.2 соответствующие параметры отмечены знаком «+»: `SCTP_ASSOCINFO`, `SCTP_GET_PEER_ADDR_INFO`, `SCTP_PEER_ADDR_PARAMS`, `SCTP_PRIMARY_ADDR`, `SCTP_RTOINFO` и `SCTP_STATUS`.

Параметр сокета `SCTP_ADAPTION_LAYER`

При инициализации ассоциации любой собеседник может указать на наличие уровня-адаптора. Это указание должно представлять из себя 32-разрядное беззнаковое целое, которое может использоваться двумя приложениями для координации локального уровня-адаптора приложений. Параметр сокета позволяет получать или устанавливать указание на наличие уровня-адаптора, которое будет предоставляться данной конечной точкой будущим собеседникам. Чтобы получить соответствующее значение, установленное собеседником, приложение должно подписаться на события уровня-адаптора.

Параметр сокета `SCTP_ASSOCINFO`

Параметр сокета `SCTP_ASSOCINFO` выполняет три функции. Во-первых, он позволяет получать сведения о существующей ассоциации. Во-вторых, с его помощью можно изменять параметры существующей ассоциации. Наконец, в-третьих, через этот параметр можно задавать значения по умолчанию для будущих ассоциаций. При получении сведений о существующей ассоциации вместо `getsockopt` следует использовать `sctp_opt_info`. Вместе с параметром при вызове функции указывается структура `sctp_assocparams`:

```
struct sctp_assocparams {
    sctp_assoc_t sasoc_assoc_id;
    uint16_t sasoc_asocmaxrxt;
    uint16_t sasoc_number_peer_destinations;
    uint32_t sasoc_peer_rwnd;
    uint32_t sasoc_local_rwnd;
    uint32_t sasoc_cookie_life;
};
```

Поля структуры имеют следующий смысл:

- `sasoc_assoc_id` хранит идентификатор ассоциации. Если при вызове `setsockopt` параметр установлен в нуль, поля `sasoc_asocmaxrxt` и `sasoc_cookie_life` трактуются как новые значения по умолчанию для сокета. Вызов `getsockopt` вернет сведения об ассоциации, если при вызове указать ее идентификатор; если же поле оставить нулевым, будут возвращены значения по умолчанию;

- `sasoc_asocmaxrxt` хранит количество повторных передач без получения подтверждений. При превышении этого ограничения передача прекращается, ассоциация закрывается и SCTP сообщает приложению о недоступности собеседника;
- `sasoc_number_peer_destinations` хранит количество адресов собеседника. Этот параметр может быть только считан, но не установлен;
- `sasoc_peer_rwnd` хранит текущее рассчитанное окно приема собеседника, то есть количество байтов, которые могут быть переданы в данный момент. Это поле изменяется динамически. Когда приложение отправляет данные, значение поля уменьшается, когда удаленное приложение считывает полученные данные, значение увеличивается. Вызовом данного параметра сокета это значение изменено быть не может;
- `sasoc_local_rwnd` хранит размер локального окна приема, о котором SCTP оповещает собеседнику. Это значение также изменяется динамически и зависит от параметра сокета `SO_SNDBUF`. Вызовом параметра `SCTP_ASSOCINFO` локальное окно изменено быть не может;
- `sasoc_cookie_life` хранит срок действия `cookie`, выданного собеседнику (в миллисекундах). Каждому `cookie` присваивается определенный срок действия, благодаря чему обеспечивается защита от атак, основанных на повторах. Значение по умолчанию равно 60 000 и может быть изменено установкой нужного значения в данном поле при условии, что в поле `sasoc_assoc_id` записано значение 0.

Рекомендации по настройке `sasoc_asocmaxrxt` для оптимальной производительности приводятся в разделе 23.11. Для защиты от атак, основанных на повторе, значение `sasoc_cookie_life` можно уменьшить, но при этом система окажется менее устойчивой к задержкам в процессе инициализации. Прочие поля полезны для отладки программ.

Параметр сокета `SCTP_AUTOCLOSE`

Этот параметр позволяет получать и устанавливать время автоматического закрытия конечной точки SCTP. Это время задается в секундах и определяет длительность существования ассоциации SCTP, по которой не передаются никакие данные. Передача данных контролируется стеком SCTP. По умолчанию функция автоматического закрытия отключена.

Параметр предназначен для использования на интерфейсах SCTP типа «один-ко-многим» (см. главу 9). Положительное значение соответствует времени поддержания неиспользуемой ассоциации в секундах, а нулевое отключает функцию автоматического закрытия. Установка параметра влияет только на будущие ассоциации, все существующие ассоциации сохраняют старые значения.

Автоматическое закрытие может использоваться сервером для закрытия неиспользуемых ассоциаций без дополнительных затрат на хранение информации о состоянии. Однако разработчик сервера должен тщательно оценить максимальную продолжительность бездействия клиентов. Если значение параметра окажется недостаточно большим, ассоциации будут закрываться слишком рано.

Параметр сокета `SCTP_DEFAULT_SEND_PARAM`

SCTP поддерживает множество дополнительных параметров отправки, которые обычно передаются в виде вспомогательных данных или используются при вызове функции `sctp_sendmsg` (который часто реализуется как библиотечный вызов, передающий вспомогательные данные пользователя). Приложение, планирующее отправку большого количества сообщений с одинаковыми параметрами, может воспользоваться параметром `SCTP_DEFAULT_SEND_PARAM` для настройки значений параметров по умолчанию и тем самым избавиться от необходимости добавлять вспомогательные данные или вызывать `sctp_sendmsg`. На вход параметра поступает структура `sctp_sndrcvinfo`:

```
struct sctp_sndrcvinfo {
    u_int16_t sinfo_stream;
    u_int16_t sinfo_ssn;
    u_int16_t sinfo_flags;
    u_int32_t sinfo_ppid;
    u_int32_t sinfo_context;
    u_int32_t sinfo_timetolive;
    u_int32_t sinfo_tsn;
    u_int32_t sinfo_cumtsn;
    sctp_assoc_t sinfo_assoc_id;
```

```
};
```

Поля структуры определяются следующим образом:

- `sinfo_stream` задает поток, в который по умолчанию направляются все сообщения;
- `sinfo_ssn` игнорируется при установке значений параметров по умолчанию. При получении сообщений функцией `recvmsg` или `sctp_recvmsg` это поле содержит значение потокового последовательного номера (stream sequence number, SSN), помещенное собеседником в порцию данных;
- `sinfo_flags` устанавливает значения всех флагов для будущих сообщений. Допустимые значения флагов приводятся в табл. 7.5;
- `sinfo_ppid` задает значение идентификатора протокола SCTP для всех будущих передач данных;
- `sinfo_context` задает значение по умолчанию для поля `sinfo_context`, которое является локальной меткой для сообщений, которые не могли быть доставлены собеседнику;
- `sinfo_timetolive` определяет время жизни отправляемых сообщений. Поле времени жизни используется стеком SCTP для того, чтобы сбрасывать сообщения, задержавшиеся в буфере отправки на слишком большой срок и не переданные ни разу. Если обе конечные точки поддерживают режим частичной надежности, параметр времени жизни влияет и на количество попыток повторной передачи, ограничивая их срок;
- `sinfo_tsn` игнорируется при установке параметров по умолчанию. При получении сообщений функцией `recvmsg` или `sctp_recvmsg` это поле содержит значение транспортного последовательного номера (transport sequence number, TSN), помещенное собеседником в порцию данных SCTP;
- `sinfo_cumtsn` игнорируется при установке параметров по умолчанию. При получении сообщений функцией `recvmsg` или `sctp_recvmsg` это поле содержит значение кумулятивного транспортного последовательного номера, вычисленного локальным стеком SCTP для удаленного собеседника;
- `sinfo_assoc_id` содержит идентификатор ассоциации, для которой требуется установка параметров по умолчанию. Для сокетов типа «один-к-одному» это поле игнорируется.

Таблица 7.5. Допустимые значения флагов SCTP (поле `sinfo_flags`)

Константа	Описание
<code>MSG_ABORT</code>	Вызывает аварийное завершение ассоциации
<code>MSG_ADDR_OVER</code>	Заставляет SCTP использовать указанный адрес вместо адреса по умолчанию
<code>MSG_EOF</code>	Корректное завершение ассоциации после отправки сообщения
<code>MSG_PR_BUFFER</code>	Включение частичной надежности в зависимости от буфера (если она вообще поддерживается)
<code>MSG_PR_SCTP</code>	Включение частичной надежности доставки для данного сообщения (если поддерживается)
<code>MSG_UNORDERED</code>	Указывает, что данное сообщение использует сервис неупорядоченной доставки

Обратите внимание, что значения параметров по умолчанию используются только тогда, когда сообщение отправляется без собственной структуры `sctp_sndrcvinfo`. Если же эта структура добавляется во вспомогательные данные при отправке сообщений, заданные в ней значения имеют приоритет перед значениями по умолчанию. Параметр `SCTP_DEFAULT_SEND_PARAM` может использоваться для получения текущих значений по умолчанию при помощи функции `sctp_opt_info`.

Параметр сокета `SCTP_DISABLE_FRAGMENTS`

В обычном режиме работы SCTP фрагментирует все сообщения, не помещающиеся в один пакет SCTP, разбивая их на несколько порций типа DATA. Установка параметра `SCTP_DISABLE_FRAGMENTS` отключает фрагментацию для данного отправителя. Если сообщение требует фрагментации, а фрагментация отключена, SCTP возвращает ошибку `EMSGSIZE` и не отсылает сообщение.

Параметр может использоваться приложениями, которые хотят самостоятельно управлять размерами сообщений, при условии, что любое из этих сообщений может поместиться в IP-пакет. Приложение должно быть готово обработать ошибку, обеспечив фрагментацию на уровне приложения или изменение размера сообщений.

Параметр сокета `SCTP_EVENTS`

Этот параметр сокета позволяет включать, выключать и определять состояние подписки на различные уведомления SCTP. Уведомление SCTP представляет собой сообщение, отправляемое стеком SCTP приложению. Сообщение считывается как и обычные данные, однако в поле `msg_flags` при вызове функции `recvmsg` должно находиться значение `MSG_NOTIFICATION`. Приложение, не готовое к использованию `recvmsg` или `sctp_recvmsg`, не должно включать подписку на события. Параметр позволяет управлять событиями восьми различных типов и передавать структуру `sctp_event_subscribe`. Нулевое значение соответствует отключению подписки, а единица — включению.

Структура `sctp_event_subscribe` определяется следующим образом:

```
struct sctp_event_subscribe {
    u_int8_t sctp_data_io_event;
    u_int8_t sctp_association_event;
    u_int8_t sctp_address_event;
    u_int8_t sctp_send_failure_event;
    u_int8_t sctp_peer_error_event;
    u_int8_t sctp_shutdown_event;
    u_int8_t sctp_partial_delivery_event;
    u_int8_t sctp_adaption_layer_event;
};
```

В табл. 7.6 описано назначение различных событий. Подробнее об уведомлениях вы узнаете в разделе 9.14.

Таблица 7.6. События SCTP

Константа	Описание
<code>sctp_data_io_event</code>	Включение и отключение доставки <code>sctp_sndrcvinfo</code> с каждым вызовом <code>recvmsg</code>
<code>sctp_association_event</code>	Включение и отключение уведомлений о состоянии ассоциации
<code>sctp_address_event</code>	Включение и отключение уведомлений об адресах
<code>sctp_send_failure_event</code>	Включение и отключение уведомлений об ошибках доставки сообщений
<code>sctp_peer_error_event</code>	Включение и отключение уведомлений об ошибках протокола собеседника
<code>sctp_shutdown_event</code>	Включение и отключение уведомлений о завершении ассоциации
<code>sctp_partial_delivery_event</code>	Включение и отключение уведомлений о частичной доставке
<code>sctp_adaption_layer_event</code>	Включение и отключение уведомлений уровня-адаптера

Параметр сокета `SCTP_GET_PEER_ADDR_INFO`

Этот параметр позволяет получить информацию о собеседнике, которая включает окно приема, слаженные значения RTT и MTU. Параметр может быть применен только к конкретному адресу собеседника. Вызывающее приложение заполняет поле `spinfo_address` структуры `sctp_paddrinfo` интересующим его адресом собеседника. Для максимальной переносимости рекомендуется работать с функцией `sctp_opt_info`, а не `getsockopt`. Формат структуры `sctp_paddrinfo` описан ниже:

```
struct sctp_paddrinfo {
    sctp_assoc_t spinfo_assoc_id;
    struct sockaddr_storage spinfo_address;
    int32_t spinfo_state;
    uint32_t spinfo_cwnd;
    u_int32_t spinfo_srtt;
    u_int32_t spinfo_rto;
    u_int32_t spinfo_mtu;
};
```

Приложению возвращаются следующие сведения:

- `spinfo_assoc_id` содержит информацию об идентификаторе ассоциации, которая доставляется также в уведомлении об установке ассоциации (`SCTP_COMM_UP`). Уникальный идентификатор ассоциации может использоваться для обращения к ней в большинстве функций SCTP;

- `spinfo_address` позволяет приложению указать конкретный адрес собеседника, для которого оно хочет получить сведения. По возвращении из `getsockopt` или `sctp_opt_info` значение структуры должно

оставаться неизменным;

- `sinfo_state` может содержать одно или несколько значений (табл. 7.7).

Таблица 7.7. Состояния адреса собеседника SCTP

Константа	Описание
<code>SCTP_ACTIVE</code>	Адрес активен и доступен
<code>SCTP_INACTIVE</code>	В настоящий момент адрес недоступен
<code>SCTP_ADDR_UNCONFIRMED</code>	Доставка данных или проверочных сообщений на данный адрес не была подтверждена

Неподтвержденным считается адрес, перечисленный собеседником в списке действующих, но не проверенный локальным SCTP. Для проверки адреса требуется, чтобы отправленные на него данные или проверочные сообщения были подтверждены. Для непроверенного адреса не может быть указано корректное значение тайм-аута повторной передачи (RTO). Активными считаются адреса, доступные для передачи данных.

■ `sinfo_cwnd` хранит текущий размер окна приема для данного адреса. Описание процедуры расчета параметра `cwnd` приводится в [117, с. 177];

- `sinfo_srtt` хранит текущую оценку сглаженного RTT для данного адреса;
- `sinfo_rto` хранит текущее значение тайм-аута повторной передачи для данного адреса;
- `sinfo_mtu` хранит текущую транспортную MTU, определенную по соответствующему алгоритму.

Параметр полезно использовать для получения идентификатора ассоциации по структуре с IP-адресом собеседника. Это будет продемонстрировано в главе 23. Кроме того, приложение может отслеживать функционирование всех адресов собеседника с несколькими интерфейсами и выбирать лучший из них в качестве адреса по умолчанию. Наконец, все эти сведения полезны для ведения журналов и отладки.

Параметр сокета `SCTP_I_WANT_MAPPED_V4_ADDR`

Этот флаг позволяет включать и отключать отображение адресов IPv4 для сокетов типа `AF_INET6`. Если параметр включен (а по умолчанию это именно так), все адреса IPv4 преобразуются в адреса IPv6 перед отправкой приложению. Если же параметр отключен, сокет SCTP не будет отображать адреса IPv4, а вместо этого будет просто передавать их в структуре `sockaddr_in`.

Параметр сокета `SCTP_INITMSG`

Параметр позволяет устанавливать и считывать параметры инициализации, по умолчанию применяемые к сокетам при отправке сообщения INIT. Вместе с параметром передается структура `sctp_initmsg`, определяемая следующим образом:

```
struct sctp_initmsg {  
    uint16_t sinit_num_ostreams;  
    uint16_t sinit_max_instreams;  
    uint16_t sinit_max_attempts;  
    uint16_t sinit_max_init_timeo;  
};
```

Поля структуры определяются следующим образом:

■ `sinit_num_ostreams` содержит количество исходящих потоков SCTP, запрашиваемое приложением. Это значение не подтверждается, пока не будет завершено рукопожатие, и может быть уменьшено в соответствии с возможностями собеседника;

■ `sinit_max_instreams` отражает максимальное количество входящих потоков, которое готово обеспечить приложение. Это значение может быть перекрыто стеком SCTP, если оно превышает максимальное количество потоков, поддерживаемое самим стеком;

■ `sinit_max_attempts` выражает количество попыток передачи начального сообщения INIT перед тем, как собеседник будет признан недоступным;

■ `sinit_max_init_timeo` задает максимальный тайм-аут повторной передачи для сообщений INIT. Это значение используется вместо `RTO_MAX` в качестве ограничения сверху на тайм-аут повторной передачи. Выражается в миллисекундах.

Обратите внимание, что установленные в 0 поля структуры игнорируются сокетом SCTP. При использовании сокета типа «один-ко-многим» (см. раздел 9.2) приложение может передать структуру `sctp_initmsg` во вспомогательных данных при неявной установке ассоциации.

Параметр сокета `SCTP_MAXBURST`

Этот параметр позволяет приложению устанавливать и считывать максимальный размер набора пакетов (maximum burst size). SCTP никогда не отправляет более, чем `SCTP_MAXBURST` пакетов одновременно, что предотвращает переполнение сети. Ограничение может применяться либо путем уменьшения окна до текущего количества пакетов «в пути» (in flight) плюс максимальный размер набора, помноженный на транспортную MTU, либо в качестве отдельного параметра, если при каждой возможности отправки будет пересыпаться не более `SCTP_MAXBURST` пакетов.

Параметр сокета `SCTP_MAXSEG`

Параметр позволяет приложению считывать и устанавливать максимальный размер фрагмента, аналогично `TCP_MAXSEG` (см. раздел 7.8).

Когда стек SCTP получает от приложения-отправителя сообщение, размер которого превышает значение этого параметра, это сообщение разбивается на несколько фрагментов, которые доставляются на вторую конечную точку по отдельности. Обычно SCTP создает фрагменты такого размера, чтобы они не превышали минимальную MTU для всех адресов собеседника. Параметр позволяет еще сильнее уменьшить это значение. Учтите, что стек SCTP может фрагментировать даже такое сообщение, размер которого не превышает `SCTP_MAXSEG`. Это произойдет в том случае, если MTU для одного из адресов собеседника окажется меньше значения `SCTP_MAXSEG`.

Параметр действует для всех адресов конечной точки и может влиять на несколько ассоциаций при работе с интерфейсами типа «один-ко-многим».

Параметр сокета `SCTP_NODELAY`

Установка параметра отключает алгоритм Нагла протокола SCTP. По умолчанию параметр выключен, то есть алгоритм Нагла включен. С протоколом SCTP этот алгоритм работает так же, как и с TCP, за тем исключением, что он пытается объединять порции данных, а не отдельные байты. Подробнее см. описание параметра `TCP_NODELAY`.

Параметр сокета `SCTP_PEER_ADDR_PARAMS`

Параметр позволяет приложению считывать и устанавливать различные параметры ассоциации. Приложение должно заполнить поле идентификатора ассоциации в структуре `sctp_paddrparams` и передать ее вместе с параметром сокета. Формат структуры приведен ниже:

```
struct sctp_paddrparams {  
    sctp_assoc_t spp_assoc_id;  
    struct sockaddr_storage spp_address;  
    u_int32_t spp_hbinterval;  
    u_int16_t spp_pathmaxrxt;  
};
```

Поля структуры имеют следующий смысл:

- `spp_assoc_id` содержит идентификатор ассоциации, параметры которойчитываются или устанавливаются. Если это значение равно нулю, приложение будет работать с параметрами по умолчанию, а не с конкретной ассоциацией;
- `spp_address` указывает IP-адрес, для которого запрашиваются или устанавливаются параметры. Если значение поля равно нулю, оно игнорируется;
- `spp_hbinterval` задает интервал между проверочными сообщениями (heartbeats). Значение `SCTP_NO_HB` отключает проверочные сообщения. Значение `SCTP_ISSUE_HB` приводит к внеочередной

отправке проверочного сообщения. Все остальные значения задают интервал проверки в миллисекундах. При установке параметров по умолчанию задание константы SCTP_ISSUE_NB не допускается;

■ spp_hbpathmaxrxt определяет максимальное количество повторных передач, после которых адресат считается недоступным (INACTIVE). Если основной адрес собеседника признается недоступным, в качестве нового основного адреса выбирается один из доступных адресов.

Параметр сокета SCTP_PRIMARY_ADDR

Параметр позволяет узнать или установить адрес, используемый локальной конечной точкой SCTP в качестве основного. Основной адрес используется в качестве адреса назначения во всех сообщениях, передаваемых собеседнику. Приложение должно заполнить структуру sctp_setprim идентификатором ассоциации и адресом собеседника.

```
struct sctp_setprim {  
    sctp_assoc_t ssp_assoc_id;  
    struct sockaddr_storage ssp_addr;  
};
```

Поля структуры имеют следующий смысл:

■ ssp_assoc_id указывает идентификатор ассоциации, для которой следует установить или считать основной адрес. В случае сокета типа «один-к-одному» это поле игнорируется;

■ ssp_addr определяет основной адрес, который обязательно должен принадлежать собеседнику. Если используется функция setsockopt, значение поля трактуется как новый основной адрес собеседника.

Получение значения этого параметра для сокета типа «один-к-одному» с единственным локальным адресом эквивалентно вызову функции getsockname.

Параметр сокета SCTP_RTOINFO

Параметр используется для считывания и установки различных тайм-аутов для конкретной ассоциации или используемых по умолчанию для конечной точки. Для считывания параметров по соображениям переносимости следует использовать функцию sctp_opt_info, а не getsockopt. Перед вызовом необходимо заполнить структуру sctp_rtoinfo, которая определяется следующим образом:

```
struct sctp_rtoinfo {  
    sctp_assoc_t srto_assoc_id;  
    uint32_t srto_initial;  
    uint32_t srto_max;  
    uint32_t srto_min;  
};
```

Поля структуры имеют следующий смысл:

■ srto_assoc_id содержит либо идентификатор конкретной ассоциации, либо 0. В последнем случае работа осуществляется со значениями по умолчанию;

■ srto_initial хранит начальное значение RTO для конкретного адреса собеседника. Это значение используется при отправке порции INIT. Измеряется поле в миллисекундах и по умолчанию равно 3000;

■ srto_max содержит максимальное значение RTO, используемое при изменении таймера повторной передачи. Если рассчитанное значение оказывается больше максимального RTO, в качестве нового таймаута используется именно максимальное значение. По умолчанию это поле имеет значение 60 000 мс;

■ srto_min содержит минимальное значение RTO, используемое при первом запуске таймера повторной передачи. Когда таймер RTO изменяется, новое значение обязательно сравнивается с минимальным. По умолчанию это поле имеет значение 1000 мс.

Запись 0 в поля srto_initial, srto_max и srto_min означает, что менять текущие параметры по умолчанию не требуется. Все значения измеряются в миллисекундах. Руководство по установке таймеров для достижения максимальной производительности приводится в разделе 23.11.

Параметр сокета SCTP_SET_PEER_PRIMARY_ADDR

Установка этого параметра приводит к отправке собеседнику сообщения, запрашивающего установку конкретного локального адреса в качестве основного. Процесс должен заполнить структуру

`sctp_setpeerprim` и указать в ней идентификатор ассоциации и локальный адрес, который должен быть сделан основным. Этот адрес должен быть привязан к данной конечной точке. Структура `sctp_setpeerprim` определяется следующим образом:

```
struct sctp_setpeerprim {  
    sctp_assoc_t sspp_assoc_id;  
    struct sockaddr_storage sspp_addr;  
};
```

Ниже приводится описание полей структуры.

- `sspp_assoc_id` указывает идентификатор ассоциации, для которой требуется установить новый основной адрес. При работе с сокетом типа «один-к-одному» это поле игнорируется;
- `sspp_addr` содержит локальный адрес, который должен использоваться собеседником в качестве основного.

Поддержка этой функции SCTP не является обязательной. Если локальная конечная точка не поддерживает параметр, процессу будет возвращена ошибка `EOPNOTSUPP`. Если же параметр не поддерживается удаленной конечной точкой, ошибка будет другой: `EINVAL`. Обратите внимание, что данный параметр не может использоваться для считывания основного адреса; он служит только для установки нового адреса в качестве основного.

Параметр сокета `SCTP_STATUS`

Этот параметр сокета служит для получения информации о текущем статусе ассоциации SCTP. Для обеспечения максимальной переносимости пользуйтесь функцией `sctp_opt_info`, а не `getaddrinfo`. Приложение должно предоставить структуру `sctp_status`, указав идентификатор ассоциации `sstat_assoc_id`. Структура будет заполнена информацией о выбранной ассоциации и возвращена приложению. Формат структуры `sctp_status` таков:

```
struct sctp_status {  
    sctp_assoc_t sstat_assoc_id;  
    int32_t sstat_state;  
    u_int32_t sstat_rwnd;  
    u_int16_t sstat_unackdata;  
    u_int16_t sstat_penddata;  
    u_int16_t sstat_instrms;  
    u_int16_t sstat_outstrms;  
    u_int32_t sstat_fragmentation_point;  
    struct sctp_paddrinfo sstat_primary;  
};
```

Поля структуры имеют следующий смысл:

- `sstat_assoc_id` содержит идентификатор ассоциации;
- `sstat_state` содержит константу, обозначающую состояние ассоциации (табл. 7.8). Подробное описание состояний конечной точки SCTP, чередующихся при установке и завершении ассоциации, приводится на рис. 2.8;
- `sstat_rwnd` содержит текущее вычисленное значение приемного окна собеседника;
- `sstat_unackdata` содержит количество неподтвержденных порций данных, ждущих ответа собеседника;
- `sstat_penddata` содержит количество непрочитанных порций данных, подготовленных локальной конечной точкой SCTP для приложения;
- `sstat_instrms` содержит количество потоков, используемых собеседником для передачи данных на данную конечную точку;
- `sstat_outstrms` содержит количество потоков, по которым данная конечная точка может передавать данные собеседнику;
- `sstat_fragmentation_point` содержит текущее значение границы фрагментации пользовательских сообщений, используемое локальной конечной точкой SCTP. Это значение обычно равняется минимальной MTU для всех адресатов или еще меньшей величине, установленной при помощи параметра `SCTP_MAXSEG`;
- `sstat_primary` содержит текущий основной адрес. Основной адрес используется по умолчанию для отправки данных собеседнику.

Таблица 7.8. Состояния SCTP

Константа	Описание
SCTP_CLOSED	Ассоциация закрыта
SCTP_COOKIE_WAIT	Ассоциация отправила пакет INIT
SCTP_COOKIE_ECHOED	Ассоциация отправила эхо-ответ cookie
SCTP_ESTABLISHED	Ассоциация установлена
SCTP_SHUTDOWN_PENDING	Ассоциация ждет отправки сообщения о завершении
SCTP_SHUTDOWN_SENT	Ассоциация отправила сообщение о завершении
SCTP_SHUTDOWN_RECEIVED	Ассоциация получила сообщение о завершении
SCTP_SHUTDOWN_ACK_SENT	Ассоциация ждет пакета SHUTDOWN-COMPLETE

Эти параметры полезны для диагностики соединения и определения характеристик текущего сеанса. Например, функция `sctp_get_no_strms` в разделе 10.2 будет считывать `sstat_outstrms` для определения количества доступных для отправки данных потоков. Низкое значение `sstat_rwnd` или высокое значение `sstat_unackdata` позволяет сделать вывод о заполнении приемного буфера собеседника, так что приложение может вовремя замедлить передачу данных. Поле `sstat_fragmentation_point` может использоваться некоторыми приложениями для уменьшения количества пакетов, создаваемых SCTP, путем уменьшения размеров сообщений.

7.11. Функция `fcntl`

Сокращение `fcntl` означает «управление файлами» (file control). Эта функция выполняет различные операции управления дескрипторами. Перед описанием этой функции и ее влияния на сокет нам нужно составить некоторое более общее представление о ее возможностях. В табл. 7.9 приводятся различные операции, выполняемые функциями `fcntl` и `ioctl` и маршрутизирующими сокетами.

Таблица 7.9. Операции функций `fcntl` и `ioctl` и маршрутизирующих сокетов

Операция	<code>fcntl</code>	<code>ioctl</code>	Маршрутизирующий сокет	Posix.1g
Установка сокета для неблокируемого ввода-вывода	<code>F_SETFL</code> , <code>O_NONBLOCK</code>	<code>FIONBIO</code>		<code>fcntl</code>
Установка сокета для ввода-вывода, управляемого сигналом	<code>F_SETFL</code> , <code>O_ASYNC</code>	<code>FIOASYNC</code>		<code>fcntl</code>
Установка владельца сокета	<code>F_SETOWN</code>	<code>SIOCSPGRP</code> или <code>FIOSETOWN</code>		<code>fcntl</code>
Получение владельца сокета	<code>F_GETOWN</code>	<code>SIOCGPGRP</code> или <code>FIOGETOWN</code>		<code>fcntl</code>
Получение текущего количества байтов в приемном буфере сокета		<code>FIONREAD</code>		
Проверка, находится ли процесс на отметке внеполосных данных		<code>SIOCATMARK</code>		<code>socketmark</code>
Получение списка интерфейсов		<code>SIOCGIFCONF</code>	<code>Sysctl</code>	
Операции интерфейсов		<code>SIOC[GS]IFxxx</code>		
Кэш-операции ARP		<code>SIOCxARP</code>	<code>RTM_xxx</code>	
Операции таблицы маршрутизации		<code>SIOGxxxRT</code>	<code>RTM_xxx</code>	

Первые шесть операций могут применяться к сокетам любым процессом, следующие две (операции над интерфейсами) используются реже, а последние две (ARP и таблица маршрутизации) выполняются администрирующими программами, такими как `ifconfig` и `route`. О различных операциях функции `ioctl` мы поговорим подробнее в главе 17, а о маршрутизирующих сокетах — в главе 18.

Существует множество способов выполнения первых четырех операций, но, как указано в последней колонке, стандарт POSIX определяет, что функция `fcntl` является предпочтительным способом. Отметим также, что POSIX предлагает функцию `socketmark` (см. раздел 24.3) как наиболее предпочтительный

способ тестирования на предмет пребывания процесса на отметке внеполосных данных. Оставшиеся операции с пустой последней колонкой не стандартизованы POSIX.

ПРИМЕЧАНИЕ

Отметим также, что первые две операции, устанавливающие сокет для неблокируемого ввода-вывода и для ввода-вывода, управляемого сигналом, традиционно применялись с использованием команд FNDELAY и FASYNC функции fcntl. POSIX определяет константы O_xxx.

Функция fcntl предоставляет следующие возможности, относящиеся к сетевому программированию:

- Неблокируемый ввод-вывод. Мы можем установить флаг состояния файла O_NONBLOCK, используя команду F_SETFL для отключения блокировки сокета. Неблокируемый ввод-вывод мы описываем в главе 16.
- Управляемый сигналом ввод-вывод. Мы можем установить флаг состояния файла O_ASYNC, используя команду F_SETFL, после чего при изменении состояния сокета будет генерироваться сигнал SIGIO. Мы рассмотрим это в главе 25.
- Команда F_SETOWN позволяет нам установить владельца сокета (идентификатор процесса или идентификатор группы процессов), который будет получать сигналы SIGIO и SIGURG. Первый сигнал генерируется, если для сокета включен управляемый сигналом ввод-вывод (см. главу 25), второй — когда для сокета приходят новые внеполосные (out-of-band data) данные (см. главу 24). Команда F_GETOWN возвращает текущего владельца сокета.

ПРИМЕЧАНИЕ

Термин «владелец сокета» определяется POSIX. Исторически реализации, происходящие от Беркли, называли его «идентификатор группы процессов сокета», потому что переменная, хранящая этот идентификатор, — это элемент so_pgid структуры socket [128, с. 438].

```
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd, ... /* int arg */);
```

Возращает: в случае успешного выполнения результат зависит от аргумента cmd, -1 в случае ошибки

Каждый дескриптор (включая сокет) имеет набор флагов, которые можно получить с помощью команды F_GETFL и установить с помощью команды F_SETFL. На сокет влияют следующие два флага:

- O_NONBLOCK — неблокируемый ввод-вывод;
- O_ASYNC — ввод-вывод, управляемый сигналом.

Позже мы опишем оба эти флага подробнее. Отметим, что типичный код, который устанавливает неблокируемый ввод-вывод с использованием функции fcntl, выглядит следующим образом:

```
int flags;
```

```
/* Делаем сокет неблокируемым */
if ((flags = fcntl(fd, F_GETFL, 0)) < 0)
    err_sys("F_GETFL error");
flags |= O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

Учтите, что вам может встретиться код, который просто устанавливает желаемый флаг:

```
/* Неправильный способ сделать сокет неблокируемым */
if (fcntl(fd, F_SETFL, O_NONBLOCK) < 0)
    err_sys("F_SETFL error");
```

Хотя при этом и устанавливается флаг отключения блокировки, также снимаются все остальные флаги состояния файла. Единственный корректный способ установить один из этих флагов состояния файла —

получить текущие флаги, с помощью операции логического ИЛИ добавить новый флаг, а затем установить флаги.

Следующий код сбрасывает флаг отключения блокировки в предположении, что переменная `flags` была задана с помощью вызова функции `fcntl`, показанного ранее:

```
flags &= ~O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

Сигналы `SIGIO` и `SIGURG` отличаются от других тем, что они генерируются для сокета, только если сокету был присвоен владелец с помощью команды `F_SETOWN`. Целое значение аргумента `arg` для команды `F_SETOWN` может быть либо положительным, задающим идентификатор процесса, получающего сигнал, либо отрицательным, абсолютное значение которого — это идентификатор группы процессов, получающей сигнал. Команда `F_GETOWN` возвращает владельца сокета, так как возвращаемое значение функции `fcntl` — либо идентификатор процесса (положительное возвращаемое значение), либо идентификатор группы процессов (отрицательное значение, отличное от `-1`). Разница между заданием процесса и группы процессов, получающих сигнал, в том, что в первом случае сигнал будет получен только одиночным процессом, тогда как во втором случае его получают все процессы в группе.

Когда создается новый сокет с помощью функции `socket`, у него нет владельца. Сокет, создаваемый из прослушиваемого сокета, наследует от него принадлежность владельцу (как и многие другие параметры сокетов [128, с. 462–463].

7.12. Резюме

Параметры сокетов лежат в широком диапазоне от очень общих (`SO_ERROR`) до очень специфических (параметры заголовка IP). Наиболее общеупотребительные параметры сокетов, которые нам могут встретиться, — это `SO_KEEPALIVE`, `SO_RCVBUF`, `SO_SNDBUF` и `SO_REUSEADDR`. Последний должен всегда задаваться для сервера TCP до того, как сервер вызовет функцию `bind` (см. листинг 11.6). Параметр `SO_BROADCAST` и десять параметров сокетов многоадресной передачи предназначены только для приложений, передающих соответственно широковещательные или многоадресные сообщения.

Параметр сокета `SO_KEEPALIVE` устанавливается многими серверами TCP и автоматически закрывает наполовину открытое соединение. Замечательное свойство этого параметра в том, что он обрабатывается на уровне TCP, не требуя на уровне приложения наличия таймера, измеряющего период отсутствия активности. Однако недостаток этого параметра в том, что он не видит разницы между выходом собеседника из строя и временной потерей соединения с ним. SCTP предоставляет 17 параметров сокетов, с помощью которых приложение может управлять транспортным уровнем. `SCTP_NODELAY` и `SCTP_MAXSEG` аналогичны `TCP_NODELAY` и `TCP_MAXSEG`, и выполняют схожие функции. Остальные 17 параметров позволяют приложению более точно контролировать поведение стека SCTP. Большинство этих параметров будет рассмотрено в главе 23.

Параметр сокета `SO_LINGER` расширяет наши возможности в отношении контроля над функцией `close` — мы можем отложить ее завершение на некоторое время. Кроме того, этот параметр позволяет нам отправить сегмент RST вместо обычной последовательности из четырех пакетов, завершающих соединение TCP. Следует соблюдать осторожность при отправке сегментов RST, поскольку в этом случае не наступает состояние TCP `TIME_WAIT`. Бывает, что этот параметр сокета не обеспечивает необходимой нам информации, и тогда требуется реализовать подтверждение на уровне приложения.

У каждого сокета TCP имеется буфер отправки и буфер приема, а у каждого сокета UDP есть буфер приема. Параметры сокета `SO_SNDBUF` и `SO_RCVBUF` позволяют нам изменять размеры этих буферов. Основное применение эти функции находят при передаче большого количества данных по каналам с повышенной пропускной способностью, которые представляют собой соединения TCP либо с широкой полосой пропускания, либо с большой задержкой, часто с использованием расширений из RFC 1323. Сокеты UDP, наоборот, могут стремиться увеличить размер приемного буфера, чтобы позволить ядру установить в очередь больше дейтаграмм, если приложение занято.

Упражнения

1. Напишите программу, которая выводит заданные по умолчанию размеры буферов отправки и приема TCP, UDP и SCTP, и запустите ее в системе, к которой у вас имеется доступ.

2. Измените листинг 1.1 следующим образом. Перед вызовом функции `connect` вызовите функцию `getsockopt`, чтобы получить размер приемного буфера сокета и MSS. Выведите оба значения. После успешного завершения функции извлеките значения тех же двух параметров сокета и выведите их. Изменились ли значения? Почему? Запустите программу, соединяющуюся с сервером в вашей локальной сети, и программу, соединяющуюся с сервером в удаленной сети. Изменяется ли MSS? Почему? Запустите также программу на разных узлах, к которым у вас есть доступ.

3. Запустите наш сервер TCP, приведенный в листингах 5.1 и 5.2, и наш клиент из листингов 5.3 и 5.4. Измените функцию `main` клиента, чтобы установить параметр сокета `SO_LINGER` перед вызовом функции `exit`, задав `l_onoff` равным 1, а `l_linger` — равным 0. Запустите сервер, а затем запустите клиент. Введите строку или две на стороне клиента для проверки работоспособности, а затем завершите работу клиента, введя символ конца файла. Что происходит? После завершения работы клиента запустите программу `netstat` на узле клиента и посмотрите, проходит ли сокет через состояние `TIME_WAIT`.

4. Будем считать, что два клиента TCP запускаются одновременно. Оба устанавливают параметр сокета `SO_REUSEADDR` и затем с помощью функции `bind` связываются с одним и тем же локальным IP-адресом и одним и тем же локальным портом (допустим, 1500). Но один из клиентов соединяется с помощью функции `connect` с адресом 198.69.10.2, порт 7000, а второй — с адресом 198.69.10.2 (тот же IP-адрес собеседника), порт 8000. Опишите возникающую ситуацию гонок.

5. Получите исходный код для примеров в этой книге (см. предисловие) и откомпилируйте программу `sock` (см. раздел B.3). Сначала классифицируйте свой узел как узел, не поддерживающий многоадресную передачу, затем — как поддерживающий многоадресную передачу, но не поддерживающий параметр `SO_REUSEPORT`, и наконец, как узел, поддерживающий многоадресную передачу с предоставлением параметра `SO_REUSEPORT`. Попытайтесь запустить несколько экземпляров программы `sock` в качестве сервера TCP (параметр `-s` командной строки) на одном и том же порте, связывая универсальный адрес, один из адресов интерфейсов вашего узла и адрес закольцовки (loopback address). Нужно ли вам задавать параметр `SO_REUSEADDR` (параметр `-A` командной строки)? Используйте программу `netstat` для просмотра прослушиваемых сокетов.

6. Продолжайте предыдущий пример, но запустите сервер UDP (параметр `-u` командной строки) и попытайтесь запустить два экземпляра, связанные с одними и теми же локальным IP-адресом и портом. Если ваша реализация поддерживает параметр `SO_REUSEPORT`, попытайтесь использовать ее (параметр `-T` командной строки).

7. Многие версии утилиты `ping` имеют флаг `-d`, задающий параметр сокета `SO_DEBUG`. В чем его назначение?

8. Продолжая пример в конце нашего обсуждения параметра сокета `TCP_NODELAY`, предположим, что клиент выполняет две операции записи с помощью функции `write`: первую для 4 байт данных и вторую для 396 байт. Также будем считать, что время задержки ACK — 100 мс, период RTT между клиентом и сервером равен 100 мс, а время обработки сервером каждого клиентского запроса — 50 мс. Нарисуйте временную диаграмму, показывающую взаимодействие алгоритма Нагла с задержанными сегментами ACK.

9. Снова выполните предыдущее упражнение, считая, что установлен параметр сокета `TCP_NODELAY`.

10. Снова выполните упражнение 8, считая, что процесс вызывает функцию `writev` один раз для обоих буферов (4-байтового и 396-байтового).

11. Прочтите RFC 1122 [10], чтобы определить рекомендуемый интервал для задержанных сегментов ACK.

12. В какой из версий наш сервер тратит больше времени — в листинге 5.1 или 5.2? Что происходит, если сервер устанавливает параметр сокета `SO_KEEPALIVE`, через соединение не происходит обмена данными, узел клиента выходит из строя и не перезагружается?

13. В какой из версий наш клиент тратит больше времени — в листинге 5.3 или 5.4? Что происходит, если клиент устанавливает параметр сокета `SO_KEEPALIVE`, через соединение не происходит обмена данными и узел сервера выходит из строя и не перезагружается?

14. В какой из версий наш клиент тратит больше времени — в листинге 5.3 или 6.2? Что происходит, если клиент устанавливает параметр сокета `SO_KEEPALIVE`, через соединение не происходит обмена данными и узел сервера выходит из строя и не перезагружается?

15. Будем считать, что и клиент, и сервер устанавливают параметр сокета `SO_KEEPALIVE`. Между собеседниками поддерживается соединение, но через это соединение не происходит обмена данными между приложениями. Когда проходят условленные 2 ч и требуется проверить наличие связи, сколькими сегментами TCP обмениваются собеседники?

16. Почти все реализации определяют константу `SO_ACCEPTCONN` в заголовочном файле `<sys/socket.h>`, но мы не описывали этот параметр. Прочтите [69], чтобы понять, зачем этот параметр существует.

Глава 8

Основные сведения о сокетах UDP

8.1. Введение

Приложения, использующие TCP и UDP, фундаментально отличаются друг от друга, потому что UDP является ненадежным протоколом дейтаграмм, не ориентированным на установление соединения, и этим принципиально непохож на ориентированный на установление соединения и надежную передачу потока байтов TCP. Тем не менее есть случаи, когда имеет смысл использовать UDP вместо TCP. Подобные случаи мы рассматриваем в разделе 22.4. Некоторые популярные приложения построены с использованием UDP, например DNS (Domain Name System — система доменных имен), NFS (сетевая файловая система — Network File System) и SNMP (Simple Network Management Protocol — простой протокол управления сетью).

На рис. 8.1 показаны вызовы функций для типичной схемы клиент-сервер UDP. Клиент не устанавливает соединения с сервером. Вместо этого клиент лишь отправляет серверу дейтаграмму, используя функцию `sendto` (она описывается в следующем разделе), которой нужно задать адрес получателя (сервера) в качестве аргумента. Аналогично, сервер не устанавливает соединения с клиентом. Вместо этого сервер лишь вызывает функцию `recvfrom`, которая ждет, когда придут данные от какого-либо клиента. Функция `recvfrom` возвращает адрес клиента (для данного протокола) вместе с дейтаграммой, и таким образом сервер может отправить ответ именно тому клиенту, который прислал дейтаграмму.

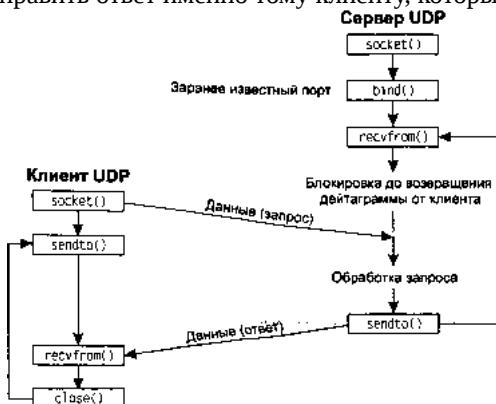


Рис. 8.1. Функции сокета для модели клиент-сервер UDP

Рисунок 8.1 иллюстрирует временную диаграмму типичного сценария обмена UDP-дейтаграммами между клиентом и сервером. Мы можем сравнить этот пример с типичным обменом по протоколу TCP, изображенным на рис. 4.1.

В этой главе мы опишем новые функции, применяемые с сокетами UDP, — `recvfrom` и `sendto`, и переделаем нашу модель клиент-сервер для применения UDP. Кроме того, мы рассмотрим использование функции `connect` с сокетом UDP и концепцию асинхронных ошибок.

8.2. Функции `recvfrom` и `sendto`

Эти две функции аналогичны стандартным функциям `read` и `write`, но требуют трех дополнительных аргументов.

```
#include <sys/socket.h>
```

```
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
    struct sockaddr *from, socklen_t *addrlen);
ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
    const struct sockaddr *to, socklen_t addrlen);
```

Обе функции возвращают количество записанных или прочитанных байтов в случае успешного выполнения, -1 в случае ошибки

Первые три аргумента, sockfd, buff и nbytes, идентичны первым трем аргументам функций read и write: дескриптор, указатель на буфер, из которого производится чтение или в который происходит запись, и число байтов для чтения или записи.

Мы расскажем об аргументе flags в главе 14, где мы рассматриваем функции recv, send, recvmsg и sendmsg, поскольку сейчас в нашем простом примере они не нужны. Пока мы всегда будем устанавливать аргумент flags в нуль.

Аргумент to для функции sendto — это структура адреса сокета, содержащая адрес протокола (например, IP-адрес и номер порта) адресата. Размер этой структуры адреса сокета задается аргументом addrlen. Функция recvfrom заполняет структуру адреса сокета, на которую указывает аргумент from, записывая в нее протокольный адрес отправителя дейтаграммы. Число байтов, хранящихся в структуре адреса сокета, также возвращается вызывающему процессу в целом числе, на которое указывает аргумент addrlen. Обратите внимание, что последний аргумент функции sendto является целочисленным значением, в то время как последний аргумент функции recvfrom — это указатель на целое значение (аргумент типа «значение-результат»).

Последние два аргумента функции recvfrom аналогичны двум последним аргументам функции accept: содержащие структуры адреса сокета по завершении сообщают нам, кто отправил дейтаграмму (в случае UDP) или кто инициировал соединение (в случае TCP). Последние два аргумента функции sendto аналогичны двум последним аргументам функции connect: мы заполняем структуру адреса сокета протокольным адресом получателя дейтаграммы (в случае UDP) или адресом узла, с которым будет устанавливаться соединение (в случае TCP).

Обе функции возвращают в качестве значения функции длину данных, которые были прочитаны или записаны. При типичном использовании функции recvfrom с протоколом дейтаграмм возвращаемое значение — это объем пользовательских данных в полученной дейтаграмме.

Дейтаграмма может иметь нулевую длину. В случае UDP при этом возвращается дейтаграмма IP, содержащая заголовок IP (обычно 20 байт для IPv4 или 40 байт для IPv6), 8-байтовый заголовок UDP и никаких данных. Это также означает, что возвращаемое из функции recvfrom нулевое значение вполне приемлемо для протокола дейтаграмм: оно не является признаком того, что собеседник закрыл соединение, как это происходит при возвращении нулевого значения из функции read на сокете TCP. Поскольку протокол UDP не ориентирован на установление соединения, то в нем и не существует такого события, как закрытие соединения.

Если аргумент from функции recvfrom является пустым указателем, то соответствующий аргумент длины (addrlen) также должен быть пустым указателем, и это означает, что нас не интересует адрес отправителя данных.

И функция recvfrom, и функция sendto могут использоваться с TCP, хотя обычно в этом нет необходимости.

8.3. Эхо-сервер UDP: функция main

Теперь мы переделаем нашу простую модель клиент-сервер из главы 5, используя UDP. Диаграмма вызовов функций в программах наших клиента и сервера UDP показана на рис. 8.1. На рис. 8.2 представлены используемые функции. В листинге 8.1^[1] показана функция сервера main.

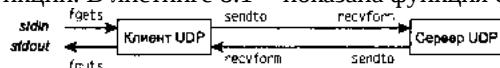


Рис. 8.2. Простая модель клиент-сервер, использующая UDP

Листинг 8.1. Эхо-сервер UDP

```
//udpciserv/udpserv01.c
1 #include "unp.h"

2
3 intmain(int argc, char **argv)
4 {
5     int sockfd;
6     struct sockaddr_in servaddr, cliaddr;

7     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
```

```

8 bzero(&servaddr, sizeof(servaddr));
9 servaddr.sin_family = AF_INET;
10 servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
11 servaddr.sin_port = htons(SERV_PORT);

12 Bind(sockfd, (SA*)&servaddr, sizeof(servaddr));

13 dg_echo(sockfd, (SA*)&cliaddr, sizeof(cliaddr));
14 }

```

Создание сокета UDP, связывание с заранее известным портом при помощи функции bind

7-12 Мы создаем сокет UDP, задавая в качестве второго аргумента функции socket значение SOCK_DGRAM (сокет дейтаграмм в протоколе IPv4). Как и в примере сервера TCP, адрес IPv4 для функции bind задается как INADDR_ANY, а заранее известный номер порта сервера — это константа SERV_PORT из заголовка unp.h.

13 Затем вызывается функция dg_echo для обработки клиентского запроса сервером.

8.4. Эхо-сервер UDP: функция dg_echo

В листинге 8.2 показана функция dg_echo.

Листинг 8.2. Функция dg_echo: отражение строк на сокете дейтаграмм

```

//lib/dg_echo.c
1 #include "unp.h"

2 void
3 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
4 {
5     int n;
6     socklen_t len;
7     char mesg[MAXLINE];

8     for (;;) {
9         len = clilen;
10        n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

11        Sendto(sockfd, mesg, n, 0, pcliaddr, len);
12    }
13 }

```

Чтение дейтаграммы, отражение отправителю

8-12 Эта функция является простым циклом, в котором очередная дейтаграмма, приходящая на порт сервера, читается функцией recvfrom и с помощью функции sendto отправляется обратно.

Несмотря на простоту этой функции, нужно учесть ряд важных деталей. Во-первых, эта функция никогда не завершается. Поскольку UDP — это протокол, не ориентированный на установление соединения, в нем не существует никаких аналогов признака конца файла, используемого в TCP.

Во-вторых, эта функция позволяет *создать последовательный сервер*, а не параллельный, который мы получали в случае TCP. Поскольку нет вызова функции fork, один процесс сервера выполняет обработку всех клиентов. В общем случае большинство серверов TCP являются параллельными, а большинство серверов UDP — последовательными.

Для сокета на уровне UDP происходит неявная буферизация дейтаграмм в виде очереди. Действительно, у каждого сокета UDP имеется буфер приема, и каждая дейтаграмма, приходящая на этот

сокет, помещается в его буфер приема. Когда процесс вызывает функцию `recvfrom`, очередная дейтаграмма из буфера возвращается процессу в порядке FIFO (First In, First Out — первым пришел, первым обслужен). Таким образом, если множество дейтаграмм приходит на сокет до того, как процесс может прочитать данные, уже установленные в очередь для сокета, то приходящие дейтаграммы просто добавляются в буфер приема сокета. Но этот буфер имеет ограниченный размер. Мы обсуждали этот размер и способы его увеличения с помощью параметра сокета `SO_RCVBUF` в разделе 7.5.

На рис. 8.3 показано обобщение нашей модели TCP клиент-сервер из главы 5, когда два клиента устанавливают соединения с сервером.

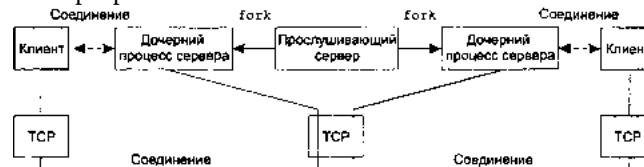


Рис. 8.3. Обобщение модели TCP клиент-сервер с двумя клиентами

Здесь имеется два присоединенных сокета, и каждый из присоединенных сокетов на узле сервера имеет свой собственный буфер приема. На рис. 8.4 показан случай, когда два клиента отправляют дейтаграммы серверу UDP.

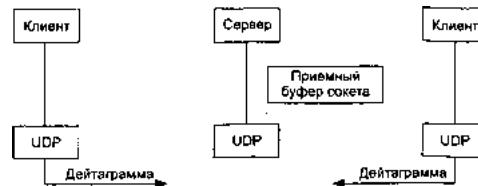


Рис. 8.4. Обобщение модели UDP клиент-сервер с двумя клиентами

Существует только один процесс сервера, и у него имеется один сокет, на который сервер получает все приходящие дейтаграммы и с которого отправляет все ответы. У этого сокета имеется буфер приема, в который помещаются все приходящие дейтаграммы.

Функция `main` в листинге 8.1 является зависящей от протокола (она создает сокет семейства `AF_INET`, а затем выделяет и инициализирует структуру адреса сокета IPv4), но функция `dg_echo` от протокола не зависит. Причина, по которой функция `dg_echo` не зависит от протокола, заключается в том, что вызывающий процесс (в нашем случае функция `main`) должен разместить в памяти структуру адреса сокета корректного размера, и указатель на эту структуру вместе с ее размером передаются в качестве аргументов функции `dg_echo`. Функция `dg_echo` никогда не углубляется в эту структуру: она просто передает указатель на нее функциям `recvfrom` и `sendto`. Функция `recvfrom` заполняет эту структуру, вписывая в нее IP-адрес и номер порта клиента, и поскольку тот же указатель (`pcliaddr`) затем передается функции `sendto` в качестве адреса получателя, таким образом дейтаграмма отражается обратно клиенту, отправившему дейтаграмму.

8.5. Эхо-клиент UDP: функция `main`

Функция `main` клиента UDP показана в листинге 8.3.

Листинг 8.3. Эхо-клиент UDP

```

//udpcliserv/udpcliserv01.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;
6     struct sockaddr_in servaddr;

7     if (argc != 2)
8         err_quit("usage: udpccli <Ipaddress>");

9     bzero(&servaddr, sizeof(servaddr));
10    servaddr.sin_family = AF_INET;
  
```

```

11 servaddr.sin_port = htons(SERV_PORT);
12 Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

13 sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

14 dg_cli(stdin, sockfd, (SA*)&servaddr, sizeof(servaddr));

15 exit(0);
16 }

```

Заполнение структуры адреса сокета адресом сервера

9-12 Структура адреса сокета IPv4 заполняется IP-адресом и номером порта сервера. Эта структура будет передана функции dg_cli. Она определяет, куда отправлять дейтаграммы.

13-14 Создается сокет UDP и вызывается функция dg_cli.

8.6. Эхо-клиент UDP: функция dg_cli

В листинге 8.4 показана функция dg_cli, которая выполняет большую часть работы на стороне клиента.

Листинг 8.4. Функция dg_cli: цикл обработки клиента

```

//lib/dg_cli.c
1 #include "unp.h"

2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int n;
6     char sendline[MAXLINE], recvline[MAXLINE + 1];

7     while (Fgets(sendline, MAXLINE, fp) != NULL) {

8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

9         n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

10        recvline[n] = 0; /* завершающий нуль */
11        Fputs(recvline, stdout);
12    }
13 }

```

7-12 В цикле обработки на стороне клиента имеется четыре шага: чтение строки из стандартного потока ввода при помощи функции fgets, отправка строки серверу с помощью функции sendto, чтение отраженного ответа сервера с помощью функции recvfrom и помещение отраженной строки в стандартный поток вывода с помощью функции fputs.

Наш клиент не запрашивал у ядра присваивания динамически назначаемого порта своему сокету (тогда как для клиента TCP это имело место при вызове функции connect). В случае сокета UDP при первом вызове функции sendto ядро выбирает динамически назначаемый порт, если с этим сокетом еще не был связан никакой локальный порт. Как и в случае TCP, клиент может вызывать функцию bind явно, но это делается редко.

Обратите внимание, что при вызове функции recvfrom в качестве пятого и шестого аргументов задаются пустые указатели. Таким образом мы сообщаем ядру, что мы не заинтересованы в том, чтобы знать, кто отправил ответ. Существует риск, что любой процесс, находящийся как на том же узле, так и на любом другом, может отправить на IP-адрес и порт клиента дейтаграмму, которая будет прочитана клиентом, предполагающим, что это ответ сервера. Эту ситуацию мы рассмотрим в разделе 8.8.

Как и в случае функции сервера `dg_echo`, функция клиента `dg_cli` является не зависящей от протокола, но функция `main` клиента зависит от протокола. Функция `main` размещает в памяти и инициализирует структуру адреса сокета, относящегося к определенному типу протокола, а затем передает функции `dg_cli` указатель на структуру вместе с ее размером.

8.7. Потерянные дейтаграммы

Клиент и сервер UDP в нашем примере являются ненадежными. Если дейтаграмма клиента потеряна (допустим, она проигнорирована неким маршрутизатором между клиентом и сервером), клиент навсегда заблокируется в своем вызове функции `recvfrom` внутри функции `dg_cli`, ожидая от сервера ответа, который никогда не придет. Аналогично, если дейтаграмма клиента приходит к серверу, но ответ сервера потерян, клиент навсегда заблокируется в своем вызове функции `recvfrom`. Единственный способ предотвратить эту ситуацию — поместить тайм-аут в клиентский вызов функции `recvfrom`. Мы рассмотрим это в разделе 14.2.

Простое помещение тайм-аута в вызов функции `recvfrom` — еще не полное решение. Например, если заданное время ожидания истекло, а ответ не получен, мы не можем сказать точно, в чем дело — или наша дейтаграмма не дошла до сервера, или же ответ сервера не пришел обратно. Если бы запрос клиента содержал требование типа «перевести определенное количество денег со счета А на счет Б» (в отличие от случая с нашим простым эхо-сервером), то тогда между потерей запроса и потерей ответа существовала бы большая разница. Более подробно о добавлении надежности в модель клиент-сервер UDP мы расскажем в разделе 22.5.

8.8. Проверка полученного ответа

В конце раздела 8.6 мы упомянули, что любой процесс, который знает номер динамически назначаемого порта клиента, может отправлять дейтаграммы нашему клиенту, и они будут перемешаны с нормальными ответами сервера. Все, что мы можем сделать, — это изменить вызов функции `recvfrom`, представленный в листинге 8.4, так, чтобы она возвращала IP-адрес и порт отправителя ответа, и игнорировать любые дейтаграммы, приходящие не от того сервера, которому мы отправляем дейтаграмму. Однако здесь есть несколько ловушек, как мы дальше увидим.

Сначала мы изменяем функцию клиента `main` (см. листинг 8.3) для работы со стандартным эхо-сервером (см. табл. 2.1). Мы просто заменяем присваивание

```
servaddr.sin_port = htons(SERV_PORT);  
присваиванием  
servaddr.sin_port = htons(7);
```

Теперь мы можем использовать с нашим клиентом любой узел, на котором работает стандартный эхо-сервер.

Затем мы переписываем функцию `dg_cli`, с тем чтобы она размещала в памяти другую структуру адреса сокета для хранения структуры, возвращаемой функцией `recvfrom`. Мы показываем ее в листинге 8.5.

Листинг 8.5. Версия функции `dg_cli`, проверяющая возвращаемый адрес сокета

```
//udpcliserv/dgcliaddr.c  
1 #include "unp.h"  
  
2 void  
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)  
4 {  
5     int n;  
6     char sendline[MAXLINE], recvline[MAXLINE + 1];  
7     socklen_t len;  
8     struct sockaddr *preply_addr;  
  
9     preply_addr = Malloc(servlen);  
  
10    while (Fgets(sendline, MAXLINE, fp) != NULL) {
```

```

11     Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

12     len = servlen;
13     n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
14     if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
15         printf("reply from %s (ignored)\n",
16         continue;
17     }
18     recvline[n] = 0; /* завершающий нуль */
19     Fputs(recvline, stdout);
20 }
21 }
```

Размещение другой структуры адреса сокета в памяти

9 Мы размещаем в памяти другую структуру адреса сокета при помощи функции `malloc`. Обратите внимание, что функция `dg_cli` все еще является не зависящей от протокола. Поскольку нам не важно, с каким типом структуры адреса сокета мы имеем дело, мы используем в вызове функции `malloc` только ее размер.

Сравнение возвращаемых адресов

12-13 В вызове функции `recvfrom` мы сообщаем ядру, что нужно возвратить адрес отправителя дейтаграммы. Сначала мы сравниваем длину, возвращаемую функцией `recvfrom` в аргументе типа «значение-результат», а затем сравниваем сами структуры адреса сокета при помощи функции `memcmp`.

Новая версия нашего клиента работает замечательно, если сервер находится на узле с одним единственным IP-адресом. Но эта программа может не сработать, если сервер имеет несколько сетевых интерфейсов (multihommed server). Запускаем эту программу, обращаясь к узлу `freebsd4`, у которого имеется два интерфейса и два IP-адреса:

```

macosx % host freebsd4
freebsd4.unpbook.com has address 172.24.37.94
freebsd4.unpbook.com has address 135.197.17.100
macosx % udpcli02 135.197.17.100
hello
reply from 172.24.37.94:7 (ignored)
goodbye
reply from 172.24.37.94:7 (ignored)
```

По рис. 1.7 видно, что мы задали IP-адрес из другой подсети. Обычно это допустимо. Большинство реализаций IP принимают приходящую IP-дейтаграмму, предназначенную для любого из IP-адресов узла, независимо от интерфейса, на который она приходит [128, с. 217-219]. Документ RFC 1122 [10] называет это моделью системы с гибкой привязкой (weak end system model). Если система должна реализовать то, что в этом документе называется моделью системы с жесткой привязкой (strong end system model), она принимает приходящую дейтаграмму, только если дейтаграмма приходит на тот интерфейс, которому она адресована.

IP-адрес, возвращаемый функцией `recvfrom` (IP-адрес отправителя дейтаграммы UDP), не является IP-адресом, на который мы посыпали дейтаграмму. Когда сервер отправляет свой ответ, IP-адрес получателя — это адрес 172.24.37.94. Функция маршрутизации внутри ядра на узле `freebsd4` выбирает адрес 172.24.37.94 в качестве исходящего интерфейса. Поскольку сервер не связал IP-адрес со своим сокетом (сервер связал со своим сокетом универсальный адрес, что мы можем проверить, запустив программу `netstat` на узле `freebsd4`), ядро выбирает адрес отправителя дейтаграммы IP. Этим адресом становится первичный IP-адрес исходящего интерфейса [128, с. 232-233]. Если мы отправляем дейтаграмму не на первичный IP-адрес интерфейса (то есть на альтернативное имя, *псевдоним*), то наша проверка, показанная в листинге 8.5, также окажется неудачной.

Одним из решений будет проверка клиентом доменного имени отвечающего узла вместо его IP-адреса. Для этого имя сервера ищется в DNS (см. главу 11) на основе IP-адреса, возвращаемого функцией `recvfrom`. Другое решение — сделать так, чтобы сервер UDP создал по одному сокету для каждого IP-адреса, сконфигурированного на узле, связал с помощью функции `bind` этот IP-адрес с сокетом, вызвал функцию `select` для каждого из всех этих сокетов (ожидая, когда какой-либо из них станет готов для чтения), а затем ответил с сокета, готового для чтения. Поскольку сокет, используемый для ответа, связан с IP-адресом, который является адресом получателя клиентского запроса (иначе дейтаграмма не была бы доставлена на сокет), мы можем быть уверены, что адреса отправителя ответа и получателя запроса совпадают. Мы показываем эти примеры в разделе 22.6.

ПРИМЕЧАНИЕ

В системе Solaris с несколькими сетевыми интерфейсами IP-адрес отправителя ответа сервера — это IP-адрес получателя клиентского запроса. Сценарий, описанный в данном разделе, относится к реализациям, происходящим от Беркли, которые выбирают IP-адрес отправителя, основываясь на исходящем интерфейсе.

8.9. Запуск клиента без запуска сервера

Следующий сценарий, который мы рассмотрим, — это запуск клиента без запуска сервера. Если мы сделаем так и введем одну строку на стороне клиента, ничего не будет происходить. Клиент навсегда блокируется в своем вызове функции `recvfrom`, ожидая ответа сервера, который никогда не придет. Но в данном примере это не имеет значения, поскольку сейчас мы стремимся глубже понять протоколы и выяснить, что происходит с нашим сетевым приложением.

Сначала мы запускаем программу `tcpdump` на узле `macosx`, а затем — клиент на том же узле, задав в качестве узла сервера `freebsd4`. Потом мы вводим одну строку, но эта строка не отражается сервером.

```
macosx % udpcli01 172.24.37.94
hello, world мы вводим эту строку,
но ничего не получаем в ответ
```

В листинге 8.6 показан вывод программы `tcpdump`.

Листинг 8.6. Вывод программы `tcpdump`, когда процесс сервера не запускается на узле сервера

```
01 0.0          arp who-has freebsd4 tell macosx
02 0.003576 (0.0036) arp reply freebsd4 is-at 0:40:5:42:d6:de
```

```
03 0.003601 (0.0000) macosx.51139 > freebsd4.9877: udp 13
04 0.009781 (0.0062) freebsd4 > macosx: icmp: freebsd4 udp port 9877 unreachable
```

В первую очередь мы замечаем, что запрос и ответ ARP получены до того, как узел клиента смог отправить дейтаграмму UDP узлу сервера. (Мы оставили этот обмен в выводе программы, чтобы еще раз подчеркнуть, что до отправки IP-дейтаграммы всегда следует отправка запроса и получение ответа по протоколу ARP.)

В строке 3 мы видим, что дейтаграмма клиента отправлена, но узел сервера отвечает в строке 4 сообщением ICMP о недоступности порта. (Длина 13 включает 12 символов плюс символ новой строки.) Однако эта ошибка ICMP не возвращается клиентскому процессу по причинам, которые мы кратко перечислим чуть ниже. Вместо этого клиент навсегда блокируется в вызове функции `recvfrom` в листинге 8.4. Мы также отмечаем, что в ICMPv6 имеется ошибка «Порт недоступен», аналогичная ошибке ICMPv4 (см. табл. A.5 и A.6), поэтому результаты, представленные здесь, аналогичны результатам для IPv4.

Эта ошибка ICMP является *асинхронной ошибкой*. Ошибка была вызвана функцией `sendto`, но функция `sendto` завершилась нормально. Вспомните из раздела 2.9, что нормальное возвращение из операции вывода UDP означает только то, что дейтаграмма была добавлена к очереди вывода канального уровня. Ошибка ICMP не возвращается, пока не пройдет определенное количество времени (4 мс для листинга 8.6), поэтому она и называется асинхронной.

Основное правило состоит в том, что асинхронные ошибки не возвращаются для сокета UDP, если сокет не был присоединен. Мы показываем, как вызвать функцию `connect` для сокета UDP, в разделе 8.11. Не все понимают, почему было принято это решение, когда сокеты были впервые реализованы. (Соображения о реализациях обсуждаются на с. 748–749 [128].) Рассмотрим клиент UDP, последовательно

отправляющий три дейтаграммы трем различным серверам (то есть на три различных IP-адреса) через один сокет UDP. Клиент входит в цикл, вызывающий функцию `recvfrom` для чтения ответов. Две дейтаграммы доставляются корректно (то есть сервер был запущен на двух из трех узлов), но на третьем узле не был запущен сервер, и третий узел отвечает сообщением ICMP о недоступности порта. Это сообщение об ошибке ICMP содержит IP-заголовок и UDP-заголовок дейтаграммы, вызвавшей ошибку. (Сообщения об ошибках ICMPv4 и ICMPv6 всегда содержат заголовок IP и весь заголовок UDP или часть заголовка TCP, чтобы дать возможность получателю сообщения определить, какой сокет вызвал ошибку. Это показано на рис. 28.5 и 28.6.) Клиент, отправивший три дейтаграммы, должен знать получателя дейтаграммы, вызвавшей ошибку, чтобы точно определить, какая из трех дейтаграмм вызвала ошибку. Но как ядро может сообщить эту информацию процессу? Единственное, что может возвратить функция `recvfrom`, — это значение переменной `errno`. Но функция `recvfrom` не может вернуть в ошибке IP-адрес и номер порта получателя UDP-дейтаграммы. Следовательно, было принято решение, что эти асинхронные ошибки возвращаются процессу, только если процесс присоединил сокет UDP лишь к одному определенному собеседнику.

ПРИМЕЧАНИЕ

Linux возвращает большинство ошибок ICMP о недоступности порта даже для неприсоединенного сокета, если не включен параметр сокета `SO_DSBCOMPAT`. Возвращаются все ошибки о недоступности получателя, показанные в табл. А.5, за исключением ошибок с кодами 0, 1, 4, 5, 11 и 12.

Мы вернемся к проблеме асинхронных ошибок с сокетами UDP в разделе 28.7 и покажем простой способ получения этих ошибок на неприсоединенном сокете при помощи нашего собственного демона.

8.10. Итоговый пример клиент-сервера UDP

На рис. 8.5 крупными черными точками показаны четыре значения, которые должны быть заданы или выбраны, когда клиент отправляет дейтаграмму UDP.

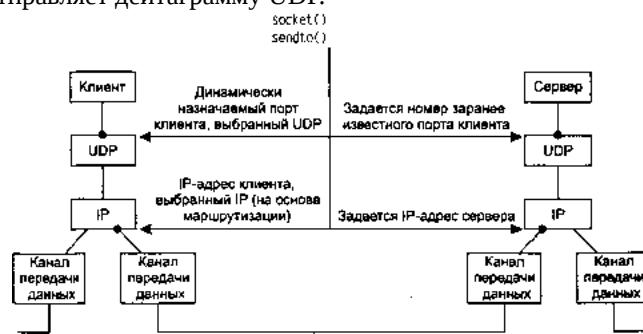


Рис. 8.5. Обобщение модели клиент-сервер UDP с точки зрения клиента

Клиент должен задать IP-адрес сервера и номер порта для вызова функции `sendto`. Обычно клиентский IP-адрес и номер порта автоматически выбираются ядром, хотя мы отмечали, что клиент может вызвать функцию `bind`. Мы также отмечали, что если эти два значения выбираются для клиента ядром, то динамически назначаемый порт клиента выбирается один раз — при первом вызове функции `sendto`, и более никогда не изменяется. Однако IP-адрес клиента может меняться для каждой дейтаграммы UDP, которую отправляет клиент, если предположить, что клиент не связывает с сокетом определенный IP-адрес при помощи функции `bind`. Причину объясняет рис. 8.5: если узел клиента имеет несколько сетевых интерфейсов, клиент может переключаться между ними (на рис. 8.5 один адрес относится к канальному уровню, изображеному слева, другой — к изображенному справа). В худшем варианте этого сценария IP-адрес клиента, выбираемый ядром на основе исходящего канального уровня, будет меняться для каждой дейтаграммы.

Что произойдет, если клиент с помощью функции `bind` свяжет IP-адрес со своим сокетом, но ядро решит, что исходящая дейтаграмма должна быть отправлена с какого-то другого канального уровня? В этом

случае дейтаграмма IP будет содержать IP-адрес отправителя, отличный от IP-адреса исходящего канального уровня (см. упражнение 8.6).

На рис. 8.6 представлены те же четыре значения, но с точки зрения сервера.

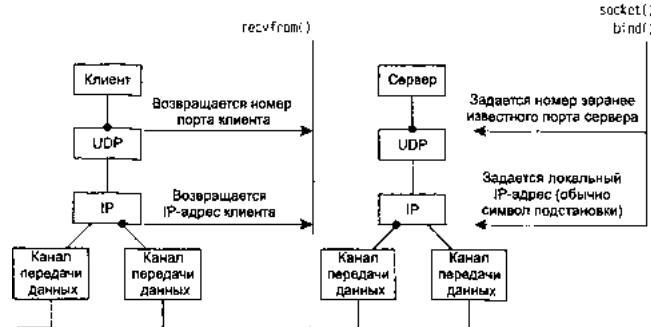


Рис. 8.6. Обобщение модели клиент-сервер UDP с точки зрения сервера

Сервер может узнать по крайней мере четыре параметра для каждой полученной дейтаграммы: IP-адрес отправителя, IP-адрес получателя, номер порта отправителя и номер порта получателя. Вызовы, возвращающие эти сведения серверам TCP и UDP, приведены в табл. 8.1.

Таблица 8.1. Информация, доступная серверу из приходящей дейтаграммы IP

IP-дейтаграмма клиента TCP-сервер UDP-сервер

IP-адрес отправителя	accept	recvfrom
Номер порта отправителя	accept	recvfrom
IP-адрес получателя	getsockname	recvmsg
Номер порта получателя	getsockname	getsockname

У сервера TCP всегда есть простой доступ ко всем четырем фрагментам информации для присоединенного сокета, и эти четыре значения остаются постоянными в течение всего времени жизни соединения. Однако в случае соединения UDP IP-адрес получателя можно получить только с помощью установки параметра сокета `IP_RECVSTAADDR` для IPv4 или `IPV6_PKTINFO` для IPv6 и последующего вызова функции `recvmsg` вместо функции `recvfrom`. Поскольку протокол UDP не ориентирован на установление соединения, IP-адрес получателя может меняться для каждой дейтаграммы, отправляемой серверу. Сервер UDP может также получать дейтаграммы, предназначенные для одного из широковещательных адресов узла или для адреса многоадресной передачи, что мы обсуждаем в главах 20 и 21. Мы покажем, как определить адрес получателя дейтаграммы UDP, в разделе 20.2, после того как опишем функцию `recvmsg`.

8.11. Функция connect для UDP

В конце разделе 8.9 мы упомянули, что асинхронные ошибки не возвращаются на сокете UDP, если сокет не был присоединен. На самом деле мы можем вызвать функцию `connect` для сокета UDP (см. раздел 4.3). Но это не приведет ни к чему похожему на соединение TCP: здесь не существует трехэтапного рукопожатия. Ядро просто проверяет, нет ли сведений о заведомой недоступности адресата, после чего записывает IP-адрес и номер порта собеседника, которые содержатся в структуре адреса сокета, передаваемой функции `connect`, и немедленно возвращает управление вызывающему процессу.

ПРИМЕЧАНИЕ

Перегрузка функции `connect` этой новой возможностью для сокетов UDP может внести путаницу. Если используется соглашение о том, что `sockname` — это адрес локального протокола, а `peername` — адрес удаленного протокола, то лучше бы эта функция называлась `setpeername`. Аналогично, функции `bind` больше подошло бы название `setsockname`.

С учетом этого необходимо понимать разницу между двумя видами сокетов UDP.

- **Неприсоединенный (unconnected) сокет UDP** — это сокет UDP, создаваемый по умолчанию.
- **Присоединенный (connected) сокет UDP** — результат вызова функции `connect` для сокета UDP.

Присоединенному сокету UDP свойственны три отличия от неприсоединенного сокета, который создается по умолчанию.

1. Мы больше не можем задавать IP-адрес получателя и порт для операции вывода. То есть мы используем вместо функции `sendto` функцию `write` или `send`. Все, что записывается в присоединенный сокет UDP, автоматически отправляется на адрес (например, IP-адрес и порт), заданный функцией `connect`.

ПРИМЕЧАНИЕ

Аналогично TCP, мы можем вызвать функцию `sendto` для присоединенного сокета UDP, но не можем задать адрес получателя. Пятый аргумент функции `sendto` (указатель на структуру адреса сокета) должен быть пустым указателем, а шестой аргумент (размер структуры адреса сокета) должен быть нулевым. В стандарте POSIX определено, что когда пятый аргумент является пустым указателем, шестой аргумент игнорируется.

2. Вместо функции `recvfrom` мы используем функцию `read` или `recv`. Единственные дейтаграммы, возвращаемые ядром для операции ввода через присоединенный сокет UDP, — это дейтаграммы, приходящие с адреса, заданного в функции `connect`. Дейтаграммы, предназначенные для адреса локального протокола присоединенного сокета UDP (например, IP-адрес и порт), но приходящие с адреса протокола, отличного от того, к которому сокет был присоединен с помощью функции `connect`, не передаются присоединенному сокету. Это ограничивает присоединенный сокет UDP, позволяя ему обмениваться дейтаграммами с одним и только одним собеседником.

ПРИМЕЧАНИЕ

Точнее, обмен дейтаграммами происходит только с одним IP-адресом, а не с одним собеседником, поскольку это может быть IP-адрес многоадресной передачи, представляющий, таким образом, группу собеседников.

3. Асинхронные ошибки возвращаются процессу только при операциях с присоединенным сокетом UDP. В результате, как мы уже говорили, неприсоединенный сокет UDP не получает никаких асинхронных ошибок.

В табл. 8.2 сводятся воедино свойства, перечисленные в первом пункте, применительно к 4.4BSD.

Таблица 8.2. Сокеты TCP и UDP: может ли быть задан адрес протокола получателя

Тип сокета	<code>write</code> или <code>send</code>	<code>sendto</code> , без указания получателя	<code>sendto</code> , с указанием получателя
Сокет TCP	Да	Да	EISCONN
Сокет UDP, присоединенный	Да	Да	EISCONN
Сокет UDP, неприсоединенный		EDESTADDRREQ	Да

ПРИМЕЧАНИЕ

POSIX определяет, что операция вывода, не задающая адрес получателя на неприсоединенном сокете UDP, должна возвращать ошибку ENOTCONN, а не EDESTADDRREQ.

Solaris 2.5 допускает функцию `sendto`, которая задает адрес получателя для присоединенного сокета UDP. POSIX определяет, что в такой ситуации должна возвращаться ошибка EISCONN.

На рис. 8.7 обобщается информация о присоединенном сокете UDP.

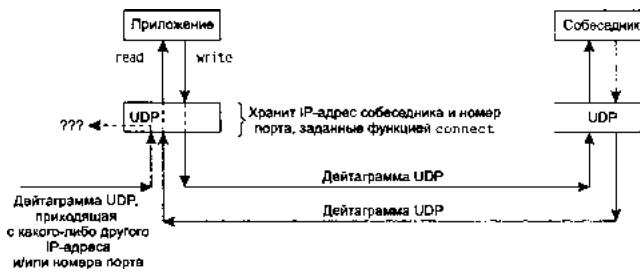


Рис. 8.7. Присоединенный сокет UDP

Приложение вызывает функцию `connect`, задавая IP-адрес и номер порта собеседника. Затем оно использует функции `read` и `write` для обмена данными с собеседником.

Дейтаграммы, приходящие с любого другого IP-адреса или порта (который мы обозначаем как «??» на рис. 8.7), не передаются на присоединенный сокет, поскольку либо IP-адрес, либо UDP-порт отправителя не совпадают с адресом протокола, с которым сокет соединяется с помощью функции `connect`. Эти дейтаграммы могут быть доставлены на какой-то другой сокет UDP на узле. Если нет другого совпадающего сокета для приходящей дейтаграммы, UDP проигнорирует ее и сгенерирует ICMP-сообщение о недоступности порта.

Обобщая вышесказанное, мы можем утверждать, что клиент или сервер UDP может вызвать функцию `connect`, только если этот процесс использует сокет UDP для связи лишь с одним собеседником. Обычно именно клиент UDP вызывает функцию `connect`, но существуют приложения, в которых сервер UDP связывается с одним клиентом на длительное время (например, TFTP), и в этом случае и клиент, и сервер вызывают функцию `connect`.

Еще один пример долгосрочного взаимодействия — это DNS (рис. 8.8).



Рис. 8.8. Пример клиентов и серверов DNS и функции `connect`

Клиент DNS может быть сконфигурирован для использования одного или более серверов, обычно с помощью перечисления IP-адресов серверов в файле `/etc/resolv.conf`. Если в этом файле указан только один сервер (на рисунке этот клиент изображен в крайнем слева прямоугольнике), клиент может вызвать функцию `connect`, но если перечислено множество серверов (второй справа прямоугольник на рисунке), клиент не может вызвать функцию `connect`. Обычно сервер DNS обрабатывает также любые клиентские запросы, следовательно, серверы не могут вызывать функцию `connect`.

Многократный вызов функции `connect` для сокета UDP

Процесс с присоединенным сокетом UDP может снова вызвать функцию `connect`. Для этого сокета, чтобы:

- задать новый IP-адрес и порт;
- отсоединить сокет.

Первый случай, задание нового собеседника для присоединенного сокета UDP, отличается от использования функции `connect` с сокетом TCP: для сокета TCP функция `connect` может быть вызвана только один раз.

Чтобы отсоединить сокет UDP, мы вызываем функцию `connect`, но присваиваем элементу семейства структуры адреса сокета (`sin_family` для IPv4 или `sin6_family` для IPv6) значение `AF_UNSPEC`. Это может привести к ошибке `EAFNOSUPPORT` [128, с. 736], но это нормально. Именно процесс вызова функции `connect` на уже присоединенном сокете UDP позволяет отсоединить сокет [128, с. 787–788].

ПРИМЕЧАНИЕ

В руководстве BSD по поводу функции `connect` традиционно говорилось: «Сокеты дейтаграмм могут разрывать связь, соединяясь с недействительными адресами, такими как

пустые адреса». К сожалению, ни в одном руководстве не сказано, что представляет собой «пустой адрес», и не упоминается, что в результате возвращается ошибка (что нормально). Стандарт POSIX явно указывает, что семейство адресов должно быть установлено в AF_UNSPEC, но затем сообщает, что этот вызов функции connect может возвратить, а может и не возвратить ошибку EAFNOSUPPORT.

Производительность

Когда приложение вызывает функцию sendto на неприсоединенном сокете UDP, ядра реализаций, происходящих от Беркли, временно соединяются с сокетом, отправляют дейтаграмму и затем отсоединяются от сокета [128, с. 762–763]. Таким образом, вызов функции sendto для последовательной отправки двух дейтаграмм на неприсоединенном сокете включает следующие шесть шагов, выполняемых ядром:

- присоединение сокета;
- вывод первой дейтаграммы;
- отсоединение сокета;
- присоединение сокета;
- вывод второй дейтаграммы;
- отсоединение сокета.

ПРИМЕЧАНИЕ

Другой момент, который нужно учитывать, — количество поисков в таблице маршрутизации. Первое временное соединение производит поиск в таблице маршрутизации IP-адреса получателя и сохраняет (кэширует) эту информацию. Второе временное соединение отмечает, что адрес получателя совпадает с кэшированным адресом из таблицы маршрутизации (мы считаем, что обеим функциям sendto задан один и тот же получатель), и ему не нужно снова проводить поиск в таблице маршрутизации [128, с. 737–738].

Когда приложение знает, что оно будет отправлять множество дейтаграмм одному и тому же собеседнику, эффективнее будет присоединить сокет явно. Вызов функции connect, за которым следуют два вызова функции write, теперь будет включать следующие шаги, выполняемые ядром:

- присоединение сокета;
- вывод первой дейтаграммы;
- вывод второй дейтаграммы.

В этом случае ядро копирует структуру адреса сокета, содержащую IP-адрес получателя и порт, только один раз, а при двойном вызове функции sendto копирование выполняется дважды. В [89] отмечается, что на временное присоединение отсоединенного сокета UDP приходится примерно треть стоимости каждой передачи UDP.

8.12. Функция dg_cli (продолжение)

Вернемся к функции dg_cli, показанной в листинге 8.4, и перепишем ее, с тем чтобы она вызывала функцию connect. В листинге 8.7 показана новая функция.

Листинг 8.7. Функция dg_cli, вызывающая функцию connect

```
//udpciserv/dgcliconnect.c
1 #include "unp.h"

2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int n;
6     char sendline[MAXLINE], recvline[MAXLINE + 1];
7     Connect(sockfd, (SA*)pservaddr, servlen);
```

```

8 while (Fgets(sendline, MAXLINE, fp) != NULL) {
9     Write(sockfd, sendline, strlen(sendline));
10    n = Read(sockfd, recvline, MAXLINE);
11    recvline[n] = 0; /* завершающий нуль */
12    Fputs(recvline, stdout);
13 }
14 }
```

Изменения по сравнению с предыдущей версией — это добавление вызова функции `connect` и замена вызовов функций `sendto` и `recvfrom` вызовами функций `write` и `read`. Функция `dg_cli` остается не зависящей от протокола, поскольку она не вникает в структуру адреса сокета, передаваемую функции `connect`. Наша функция `main` клиента, показанная в листинге 8.3, остается той же.

Если мы запустим программу на узле `macosx`, задав IP-адрес узла `freebsd4` (который не запускает наш сервер на порте 9877), мы получим следующий вывод:

```
macosx % udpccli04 172.24.37.94
hello, world
read error: Connection refused
```

Первое, что мы замечаем, — мы *не* получаем ошибку, когда запускаем процесс клиента. Ошибка происходит только после того, как мы отправляем серверу первую дейтаграмму. Именно отправка этой дейтаграммы вызывает ошибку ICMP от узла сервера. Но когда клиент TCP вызывает функцию `connect`, задавая узел сервера, на котором не запущен процесс сервера, функция `connect` возвращает ошибку, поскольку вызов функции `connect` вызывает отправку первого пакета трехэтапного рукопожатия TCP, и именно этот пакет вызывает получение сегмента RST от собеседника (см. раздел 4.3).

В листинге 8.8 показан вывод программы `tcpdump`.

Листинг 8.8. Вывод программы `tcpdump` при запуске функции `dg_cli`

```
macosx % tcpdump
01 0.0          macosx.51139 > freebsd4 9877:udp 13
02 0.006180 ( 0.0062) freebsd4 > macosx: icmp: freebsd4 udp port 9877 unreachable
```

В табл. А.5 мы также видим, что возникшую ошибку ICMP ядро сопоставляет ошибке `ECONNREFUSED`, которая соответствует выводу строки сообщения `Connection refused` (В соединении отказано) функцией `err_sys`.

ПРИМЕЧАНИЕ

К сожалению, не все ядра возвращают сообщения ICMP присоединенному сокету UDP, как мы показали в этом разделе. Обычно ядра реализаций, происходящих от Беркли, возвращают эту ошибку, а ядра System V — не возвращают. Например, если мы запустим тот же клиент на узле Solaris 2.4 и с помощью функции `connect` соединимся с узлом, на котором не запущен наш сервер, то с помощью программы `tcpdump` мы сможем убедиться, что ошибка ICMP о недоступности порта возвращается узлом сервера, но вызванная клиентом функция `read` никогда не завершается. Эта ситуация была исправлена в Solaris 2.5. UnixWare не возвращает ошибку, в то время как AIX, Digital Unix, HP-UX и Linux возвращают.

8.13. Отсутствие управления потоком в UDP

Теперь мы проверим, как влияет на работу приложения отсутствие какого-либо управления потоком в UDP. Сначала мы изменим нашу функцию `dg_cli` так, чтобы она отправляла фиксированное число дейтаграмм. Она больше не будет читать из стандартного потока ввода. В листинге 8.9 показана новая версия функции. Эта функция отправляет серверу 2000 дейтаграмм UDP по 1400 байт каждая.

Листинг 8.9. Функция `dg_cli`, отсылающая фиксированное число дейтаграмм серверу

```
//udpcliserv/dgcliloo1.c
1 #include "unp.h"
```

```

2 #define NDG 2000 /* количество дейтаграмм для отправки */
3 #define DGLEN 1400 /* длина каждой дейтаграммы */

4 void
5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
6 {
7     int i;
8     char sendline[DGLEN];

9     for (i = 0; i < NDG; i++) {
10        Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
11    }
12 }
```

Затем мы изменяем сервер так, чтобы он получал дейтаграммы и считал число полученных дейтаграмм. Сервер больше не отражает дейтаграммы обратно клиенту. В листинге 8.10 показана новая функция dg_echo. Когда мы завершаем процесс сервера нажатием клавиши прерывания на терминале (что приводит к отправке сигнала SIGINT процессу), сервер выводит число полученных дейтаграмм и завершается.

Листинг 8.10. Функция dg_echo, считающая полученные дейтаграммы

```

//udpcliserv/dgecholoop1.c
1 #include "unp.h"

2 static void recvfrom_int(int);
3 static int count;

4 void
5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
6 {
7     socklen_t len;
8     char mesg[MAXLINE];

9     Signal (SIGINT, recvfrom_int);

10    for (;;) {
11        len = clilen;
12        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

13        count++;
14    }
15 }

16 static void
17 recvfrom_int(int signo)
18 {
19     printf("\nreceived %d datagrams\n", count);
20     exit(0);
21 }
```

Теперь мы запускаем сервер на узле freebsd, который представляет собой медленный компьютер SPARCStation. Клиент мы запускаем в значительно более быстрой системе RS/6000 с операционной системой aix. Они соединены друг с другом напрямую каналом Ethernet на 100 Мбит/с. Кроме того, мы запускаем программу netstat -s на узле сервера и до, и после запуска клиента и сервера, поскольку выводимая статистика покажет, сколько дейтаграмм мы потеряли. В листинге 8.11 показан вывод сервера.

Листинг 8.11. Вывод на узле сервера

```
freebsd % netstat -s -p udp
udp:
```

```

71208 datagrams received
0 with incomplete header
0 with bad data length field
0 with bad checksum
0 with no checksum
832 dropped due to no socket
16 broadcast/multicast datagrams dropped due to no socket
1971 dropped due to full socket buffers
0 not for hashed pcb
68389 delivered
137685 datagrams output
freebsd % udpserv06 запускаем наш сервер
        клиент посыпает дейтаграммы
^C для окончания работы клиента вводим наш символ прерывания
freebsd % netstat -s -p udp
udp
73208 datagrams received
0 with incomplete header
0 with bad data length field
0 with bad checksum
0 with no checksum
832 dropped due to no socket
16 broadcast/multicast datagrams dropped due to no socket
3941 dropped due to full socket buffers
0 not for hashed pcb
68419 delivered
137685 datagrams output

```

Клиент отправил 2000 дейтаграмм, но приложение-сервер получило только 30 из них, что означает уровень потерь 98%. Ни сервер, ни клиент не получают сообщения о том, что эти дейтаграммы потеряны. Как мы и говорили, UDP не имеет возможности управления потоком — он ненадежен. Как мы показали, для отправителя UDP не составляет труда переполнить буфер получателя.

Если мы посмотрим на вывод программы **netstat**, то увидим, что общее число дейтаграмм, полученных узлом сервера (не приложением-сервером) равно 2000 (73 208 – 71 208). Счетчик **dropped due to full socket buffers** (отброшено из-за переполнения буферов сокета) показывает, сколько дейтаграмм было получено UDP и проигнорировано из-за того, что приемный буфер принимающего сокета был полон [128, с. 775]. Это значение равно 1970 (3941 – 1971), что при добавлении к выводу счетчика дейтаграмм, полученных приложением (30), дает 2000 дейтаграмм, полученных узлом. К сожалению, счетчик дейтаграмм, отброшенных из-за заполненного буфера, в программе **netstat** распространяется на всю систему. Не существует способа определить, на какие приложения (например, какие порты UDP) это влияет.

Число дейтаграмм, полученных сервером в этом примере, недетерминировано. Оно зависит от многих факторов, таких как нагрузка сети, загруженность узла клиента и узла сервера.

Если мы запустим тот же клиент и тот же сервер, но на этот раз клиент на медленной системе Sun, а сервер на быстрой системе RS/6000, никакие дейтаграммы не теряются.

```

aix % udpserv06
^? после окончания работы клиента вводим наш символ прерывания
received 2000 datagrams

```

Приемный буфер сокета UDP

Число дейтаграмм UDP, установленных в очередь UDP, для данного сокета ограничено размером его приемного буфера. Мы можем изменить его с помощью параметра сокета **SO_RCVBUF**, как мы показали в разделе 7.5. В FreeBSD по умолчанию размер приемного буфера сокета UDP равен 42 080 байт, что допускает возможность хранения только 30 из наших 1400-байтовых дейтаграмм. Если мы увеличим размер приемного буфера сокета, то можем рассчитывать, что сервер получит дополнительные

дейтаграммы. В листинге 8.12 представлена измененная функция dg_echo из листинга 8.10, которая увеличивает размер приемного буфера сокета до 240 Кбайт. Если мы запустим этот сервер в системе Sun, а клиент — в системе RS/6000, то счетчик полученных дейтаграмм будет иметь значение 103. Поскольку это лишь немногим лучше, чем в предыдущем примере с размером буфера, заданным по умолчанию, ясно, что мы пока не получили решения проблемы.

Листинг 8.12. Функция dg_echo, увеличивающая размер приемного буфера сокета

```
//udpcliserv/dgecholoop2.c
1 #include "unp.h"

2 static void recvfrom_int(int);
3 static int count;

4 void
5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
6 {
7     int n;
8     socklen_t len;
9     char mesg[MAXLINE];

10    Signal(SIGINT, recvfrom_int);

11    n = 240 * 1024;
12    Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

13    for (;;) {
14        len = clilen;
15        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

16        count++;
17    }
18 }

19 static void
20 recvfrom_int(int signo)
21 {
22     printf("\nreceived %d datagrams\n", count);
23     exit(0);
24 }
```

ПРИМЕЧАНИЕ

Почему мы устанавливаем размер буфера приема сокета равным 240×1024 байт в листинге 8.12? Максимальный размер приемного буфера сокета в BSD/OS 2.1 по умолчанию равен 262 144 байта (256×1024), но из-за способа размещения буфера в памяти (описанного в главе 2 [128]) он в действительности ограничен до 246 723 байт. Многие более ранние системы, основанные на 4.3BSD, ограничивали размер буфера приема сокета примерно до 52 000 байт.

8.14. Определение исходящего интерфейса для UDP

С помощью присоединенного сокета UDP можно также задавать исходящий интерфейс, который будет использован для отправки дейтаграмм к определенному получателю. Это объясняется побочным эффектом функции connect, примененной к сокету UDP: ядро выбирает локальный IP-адрес (предполагается, что процесс еще не вызвал функцию bind для явного его задания). Локальный адрес выбирается в процессе поиска адреса получателя в таблице маршрутизации, причем берется основной IP-адрес интерфейса, с которого, согласно таблице, будут отправляться дейтаграммы.

В листинге 8.13 показана простая программа UDP, которая с помощью функции connect соединяется с заданным IP-адресом и затем вызывает функцию getsockname, выводя локальный IP-адрес и порт.

Листинг 8.13. Программа UDP, использующая функцию connect для определения исходящего интерфейса

```
//udpcliserv/udpcli09.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;
6     socklen_t len;
7     struct sockaddr_in cliaddr, servaddr;

8     if (argc != 2)
9         err_quit("usage: udpcli <Ipaddress>");

10    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

11    bzero(&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_port = htons(SERV_PORT);
14    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

15    Connect(sockfd, (SA*)&servaddr, sizeof(servaddr));

16    len = sizeof(cliaddr);
17    Getsockname(sockfd, (SA*)&cliaddr, &len);
18    printf("local address %s\n", Sock_ntop((SA*)&cliaddr, len));

19    exit(0);
20 }
```

Если мы запустим программу на узле freebsd с несколькими сетевыми интерфейсами, то получим следующий вывод:

```
freebsd % udpcli09 206.168.112.96
local address 12.106.32.254:52329
```

```
freebsd % udpcli09 192.168.42.2
local address 192.168.42.1:52330
```

```
freebsd % udpcli09 127.0.0.1
local address 127.0.0.1:52331
```

По рис. 1.7 видно, что когда мы запускаем программу первые два раза, аргументом командной строки является IP-адрес в разных сетях Ethernet. Ядро присваивает локальный IP-адрес первичному адресу интерфейса в соответствующей сети Ethernet. При вызове функции connect на сокете UDP ничего не отправляется на этот узел — это полностью локальная операция, которая сохраняет IP-адрес и порт собеседника. Мы также видим, что вызов функции connect на неприсоединенном сокете UDP также присваивает сокету динамически назначаемый порт.

ПРИМЕЧАНИЕ

К сожалению, эта технология действует не во всех реализациях, что особенно касается ядер, происходящих от SVR4. Например, это не работает в Solaris 2.5, но работает в AIX, Digital Unix, Linux, Mac OS X и Solaris 2.6.

8.15. Эхо-сервер TCP и UDP, использующий функцию select

Теперь мы объединим наш параллельный эхо-сервер TCP из главы 5 и наш последовательный эхо-сервер UDP из данной главы в один сервер, использующий функцию `select` для мультиплексирования сокетов TCP и UDP. В листинге 8.14 представлена первая часть этого сервера.

Листинг 8.14. Первая часть эхо-сервера, обрабатывающего сокеты TCP и UDP при помощи функции `select`

```
//udpcliserv/udpservselect01.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int listenfd, connfd, udpfd, nready, maxfdp1;
6     char mesg[MAXLINE];
7     pid_t childpid;
8     fd_set rset;
9     ssize_t n;
10    socklen_t len;
11    const int on = 1;
12    struct sockaddr_in cliaddr, servaddr;
13    void sig_chld(int);

14    /* создание прослушиваемого сокета TCP */
15    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

16    bzero(&servaddr, sizeof(servaddr));
17    servaddr.sin_family = AF_INET;
18    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19    servaddr.sin_port = htons(SERV_PORT);

20    Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21    Bind(listenfd, (SA*)&servaddr, sizeof(servaddr));

22    Listen(listenfd, LISTENQ);

23    /* создание сокета UDP */
24    udpfd = Socket(AF_INET, SOCK_DGRAM, 0);

25    bzero(&servaddr, sizeof(servaddr));
26    servaddr.sin_family = AF_INET;
27    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
28    servaddr.sin_port = htons(SERV_PORT);

29    Bind(udpfd, (SA*)&servaddr, sizeof(servaddr));
```

Создание прослушиваемого сокета TCP

14-22 Создается прослушиваемый сокет TCP, который связывается с заранее известным портом сервера. Мы устанавливаем параметр сокета `SO_REUSEADDR` в случае, если на этом порте существуют соединения.

Создание сокета UDP

23-29 Также создается сокет UDP и связывается с тем же портом. Даже если один и тот же порт используется для сокетов TCP и UDP, нет необходимости устанавливать параметр сокета `SO_REUSEADDR` перед этим вызовом функции `bind`, поскольку порты TCP не зависят от портов UDP.

В листинге 8.15 показана вторая часть нашего сервера.

Листинг 8.15. Вторая половина эхо-сервера, обрабатывающего TCP и UDP при помощи функции `select`

```
1  udpcliserv/udpservselect01.c
2  ...
3  30  Signal(SIGCHLD, sig_chld); /* требуется вызвать waitpid() */
4
5  31  FD_ZERO(&rset);
6  32  maxfdp1 = max(listenfd, udpfd) + 1;
7  33  for (;;) {
8  34    FD_SET(listenfd, &rset);
9  35    FD_SET(udpfd, &rset);
10  36    if ((nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
11      if (errno == EINTR)
12        continue; /* назад в for() */
13      else
14        err_sys("select error");
15    }
16    if (FD_ISSET(listenfd, &rset)) {
17      len = sizeof(cliaddr);
18      connfd = Accept(listenfd, (SA*)&cliaddr, &len);
19
20      if ((childpid = Fork()) == 0) { /* дочерний процесс */
21        Close(listenfd); /* закрывается прослушиваемый сокет */
22        str_echo(connfd); /* обработка запроса */
23        exit(0);
24      }
25      Close(connfd); /* родитель закрывает присоединенный сокет */
26    }
27    if (FD_ISSET(udpfd, &rset)) {
28      len = sizeof(cliaddr);
29      n = Recvfrom(udpfd, mesg, MAXLINE, 0, (SA*)&cliaddr, &len);
30
31      Sendto(udpfd, mesg, n, 0, (SA*)&cliaddr, len);
32    }
33  }
```

Установка обработчика сигнала `SIGCHLD`

30 Для сигнала `SIGCHLD` устанавливается обработчик, поскольку соединения TCP будут обрабатываться дочерним процессом. Этот обработчик сигнала мы показали в листинге 5.8.

Подготовка к вызову функции `select`

31-32 Мы инициализируем набор дескрипторов для функции `select` и вычисляем максимальный из двух дескрипторов, готовности которого будем ожидать.

Вызов функции `select`

34-41 Мы вызываем функцию `select`, ожидая только готовности к чтению прослушиваемого сокета TCP или сокета UDP. Поскольку наш обработчик сигнала `sig_chld` может прервать вызов функции `select`, обрабатываем ошибку `EINTR`.

Обработка нового клиентского соединения

42-51 С помощью функции `accept` мы принимаем новое клиентское соединение, а когда прослушиваемый сокет TCP готов для чтения, с помощью функции `fork` порождаем дочерний процесс и вызываем нашу функцию `str_echo` в дочернем процессе. Это та же последовательность действий, которую мы выполняли в главе 5.

Обработка приходящей дейтаграммы

52-57 Если сокет UDP готов для чтения, дейтаграмма пришла. Мы читаем ее с помощью функции `recvfrom` и отправляем обратно клиенту с помощью функции `sendto`.

8.16. Резюме

Преобразовать наши эхо-клиент и эхо-сервер так, чтобы использовать UDP вместо TCP, оказалось несложно. Но при этом мы лишились множества возможностей, предоставляемых протоколом TCP: определение потерянных пакетов и повторная передача, проверка, приходят ли пакеты от корректного собеседника, и т.д. Мы возвратимся к этой теме в разделе 22.5 и увидим, как можно улучшить надежность приложения UDP.

Сокеты UDP могут генерировать асинхронные ошибки, то есть ошибки, о которых сообщается спустя некоторое время после того, как пакет был отправлен. Сокеты TCP всегда сообщают приложению о них, но в случае UDP для получения этих ошибок сокет должен быть присоединенным.

В UDP отсутствует возможность управления потоком, что очень легко продемонстрировать. Обычно это не создает проблем, поскольку многие приложения UDP построены с использованием модели «запрос-ответ» и не предназначены для передачи большого количества данных.

Есть еще ряд моментов, которые нужно учитывать при написании приложений UDP, но мы рассмотрим их в главе 22 после описания функций интерфейсов, широковещательной и многоадресной передачи.

Упражнения

1. Допустим, у нас имеется два приложения, одно использует TCP, а другое — UDP. В приемном буфере сокета TCP находится 4096 байт данных, а в приемном буфере для сокета UDP — две дейтаграммы по 2048 байт. Приложение TCP вызывает функцию `read` с третьим аргументом 4096, а приложение UDP вызывает функцию `recvfrom` с третьим аргументом 4096. Есть ли между этими вызовами какая-нибудь разница?

2. Что произойдет в листинге 8.2, если мы заменим последний аргумент функции `sendto` (который мы обозначили `len`) аргументом `c1ilen`?

3. Откомпилируйте и запустите сервер UDP из листингов 8.1 и 8.4, а затем — клиент из листингов 8.3 и 8.4. Убедитесь в том, что клиент и сервер работают вместе.

4. Запустите программу `ping` в одном окне, задав параметр `-i 60` (отправка одного пакета каждые 60 секунд; некоторые системы используют ключ `I` вместо `i`), параметр `-v` (вывод всех полученных сообщений об ошибках ICMP) и задав адрес закольцовки на себя (обычно 127.0.0.1). Мы будем использовать эту программу, чтобы увидеть ошибку ICMP недоступности порта, возвращаемую узлом сервера. Затем запустите наш клиент из предыдущего упражнения в другом окне, задав IP-адрес некоторого узла, на котором не запущен сервер. Что происходит?

5. Рассматривая рис. 8.3, мы сказали, что каждый присоединенный сокет TCP имеет свой собственный буфер приема. Как вы думаете, есть ли у прослушиваемого сокета свой собственный буфер приема?

6. Используйте программу `sock` (см. раздел B.3) и такое средство, как, например, `tcpdump` (см. раздел B.5), чтобы проверить утверждение из раздела 8.10: если клиент с помощью функции `bind` связывает IP-адрес со своим сокетом, но отправляет дейтаграмму, исходящую от другого интерфейса, то результирующая дейтаграмма содержит IP-адрес, который был связан с сокетом, даже если он не соответствует исходящему интерфейсу.

7. Откомпилируйте программы из раздела 8.13 и запустите клиент и сервер на различных узлах. Помещайте `printf` в клиент каждый раз, когда дейтаграмма записывается в сокет. Изменяет ли это процент полученных пакетов? Почему? Вызывайте `printf` из сервера каждый раз, когда дейтаграмма читается из сокета. Изменяет ли это процент полученных пакетов? Почему?

8. Какова наибольшая длина, которую мы можем передать функции `sendto` для сокета UDP/IPv4, то есть каково наибольшее количество данных, которые могут поместиться в дейтаграмму UDP/IPv4? Что изменяется в случае UDP/IPv6?

Измените листинг 8.4, с тем чтобы отправить одну дейтаграмму UDP максимального размера, считать ее обратно и вывести число байтов, возвращаемых функцией `recvfrom`.

9. Измените листинг 8.15 таким образом, чтобы он соответствовал RFC 1122: для сокета UDP следует использовать параметр `IP_RECVSTADDR`.

Глава 9

Основы сокетов SCTP

9.1. Введение

SCTP — новый транспортный протокол, принятый IETF в качестве стандарта в 2000 году. (Для сравнения, протокол TCP был стандартизован в 1981 году.) Изначально SCTP проектировался с учетом потребностей растущего рынка IP-телефонии, и предназначался, в частности, для передачи телефонного сигнала через Интернет. Требования, которым должен был отвечать SCTP, описываются в RFC 2719 [84]. SCTP — надежный протокол, ориентированный на передачу сообщений, предоставляющий возможность работать с несколькими потоками каждой паре конечных точек, а также обеспечивающий поддержку концепции многоинтерфейсного узла на транспортном уровне. Поскольку это относительно новый протокол, он распространен не так широко, как TCP и UDP, однако он обладает особенностями, облегчающими проектирование некоторых видов приложений. Выбору между SCTP и TCP будет посвящен раздел 23.12.

Несмотря на принципиальную разницу между SCTP и TCP, с точки зрения приложения интерфейс SCTP типа «один-к-одному» почти ничем не отличается от интерфейса TCP. Это делает перенос приложений достаточно тривиальным, однако при таком переносе некоторые усовершенствованные функции SCTP остаются незадействованными. Интерфейс типа «один-ко-многим» задействует эти функции «на всю катушку», но переход к нему может потребовать значительной переделки существующих приложений. Новый интерфейс рекомендуется использовать большинству новых приложений, разрабатываемых в расчете на SCTP.

Эта глава описывает дополнительные элементарные функции сокетов, которые могут использоваться с SCTP. Сначала мы опишем две модели интерфейса, доступные разработчику приложения. В главе 10 мы разработаем новую версию эхо-сервера, использующую модель «один-ко-многим». Кроме того, мы опишем новые функции, которые предназначены только для SCTP. Особое внимание будет уделено функции `shutdown` и различиям процедуры завершения ассоциации SCTP от процедуры завершения соединения TCP. В разделе 23.4 мы рассмотрим пример использования уведомлений для оповещения приложения о важных событиях, связанных с протоколом (помимо прибытия новых пользовательских данных).

Интерфейс функций SCTP еще не стабилизировался полностью, что объясняется молодостью этого протокола. На момент написания этой книги описываемые в ней интерфейсы считались стабилизировавшимися, однако они еще не были распространены так широко, как остальные части API сокетов. Те, кто работает с приложениями, ориентированными исключительно на SCTP, должны быть готовы устанавливать обновления для ядра или для операционной системы в целом, а приложения, рассчитанные на повсеместное использование, должны уметь работать с TCP, потому что протокол SCTP пока что доступен далеко не на всех системах.

9.2. Модели интерфейса

Сокеты SCTP бывают двух типов: «один-к-одному» и «один-ко-многим». Сокету типа «один-к-одному» всегда сопоставляется ровно одна ассоциация SCTP. Вспомните, что в разделе 2.5 мы отмечали, что ассоциация является соединением между двумя системами, которое может задействовать более двух IP-адресов, если хотя бы одна из систем имеет несколько интерфейсов. Связь между сокетом и ассоциацией SCTP такая же, как между сокетом и соединением TCP. Сокету типа «один-ко-многим» может сопоставляться одновременно несколько активных ассоциаций. То же самое имеет место и в UDP, где сокет, привязанный к конкретному порту, может получать дейтаграммы от нескольких конечных точек UDP, передающих данные одновременно.

Выбор интерфейса при разработке приложения должен осуществляться с учетом нескольких факторов:

- тип сервера (последовательный или параллельный);
- количество дескрипторов сокетов, с которыми должен работать сервер;
- важно ли оптимизировать работу приложения, разрешив передачу данных в третьем (и, возможно, четвертом) пакете четырехэтапного рукопожатия;

- для какого количества соединений существует необходимость хранить информацию о состоянии.

ПРИМЕЧАНИЕ

Когда API сокетов для протокола SCTP еще только разрабатывался, сокеты разных типов назывались по-разному. Читатели до сих пор могут столкнуться со старой терминологией в документации или исходном коде. Изначально сокет типа «один-к-одному» назывался сокетом типа TCP (TCP-style socket), а сокет типа «один-ко-многим» — сокетом типа UDP (UDP-style socket).

Впоследствии от этих терминов пришлось отказаться, так как они создавали впечатление, что SCTP будет вести себя, как TCP или UDP, при использовании сокетов соответствующих типов. На самом деле имелось в виду только одно различие между TCP и UDP: возможность одновременной работы с несколькими адресатами на транспортном уровне. Современные термины («один-к-одному» и «один-ко-многим») фокусируют наше внимание на главном отличии двух типов сокетов.

Наконец, обратите внимание, что некоторые авторы используют термин «несколько-к-одному» вместо «один-ко-многим». Эти термины взаимозаменяемы.

Сокет типа «один-к-одному»

Данный тип сокета был разработан специально для облегчения переноса существующих приложений с TCP на SCTP. Его модель практически идентична описанной в главе 4. Существуют, конечно, некоторые отличия, о которых следует помнить (в особенности, при переносе приложений):

1. Все параметры сокетов должны быть преобразованы к соответствующим эквивалентам SCTP. Чаще всего используются параметры TCP_NODELAY и TCP_MAXSEG, вместо которых следует задавать SCTP_NODELAY и SCTP_MAXSEG.

2. Протокол SCTP сохраняет границы сообщений, поэтому приложению не приходится кодировать их самостоятельно. Например, приложение, основанное на TCP, может отправлять записи, чередуя двухбайтовые поля длины с полями данных переменной длины (каждое поле записывается в буфер отправки отдельным вызовом `write`). Если так поступить с SCTP, адресат получит два отдельных сообщения, то есть функция `read` возвратится дважды: один раз с двухбайтовым сообщением (поле длины), а второй — с сообщением неопределенной длины.

3. Некоторые TCP-приложения используют половинное закрытие для извещения собеседника о конце считываемых данных. Для переноса таких приложений на SCTP потребуется переписать их таким образом, чтобы сигнал о конце данных передавался в обычном потоке.

4. Функция `send` может использоваться обычным образом. Функции `sendto` и `sendmsg` трактуют информацию об адресе получателя как приоритетную перед основным адресом собеседника (см. раздел 2.8).

Типичное приложение, работающее в стиле «один-к-одному», будет вести себя так, как показано на временной диаграмме рис. 9.1. Запущенный сервер открывает сокет, привязывается к адресу, после чего ожидает подсоединения клиента в системном вызове `accept`. Через некоторое время запускается клиент, который открывает свой сокет и инициирует установление ассоциации с сервером. Предполагается, что клиент отправляет серверу запрос, сервер обрабатывает этот запрос и отправляет свой ответ обратно клиенту. Взаимодействие продолжается до тех пор, пока клиент не начнет процедуру завершения ассоциации. После закрытия ассоциации сервер либо завершает работу, либо ожидает установления новой ассоциации. Из сравнения с временной диаграммой TCP (см. рис. 4.1) становится ясно, что обмен пакетами через сокет SCTP типа «один-к-одному» осуществляется приблизительно так же.

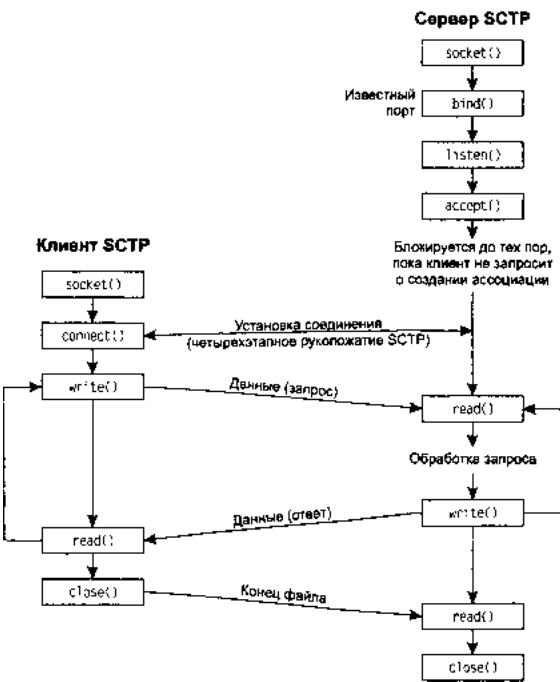


Рис. 9.1. Временная диаграмма для сокета SCTP типа «один-к-одному»

Сокет SCTP типа «один-к-одному» является IP-сокетом (семейство AF_INET или AF_INET6) со значением типа SOCK_STREAM и значением протокола IPPROTO_SCTP.

Сокет типа «один-ко-многим»

Сокет типа «один-ко-многим» дает разработчику приложения возможность написать сервер, не использующий большого количества дескрипторов сокетов. Один дескриптор для такого сервера будет представлять несколько ассоциаций, подобно сокету UDP, способному принимать дейтаграммы от множества клиентов. Для обращения к конкретной ассоциации, установленной для сокета типа «один-ко-многим», используется идентификатор. Идентификатор ассоциации представляет собой значение типа `sctp_assoc_t` (обычно это целое число). Значение идентификатора скрывается от приложения, то есть оно не должно использовать идентификатор, если тот еще не был предоставлен приложению ядром.

При написании приложения, использующего сокеты данного типа, рекомендуется помнить о следующих важных моментах:

1. Когда клиент закрывает ассоциацию, она автоматически закрывается и на стороне сервера. При этом удаляются все сведения о состоянии ассоциации в ядре.

2. Только при использовании типа «один-ко-многим» возможна передача данных в третьем и четвертом пакетах четырехэтапного рукопожатия (см. упражнение 9.3).

3. Вызов `sendto`, `sendmsg` или `sctp_sendmsg` для адресата, с которым еще не установлена ассоциация, приведет к попытке активного открытия, в результате чего будет создана новая ассоциация с указанным адресом. Это происходит даже в том случае, если приложение, вызвавшее `send`, перед этим вызвало для того же сокета функцию `listen`, запросив пассивное открытие.

4. Приложение должно использовать функции `sendto`, `sendmsg` и `sctp_sendmsg`, но не `send` и `write`. (Если вы создали сокет типа «один-к-одному» вызовом `sctp_peeloff`, то `send` и `write` вызывать можно.)

5. При вызове одной из функций отправки данных используется основной адрес получателя, выбранный системой в момент установки ассоциации (раздел 2.8), если вызывающий процесс не установил флаг `MSG_ADDR_OVER` в структуре `sctp_sndrcvinfo`. Для этого необходимо вызвать функцию `sendmsg` с вспомогательными данными или воспользоваться функцией `sctp_sendmsg`.

6. Уведомление о событиях для ассоциации может быть включено по умолчанию, так что если приложению не требуется получать эти уведомления, оно должно явным образом отключить их при помощи параметра сокета `SCTP_EVENTS`. (Одно из множества уведомлений SCTP обсуждается в разделе 9.14.) По умолчанию единственным включенным событием является `sctp_data_io_event`. Уведомление о

нем передается в виде вспомогательных данных при вызове `recvmsg` и `sctp_recvmsg`. Это относится к сокетам обоих типов.

ПРИМЕЧАНИЕ

Когда интерфейс API сокетов SCTP находился на стадии разработки, для сокетов типа «один-ко-многим» по умолчанию было включено еще и уведомление об установке ассоциации. В более поздних версиях документации API говорится о том, что по умолчанию для сокетов обоих типов отключены все уведомления, за исключением `sctp_data_io_event`. Однако не все реализации могут соответствовать этому утверждению. Хорошим тоном будет включать все нужные уведомления и отключать ненужные в явном виде. Благодаря этому разработчик получает гарантию того, что приложение будет вести себя так, как он этого хочет, в любой операционной системе.

Типичная временная диаграмма для сокета типа «один-ко-многим» приведена на рис. 9.2. Сначала запускается сервер, который создает сокет, привязывает его к адресу, вызывает функцию `listen` для того, чтобы разрешить клиентам устанавливать ассоциации, после чего он вызывает `sctp_recvmsg` и приостанавливается в ожидании первого сообщения. В свою очередь, клиент открывает сокет и вызывает функцию `sctp_sendto`, которая неявно инициирует ассоциацию и вкладывает данные в третий пакет четырехэтапного рукопожатия. Сервер получает запрос, обрабатывает его и отсылает свой ответ. Клиент получает ответ сервера и закрывает сокет, тем самым закрывая и ассоциацию. Сервер переходит к ожиданию следующего сообщения.

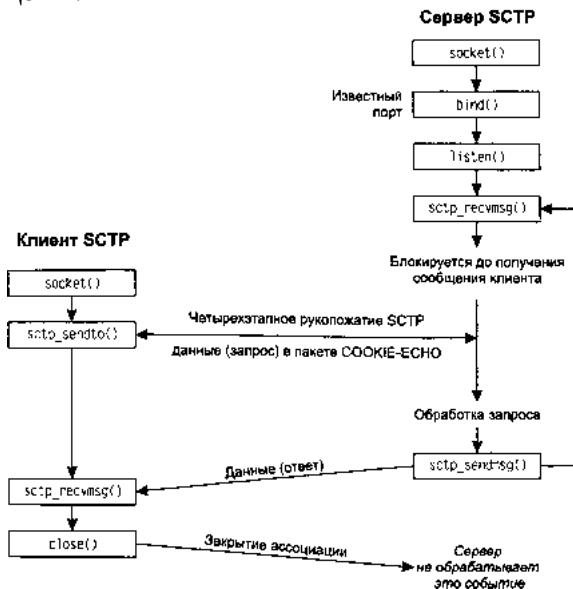


Рис. 9.2. Временная диаграмма работы сокета типа «один-ко-многим»

В этом примере рассматривается последовательный сервер, один программный поток которого обрабатывает сообщения, полученные через несколько ассоциаций. SCTP позволяет использовать сокет типа «один-ко-многим» с функцией `sctp_peeloff` (см. раздел 9.12) для реализации комбинированной параллельно- последовательной модели сервера.

- Функция `sctp_peeloff` позволяет выделить конкретную ассоциацию (например, долговременный сеанс связи) из сокета типа «один-ко-многим» в отдельный сокет типа «один-к-одному».

- Полученный таким образом сокет типа «один-к-одному» может быть передан новому потоку или порожденному процессу (как в модели параллельного сервера).

- Основной поток обрабатывает сообщения от всех остальных ассоциаций в последовательном режиме.

Сокет SCTP типа «один-ко-многим» является IP-сокетом (семейство `AF_INET` или `AF_INET6`) со значением типа `SOCK_SEQPACKET` и значением протокола `IPPROTO_SCTP`.

9.3. Функция sctp_bindx

Сервер SCTP может привязаться к некоторому подмножеству IP-адресов узла, на котором он запущен. Серверы TCP и UDP могли привязываться либо к одному, либо ко всем адресам узла, но не могли указывать конкретный набор адресов. Функция `sctp_bindx` делает программирование приложений более гибким, предоставляя возможность связывать сокет SCTP с заданными адресами.

```
#include <netinet/sctp.h>
```

```
int sctp_bindx(int sockfd, const struct sockaddr *addrs, int addrcnt, int flags);
```

Возвращает: 0 в случае успешного завершения, -1 в случае ошибки

Аргумент `sockfd` представляет собой дескриптор сокета, возвращаемый функцией `socket`. Второй аргумент — указатель на упакованный список адресов. Каждая структура адреса сокета помещается в буфер непосредственно после предшествующей структуры, без всяких дополняющих нулей (пример приводится на рис. 9.3).

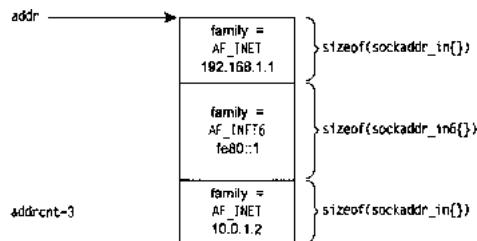


Рис. 9.3. Формат упакованного списка адресов для функций SCTP

Количество адресов, передаваемых `sctp_bindx`, указывается в параметре `addrcnt`. Параметр `flags` сообщает функции `sctp_bindx` о необходимости выполнения действий, перечисленных в табл. 9.1.

Таблица 9.1. Флаги функции `sctp_bindx`

Значение аргумента `flags` Описание

`SCTP_BINDEX_ADD_ADDR` Добавляет адреса к уже определенным для сокета

`SCTP_BINDEX_REMOVE_ADDR` Удаляет адреса из списка адресов сокета

Функцию `sctp_bindx` можно вызывать независимо от того, привязан ли сокет к каким-нибудь адресам. Для несвязанного сокета вызов `sctp_bindx` приведет к привязке указанного набора адресов. При работе с уже связанным сокетом указание флага `SCTP_BINDEX_ADD_ADDR` позволяет добавить адреса к данному дескриптору. Флаг `SCTP_BINDEX_REMOVE_ADDR` предназначен для удаления адресов из списка связанных с данным дескриптором. Если `sctp_bindx` вызывается для прослушиваемого сокета, новая конфигурация будет использоваться только для новых ассоциаций; вызов никак не затронет уже установленные ассоциации. Флаги `sctp_bindx` взаимно исключают друг друга: если указать оба, функция вернет ошибку `EINVAL`. Номер порта во всех структурах адреса сокета должен быть одним и тем же. Он должен совпадать с тем номером порта, который был связан с данным сокетом ранее. В противном случае `sctp_bindx` тоже вернет ошибку `EINVAL`.

Если конечная точка поддерживает динамическую адресацию, вызов `sctp_bindx` с флагом `SCTP_BINDEX_REMOVE_ADDR` или `SCTP_BINDEX_ADD_ADDR` приведет к передаче собеседнику сообщения о необходимости изменения списка адресов. Поскольку изменение списка адресов для установленной ассоциации не является обязательным, реализации, не поддерживающие эту функцию, будут при попытке ее использования возвращать ошибку `EOPNOTSUPP`. Обратите внимание, что для нормальной работы динамической адресации она должна поддерживаться обеими сторонами. Все это полезно в том случае, если система поддерживает динамическое предоставление интерфейсов: когда открывается доступ к новому интерфейсу Ethernet, приложение может вызвать `SCTP_BINDEX_ADD_ADDR` и начать работать с этим интерфейсом по уже установленным ассоциациям.

9.4. Функция sctp_connectx

```
#include <netinet/sctp.h>
```

```
int sctp_connectx(int sockfd, const struct sockaddr *addrs, int addrcnt);
```

Возвращает: 0 в случае успешного завершения, -1 в случае ошибки

Функция `sctp_connectx` используется для соединения с многоинтерфейсным узлом. При ее вызове мы должны указать адреса собеседника в параметре `addrs` (количество адресов определяется параметром `addrcnt`). Формат структуры `addrs` представлен на рис. 9.3. Стек SCTP устанавливает ассоциацию, используя один или несколько адресов из переданного списка. Все адреса `addrs` считаются действующими и подтвержденными.

9.5. Функция `sctp_getpaddrs`

Функция `getpeername` не предназначена для использования протоколом, рассчитанным на работу с многоинтерфейсными узлами. Для сокетов SCTP она способна вернуть лишь основной адрес собеседника. Если нужны все адреса, следует вызывать функцию `sctp_getpaddrs`.

```
#include <netinet/sctp.h>
```

```
int sctp_getpaddrs(int sockfd, sctp_assoc_t id, struct sockaddr **addrs);
```

Возвращает: 0 в случае успешного завершения, -1 в случае ошибки

Аргумент `sockfd` представляет собой дескриптор сокета, возвращаемый функцией `socket`. Второй аргумент задает идентификатор ассоциации для сокетов типа «один-ко-многим». Для сокетов типа «один-к-одному» этот аргумент игнорируется. `addrs` — адрес указателя, который функция `sctp_getpaddrs` заполнит упакованным списком адресов, выделив под него локальный буфер (см. рис. 9.3 и листинг 23.12). Для освобождения буфера, созданного `sctp_getpaddrs`, следует использовать вызов `sctp_freepaddrs`.

9.6. Функция `sctp_freepaddrs`

Функция `sctp_freepaddrs` освобождает ресурсы, выделенные вызовом `sctp_getpaddrs`.

```
#include <netinet/sctp.h>
```

```
void sctp_freepaddrs(struct sockaddr *addrs);
```

Здесь аргумент `addrs` — указатель на массив адресов, возвращаемый `sctp_getpaddrs`.

9.7. Функция `sctp_getladdrs`

Функция `sctp_getladdrs` может использоваться для получения списка локальных адресов, относящихся к определенной ассоциации. Эта функция бывает необходима в тех случаях, когда приложению требуется узнать, какие именно локальные адреса оно использует (набор адресов, напомним, может быть произвольным подмножеством всех адресов системы).

```
#include <netinet/sctp.h>
```

```
int sctp_getladdrs(int sockfd, sctp_assoc_t id, struct sockaddr **addrs);
```

Возвращает: количество локальных адресов, помещенных в addrs, или -1 в случае ошибки.

Здесь `sockfd` — дескриптор сокета, возвращаемый функцией `socket`. Аргумент `id` — идентификатор ассоциации для сокетов типа «один-ко-многим». Поле `id` игнорируется для сокетов типа «один-к-одному». Параметр представляет собой адрес указателя на буфер, выделяемый и заполняемый функцией `sctp_getladdrs`. В этот буфер помещается упакованный список адресов. Структура списка представлена на рис. 9.3 и в листинге 23.12. Для освобождения буфера процесс должен вызвать функцию `sctp_freladdrs`.

9.8. Функция `sctp_freladdrs`

Функция `sctp_freladdrs` освобождает ресурсы, выделенные при вызове `sctp_getladdrs`.

```
#include <netinet/sctp.h>
```

```
void sctp_freladdrs(struct sockaddr *addrs);
```

Здесь `addrs` указывает на список адресов, возвращаемый `sctp_getladdrs`.

9.9. Функция `sctp_sendmsg`

Приложение может управлять параметрами SCTP, используя функцию `sendmsg` со вспомогательными данными (см. главу 14). Однако из-за неудобств, связанных с применением вспомогательных данных, многие реализации SCTP предоставляют дополнительный библиотечный вызов (который на самом деле может быть и системным вызовом), упрощающий обращение к расширенным функциям SCTP. Вызов функции должен иметь следующий формат:

```
ssize_t sctp_sendmsg(int sockfd, const void *msg, size_t msgsz,
                      const struct sockaddr *to, socklen_t tolen, uint32_t ppid,
                      uint32_t flags, uint16_t stream, uint32_t timetolive,
                      uint32_t context);
```

Возвращает: количество записанных байтов в случае успешного завершения, -1 в случае ошибки

Использование `sctp_sendmsg` значительно упрощает отправку параметров, но требует указания большего количества аргументов. В поле `sockfd` помещается дескриптор сокета, возвращенный системным вызовом `socket`. Аргумент `msg` указывает на буфер размера `msgsz`, содержимое которого должно быть передано собеседнику. В поле `tolen` помещается длина адреса, передаваемого через аргумент `to`. В поле `ppid` помещается идентификатор протокола, который будет передан вместе с порцией данных. Поле `flags` передается стеку SCTP. Разрешенные значения этого поля приводятся в табл. 7.5.

Номер потока SCTP указывается вызывающим приложением в аргументе `stream`. Процесс может указать время жизни сообщения в миллисекундах в поле `timetolive`. Значение 0 соответствует бесконечному времени жизни. Пользовательский контекст, при наличии такого, может быть указан в поле `context`. Пользовательский контекст связывает неудачную передачу сообщения (о которой получено уведомление) с локальным контекстом, имеющим отношение к приложению. Например, чтобы отправить сообщение в поток 1 с флагом отправки `MSG_PR_SCTP_TTL`, временем жизни равным 1000 мс, идентификатором протокола 24 и контекстом 52, процесс должен сделать следующий вызов:

```
ret =
sctp_sendmsg(sockfd, data, datasz, &dest, sizeof(dest), 24,
              MSG_PR_SCTP_TTL, 1, 1000, 52);
```

Этот подход значительно проще выделения памяти под необходимые вспомогательные данные и настройки структур, входящих в `msghdr`. Обратите внимание, что если функция `sctp_sendmsg` реализована через вызов `sendmsg`, то поле `flags` в последнем устанавливается равным 0.

9.10. Функция `sctp_recvmsg`

Функция `sctp_recvmsg`, подобно `sctp_sendmsg`, предоставляет удобный интерфейс к расширенным возможностям SCTP. С ее помощью пользователь может получить не только адрес собеседника, но и поле `msg_flags`, которое обычно заполняется при вызове `recvmsg` (например, `MSG_NOTIFICATION`, `MSG_EOR` и так далее). Кроме того, функция дает возможность получить структуру `sctp_sndrcvinfo`, которая сопровождает сообщение, считанное в буфер. Обратите внимание, что если приложение хочет получать информацию, содержащуюся в структуре `sctp_sndrcvinfo`, оно должно быть подписано на событие `sctp_data_io_event` с параметром сокета `SCTP_EVENTS` (по умолчанию эта подписка включена).

```
ssize_t sctp_recvmsg(int sockfd, void *msg, size_t msgsz,
                     struct sockaddr *from, socklen_t *fromlen,
                     struct sctp_sndrcvinfo *sinfo, int *msg_flags);
```

Возвращает: количество считанных байтов в случае успешного завершения, -1 в случае ошибки

По возвращении из этого вызова аргумент `msg` оказывается заполненным не более, чем `msgsz` байтами данных. Адрес отправителя сообщения помещается в аргумент `from`, а размер адреса — в аргумент `fromlen`. Флаги сообщения будут помещены в аргумент `msg_flags`. Если уведомление `sctp_data_io_event` включено (а по умолчанию это так и есть), структура `sctp_sndrcvinfo` заполняется подробными сведениями о сообщении. Обратите внимание, что если функция `sctp_recvmsg` реализована через вызов `recvmsg`, то поле `flags` в последнем устанавливается равным нулю.

9.11. Функция `sctp_opt_info`

Эта функция предназначена для тех приложений, которым недостаточно возможностей, предоставляемых функциями `getsockopt` для протокола SCTP. Дело в том, что некоторые параметры сокетов SCTP (например, `SCTP_STATUS`) требуют использования переменных типа «значение-результат» для

передачи идентификатора ассоциации. Если функция `getsockopt` не поддерживает работу с такими переменными, разработчику придется вызывать `sctp_opt_info`. В системах типа FreeBSD, разрешающих указывать переменные типа «значение-результат» с параметрами сокетов, функция `sctp_opt_info` представляет собой оболочку, передающую аргументы функции `getsockopt` в нужном формате. В целях обеспечения переносимости разработчикам приложений рекомендуется использовать `sctp_opt_info` для всех параметров, требующих работы с переменными типа «значение-результат» (см. раздел 7.10).

```
int sctp_opt_info(int sockfd, sctp_assoc_t assoc_id, int opt,
                  void *arg, socklen_t *siz);
```

Возращает: 0 в случае успешного завершения, -1 в случае ошибки

Здесь `sockfd` — дескриптор сокета, с параметрами которого хочет работать пользователь. Аргумент `assoc_id` задает идентификатор ассоциации, которую нужно выделить из списка всех ассоциаций данного сокета. Аргумент `opt` задает параметр сокета для SCTP (список параметров приводится в разделе 7.10). `Arg` — аргумент параметра сокета, `siz` — указатель на переменную типа `socklen_t`, в которой хранится размер аргумента параметра сокета.

9.12. Функция `sctp_peeloff`

Как отмечалось ранее, любую ассоциацию, установленную через сокет типа «один-ко-многим», можно выделить в собственный сокет типа «один-к-одному». По семантике новая функция подобна `accept` с дополнительным аргументом. Процесс передает дескриптор `sockfd` сокета типа «один-ко-многим» и идентификатор `id` выделяемой ассоциации. Функция возвращает дескриптор нового сокета. Этот дескриптор имеет тип «один-к-одному», и он изначально связан с выбранной ассоциацией.

```
int sctp_peeloff(int sockfd, sctp_assoc_t id);
```

Возращает: дескриптор нового сокета в случае успешного завершения, -1 в случае ошибки

9.13. Функция `shutdown`

Обсуждавшаяся в разделе 9.6 функция `shutdown` может использоваться с конечной точкой SCTP, использующей интерфейс типа «один-к-одному». Поскольку архитектура SCTP не предусматривает наполовину закрытого состояния, реакция на вызов `shutdown` конечной точки SCTP отличается от реакции TCP. Когда конечная точка SCTP инициирует процедуру завершения ассоциации, оба собеседника должны закончить передачу данных, находящихся в очереди, после чего закрыть ассоциацию. Конечная точка, выполнявшая активное открытие, может вызвать `shutdown` вместо `close` для того, чтобы впоследствии подключиться к новому собеседнику. В отличие от TCP, закрывать сокет функцией `close`, а затем создавать его снова здесь не требуется. SCTP разрешает конечной точке вызвать `shutdown`, а после завершения этой функции — открывать новые ассоциации через тот же сокет. Обратите внимание, что если конечная точка не дождется завершения последовательности закрытия ассоциации, установка нового соединения закончится неудачей. На рис. 9.4 приведена типичная временная диаграмма вызовов для этого сценария.

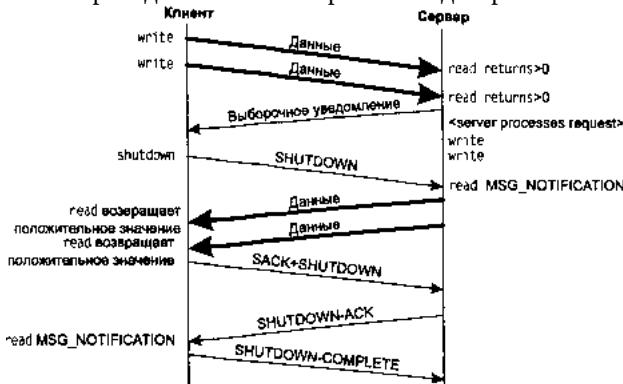


Рис. 9.4. Закрытие ассоциации SCTP вызовом `shutdown`

Обратите внимание, что на рис. 9.4 мы подразумеваем, что процесс подписан на события `MSG_NOTIFICATION`. Если же он не подписался на эти события, функция `read` считает нулевое количество

байтов. Результаты вызова `shutdown` для TCP были описаны в разделе 6.6. В документации *howto* на функцию `shutdown` для SCTP перечислены следующие константы:

- `SHUT_RD` — та же семантика, что и для TCP (см. раздел 6.6); никаких особых действий протокол SCTP не предусматривает;
- `SHUT_WR` — запрещает отправку сообщений и инициирует процедуру завершения ассоциации SCTP. Этот параметр не дает возможности работать в наполовину закрытом состоянии, однако позволяет локальной конечной точке считать все данные, которые собеседник отправит до получения сообщения SCTP `SHUTDOWN`;
- `SHUT_RDWR` — запрещает вызовы `read` и `write` и инициирует процедуру завершения ассоциации SCTP. Данные, передававшиеся в момент вызова `shutdown` на локальную конечную точку, будут подтверждены и сброшены без всякого уведомления процесса.

9.14. Уведомления

SCTP предоставляет разработчику приложений большое количество разнообразных уведомлений. С их помощью процесс может отслеживать состояние ассоциаций, с которыми он работает. Уведомления сообщают о событиях транспортного уровня, включая изменения состояния сети, установку ассоциаций, протокольные ошибки удаленного узла и неудачи при доставке сообщений. По умолчанию уведомления обо всех событиях отключены для сокетов обоих типов. Исключение делается для события `sctp_data_io_event`. Пример использования уведомлений будет приведен в разделе 23.7.

Параметр сокета `SCTP_EVENTS` позволяет подписаться на восемь событий. Из них семь штук генерируют дополнительные данные, которые процесс может получить через обычный дескриптор сокета. Уведомления добавляются к обычным данным, приходящим на соответствующий сокет, по мере того, как происходят события, генерирующие эти уведомления. При чтении из сокета, для которого включена подписка на уведомления, пользовательские данные и сообщения смешиваются друг с другом. Чтобы различить их, процесс должен использовать функции `recvmsg` или `sctp_recvmsg`. Для уведомлений о событиях поле `msg_flags` содержит флаг `MSG_NOTIFICATION`. Этот флаг говорит приложению о том, что считанное сообщение представляет собой не обычные данные, принятые от собеседника, а уведомление о каком-либо событии от локального стека SCTP.

Уведомление любого типа имеет следующий формат. Первые восемь байтов идентифицируют тип уведомления и его полную длину. Включение подписки на событие `sctp_data_io_event` приводит к тому, что с каждой операцией чтения пользовательских данных процесс принимает структуру `sctp_sndrcvinfo`. Вызовом `recvmsg` эта структура помещается во вспомогательные данные. Приложение может также вызвать `sctp_recvmsg`, которая использует указатель на структуру `sctp_sndrcvinfo`.

Два уведомления содержат поле кода причины ошибки SCTP (SCTP error cause field). Значения этого поля перечислены в разделе 3.3.10 RFC 2960 [118] и в разделе «CAUSE CODES» (коды причин) документа <http://www.iana.org/assignments/sctp-parameters>.

Уведомления определяются следующим образом.

```
struct sctp_tlv {  
    u_int16_t sn_type;  
    u_int16_t sn_flags;  
    u_int32_t sn_length;  
};  
  
/* уведомление о событии */  
union sctp_notification {  
    struct sctp_tlv sn_header;  
    struct sctp_assoc_change sn_assoc_change;  
    struct sctp_paddr_change sn_paddr_change;  
    struct sctp_remote_error sn_remote_error;  
    struct sctp_send_failed sn_send_failed;  
    struct sctp_shutdown_event sn_shutdown_event;  
    struct sctp_adaption_event sn_adaption_event;  
    struct sctp_pdapi_event sn_pdapi_event;  
};
```

Обратите внимание, что для интерпретации значения типа используется поле `sn_header`. Таблица 9.2 содержит значения, которые могут помещаться в поля `sn_header`, `sn_type`, а также соответствующие значения поля подписки, которые используются с параметром сокета `SCTP_EVENTS`.

Таблица 9.2. Тип и поле подписки

<code>sn_type</code>	Поле подписки
<code>SCTP_ASSOC_CHANGE</code>	<code>sctp_association_event</code>
<code>SCTP_PEER_ADDR_CHANGE</code>	<code>sctp_address_event</code>
<code>SCTP_REMOTE_ERROR</code>	<code>sctp_peer_error_event</code>
<code>SCTP_SEND_FAILED</code>	<code>sctp_send_failure_event</code>
<code>SCTP_SHUTDOWN_EVENT</code>	<code>sctp_shutdown_event</code>
<code>SCTP_ADAPTION_INDICATON</code>	<code>sctp_adaption_layer_event</code>
<code>SCTP_PARTIAL_DELIVERY_EVENT</code>	<code>sctp_partial_delivery_event</code>

У каждого уведомления имеется своя собственная структура, которая содержит подробную информацию о произошедшем событии.

■ **SCTP_ASSOC_CHANGE**

Это уведомление сообщает приложению о том, что произошло изменение, связанное с ассоциациями: возникла новая ассоциация или завершилась существующая. Структура данных имеет следующий формат:

```
struct sctp_assoc_change {
    u_int16_t sac_type;
    u_int16_t sac_flags;
    u_int32_t sac_length;
    u_int16_t sac_state;
    u_int16_t sac_error;
    u_int16_t sac_outbound_streams;
    u_int16_t sac_inbound_streams;
    sctp_assoc_t sac_assoc_id;
    uint8_t sac_info[];
};
```

Поле `sac_state` определяет тип события, связанного с ассоциацией. Оно может принимать следующие значения:

□ `SCTP_COMM_UP` — создана новая ассоциация. Поля входящих и исходящих потоков (`inbound_streams` и `outbound_streams`) говорят о том, сколько потоков доступно в соответствующих направлениях. Идентификатор ассоциации позволяет взаимодействовать со стеком SCTP;

□ `SCTP_COMM_LOST` — ассоциация закрыта из-за превышения порога недоступности (заданное количество раз был превышен тайм-аут) или собеседник выполнил аварийное закрытие ассоциации (при помощи параметра сокета `SO_LINGER` или вызовом `sendmsg` с флагом `MSG_ABORT`). Пользовательские данные могут быть помещены в поле `sac_info`;

□ `SCTP_RESTART` — собеседник перезапущен. Наиболее типичной причиной этого уведомления бывает выход из строя и перезапуск собеседника. Приложение должно проверить количество потоков в обоих направлениях, потому что при перезапуске эти значения могут измениться;

□ `SCTP_SHUTDOWN_COMP` — закончено завершение соединения, инициированное локальной конечной точкой (вызовом `shutdown` или `sendmsg` с флагом `MSG_EOF`). После получения этого сообщения сокет типа «один-к-одному» может быть использован для подключения к другому собеседнику;

□ `SCTP_CANT_STR_ASSOC` — собеседник не ответил при попытке установления ассоциации.

Поле `sac_error` содержит коды причин ошибок протокола SCTP, которые могли привести к изменению состояния ассоциации. Поля `sac_inbound_streams` и `sac_outbound_streams` говорят о том, какое количество потоков в каждом направлении было согласовано во время установки ассоциации. Поле `sac_assoc_id` содержит уникальный идентификатор ассоциации, который может использоваться как при работе с параметрами сокета, так и в последующих уведомлениях. Наконец, поле `sac_info` может содержать дополнительные пользовательские сведения. Например, если ассоциация была разорвана собеседником в связи с ошибкой, определенной пользователем, код этой ошибки будет помещен в поле `sac_info`.

■ **SCTP_PEER_ADDR_CHANGE** Это уведомление говорит об изменении состояния одного из адресов собеседника. Изменение может заключаться либо в отказе (отсутствии подтверждения отправленных на

этот адрес данных), либо в восстановлении (ответе отказавшего ранее адреса). Структура данных имеет следующий формат:

```
struct sctp_paddr_change {  
    u_int16_t spc_type;  
    u_int16_t spc_flags;  
    u_int32_t spc_length;  
    struct sockaddr_storage spc_aaddr;  
    u_int32_t spc_state;  
    u_int32_t spc_error;  
    sctp_assoc_t spc_assoc_id;  
};
```

Поле `spc_aaddr` содержит адрес собеседника, с которым связано данное событие. Поле `spc_state` может принимать одно из значений, перечисленных в табл. 9.3.

Таблица 9.3. Уведомление о состоянии адреса собеседника

spc_state	Значение
SCTP_ADDR_ADDED	Адрес добавлен к ассоциации
SCTP_ADDR_AVAILABLE	Адрес доступен
SCTP_ADDR_CONFIRMED	Адрес подтвержден и считается действующим
SCTP_ADDR_MADE_PRIM	Адрес сделан основным
SCTP_ADDR_REMOVED	Адрес удален из списка адресов ассоциации
SCTP_ADDR_UNREACHABLE	Адрес недоступен

Данные, отправленные на недоступный (`SCTP_ADDR_UNREACHABLE`) адрес, будут направляться на альтернативный адрес. Некоторые состояния доступны только в тех реализациях SCTP, которые поддерживают динамическую адресацию (в частности, `SCTP_ADDR_ADDED` и `SCTP_ADDR_REMOVED`).

Поле `spc_error` содержит код ошибки, дающий больше сведений о событии, а поле `spc_assoc_id`, как обычно, хранит идентификатор ассоциации.

■ **SCTP_REMOTE_ERROR**

Собеседник может отправить на локальную конечную точку сообщение об ошибке. Такие сообщения могут описывать различные ошибочные состояния ассоциации. Если это уведомление включено, вся сбояная порция данных передается приложению в сетевом формате. Сообщение имеет следующий формат:

```
struct sctp_remote_error {  
    u_int16_t sre_type;  
    u_int16_t sre_flags;  
    u_int32_t sre_length;  
    u_int16_t sre_error;  
    sctp_assoc_t sre_assoc_id;  
    u_int8_t sre_data[];  
};
```

Поле `sre_error` содержит код причины ошибки протокола SCTP; `sre_assoc_id` — идентификатор ассоциации, а `sre_data` — ошибочную порцию данных в сетевом формате.

■ **SCTP_SEND_FAILED**

Сообщение, которое невозможно доставить собеседнику, возвращается отправителю в этом уведомлении. За таким уведомлением обычно следует уведомление об отказе ассоциации. В большинстве случаев доставка сообщения оказывается невозможной именно по причине отказа ассоциации. Если же используется режим частичной надежности SCTP, сообщение может быть возвращено и в том случае, если отказа ассоциации реально не произошло.

Данные, возвращаемые приложению с этим уведомлением, имеют следующий формат:

```
struct sctp_send_failed {  
    u_int16_t ssf_type;  
    u_int16_t ssf_flags;  
    u_int32_t ssf_length;  
    u_int32_t ssf_error;  
    struct sctp_sndrcvinfo ssf_info;  
    sctp_assoc_t ssf_assoc_id;
```

```
    u_int8_t ssf_data[];  
};
```

Поле `ssf_flags` может иметь одно из двух значений:

- `SCTP_DATA_UNSENT` — сообщение не было послано собеседнику (управление потоком не позволило отправить сообщение до истечения его времени жизни);

- `SCTP_DATA_SENT` — сообщение было передано по крайней мере один раз, но собеседник не подтвердил его получение. Собеседник мог получить сообщение, но он не смог подтвердить его.

Эта разница может быть существенной для протоколов обработки транзакций, которые при восстановлении соединения могут предпринимать разные действия в зависимости от того, было принято конкретное сообщение или нет. Поле `ssf_error` может содержать код ошибки, относящейся к конкретному уведомлению, или быть нулевым. Поле `ssf_info` содержит сведения, переданные ядру при отправке данных (например, номер потока, контекст и так далее). Поле `ssf_assoc_id` содержит идентификатор ассоциации, а в поле `ssf_data` помещается недоставленное сообщение.

■ `SCTP_SHUTDOWN_EVENT`

Это уведомление передается приложению при приеме от собеседника порции `SHUTDOWN`. После этой порции никакие новые данные на том же сокете получены быть не могут. Все данные, уже помещенные в очередь, будут переданы собеседнику, после чего ассоциация будет закрыта. Уведомление имеет следующий формат:

```
struct sctp_shutdown_event {  
    uint16_t sse_type;  
    uint16_t sse_flags;  
    uint32_t sse_length;  
    sctp_assoc_t sse_assoc_id;  
};
```

Поле `sse_assoc_id` содержит идентификатор ассоциации, которая закрывается и потому не может более использоваться для передачи данных.

■ `SCTP_ADAPTION_INDICATION`

Некоторые реализации поддерживают параметр *индикации адаптирующего уровня* (*adaption layer indication*). Этот параметр передается в пакетах `INIT` и `INIT-ACK` и уведомляет собеседника о выполняемой адаптации приложения. Уведомление имеет следующий формат:

```
struct sctp_adaption_event {  
    u_int16_t sai_type;  
    u_int16_t sai_flags;  
    u_int32_t sai_length;  
    u_int32_t sai_adaption_ind;  
    sctp_assoc_t sai_assoc_id;  
};
```

Поле `sai_assoc_id` содержит обычный идентификатор ассоциации. Поле `sai_adaption_ind` представляет собой 32-разрядное целое число, переданное собеседником локальной конечной точке в сообщении `INIT` или `INIT-ACK`. Уровень адаптации для исходящих сообщений устанавливается при помощи параметра сокета `SCTP_ADAPTION_LAYER` (см. раздел 7.10). Все это описано в стандарте [116], а пример использования параметра для удаленного прямого доступа к памяти и прямой записи данных описывается в [115].

■ `SCTP_PARTIAL_DELIVERY_EVENT`

Интерфейс частичной доставки используется для передачи больших сообщений пользователю через буфер сокета. Представьте, что процесс отправил сообщение размером 4 Мбайт. Сообщение такого размера может сильно перегрузить системные ресурсы. Реализация `SCTP` не смогла бы обработать такое сообщение, если бы у нее не было механизма доставки сообщений по частям до полного их получения. Реализация, обеспечивающая частичную доставку, называется *интерфейсом частичной доставки* (*partial delivery API*). `SCTP` передает данные приложению, не устанавливая флаги в поле `msg_flags` до тех пор, пока не будет готов последний сегмент сообщения. Для этого сегмента устанавливается флаг `MSG_EOR` (конец записи). Обратите внимание, что если приложение рассчитывает принимать большие сообщения, оно должно использовать функции `recvmsg` и `sctp_recvmsg`, чтобы иметь возможность проверять поле `msg_flags` на наличие флага окончания записи.

В некоторых ситуациях интерфейсу частичной доставки может потребоваться информировать приложение о состоянии сообщения. Например, если при доставке большого сообщения произошел сбой,

приложению доставляется уведомление SCTP_PARTIAL_DELIVERY_EVENT, имеющее следующий формат:

```
struct sctp_pdapi_event {  
    uint16_t pdapi_type;  
    uint16_t pdapi_flags;  
    uint32_t pdapi_length;  
    uint32_t pdapi_indication;  
    sctp_assoc_t pdapi_assoc_id;  
};
```

Идентификатор pdapi_assoc_id указывает на ассоциацию, к которой относится принятое уведомление. Поле pdapi_indication содержит сведения о произошедшем событии. На данный момент поле может иметь единственное значение SCTP_PARTIAL_DELIVERY_ABORTED, указывающее на аварийное завершение частичной доставки сообщения, обрабатываемого в данный момент.

9.15. Резюме

SCTP предлагает разработчику приложений два вида интерфейсов: «один-к-одному», облегчающий миграцию существующих TCP-приложений на SCTP, и «один-ко-многим», реализующий все новые возможности SCTP. Функция sctp_peeloff позволяет выделять ассоциации из множественных сокетов в одиночные. Кроме того, SCTP предоставляет множество уведомлений о событиях транспортного уровня, на которые приложение при необходимости может подписываться. События помогают приложению управлять ассоциациями, с которыми оно работает.

Поскольку протокол SCTP ориентирован на многоинтерфейсные узлы, не все стандартные функции сокетов, рассмотренные в главе 4, оказываются эффективны при работе с ним. Функции sctp_bindx, sctp_connectx, sctp_getladdrs и sctp_getpaddrs позволяют управлять адресами и ассоциациями. Функции sctp_sendmsg и sctp_recvmsg упрощают использование расширенных возможностей SCTP. В главах 10 и 23 мы приведем примеры, наглядно демонстрирующие рассмотренные в этой главе новые концепции.

Упражнения

1. В какой ситуации разработчик приложения скорее всего воспользуется функцией sctp_peeloff?
2. Говоря о сокетах типа «один-ко-многим», мы утверждаем, что на стороне сервера также происходит автоматическое закрытие. Почему это верно?
3. Почему передача пользовательских данных в третьем пакете рукопожатия возможна только для сокетов типа «один-ко-многим»? (Подсказка: нужно иметь возможность отправлять данные во время установки ассоциации.)
4. В какой ситуации пользовательские данные могут быть переданы в третьем и четвертом пакетах четырехэтапного рукопожатия?
5. В разделе 9.7 говорится о том, что набор локальных адресов может быть подмножеством связанных адресов. В какой ситуации это возможно?

Глава 10

Пример SCTP-соединения клиент-сервер

10.1. Введение

Воспользуемся некоторыми элементарными функциями из глав 4 и 9 для написания полнофункционального приложения SCTP с архитектурой клиент-сервер типа «один-ко-многим». Сервер из нашего примера будет аналогичен эхо-серверу из главы 5. Приложение будет функционировать следующим образом:

1. Клиент считывает строку текста из стандартного потока ввода и отсылает ее серверу. Стока имеет формат `[#]text`, где номер в скобках обозначает номер потока SCTP, по которому должно быть отправлено это текстовое сообщение.
2. Сервер принимает текстовое сообщение из сети, увеличивает номер потока, по которому было получено сообщение, на единицу и отправляет сообщение обратно клиенту через поток с новым номером.
3. Клиент считывает полученную строку и выводит ее в стандартный поток вывода, добавляя к ней номер потока и порядковый номер для данного потока.

Наше приложение вместе с функциями, используемыми для операций ввода и вывода, изображено на рис. 10.1.

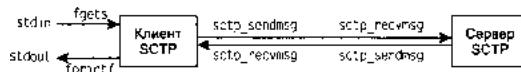


Рис. 10.1. Простое потоковое приложение SCTP с архитектурой клиент-сервер

Две стрелки между клиентом и сервером обозначают два односторонних потока (ассоциация в целом является полностью двусторонней). Функции `fgets` и `fputs` входят в стандартную библиотеку ввода-вывода. Мы не пользуемся функциями `written` и `readline` из раздела 3.9, потому что в них нет необходимости. Вместо них мы вызываем `sctp_sendmsg` и `sctp_recvmsg` из разделов 9.9 и 9.10 соответственно.

Сервер в нашем примере будет относиться к типу «один-ко-многим». Этот вариант был выбран нами по одной важной причине. Примеры из главы 5 могут быть переделаны под SCTP внесением крайне незначительных изменений: достаточно изменить вызов `socket`, указав в качестве третьего аргумента `IPPROTO_SCTP` вместо `IPPROTO_TCP`. Однако приложение, полученное таким образом, не использовало бы дополнительные возможности, предоставляемые SCTP, за исключением поддержки многоинтерфейсных узлов. Написав сервер типа «один-ко-многим», мы смогли показать все достоинства SCTP.

10.2. Потоковый эхо-сервер SCTP типа «один-ко-многим»: функция main

Наши клиент и сервер SCTP вызывают функции в последовательности, представленной на рис. 9.2. Код последовательного сервера представлен в листинге 10.1^[1].

Листинг 10.1. Потоковый эхо-сервер SCTP

```
//sctp/sctpserv01.c
1 #include "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int sock_fd, msg_flags;
6     char readbuf[BUFFSIZE];
7     struct sockaddr_in servaddr, cliaddr;
8     struct sctp_sndrcvinfo sri;
9     struct sctp_event_subscribe evnts;
10    int stream_increment=1;
11    socklen_t len;
12    size_t rd_sz;
13    if (argc == 2)
```

```

14     stream_increment = atoi(argv[1]);
15     sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
16     bzero(&servaddr, sizeof(servaddr));
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19     servaddr.sin_port = htons(SERV_PORT);

20     Bind(sock_fd, (SA*)&servaddr, sizeof(servaddr));

21     bzero(&evnts, sizeof(evnts));
22     evnts.sctp_data_io_event = 1;
23     Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof(evnts));

24     Listen(sock_fd, LISTENQ);
25     for (;;) {
26         len = sizeof(struct sockaddr_in);
27         rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof(readbuf),
28             (SA*)&cliaddr, &len, &sri, &msg_flags);
29         if (stream_increment) {
30             sri.sinfo_stream++;
31             if (sri.sinfo_stream >=
32                 sctp_get_no_streams(sock_fd, (SA*)&cliaddr, len))
33                 sri.sinfo_stream = 0;
34         }
35         Sctp_sendmsg(sock_fd, readbuf, rd_sz,
36             (SA*)&cliaddr, len,
37             sri.sinfo_ppid,
38             sri.sinfo_flags, sri.sinfo_stream, 0, 0);
39     }
40 }
```

Настройка приращения номера потока

13-14 По умолчанию наш сервер отвечает клиенту через поток, номер которого на единицу больше номера потока, по которому было получено сообщение. Если приложению в строке вызова передается целочисленный аргумент, он интерпретируется как значение флага *stream_increment*, с помощью которого приращение номера потока можно отключить. Мы воспользуемся этим параметром командной строки, когда будем говорить о блокировании в разделе 10.5.

Создание сокета SCTP

15 Создается сокет SCTP типа «один-ко-многим».

Связывание с адресом

16-20 Структура адреса сокета Интернета заполняется универсальным адресом (*INADDR_ANY*) и номером заранее известного порта сервера *SERV_PORT*. Связывание с универсальным адресом означает, что конечная точка SCTP будет использовать все доступные локальные адреса для всех создаваемых ассоциаций. Для многоинтерфейсных узлов это означает, что удаленная конечная точка сможет устанавливать ассоциации и передавать пакеты на любой локальный интерфейс. Выбор номера порта SCTP основывался на рис. 2.10. Обратите внимание, что ход рассуждений для сервера тот же, что и в одном из предшествовавших примеров в разделе 5.2.

Подписка на уведомления

21-23 Сервер изменяет параметры подписки на уведомления для сокета SCTP. Сервер подписывается только на событие `sctp_data_io_event`, что позволяет ему получать структуру `sctp_sndrcvinfo`. По ее содержимому сервер сможет определять номер потока полученного сообщения.

Разрешение установки входящих ассоциаций

24 Сервер разрешает устанавливать входящие ассоциации, вызывая функцию `listen`. Затем управление передается главному циклу.

Ожидание сообщения

26-28 Сервер инициализирует размер структуры адреса сокета клиента, после чего блокируется в ожидании сообщения от какого-либо удаленного собеседника.

Увеличение номера потока

29-34 Сервер проверяет состояние флага `stream_increment` и определяет, нужно ли увеличивать номер потока. Если флаг установлен (никакие аргументы в командной строке не передавались), сервер увеличивает номер потока, по которому было получено сообщение, на единицу. Если полученное число достигает предельного количества потоков (получаемого вызовом `sctp_get_no_strms`), сервер сбрасывает номер потока в 0. Функция `sctp_get_no_strms` в листинге не приведена. Она использует параметр `SCTP_STATUS` (см. раздел 7.10) для определения согласованного количества потоков.

Отправка ответа

35-38 Сервер отсылает сообщения, используя идентификатор протокола, флаги и номер потока (который, возможно, был увеличен), хранящиеся в структуре `sri`.

Заметьте, что нашему серверу не нужны уведомления об установке ассоциаций, поэтому он отключает все события, которые привели бы к передаче сообщений в буфер сокета. Сервер полагается на сведения из структуры `sctp_sndrcvinfo`, а обратный адрес берет из переменной `cliaddr`. Этого оказывается достаточно для отправки эхо-ответа собеседнику через установленную им ассоциацию.

Программа работает до тех пор, пока пользователь не завершит ее передачей сигнала.

10.3. Потоковый эхо-клиент SCTP типа «один-ко-многим»: функция `main`

В листинге 10.2 приведена функция `main` нашего клиента SCTP.

Листинг 10.2. Потоковый эхо-клиент SCTP

```
//sctp/sctpclient01.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sock_fd;
6     struct sockaddr_in servaddr;
7     struct sctp_event_subscribe evnts;
8     int echo_to_all=0;

9     if (argc < 2)
```

```

10    err_quit("Missing host argument - use '%s host [echo]'\n", argv[0]);
11  if (argc > 2) {
12    printf("Echoing messages to all streams\n");
13    echo_to_all = 1;
14  }
15  sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
16  bzero(&servaddr, sizeof(servaddr));
17  servaddr.sin_family = AF_INET;
18  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19  servaddr.sin_port = htons(SERV_PORT);
20  Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

21  bzero(&evnts, sizeof(evnts));
22  evnts.sctp_data_io_event = 1;
23  Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof(evnts));
24  if (echo_to_all == 0)
25    sctpstr_cli(stdin, sock_fd, (SA*)&servaddr, sizeof(servaddr));
26  else
27    sctpstr_cli_echoall(stdin, sock_fd, (SA*)&servaddr,
28    sizeof(servaddr));
29  Close(sock_fd);
30  return(0);
31 }
```

Проверка аргументов и создание сокета

9-15 Клиент проверяет переданные ему при запуске аргументы командной строки. Сначала проверяется, указан ли в строке IP-адрес узла, на который нужно отправлять сообщения. Затем проверяется, указан ли параметр отправки эхо-сообщений всем (мы воспользуемся им в разделе 10.5). Наконец, клиент создает сокет SCTP типа «один-ко-многим».

Подготовка адреса сервера

16-20 Клиент преобразует IP-адрес сервера, переданный ему в командной строке, с помощью функции `inet_pton`. К адресу он добавляет заранее известный номер порта сервера. Полученная структура используется для всех обращений к данному серверу.

Подписка на уведомления

21-23 Клиент явно указывает, какие именно уведомления он хочет получать от созданного сокета SCTP. События `MSG_NOTIFICATION` ему не нужны, поэтому он отключает их, оставляя лишь структуру `sctp_sndrcvinfo`.

Вызов функции обработки сообщений

24-28 Если флаг `echo_to_all` не установлен, клиент вызывает функцию `sctpstr_cli`, которая будет обсуждаться в разделе 10.4. В противном случае вызывается `sctpstr_cli_echoall` (раздел 10.5, где рассматривается применение потоков SCTP).

Завершение работы

29-31 Закончив работу с сообщениями, клиент закрывает сокет SCTP, что приводит к закрытию всех ассоциаций, использующих этот сокет. Затем функция `main` завершается и возвращает код 0 — никаких ошибок не произошло.

10.4. Потоковый эхо-клиент SCTP: функция str_cli

В листинге 10.3 приведена основная функция эхо-клиента SCTP.

Листинг 10.3. Функция sctp_strcli

```
//sctp/sctp_strcli.c
1 #include "unp.h"

2 void
3 sctpstr_cli(FILE *fp, int sock_fd, struct sockaddr *to, socklen_t tolen)
4 {
5     struct sockaddr_in peeraddr;
6     struct sctp_sndrcvinfo sri;
7     char sendline[MAXLINE], recvline[MAXLINE];
8     socklen_t len;
9     int out_sz, rd_sz;
10    int msg_flags;

11    bzero(&sri, sizeof(sri));
12    while (fgets(sendline, MAXLINE, fp) != NULL) {
13        if (sendline[0] != '[') {
14            printf("Error, line must be of the form '[streamnum]text'\n");
15            continue;
16        }
17        sri.sinfo_stream = strtol(&sendline[1], NULL, 0);
18        out_sz = strlen(sendline);
19        Sctp_sendmsg(sock_fd, sendline, out_sz,
20                      to, tolen, 0, 0, sri.sinfo_stream, 0, 0);

21        len = sizeof(peeraddr);
22        rd_sz = Sctp_recvmsg(sock_fd, recvline, sizeof(recvline),
23                             (SA*)&peeraddr, &len, &sri, &msg_flags);
24        printf("From str:%d seq:%d (assoc:0x%lx):",
25               sri.sinfo_stream.sri.sinfo_ssn, (u_int)sri.sinfo_assoc_id);
26        printf("%*s", rd_sz.recvline);
27    }
28 }
```

Инициализация структуры sri и вход в цикл

11-12 Основная функция клиента начинает работу с очистки структуры `sctp_sndrcvinfo` (переменная `sri`). Затем функция входит в цикл,читывающий из дескриптора `fp`, переданного вызывающей функцией, при помощи блокирующего вызова `fgets`. Главная программа (`main`) передает этой функции `stdin` в качестве аргумента `fp`, поэтому функция считывает и обрабатывает пользовательский ввод до тех пор, пока пользователь не введет завершающий EOF (Ctrl+D). При этом функция завершается и управление передается вызвавшей функции.

Проверка ввода

13-16 Клиент проверяет введенный пользователем текст на соответствие шаблону [#]текст. Если формат строки нарушен, клиент выводит сообщение об ошибке и снова вызывает `fgets`.

Преобразование номера потока

17 Клиент записывает запрошенный пользователем номер потока из текстовой строки в поле `sinfo_stream` структуры `sri`.

Отправка сообщения

18-20 После инициализации длины структуры адреса и размера пользовательских данных клиент отсылает сообщение серверу при помощи функции `sctp_sendmsg`.

Блокирование в ожидании ответа

21-23 Клиент блокируется и ожидает получения эхо-ответа сервера.

Отображение полученного эхо-ответа

24-26 Клиент выводит на экран полученное от сервера сообщение, вместе с номером потока и последовательным номером сообщения в этом потоке. После этого клиент возвращается на начало цикла, ожидая, что пользователь введет следующую строку.

Запуск программы

Мы запустили эхо-сервер SCTP без аргументов командной строки на компьютере, работающем под управлением FreeBSD. Клиенту при запуске необходимо указать IP-адрес сервера.

```
freebsd4% sctpclient01 10.1.1.5
[0]Hello                                Отправка сообщения по потоку 0
From str:1 seq:0 (assoc:0xc99e15a0):[0]Hello Эхо-ответ сервера в потоке 1
[4]Message two                            Отправка сообщения по потоку 4
From str:5 seq:0 (assoc.0xc99e15a0):[4]Message two Эхо-ответ сервера
                                         в потоке 5
[4]Message three                          Отправка сообщения по потоку 4
From str:5 seq:1 (assoc 0xc99e15a0):[4]Message three Эхо-ответ сервера
                                         в потоке 5
^D                                       Ввод символа EOF
freebsd4%
```

Обратите внимание, что клиент отправляет сообщения по потокам 0 и 4, а сервер отвечает ему по потокам 1 и 5. Именно такое поведение и ожидается в том случае, когда наш сервер запускается без аргументов командной строки. Заметьте также, что порядковый номер сообщения по пятому потоку увеличился на единицу при приеме третьего сообщения, как и должно было произойти.

10.5. Блокирование очереди

Наш сервер позволяет отправлять текстовые сообщения по любому из нескольких потоков. Поток SCTP — это вовсе не поток байтов, как в TCP. Это последовательность сообщений, упорядоченных в пределах ассоциации. Потоки с собственным порядком используются для того, чтобы обойти блокирование очереди (*head-of-line blocking*), которое может возникать в TCP.

Блокирование возникает при потере сегмента TCP при передаче и приходе следующего за ним сегмента, который удерживается до тех пор, пока утраченный сегмент не будет передан повторно и получен адресатом. Задержка доставки последующих сегментов гарантирует, что приложение получит данные в том порядке, в котором они были отправлены. Это совершенно необходимая функция, которая, к сожалению, обладает определенными недостатками. Представьте, что семантически независимые сообщения передаются по одному соединению TCP. Например, веб-сервер может передать браузеру три картинки для

отображения на экране. Чтобы картинки выводились на экран одновременно, сервер передает сначала часть первого изображения, затем часть второго и часть третьего. Процесс повторяется до тех пор, пока все три картинки не будут переданы клиенту целиком. Что произойдет, если потерянется сегмент TCP, относящийся к первому изображению? Клиент не получит никаких данных до тех пор, пока недостающий сегмент не будет передан повторно и доставлен ему. Задержаны будут все три изображения, хотя сегмент относился только к одному из них (первому). Этую ситуацию иллюстрирует рис. 10.2.

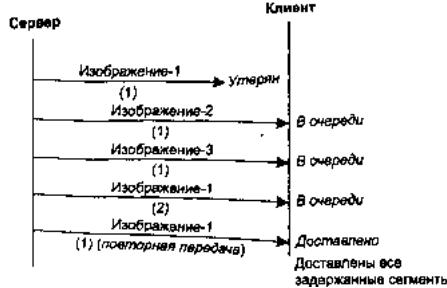


Рис. 10.2. Отправка трех изображений по одному TCP-соединению

ПРИМЕЧАНИЕ

Хотя HTTP работает иначе, были предложены расширения этого протокола, такие как SCP [108] и SMUX [33], которые обеспечивают описанную функциональность поверх TCP. Эти протоколы мультиплексирования позволяют избежать проблем, связанных с параллельными TCP-соединениями, не имеющими общей информации о состоянии [123]. Несмотря на то что создание одного TCP-соединения для каждого изображения (как обычно и делают клиенты HTTP) позволяет избежать блокирования, каждому соединению приходится тратить время на определение времени обращения и доступной пропускной способности. Потеря сегмента, относящегося к одному соединению (признак затора на линии) не обязательно приводит к замедлению передачи по остальным соединениям. В результате совокупное использование загруженных сетей падает.

Для приложения было бы лучше, если бы транспортный протокол вел себя иначе. В идеале задерживаться должны только сегменты первой картинки, тогда как сегменты второй и третьей должны доставляться так, как если бы сегмент первой картинки не был утерян вовсе.

Многопоточный режим SCTP позволяет свести к минимуму блокирование очереди. На рис. 10.3 мы показываем процесс отправки тех же трех изображений. На этот раз сервер использует потоки, так что блокируется только одно изображение, а второе и третье доставляются без помех. Первое изображение не доставляется до тех пор, пока не будет восстановлен порядок сегментов.



Рис. 10.3. Отправка трех изображений по потокам SCTP

Теперь мы можем привести полный код нашего клиента (с функцией `sctpstr_cli_echoall`, листинг 10.4), чтобы на его примере продемонстрировать устранение проблем с блокированием очереди при помощи SCTP. Новая функция аналогична `sctpstr_cli` за тем исключением, что клиент больше не требует указания номера потока в квадратных скобках в каждом сообщении. Функция передает сообщение пользователя по всем потокам, количество которых определяется константой `SERV_MAX_SCTP_STRM`. После отправки сообщения клиент ждет прихода всех ответных сообщений сервера. Запуская сервер, мы

передаем ему аргумент командной строки, указывающий на то, что сервер должен отвечать на сообщения по тем же потокам, по которым они приходят. Это позволяет пользователю отслеживать ответы и порядок их прибытия.

Листинг 10.4. Функция sctp_strcliecho

```
1 #include "unp.h"

2 #define SCTP_MAXLINE 800

3 void
4 sctpstr_cli_echoall(FILE *fp, int sock_fd, struct sockaddr to,
5     socklen_t tolen)
6 {
7     struct sockaddr_in peeraddr;
8     struct sctp_sndrcvinfo sri;
9     char sendline[SCTP_MAXLINE], recvline[SCTP_MAXLINE];
10    socklen_t len;
11    int rd_sz, i, strsz;
12    int msg_flags;

13    bzero(sendline, sizeof(sendline));
14    bzero(&sri, sizeof(sri));
15    while (fgets(sendline, SCTP_MAXLINE - 9, fp) != NULL) {
16        strsz = strlen(sendline);
17        if (sendline[strsz-1] == '\n') {
18            sendline[strsz-1] = '\0';
19            strsz--;
20        }
21        for (i=0; i<SERV_MAX_SCTP_STRM; i++) {
22            snprintf(sendline + strsz, sizeof(sendline) - strsz,
23                ".msg %d", i);
24            Sctp_sendmsg(sock_fd, sendline, sizeof(sendline),
25                to, tolen, 0, 0, i, 0, 0);
26        }
27        for (i =0; i < SERV_MAX_SCTP_STRM; i++) {
28            len = sizeof(peeraddr);
29            rd_sz = Sctp_recvmsg(sock_fd, recvline, sizeof(recvline),
30                (SA*)&peeraddr, &len, &sri, &msg_flags);
31            printf("From str:%d seq:%d (assoc:0x%08x)",
32                sri.sinfo_stream, sri.sinfo_ssn,
33                (u_int)sri.sinfo_assoc_id);
34            printf("%.8s\n", rd_sz, recvline);
35        }
36    }
37 }
```

Инициализация структур данных и ожидание ввода

13-15 Как и в предыдущем примере, клиент инициализирует структуру `sri`, предназначенную для настройки потока, с которым клиент будет работать. Кроме того, клиент обнуляет буфер данных, из которого считывается пользовательский ввод. Затем программа входит в основной цикл, блокируясь в вызове `fgets`.

Предварительная обработка сообщения

16-20 Клиент определяет размер сообщения и удаляет символ перевода строки, если таковой находится в конце буфера.

Отправка сообщения по всем потокам

21-26 Клиент отсылает сообщение с помощью функции `sctp_sendmsg`. Передается все содержимое буфера длиной `SCTP_MAXLINE`. Перед отправкой сообщения к нему добавляется строка `.msg`, и номер потока, чтобы мы могли впоследствии определить порядок получения сообщений и сравнить его с порядком отправки сообщений. Обратите внимание, что клиент отправляет сообщения по заданному количеству потоков, не проверяя, сколько потоков было согласовано с сервером. Может получиться так, что некоторые операции отправки сообщений завершатся с ошибкой, если количество потоков будет снижено по запросу собеседника.

ПРИМЕЧАНИЕ

Этот код может работать неправильно, если окна приема и отправки будут слишком малы. Если окно приема собеседника слишком мало, клиент может заблокироваться. Поскольку клиент не переходит к считыванию данных из приемного буфера, пока он не отправит все сообщения, сервер также может заблокироваться в ожидании освобождения буфера клиента. В результате обе конечные точки SCTP зависнут. Приведенная программа не рассчитана на масштабирование. Она предназначена лишь для иллюстрации потоков и блокирования очередей в простейшем варианте.

Считывание и отображение эхо-ответа

27-35 Клиент блокируется в цикле считывания всех ответных сообщений сервера, которые отображаются на экране по мере поступления. После считывания последнего сообщения сервера клиент возвращается к обработке ввода пользователя.

Запуск программы

Мы запустили клиент и сервер на разных компьютерах с FreeBSD, между которыми был установлен настраиваемый маршрутизатор (рис. 10.4). Маршрутизатор может создавать задержку и сбрасывать часть пакетов. Сначала мы запускаем программу без сброса пакетов на маршрутизаторе.

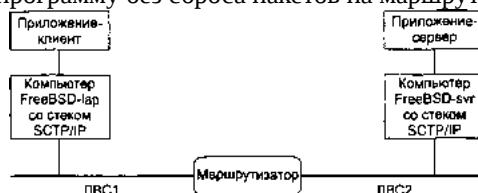


Рис. 10.4. Тестовая конфигурация сети

Мы запускаем сервер с аргументом 0 в командной строке, благодаря чему сервер не увеличивает номер потока при отправке эхо-ответа.

Затем мы запускаем клиент, передавая ему в командной строке адрес эхо-сервера и дополнительный аргумент, указывающий на необходимость отправки сообщений по всем потокам одновременно.

```
freebsd4% sctpclient01 10.1.4.1 echo
Echoing messages to all streams
Hello
From str:0 seq:0 (assoc:0xc99e15a0):Hello.msg.0
From str:1 seq:0 (assoc.0xc99e15a0):Hello.msg.1
From str:2 seq:0 (assoc:0xc99e15a0):Hello.msg.2
From str:3 seq:0 (assoc 0xc99e15a0):Hello.msg.3
From str:4 seq:0 (assoc.0xc99e15a0):Hello.msg.4
```

```
From str:5 seq:0 (assoc:0xc99e15a0):Hello.msg.5
From str:6 seq:0 (assoc:0xc99e15a0):Hello.msg.6
From str:7 seq:0 (assoc:0xc99e15a0):Hello.msg.7
From str:8 seq:0 (assoc:0xc99e15a0):Hello.msg.8
From str:9 seq:0 (assoc:0xc99e15a0).Hello.msg.9
^D
freebsd4%
```

В отсутствие потерь при передаче клиент получает ответы сервера в том же порядке, в котором отправляет запросы. Изменим параметры маршрутизатора таким образом, чтобы терять 10% всех пакетов, передаваемых в обоих направлениях, и перезапустим клиент.

```
freebsd4% sctpclient01 10.1.4.1 echo
Echoing messages to all streams
Hello
From str:0 seq:0 (assoc:0xc99e15a0):Hello.msg.0
From str:2 seq:0 (assoc:0xc99e15a0):Hello.msg.2
From str:3 seq:0 (assoc:0xc99e15a0):Hello.msg.3
From str:5 seq:0 (assoc:0xc99e15a0):Hello.msg.5
From str:1 seq:0 (assoc:0xc99e15a0):Hello.msg.1
From str:8 seq:0 (assoc:0xc99e15a0):Hello.msg.8
From str:4 seq:0 (assoc:0xc99e15a0):Hello.msg.4
From str:7 seq:0 (assoc:0xc99e15a0):Hello.msg.7
From str:9 seq:0 (assoc:0xc99e15a0):Hello.msg.9
From str:6 seq:0 (assoc:0xc99e15a0):Hello msg.6
^D
freebsd4%
```

Можно проверить, действительно ли сообщения в каждом из потоков доставляются в правильном порядке, если изменить клиента так, чтобы он отправлял по два сообщения в поток. Кроме того, мы добавим к сообщению суффикс с его номером, чтобы отличать эхо-ответы друг от друга. Измененная функция клиента представлена в листинге 10.5.

Листинг 10.5. Изменения в функции sctp_strcliecho

```
//sctp/sctp_strcliecho2.c
21 for (i =0; i < SERV_MAX_SCTP_STRM; i++) {
22     snprintf(sendline + strsz, sizeof(sendline) - strsz,
23             ".msg.%d 1", i);
24     Sctp_sendmsg(sock_fd, sendline, sizeof(sendline),
25                  to, tolen, 0, 0, i, 0, 0);
26     snprintf(sendline + strsz, sizeof(sendline) - strsz,
27             ".msg.%d 2", i);
28     Sctp_sendmsg(sock_fd, sendline, sizeof(sendline),
29                  to, tolen, 0, 0, i, 0, 0);
30 }
31 for (i = 0; i < SERV_MAX_SCTP_STRM*2, i++) {
32     len = sizeof(peeraddr);
```

Первое сообщение: добавление номера и отправка

22-25 Клиент добавляет к первому сообщению его номер, с помощью которого мы сможем отслеживать отправленные сообщения. Затем сообщение отсылается вызовом sctp_sendmsg.

Второе сообщение: добавление номера и отправка

26-29 Номер сообщения изменяется с единицы на двойку, после чего сообщение отсылается по тому же потоку.

Считывание и отображение эхо-ответа

31 Здесь требуется лишь одно незначительное изменение: количество ожидаемых ответов эхо-сервера должно быть удвоено.

Запуск измененной программы

Запустив сервер и измененный клиент, мы получаем следующий результат:

```
freebsd4% sctpclient01 10.1.4.1 echo
Echoing messages to all streams
Hello
From str:0 seq:0 (assoc:0xc99e15a0):Hello.msg.0 1
From str:0 seq:1 (assoc:0xc99e15a0):Hello.msg.0 2
From str:1 seq:0 (assoc:0xc99e15a0):Hello.msg.1 1
From str:4 seq:0 (assoc:0xc99e15a0):Hello.msg.4 1
From str:5 seq:0 (assoc:0xc99e15a0):Hello.msg.5 1
From str:7 seq:0 (assoc:0xc99e15a0):Hello.msg.7 1
From str:8 seq:0 (assoc:0xc99e15a0):Hello.msg.8 1
From str:9 seq:0 (assoc:0xc99e15a0):Hello.msg.9 1
From str:3 seq:0 (assoc:0xc99e15a0):Hello.msg.3 1
From str:3 seq:0 (assoc:0xc99e15a0):Hello.msg.3 2
From str:1 seq:0 (assoc:0xc99e15a0):Hello.msg.1 2
From str:5 seq:0 (assoc:0xc99e15a0):Hello.msg.5 2
From str:2 seq:0 (assoc:0xc99e15a0):Hello.msg.2 1
From str:6 seq:0 (assoc:0xc99e15a0):Hello.msg.6 1
From str:6 seq:0 (assoc:0xc99e15a0):Hello.msg.6 2
From str:2 seq:0 (assoc:0xc99e15a0):Hello.msg.2 2
From str:7 seq:0 (assoc:0xc99e15a0):Hello.msg.7 2
From str:8 seq:0 (assoc:0xc99e15a0):Hello.msg.8 2
From str:9 seq:0 (assoc:0xc99e15a0):Hello.msg.9 2
From str:4 seq:0 (assoc:0xc99e15a0):Hello.msg.4 2
^D
freebsd4%
```

Как видно из вывода, сообщения действительно теряются, но при этом задерживаются только те, которые относятся к конкретному потоку. Данные в остальных потоках не задерживаются. Потоки SCTP могут послужить мощным средством борьбы с блокированием очереди, позволяющим в то же время сохранять порядок данных в рамках конкретного набора сообщений.

10.6. Управление количеством потоков

Мы рассмотрели пример использования потоков SCTP, но пока что мы не знаем, каким образом можно контролировать количество потоков, запрашиваемых конечной точкой в процессе инициализации ассоциации. В предыдущих примерах мы работали с тем количеством исходящих потоков, которое было установлено в системе по умолчанию. В реализации SCTP для FreeBSD, созданной в рамках проекта KAME, это значение равно 10. А что, если серверу и клиенту нужно больше десяти потоков? В листинге 10.6 мы приводим модификацию кода сервера, позволяющую увеличивать количество потоков, запрашиваемое при создании ассоциации. Обратите внимание, что данный параметр сокета должен быть изменен до создания ассоциации.

Листинг 10.6. Вариант сервера, допускающий увеличение числа потоков

```
//sctp/sctpserv02.c
14 if (argc > 2)
15     stream_increment = atoi(argv[1]);
16 sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
17 bzero(&initm, sizeof(initm));
18 initm.sinit_num_ostreams = SERV_MORE_STRMS_SCTP;
```

```
19 Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_INITMSG, &initm, sizeof(initm));
```

Предварительная настройка

14-16 Как и в предыдущей версии программы, сервер устанавливает флаг `stream_increment` в соответствии с дополнительным параметром командной строки, после чего открывает сокет.

Изменение запрашиваемого количества потоков

17-19 Все сделанные модификации относятся именно к этим строкам. Сначала сервер обнуляет структуру `sctp_initmsg`. Это изменение гарантирует, что вызов `setsockopt` не приведет к непреднамеренному изменению каких-либо иных значений кроме того, которое нас интересует. Затем сервер устанавливает поле `sinit_max_ostreams` равным количеству запрашиваемых потоков. После этого вызывается функция `setsockopt` с параметром сокета `SCTP_INITMSG` для установки параметров сообщения INIT.

Альтернативой установке параметра сокета может быть вызов функции `sendmsg` со вспомогательными данными, запрашивающими требуемое количество потоков. Передача вспомогательных данных приведет к желаемому результату только для сокетов типа «один-ко-многим».

10.7. Управление завершением соединения

В наших примерах на клиента была возложена ответственность по завершению ассоциации, для чего ему приходилось закрывать сокет. Но закрытие сокета не всегда является желаемой операцией с точки зрения приложения. Кроме того, серверу не нужно оставлять ассоциацию открытой после отправки эхо-ответа. В описанных ситуациях применяются альтернативные механизмы завершения ассоциации. Для сокетов типа «один-ко-многим» доступно два метода: корректное и аварийное закрытие.

Если сервер хочет закрыть ассоциацию после отправки сообщения, он должен добавить флаг `MSG_EOF` в это сообщение, поместив его в поле `sinfo_flags` структуры `sctp_sndrcvinfo`. Этот флаг закрывает ассоциацию после подтверждения приема отсылаемого сообщения. Альтернативный метод состоит в установке флага `MSG_ABORT` в том же поле `sinfo_flags`. При этом происходит немедленное закрытие ассоциации с отправкой порции ABORT (аналог TCP-сегмента RST). Данные, находящиеся в буфере отправки, сбрасываются. Однако закрытие сеанса SCTP порцией ABORT не приводит к негативным последствиям типа пропущенного состояния `TIME_WAIT`, как это происходит в TCP. В листинге 10.7 показана новая версия эхо-сервера, инициирующая корректное завершение соединения одновременно с отправкой эхо-ответа клиенту. В листинге 10.8 показана версия клиента, отправляющая порцию ABORT перед закрытием сокета.

Листинг 10.7. Сервер, закрывающий ассоциацию после отправки ответа

```
//sctp/sctpserv03.c
25 for (;;) {
26     len = sizeof(struct sockaddr_in);
27     rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof(readbuf),
28         (SA*)&cliaddr, &len, &sri, &msg_flags);
29     if (stream_increment) {
30         sri.sinfo_stream++;
31         if (sri.sinfo_stream >=
32             sctp_get_no_strms(sock_fd, (SA*)&cliaddr, len))
33             sri.sinfo_stream = 0;
34     }
35     Sctp_sendmsg(sock_fd, readbuf, rd_sz,
36         (SA*)&cliaddr, len,
37         sri.sinfo_ppid,
38         (sri.sinfo_flags | MSG_EOF), sri.sinfo_stream, 0, 0);
39 }
```

Отправка ответа с закрытием ассоциации

38 Изменение кода сервера состоит в том, что мы добавляем флаг `MSG_EOF` к прочим флагам в вызове `sctp_sendmsg` операцией логического ИЛИ. Благодаря этому сервер закрывает ассоциацию после подтверждения доставки сообщения.

Листинг 10.8. Клиент, выполняющий аварийное закрытие ассоциации

```
//sctp/sctpclient02.c
25 if (echo_to_all == 0)
26 sctpstr_cli(stdin, sock_fd, (SA*)&servaddr, sizeof(servaddr));
27 else
28 sctpstr_cli_echoall(stdin, sock_fd, (SA*)&servaddr,
29     sizeof(servaddr));
30 strcpy(byemsg, "goodbye");
31 Sctp_sendmsg(sock_fd, byemsg, strlen(byemsg),
32 (SA*)&servaddr, sizeof(servaddr), 0, MSG_ABORT, 0, 0, 0);
33 Close(sock_fd);
```

Аварийное закрытие ассоциации

30-32 Клиент подготавливает сообщение об аварийном закрытии ассоциации, вызванном пользовательской ошибкой. Затем функция `sctp_sendmsg` вызывается с флагом `MSG_ABORT`. При этом отправляется порция данных `ABORT`, что приводит к немедленному закрытию ассоциации. В порцию данных включается код пользовательской ошибки и сообщение («`goodbye`») в поле причины ошибки вышележащего уровня.

Закрытие дескриптора сокета

33 Хотя ассоциация и была завершена, дескриптор сокета все равно закрыть нужно, чтобы освободить связанные с ним системные ресурсы.

10.8. Резюме

Мы изучили простой пример клиента и сервера SCTP общим объемом около 150 строк кода. Обе программы работали с сокетами SCTP типа «один-ко-многим». Сервер был написан в последовательном стиле, который часто используется при работе с такими сокетами. Он считывал сообщения и отвечал на них по тому же потоку, из которого они приходили, или по потоку с увеличенным на единицу номером. Затем мы исследовали проблему блокирования очереди, изменив программу клиента таким образом, чтобы подчеркнуть особенности ситуации и продемонстрировать использование потоков SCTP для решения проблемы. После этого мы показали, каким образом можно изменить количество потоков при помощи одного из множества параметров сокета, используемых для управления поведением SCTP. Наконец, мы снова изменили код сервера и клиента, чтобы показать корректное и аварийное закрытие ассоциации.

Углубленное исследование SCTP будет проведено в главе 23.

Упражнения

1. Что произойдет с программой в листинге 10.1, если SCTP вернет сообщение об ошибке? Каким образом вы можете устраниТЬ указанный недостаток программы?
2. Что произойдет, если сервер завершит работу, не ответив на сообщения? Может ли клиент каким-либо образом получить уведомление об этом событии?
3. В листинге 10.7 в строке 22 аргумент `out_sz` устанавливается равным 800 байт. Как вы думаете, почему мы выбрали именно это значение? Существует ли лучший способ найти оптимальное значение этого аргумента?

4. Как повлияет алгоритм Нагла (см. раздел 7.10) на нашего клиента из листинга 10.7? Не лучше ли будет отключить алгоритм Нагла для этой программы? Воплотите это изменение в код клиента и сервера.

5. В разделе 10.6 мы утверждали, что приложению следует изменять количество потоков до установки ассоциации. Что произойдет в противном случае?

6. Когда мы говорили о количестве потоков, мы подчеркнули, что только для сокетов типа «один-ко-многим» можно увеличить количество потоков при помощи вспомогательных данных. Почему это так? (Подсказка: вспомогательные данные необходимо передавать с сообщениями.)

7. Почему сервер может не отслеживать открытые ассоциации? Опасно ли это?

8. В разделе 10.7 мы изменили сервер так, что он стал закрывать ассоциацию после отправки каждого сообщения. Вызовет ли это какие-либо проблемы? Хорошее ли это решение с точки зрения архитектуры приложения?

Глава 11

Преобразования имен и адресов

11.1. Введение

Во всех предшествующих примерах мы использовали численные адреса узлов (например, 206.6.226.33) и численные номера портов для идентификации серверов (например, порт 13 для стандартного сервера времени и даты и порт 9877 для нашего эхо-сервера). Однако по ряду соображений предпочтительнее использовать имена вместо чисел: во-первых, имена проще запоминаются, во-вторых, если численный адрес поменяется, имя можно сохранить, и в-третьих, с переходом на IPv6 численные адреса становятся значительно длиннее, что увеличивает вероятность ошибки при вводе адреса вручную. В этой главе описываются функции, выполняющие преобразование имен и адресов: `gethostbyname` и `gethostbyaddr` для преобразования имен узлов и IP-адресов, и `getservbyname` и `getservbyport` для преобразования имен служб и номеров портов. Здесь же мы рассмотрим две независимые от протоколов функции `getaddrinfo` и `getnameinfo`, осуществляющие преобразование между IP-адресами и именами узлов, а также между именами служб и номерами портов.

11.2. Система доменных имен

Система доменных имен (*Domain Name System, DNS*) используется прежде всего для сопоставления имен узлов и IP-адресов. Имя узла может быть либо *простым* (simple name), таким как `solaris` или `bsdi`, либо *полным доменным именем* (fully qualified domain name, FQDN), например `solaris.unpbook.com..`

ПРИМЕЧАНИЕ

В техническом отношении FQDN может также называться абсолютным именем и должно оканчиваться точкой, но пользователи часто игнорируют точку в конце. Точка сообщает распознавателю о том, что имя является абсолютным и не требует проведения поиска по различным доменам верхних уровней.

В этом разделе мы рассмотрим только основы DNS, необходимые нам для сетевого программирования. Читатели, интересующиеся более подробным изложением вопроса, могут обратиться к главе 14 [111] и к [1]. Дополнения, требуемые для IPv6, изложены в RFC 1886 [121].

Записи ресурсов

Записи в DNS называются *записями ресурсов* (*resource records, RR*). Нас интересуют только несколько типов RR.

- А. Запись типа A преобразует имя узла в 32-разрядный адрес IPv4. Вот, например, четыре записи DNS для узла `freebsd` в домене `unpbook.com`, первая из которых — это запись типа A:

```
freebsd IN A 12.106.32.254
IN AAAA 3ffe:b80:1f8d:1:a00:20ff:fea7:686b
IN MX 5 freebsd.unpbook.com.
IN MX 10 mailhost.unpbook.com.
```

- AAAA. Запись типа AAAA, называемая «четыре А» (quad A), преобразует имя узла в 128-разрядный адрес IPv6. Название «четыре А» объясняется тем, что 128-разрядный адрес в четыре раза больше 32-разрядного адреса.

■ PTR. Запись PTR (pointer records — запись указателя) преобразует IP-адрес в имя узла. Четыре байта адреса IPv4 располагаются в обратном порядке. Каждый байт преобразуется в десятичное значение ASCII (0-255), а затем добавляется `in-addr.arpa`. Получившаяся строка используется в запросе PTR.

32 полубайта 128-разрядного адреса IPv6 также располагаются в обратном порядке. Каждый полубайт преобразуется в соответствующее шестнадцатеричное значение ASCII (0-9, a-f) и добавляется к `ip6.arpa`.

Например, две записи PTR для нашего узла freebsd будут выглядеть так:

```
254.32.106.12 in-addr.arpa  
b.6.8.6.7.a.e.f.f.0.2.0.0.a.0.1.0.0.0.d.8.f.1.0.8.b.0.e.f.f.3.ip6.arpa
```

- MX. Запись типа MX (Mail Exchange Record) определяет, что узел выступает в роли «маршрутизирующего почтового сервера» для заданного узла. В приведенном выше примере для узла solaris предоставлено две записи типа MX. Первая имеет предпочтительное значение 5, вторая — 10. Когда существует множество записей типа MX, они используются в порядке предпочтения начиная с наименьшего значения.

ПРИМЕЧАНИЕ

Мы не используем в примерах книги записей типа MX, но упоминаем о них, потому что они широко используются в реальной жизни.

- CNAME. Аббревиатура CNAME означает «каноническое имя» (canonical name). Обычно такие записи используются для присвоения имен распространенным службам, таким как ftp и www. При использовании имен служб вместо действительного имени узла перемещение службы на другой узел становится прозрачным (то есть незаметным для пользователя). Например, для нашего узла linux каноническими именами могут быть следующие записи:

```
ftp IN CNAME linux.unpbook.com.  
www IN CNAME linux.unpbook.com.
```

Сейчас прошло еще слишком мало времени с момента появления протокола IPv6, чтобы сказать, каких соглашений будут придерживаться администраторы для узлов, поддерживающих и IPv4, и IPv6. В нашем примере мы задали узлу freebsd и запись типа A, и запись типа AAAA. Автор помещает и запись типа A, и запись типа AAAA под каноническим именем узла (как показано ниже) и создает три записи RR. Первая запись RR, имя которой оканчивается на -4, содержит запись типа A; вторая, с именем, оканчивающимся на -6, содержит запись типа AAAA; а третья запись RR, имя которой оканчивается на -611, содержит запись типа AAAA с локальным в пределах физической подсети (link-local, см. главу 19) адресом узла (что иногда удобно использовать в целях отладки). Все записи для другого нашего узла будут выглядеть так:

```
aix-4 IN A 206.62.226.43  
aix IN A 206.62.226.43  
IN MX 5 aix.unpbook.com.  
IN MX 10 mailhost.unpbook.com.  
Aix-4 IN A 192.168.42.2  
aix-6 IN AAAA 3ffe:b80:1f8d:2:204:acff:fe17:bf38  
aix-611 IN AAAA fe80::204:acff:fe17:bf38
```

Эта запись дает нам дополнительный контроль над протоколом, выбранным некоторыми приложениями, как мы увидим в следующей главе.

Распознаватели и серверы имен

Организации обычно работают с одним или несколькими серверами имен (name servers). Часто в качестве сервера используется программа BIND (Berkeley Internet Name Domain). Приложения, такие как клиенты и серверы, которые мы создаем в этой книге, соединяются с сервером DNS при помощи вызова функций из библиотеки, называемой *распознавателем* (*resolver*). Обычные функции распознавателя — `gethostbyname` и `gethostbyaddr`, и обе они описаны в этой главе. Первая находит адрес узла по его имени, а вторая — наоборот.

На рис. 11.1 показано типичное расположение приложений, распознавателей и серверов имен. В некоторых системах код распознавателя содержится в системной библиотеке и встраивается в приложение, когда оно создается. В других системах имеется централизованный демон-распознаватель, к которому обращаются все приложения. Системная библиотека выполняет удаленные вызовы процедур такого распознавателя. В любом случае код приложения вызывает код распознавателя посредством обычных вызовов, чаще всего `gethostbyname` и `gethostbyaddr`.

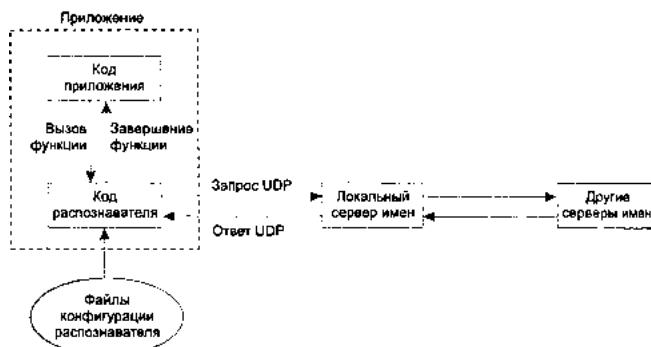


Рис. 11.1. Типичное расположение приложений, распознавателей и серверов имен

Код распознавателя считывает из файлов конфигурации, зависящих от системы, расположение серверов имен организации. (Мы говорим «серверы имен», употребляя множественное число, потому что большинство организаций работают с несколькими серверами имен, хотя мы и показываем на рисунке только один локальный сервер.) Файл `/etc/resolv.conf` обычно содержит IP-адреса локальных серверов имен.

ПРИМЕЧАНИЕ

Было бы удобно указывать в файле `/etc/resolv.conf` имена, а не IP-адреса серверов имен, потому что имена удобнее запоминать и редактировать, однако это возвратило бы нас к вечной проблеме курицы и яйца: каким образом распознать имя сервера имен?

Распознаватель посылает запрос локальному серверу имен, используя UDP. Если локальный сервер имен не знает ответа, он обычно запрашивает другие серверы имен через Интернет, также используя UDP. Если ответ слишком велик, чтобы поместиться в один UDP-пакет, распознаватель автоматически переключается на TCP.

Альтернативы DNS

Можно получить информацию об имени и адресе без использования DNS. Типичной альтернативой служат статические файлы со списком узлов (обычно файл `/etc/hosts`, как мы указываем в табл. 11.2), информационная система сети (Network Information System, NIS) и упрощенный протокол службы каталогов (Lightweight Directory Access Protocol — LDAP). К сожалению, способ конфигурирования узла для использования различных типов служб имен зависит от реализации. Solaris 2.x, HP-UX 10 и более новых версий, а также FreeBSD 5.x используют файл `/etc/nswitch.conf`, тогда как AIX использует файл `/etc/netsvc.conf`. BIND 9.9 предоставляет свою собственную версию, которая называется IRS (Information Retrieval Service — служба получения информации), использующую файл `/etc/irs.conf`. Если сервер имен должен применяться для поиска имен узлов, все эти системы используют для задания IP-адресов серверов имен файл `/etc/resolv.conf`. К счастью, эти различия обычно скрыты от программиста приложений, поэтому мы просто вызываем функции распознавателя, такие как `gethostbyname` и `gethostbyaddr`.

11.3. Функция `gethostbyname`

Узлы компьютерных сетей мы обычно идентифицируем по их именам, удобным для человеческого восприятия. Но во всех примерах книги специально использовались IP-адреса вместо имен, поэтому мы точно знаем, что входит в структуры адресов сокетов для таких функций, как `connect` и `sendto`, и что возвращаются функциями `accept` и `recvfrom`. Тем не менее большинство приложений имеют дело с именами, а не с адресами. Это особенно актуально при переходе на IPv6, поскольку адреса IPv6 (шестнадцатеричные строки) значительно длиннее адресов IPv4, записанных в точечно-десятичном

представлении. (Например, запись типа AAAA и запись типа PTR для `ip6.arpa` в предыдущем разделе показывают это со всей очевидностью.)

Самая основная функция, выполняющая поиск имени узла, — это функция `gethostbyname`. При успешном выполнении она возвращает указатель на структуру `hostent`, содержащую все адреса IPv4 для узла. Однако она может возвращать только адреса IPv4. В разделе 11.6 рассматривается функция, возвращающая адреса IPv4 и IPv6. Стандарт POSIX предупреждает, что функция `gethostbyname` может быть исключена из будущей его версии.

ПРИМЕЧАНИЕ

Маловероятно, что реализации `gethostbyname` исчезнут раньше, чем весь Интернет перейдет на протокол IPv6, а произойдет это еще очень не скоро. Однако удаление функции из стандарта POSIX гарантирует, что она не будет использоваться в новых программах. Вместо нее мы рекомендуем использовать `getaddrinfo` (раздел 11.6).

```
#include <netdb.h>

struct hostent *gethostbyname(const char *hostname);
Возвращает: непустой указатель в случае успешного выполнения, -1 в случае ошибки
Непустой указатель, возвращаемый этой функцией, указывает на следующую структуру hostent:
struct hostent {
    char *h_name;           /* официальное (каноническое) имя узла */
    char **h_aliases;       /* указатель на массив указателей на псевдонимы */
    int   h_addrtype;       /* тип адреса узла: AF_INET */
    int   h_length;         /* длина адреса: 4 */
    char **h_addr_list;    /* указатель на массив указателей с адресами IPv4 или IPv6 */
};
```

В терминах DNS функция `gethostbyname` выполняет запрос на запись типа A. Функция возвращает только адреса IPv4.

На рис. 11.2 представлено устройство структуры `hostent` и содержащаяся в ней информация, в предположении, что искомое имя узла имеет два альтернативных имени и три адреса IPv4. Все имена узла представляют собой строки языка C.

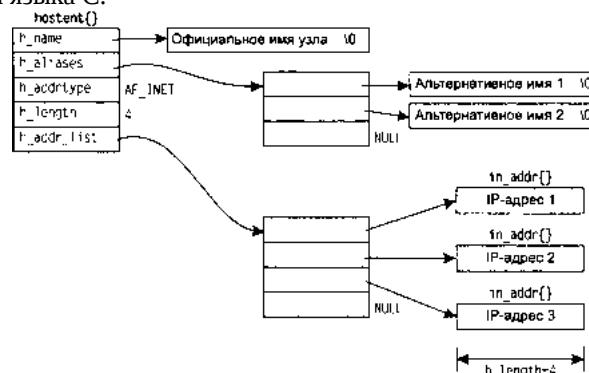


Рис. 11.2. Структура `hostent` и ее содержимое

Возвращаемое имя `h_name` называется каноническим именем узла. Например, с показанными в предыдущем разделе записями CNAME каноническое имя узла `ftp://ftp.unpbook.com` будет иметь вид `linux.unpbook.com`. Также если мы вызываем функцию `gethostbyname` с узла `aix` с неполным именем, например `solaris`, то в качестве канонического имени возвращается полное доменное имя (FQDN) `solaris.unpbook.com..`.

ПРИМЕЧАНИЕ

Некоторые версии функции `gethostbyname` допускают, что аргумент `hostname` может быть записан в виде строки десятичных чисел, разделенных точками. То есть вызов в форме `hptr = gethostbyname("206.62.226.33")`; будет работать. Этот код был добавлен, поскольку клиент Rlogin принимает только имя узла, вызывая функцию `gethostbyname`, и не принимает точечно-десятичную запись [127]. Стандарт POSIX допускает это, но не устанавливает такое поведение в качестве обязательного, поэтому переносимое приложение не может использовать указанную особенность.

Функция `gethostbyname` отличается от других функций сокетов, описанных нами, тем, что она не задает значение переменной `errno`, когда происходит ошибка. Вместо этого она присваивает глобальной целочисленной переменной `h_errno` одну из следующих констант, определяемых в заголовке `<netdb.h>`:

- `HOST_NOT_FOUND`;
- `TRY AGAIN`;
- `NO_RECOVERY`;
- `NO_DATA` (идентично `NO_ADDRESS`).

Ошибка `NO_DATA` означает, что заданное имя действительно, но у него нет записи типа A. Примером может служить имя узла, имеющего только запись типа MX.

Самые современные распознаватели предоставляют функцию `hstrerror`, которая в качестве единственного аргумента получает значение `h_errno` и возвращает указатель типа `const char*` на описание ошибки. Некоторые примеры строк, возвращаемых этой функцией, мы увидим в следующем примере.

Пример

В листинге 11.1^[1] показана простая программа, вызывающая функцию `gethostbyname` для любого числа аргументов командной строки и выводящая всю возвращаемую информацию.

Листинг 11.1. Вызов функции и вывод возвращаемой информации

```
//names/hostent.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     char *ptr, **pptr;
6     char str[INET_ADDRSTRLEN];
7     struct hostent *hptr;

8     while (--argc > 0) {
9         ptr = *++argv;
10        if ((hptr = gethostbyname(ptr)) == NULL) {
11            err_msg("gethostbyname error for host, %s: %s",
12                ptr, hstrerror(h_errno));
13            continue;
14        }
15        printf("official hostname: %s\n", hptr->h_name);

16        for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
17            printf("\talias: %s\n", *pptr);

18        switch (hptr->h_addrtype) {
19        case AF_INET:
20            pptr = hptr->h_addr_list;
21            for (; *pptr != NULL; pptr++)
22                printf("\taddress: %s\n",
23                    Inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str))));
```

```
24     break;
25
26     default:
27         err_ret("unknown address type");
28     break;
29 }
30 exit(0);
31 }
```

8-14 Функция `gethostbyname` вызывается для каждого аргумента командной строки.

15-17 Выводится каноническое имя узла, за которым идет список альтернативных имен.

18-24 Переменная `pptr` указывает на массив указателей на индивидуальные адреса. Для каждого адреса мы вызываем функцию `inet_ntop` и выводим возвращаемую строку.

Сначала мы выполняем программу с именем нашего узла `aix`, у которого имеется только один адрес IPv4:

```
freebsd % hostent aix
official hostname: aix.unpbook.com
address: 192.168.42.2
```

Обратите внимание, что официальное имя узла — это FQDN. Кроме того, хотя у узла имеется адрес IPv6, возвращается только адрес IPv4. Следующим будет веб-сервер с несколькими адресами IPv4:

```
solaris % hostent cnn.com
official hostname: cnn.com
address: 64.236.16.20
address: 64.236.16.52
address: 64.236.16.84
address: 64.236.16.116
address: 64.236.24.4
address: 64.236.24.12
address: 64.236.24.20
address: 64.236.24.28
```

Далее идет имя, представленное в разделе 11.2 как имя с записью типа CNAME:

```
solaris % hostent www
official hostname: linux.unpbook.com
alias: www.unpbook.com
address: 206.168.112.219
```

Как мы и предполагали, официальное имя узла отличается от нашего аргумента командной строки.

Чтобы увидеть строки ошибок, возвращаемые функцией `hstrerror`, мы сначала задаем несуществующее имя узла, а затем имя, имеющее только запись типа MX:

```
solaris % hostent nosuchname.invalid
gethostbyname error for host: nosuchname.invalid: Unknown host
```

```
solaris % hostent uunet.uu.net
gethostbyname error for host: uunet.uu.net: No address associated with name
```

11.4 Функция `gethostbyaddr`

Функция `gethostbyaddr` получает в качестве аргумента двоичный IP-адрес и пытается найти имя узла, соответствующее этому адресу. Ее действие обратно действию функции `gethostbyname`.

```
#include <netdb.h>
```

```
struct hostent *gethostbyaddr(const char *addr, size_t len, int family);
```

Возращает: непустой указатель в случае успешного выполнения, -1 в случае ошибки

Эта функция возвращает указатель на ту же структуру `hostent`, которую мы описывали при рассмотрении функции `gethostbyname`. Обычно в этой структуре нас интересует поле `h_name`, каноническое имя узла.

Аргумент `addr` не относится к типу `char*`, но в действительности это указатель на структуру `in_addr`, содержащую адрес IPv4. Поле `len` — это длина структуры: 4 для адресов IPv4. Аргумент `family` будет иметь значение `AF_INET`.

В терминах DNS функция `gethostbyaddr` запрашивает у сервера имен запись типа PTR в домене `in-addr.arpa`.

11.5. Функции `getservbyname` и `getservbyport`

Службы, как и узлы, также часто идентифицируются по именам. Используя в нашем коде имя службы вместо номера порта, при условии, что имена служб сопоставляются номерам портов в некотором файле (обычно `/etc/services`), мы получаем следующее преимущество. Если этой службе будет назначен другой номер порта, то нам будет достаточно изменить одну строку в файле `/etc/services`, вместо того чтобы перекомпилировать все приложения. Следующая функция, `getservbyname`, ищет службу по ее заданному имени.

ПРИМЕЧАНИЕ

Канонический список номеров портов, назначенных определенным службам, поддерживается IANA и располагается по адресу <http://www.iana.org/assignments/port-numbers> (см. раздел 2.9). Файл `/etc/services` чаще всего содержит некоторое подмножество списка IANA.

```
#include <netdb.h>

struct servent *getservbyname(const char *servname, const char *proto);
Возвращает: непустой указатель в случае успешного выполнения, NULL в случае ошибки
Функция возвращает указатель на следующую структуру:
struct servent {
    char *s_name;      /* официальное имя службы */
    char **s_aliases; /* список псевдонимов */
    int s_port;        /* номер порта, записанный в сетевом порядке байтов */
    char *s_proto;     /* протокол, который нужно использовать */
};
```

Имя службы `servname` должно быть указано обязательно. Если задан и протокол (то есть если `proto` — непустой указатель), то в структуре должен быть указан совпадающий протокол. Некоторые службы Интернета позволяют использовать и TCP, и UDP (например, DNS и все службы, представленные в табл. 2.1), в то время как другие поддерживают только один протокол (протоколу FTP требуется TCP). Если аргумент `proto` не задан и служба поддерживает несколько протоколов, то возвращаемый номер порта зависит от реализации. Обычно это не имеет значения, поскольку службы, поддерживающие множество протоколов, как правило, используют один и тот же номер порта для протоколов TCP и UDP, но вообще говоря это не гарантируется.

Более всего в структуре `servent` нас интересует поле номера порта. Поскольку номер порта возвращается в сетевом порядке байтов, мы не должны вызывать функцию `htonl` при записи его в структуру адреса сокета.

Типичные вызовы этой функции могут быть такими:

```
struct servent *sptr;

sptr = getservbyname("domain", "udp"); /* DNS с использованием UDP */
sptr = getservbyname("ftp", "tcp");    /* FTP с использованием TCP */
sptr = getservbyname("ftp", NULL);     /* FTP с использованием TCP */
sptr = getservbyname("ftp", "udp");    /* этот вызов приведет к ошибке */
```

Поскольку протоколом FTP поддерживается только TCP, второй и третий вызовы эквивалентны, а четвертый вызов приводит к ошибке. Вот соответствующие строки из файла `/etc/services`:

```
freebsd % grep -e ^ftp -e ^domain /etc/services
ftp-data 20/tcp #File Transfer [Default Data]
ftp       21/tcp #File Transfer [Control]
```

```

domain      53/tcp   #Domain Name Server
domain      53/udp   #Domain Name Server
ftp-agent  574/tcp #FTP Software Agent System
ftp-agent  574/udp #FTP Software Agent System
ftps-data  989/tcp # ftp protocol, data, over TLS/SSL
ftps       990/tcp # ftp protocol, control, over TLS/SSL
Следующая функция, getservbyport, ищет службу по заданному номеру порта и (не обязательно)
протоколу.

```

```

#include <netdb.h>

struct servent *getservbyport(int port, const char *protname);
Возвращает: непустой указатель в случае успешного выполнения, NULL в случае ошибки
Значение аргумента port должно быть записано в сетевом порядке байтов. Типичные примеры вызова
этой функции приведены ниже:
struct servent *sptr;

sptr = getservbyport(htons(53), "udp"); /* DNS с использованием UDP */
sptr = getservbyport(htons(21), "tcp"); /* FTP с использованием TCP */
sptr = getservbyport(htons(21), NULL); /* FTP с использованием TCP */
sptr = getservbyport(htons(21), "udp"); /* этот вызов приведет к ошибке */

```

Последний вызов оказывается неудачным, поскольку нет службы, использующей порт 21 с протоколом UDP.

Помните, что некоторые номера портов используются с TCP для одной службы, а с UDP — для совершенно другой, например:

```

freebsd % grep 514 /etc/services
shell  514/tcp cmd #like exec, but automatic
syslog 514/udp

```

Здесь показано, что порт 514 используется командой rsh с TCP и демоном syslog с UDP. Это характерно для портов 512-514.

Пример: использование функций gethostbyname и getservbyname

Теперь мы можем изменить код нашего TCP-клиента времени и даты, показанный в листинге 1.1, так, чтобы использовать функции gethostbyname и getservbyname и принимать два аргумента командной строки: имя узла и имя службы. Наша программа показана в листинге 11.2. Эта программа также демонстрирует желательное поведение при установлении соединения со всеми IP-адресами сервера на узле, имеющем несколько сетевых интерфейсов: попытки продолжаются до тех пор, пока соединение не будет успешно установлено или пока не будут перебраны все адреса.

Листинг 11.2. Наш клиент времени и даты, использующий функции gethostbyname и getservbyname

```

//names/daytimetcpccli1.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd, n;
6     char recvline[MAXLINE + 1];
7     struct sockaddr_in servaddr;
8     struct in_addr **pptr;
9     struct in_addr *inetaddrp[2];
10    struct in_addr inetaddr;
11    struct hostent *hp;
12    struct servent *sp;

13    if (argc != 3)

```

```

14  err_quit("usage: daytimetcpccli1 <hostname> <service>");

15 if ((hp = gethostbyname(argv[1])) == NULL) {
16   if (inet_aton(argv[1], &inetaddr) == 0) {
17     err_quit("hostname error for %s: %s", argv[1],
18             hstrerror(h_errno));
19   } else {
20     inetaddrp[0] = &inetaddr;
21     inetaddrp[1] = NULL;
22     pptr = inetaddrp;
23   }
24 } else {
25   pptr = (struct in_addr**)hp->h_addr_list;
26 }

27 if ((sp = getservbyname(argv[2], "tcp")) == NULL)
28   err_quit("getservbyname error for %s", argv[2]);

29 for (; *pptr != NULL; pptr++) {
30   sockfd = Socket(AF_INET, SOCK_STREAM, 0);

31   bzero(&servaddr, sizeof(servaddr));
32   servaddr.sin_family = AF_INET;
33   servaddr.sin_port = sp->s_port;
34   memcpy(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
35   printf("trying %s\n", Sock_ntop((SA*)&servaddr, sizeof(servaddr)));

36   if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) == 0)
37     break; /* успешное завершение */
38   err_ret("connect error");
39   close(sockfd);
40 }
41 if (*pptr == NULL)
42   err_quit("unable to connect");

43 while ((n = Read(sockfd, recvline, MAXLINE)) > 0) {
44   recvline[n] = 0; /* null terminate */
45   Fputs(recvline, stdout);
46 }
47 exit(0);
48 }

```

Вызов функций `gethostbyname` и `getservbyname`

13-28 Первый аргумент командной строки — это имя узла, передаваемое в качестве аргумента функции `gethostbyname`, а второй — имя службы, передаваемое в качестве аргумента функции `getservbyname`. Наш код подразумевает использование протокола TCP, что мы указываем во втором аргументе функции `getservbyname`. Если функция `gethostbyname` не удается найти нужное имя, мы вызываем функцию `inet_aton` (см. раздел 3.6), чтобы проверить, не является ли аргумент командной строки IP-адресом в формате ASCII. В этом случае формируется список из одного элемента — этого IP-адреса.

Перебор всех адресов

29-35 Теперь мы пишем вызовы функций `socket` и `connect` в цикле, который выполняется для каждого адреса сервера, пока попытка вызова функции `connect` не окажется успешной или пока не закончится список серверов. После вызова функции `socket` мы заполняем структуру адреса сокета Интернета IP-адресом и номером порта сервера. Хотя в целях увеличения производительности мы могли бы вынести из цикла вызов функции `bzero` и последующие два присваивания, наш код легче читать в таком виде, как он представлен сейчас. Установление соединения с сервером редко является основным источником проблем с производительностью сетевого клиента.

Вызов функции `connect`

36-39 Вызывается функция `connect`, и если вызов оказывается успешным, функция `break` завершает цикл. Если установить соединение не удается, мы выводим сообщение об ошибке и закрываем сокет. Вспомните, что дескриптор, для которого вызов функции `connect` оказался неудачным, не может больше использоваться и должен быть закрыт.

Завершение программы

41-42 Если цикл завершается, потому что ни один вызов функции `connect` не закончился успехом, программа завершает работу.

Чтение ответа сервера

43-47 Мы считываем ответ сервера и завершаем программу, когда сервер закрывает соединение. Если мы запустим эту программу, указав один из наших узлов, на котором работает сервер времени и даты, мы получим ожидаемый результат:

```
freebsd % daytimetcpccli1 aix daytime
trying 192.168.42.2:13
Sun Jul 27 22:44:19 2003
```

Но еще интереснее запустить программу, обратившись к маршрутизатору с несколькими сетевыми интерфейсами, на котором не работает сервер времени и даты:

```
solaris % daytimetcpccli1 gateway.tuc.noao.edu daytime
trying 140.252.108.1:13
connect error: Operation timed out
trying 140.252.1.4:13
connect error: Operation timed out
trying 140.252.104.1:13
connect error: Connection refused
unable to connect
```

11.6. Функция `getaddrinfo`

Функции `gethostbyname` и `gethostbyaddr` поддерживают только IPv4. Интерфейс IPv6 разрабатывался в несколько этапов (история разработки описана в разделе 11.20), и в конечном итоге получилась функция `getaddrinfo`. Последняя осуществляет трансляцию имен в адреса и служб в порты, причем возвращает она список структур `sockaddr`, а не список адресов. Такие структуры могут непосредственно использоваться функциями сокетов. Благодаря этому функция `getaddrinfo` скрывает все различия между протоколами в библиотеке функций. Приложение работает только со структурами адресов сокетов, которые заполняются `getaddrinfo`. Эта функция определяется стандартом POSIX.

ПРИМЕЧАНИЕ

Определение этой функции в POSIX происходит от более раннего предложения Кейта Склуэра (Keith Sklower) для функции, называемой `getconninfo`. Эта функция стала результатом обсуждений с Эриком Олменом (Eric Allman), Вильямом Даством (William Durst), Майклом Карелсом (Michael Karels) и Стивеном Вайсом (Steven Wise), а также более ранней реализации, написанной Эриком Олменом. Замечание о том, что указания имени узла и имени службы достаточно для соединения с этой службой независимо от деталей протокола, было сделано Маршалом Роузом (Marshall Rose) в проекте X/Open.

```
#include <netdb.h>

int getaddrinfo(const char *hostname, const char *service,
    const struct addrinfo *hints, struct addrinfo **result);
Возвращает: 0 в случае успешного выполнения, ненулевое значение в случае ошибки
(см. табл. 11.2).
```

Через указатель `result` функция возвращает указатель на связный список структур `addrinfo`, который задается в заголовочном файле `<netdb.h>`:

```
struct addrinfo {
    int      ai_flags;          /* AI_PASSIVE, AI_CANONNAME */
    int      ai_family;         /* AF_XXX */
    int      ai_socktype;       /* SOCK_XXX */
    int      ai_protocol;       /* 0 или IPPROTO_XXX для IPv4 и IPv6 */
    size_t   ai_addrlen;        /* длина ai_addr */
    char*   ai_canonname;       /* указатель на каноническое имя узла */
    struct sockaddr *ai_addr;   /* указатель на структуру адреса сокета */
    struct addrinfo *ai_next;   /* указатель на следующую структуру в связном
                                списке */
};
```

Переменная `hostname` — это либо имя узла, либо строка адреса (точечно-десятичная запись для IPv4 или шестнадцатеричная строка для IPv6). Переменная `service` — это либо имя службы, либо строка, содержащая десятичный номер порта. (См. также упражнение 11.4.)

Аргумент `hints` — это либо пустой указатель, либо указатель на структуру `addrinfo`, заполненную рекомендациями вызывающего процесса о типах информации, которую он хочет получить. Например, если заданная служба предоставляется и для TCP, и для UDP (служба `domain`, которая ссылается на сервер DNS), вызывающий процесс может присвоить элементу `ai_socktype` структуры `hints` значение `SOCK_DGRAM`. Тогда возвращение информации будет иметь место только для дейтаграммных сокетов.

Вызывающим процессом могут быть установлены значения следующих элементов структуры `hints`:

- `ai_flags` (несколько констант `AI_XXX`, объединенных операцией ИЛИ);
- `ai_family` (значение `AF_XXX`);
- `ai_socktype` (значение `SOCK_XXX`);
- `ai_protocol`.

Поле `ai_flags` может содержать следующие константы:

- `AI_PASSIVE` указывает, что сокет будет использоваться для пассивного открытия;
- `AI_CANONNAME` указывает функции на необходимость возвратить каноническое имя узла;
- `AI_NUMERICHOST` запрещает преобразование между именами и адресами. Аргумент `hostname` должен представлять собой строку адреса;

- `AI_NUMERICSERV` запрещает преобразование между именами служб и номерами портов. Аргумент `service` должен представлять собой строку с десятичным номером порта;
- `AI_V4MAPPED` вместе с `ai_family = AF_INET` указывает функции на необходимость вернуть адреса IPv4 из записей A, преобразованные к IPv6, если записи типа AAAA отсутствуют;
- `AI_ALL` при указании вместе с `AI_V4MAPPED` говорит о необходимости вернуть адреса IPv4, преобразованные к IPv6, вместе с истинными адресами IPv6;
- `AI_ADDRCONFIG` возвращает адреса, относящиеся к заданной версии IP, когда имеется несколько интерфейсов, имеющих IP-адреса другой версии.

Если аргументом структуры `hints` является пустой указатель, функция подразумевает нулевое значение для `ai_flags`, `ai_socktype` и `ai_protocol` и значение `AF_UNSPEC` для `ai_family`.

Если функция завершается успешно (0), то в переменную, на которую указывает аргумент `result`, записывается указатель на список структур `addrinfo`, связанных через указатель `ai_next`. Имеется два способа возвращения множественных структур.

1. Если существует множество адресов, связанных с узлом `hostname`, то одна структура возвращается для каждого адреса, который может использоваться с запрашиваемым семейством адресов (значение `ai_family`, если задано).

2. Если служба предоставляет для множества типов сокетов, то одна структура может быть возвращена для каждого типа сокета в зависимости от `ai_socktype`. (Заметьте, что большинство реализаций `getaddrinfo` считают, что номер порта используется только тем типом сокета, который запрашивается в `ai_socktype`. Если аргумент `ai_socktype` не определен, функция возвращает ошибку.)

Например, если структура `hints` пуста, а вы запрашиваете записи для службы `domain` на узле с двумя IP-адресами, возвращаются четыре структуры `addrinfo`:

- одна для первого IP-адреса и типа сокета `SOCK_STREAM`;
- одна для первого IP-адреса и типа сокета `SOCK_DGRAM`;
- одна для второго IP-адреса и типа сокета `SOCK_STREAM`;
- одна для второго IP-адреса и типа сокета `SOCK_DGRAM`.

Мы показываем схематическое изображение этого примера на рис. 11.3. Не существует никакого гарантированного порядка структур при возвращении множества элементов. Например, мы не можем считать, что службы TCP возвращаются перед службами UDP.

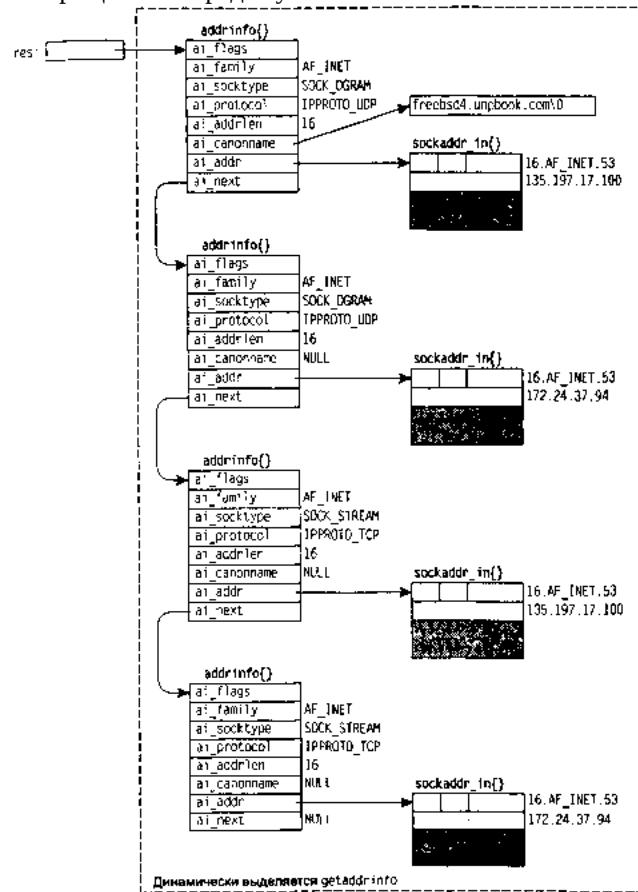


Рис. 11.3. Пример информации, возвращаемой функцией `getaddrinfo`

ПРИМЕЧАНИЕ

Хотя это и не гарантируется, реализация должна возвращать IP-адреса в том же порядке, в котором они возвращаются DNS. Некоторые распознаватели позволяют администратору

указывать порядок сортировки адресов в файле `/etc/resolv.conf`. Протокол IPv6 определяет правила выбора адресов (RFC 3484 [28]), которые могут влиять на порядок адресов, возвращаемых `getaddrinfo`.

Информация, возвращаемая в структурах `addrinfo`, готова для передачи функциям `socket` и `connect` или `sendto` (для клиента) и `bind` (для сервера). Аргументы функции `socket` — это элементы `ai_family`, `ai_socktype` и `ai_protocol`. Второй и третий аргументы функций `connect` и `bind` — это элементы `ai_addr` (указатель на структуру адреса сокета соответствующего типа, заполняемую функцией `getaddrinfo`) и `ai_addrlen` (длина этой структуры адреса сокета).

Если в структуре `hints` установлен флаг `AI_CANONNAME`, элемент `ai_canonname` первой возвращаемой структуры указывает на каноническое имя узла. В терминах DNS это обычно полное доменное имя (FQDN). Программы типа `telnet` широко используют этот флаг для того, чтобы выводить канонические имена систем, к которым производится подключение. Пользователь может указать короткое имя узла или его альтернативное имя, но он должен знать, с какой системой он в результате соединился.

На рис. 11.3 представлена возвращаемая информация для следующего вызова:

```
struct addrinfo hints, *res;
```

```
bzero(&hints, sizeof(hints));
```

```
hints.ai_flags = AI_CANONNAME;
```

```
hints.ai_family = AF_INET;
```

```
getaddrinfo("bsdi", "domain", &hints, &res);
```

На этом рисунке все, кроме переменной `res`, относится к динамически выделяемой памяти (например, с помощью функции `malloc`). Предполагается, что каноническое имя узла `freebsd4` — `freebsd4.unprbook.com`, и что этот узел имеет два адреса IPv4 в DNS.

Порт 53 предназначен для службы `domain`, и нужно учитывать, что этот номер порта будет представлен в структурах адресов сокетов в сетевом порядке байтов. Мы приводим возвращаемые значения `ai_protocol` `IPPROTO_TCP` и `IPPROTO_UDP`. Функция `getaddrinfo` может возвращать значение `ai_protocol` равное 0 для структур `SOCK_STREAM`, если этого достаточно для однозначного определения протокола (типа сокета недостаточно, например, если в системе помимо TCP реализован и SCTP), и 0 для структур `SOCK_DGRAM`, если в системе не реализованы другие протоколыдейтаграмм для IP (на момент написания этой книги стандартизованных протоколов еще не было, но два уже разрабатывались IETF). Лучше всего, если `getaddrinfo` всегда будет возвращать конкретный тип протокола.

В табл. 11.1 показано число структур `addrinfo` для каждого возвращаемого адреса, определяемое на основе заданного имени службы (которое может быть представлено десятичным номером порта) и рекомендации `ai_socktype`.

Таблица 11.1. Число структур `addrinfo`, возвращаемых для каждого IP-адреса

Элемент <code>ai_socktype</code>	Служба обозначена именем и предоставляется:						Служба обозначена именем порта
	Только TCP	Только UDP	Только SCTP	TCP и UDP	TCP и SCTP	TCP, UDP и SCTP	
0	1	1	1	2	2	3	Ошибка
<code>SOCK_STREAM</code>	1	Ошибка	1	1	2	2	2
<code>SOCK_DGRAM</code>	Ошибка	1		1	Ошибка	1	1
<code>SOCK_SEQPACKET</code>	Ошибка	Ошибка	1	Ошибка	1	1	1

Более одной структуры `addrinfo` возвращается для каждого IP-адреса только в том случае, когда поле `ai_socktype` структуры `hints` пусто и либо служба поддерживается TCP и UDP (как указано в файле `/etc/services`), либо задан номер порта для этой службы.

Если бы мы рассматривали все 64 возможных варианта сочетаний входных данных для функции `getaddrinfo` (имеется шесть входных переменных), многие сочетания оказались бы недопустимыми, а некоторые не имели бы смысла. Вместо этого рассмотрим наиболее типичные случаи.

■ **Задание имени узла и службы.** Это традиционный случай для клиента TCP и UDP. По завершении клиент TCP перебирает в цикле все возвращаемые IP-адреса, вызывая функции `socket` и `connect` для каждого из них, пока не установится соединение или пока не будут перебраны все адреса. Мы показываем такой пример с нашей функцией `tcp_connect` в листинге 11.2.

Для клиента UDP структура адреса сокета, заполняемая с помощью функции `getaddrinfo`, будет использоваться в вызове функции `sendto` или `connect`. Если клиент сообщит, что первый адрес не работает (ошибка на присоединенном сокете UDP или тайм-аут на неприсоединенном сокете), будет предпринята попытка обратиться к другому адресу.

Если клиент знает, что он обрабатывает только один тип сокета (например, клиентами Telnet и FTP обрабатываются только сокеты TCP, а клиентами TFTP — только сокеты UDP), то элементу `ai_socktype` структуры `hints` должно быть задано соответственно либо значение `SOCK_STREAM`, либо значение `SOCK_DGRAM`.

■ Типичный сервер задает службу (`service`), но не имя узла (`hostname`), и задает флаг `AI_PASSIVE` в структуре `hints`. Возвращаемая структура адреса сокета должна содержать IP-адрес, равный `INADDR_ANY` (для IPv4) или `IN6ADDR_ANY_INIT` (для IPv6). Сервер TCP затем вызывает функции `socket`, `bind` и `listen`. Если сервер хочет разместить в памяти с помощью функции `malloc` другую структуру адреса сокета, чтобы получить адрес клиента из функции `accept`, то возвращаемое значение `ai_addrlen` задает требуемый для этого размер.

Сервер UDP вызывает функции `socket`, `bind` и затем `recvfrom`. Если сервер хочет разместить в памяти с помощью функции `malloc` другую структуру адреса сокета, чтобы получить адрес клиента из функции `recvfrom`, возвращаемое значение `ai_addrlen` также задает нужный размер.

Как и в случае типичного клиентского кода, если сервер знает, что он обрабатывает только один тип сокета, то элемент `ai_socktype` структуры `hints` должен быть задан либо как `SOCK_STREAM`, либо как `SOCK_DGRAM`. Это позволяет избежать возвращения множества структур, с (возможно) неверным значением элемента `ai_socktype`.

■ До сих пор мы демонстрировали серверы TCP, создающие один прослушиваемый сокет, и серверы UDP, создающие один сокет дейтаграмм. Это тот вариант, который подразумевался в предыдущем абзаце. Альтернативным устройством является сервер, который обрабатывает множество сокетов с помощью функции `select`. В этом сценарии сервер должен последовательно перебрать все структуры из списка, возвращаемого функцией `getaddrinfo`, создать по одному сокету для каждой структуры и вызвать функцию `select`.

ПРИМЕЧАНИЕ

Проблема этой технологии состоит в том, что условие, по которому функция `getaddrinfo` возвращает множество структур, возникает, когда служба может обрабатываться как протоколом IPv4, так и протоколом IPv6 (см. табл. 11.3). Но эти два протокола не полностью независимы, как мы увидели в разделе 10.2, то есть если мы создаем прослушиваемый сокет IPv6 для данного порта, нет необходимости создавать для него прослушиваемый сокет IPv4, поскольку соединения, приходящие от клиентов IPv4, автоматически обрабатываются стеком протоколов и прослушиваемым сокетом IPv6, при условии, что параметр сокета `IPV6_V6ONLY` не установлен.

Невзирая на тот факт, что функция `getaddrinfo` «лучше», чем функции `gethostbyname` и `gethostbyaddr` (помимо того что эта функция упрощает написание кода, не зависящего от протокола, она обрабатывает и имя узла, и имя службы, и к тому же вся возвращаемая ею информация размещается в памяти динамически, а не статически), ее все же не так просто использовать, как это могло показаться. Проблема в том, что нам требуется разместить в памяти структуру `hints`, инициализировать ее нулем, заполнить необходимые поля, вызвать функцию `getaddrinfo` и затем пройти весь связный список, проверяя каждый его элемент. В последующих разделах мы предоставим более простые интерфейсы для типичных клиентов TCP и UDP и серверов, которые будем создавать в оставшейся части книги.

Функция `getaddrinfo` решает проблему преобразования имен узлов и имен служб в структуры адресов сокетов. В разделе 11.17 мы опишем обратную функцию `getnameinfo`, которая преобразует структуры адресов сокетов в имена узлов и имена служб.

11.7. Функция `gai_strerror`

Ненулевые значения ошибок, возвращаемых функцией `getaddrinfo`, имеют названия и значения, показанные в табл. 11.2. Функция `gai_strerror` получает одно из этих значений в качестве аргумента и возвращает указатель на соответствующую текстовую строку с описанием ошибки.

```
#include <netdb.h>

char *gai_strerror(int error);
Возвращает: указатель на строку с описанием ошибки
```

Таблица 11.2. Ненулевые возвращаемые значения (константы) ошибок функции getaddrinfo

Константа	Описание
EAI_AGAIN	Временный сбой при попытке разрешения имен
EAI_BADFLAGS	Недопустимое значение ai_flags
EAI_FAIL	Неисправимая ошибка при разрешении имен
EAI_FAMILY	Семейство ai_family не поддерживается
EAI_MEMORY	Ошибка при выделении памяти
EAI_NONAME	Имя узла или имя службы неизвестны или равны NULL
EAI_OVERFLOW	Переполнен буфер пользовательских аргументов (только для getnameinfo)
EAI_SERVICE	Запрошенная служба не поддерживается для данного типа сокета ai_socktype
EAI_SOCKTYPE	Тип сокета ai_socktype не поддерживается
EAI_SYSTEM	Другая системная ошибка, возвращаемая в переменной errno

11.8. Функция freeaddrinfo

Вся память, занимаемая структурами `addrinfo`, структурами `ai_addr` и строкой `ai_canonname`, которые возвращаются функцией `getaddrinfo`, динамически выделяется функцией `malloc`. Эта память освобождается при вызове функции `freeaddrinfo`.

```
#include <netdb.h>
```

```
void freeaddrinfo(struct addrinfo *ai);
```

Переменная `ai` должна указывать на первую из структур `addrinfo`, возвращаемых функцией `getaddrinfo`. Освобождается вся область памяти, занятая структурами из связного списка, вместе с динамически выделенной областью памяти, содержащей данные, на которые указывают эти структуры (например, структуры адресов сокетов и канонические имена узлов).

Предположим, что мы вызываем функцию `getaddrinfo`, проходим последовательно по всему связному списку структур `addrinfo` и находим нужную структуру. Если далее мы попытаемся сохранить нужную нам информацию простым копированием структуры `addrinfo`, а затем вызовем функцию `freeaddrinfo`, мы получим скрытую ошибку. Причина в том, что структура `addrinfo` сама указывает на динамически выделенный участок памяти (для структуры адреса сокета и, возможно, для канонического имени). Но эта область памяти, на которую указывает сохраненная нами структура, при вызове функции `freeaddrinfo` освобождается и может использоваться для хранения какой-либо иной информации.

ПРИМЕЧАНИЕ

Создание копии только самой структуры `addrinfo`, а не структур, на которые она, в свою очередь, указывает, называется поверхностным копированием (*shallow copy*). Копирование структуры `addrinfo` и всех структур, на которые она указывает, называется детальным копированием (*deep copy*).

11.9. Функция getaddrinfo: IPv6

Стандарт POSIX определяет как `getaddrinfo`, так и возвращаемые этой функцией данные для протоколов IPv4 и IPv6. Отметим следующие моменты, прежде чем свести возвращаемые значения воедино в табл. 11.3.

- Входные данные функции `getaddrinfo` могут относиться к двум различным типам, которые выбираются в зависимости от того, какой тип структуры адреса сокета вызывающий процесс хочет

получить обратно и какой тип записей нужно искать в DNS или иной базе данных.

■ Семейством адресов, указанным вызывающим процессом в структуре `hints`, задается тип структуры адреса сокета, который вызывающий процесс предполагает получить. Если вызывающий процесс задает `AF_INET`, функция не должна возвращать структуры `sockaddr_in6`, а для `AF_INET6` функция не должна возвращать структур `sockaddr_in`.

■ POSIX утверждает, что при задании семейства `AF_UNSPEC` должны возвращаться адреса, которые могут использоваться с любым семейством протоколов, допускающим применение имени узла и имени службы. Это подразумевает, что если у узла имеются как записи типа AAAA, так и записи типа A, то записи типа AAAA возвращаются как структуры `sockaddr_in6`, а записи типа A — как структуры `sockaddr_in`. Нет смысла возвращать еще и записи типа A как адреса IPv4, преобразованные к виду IPv6, в структурах `sockaddr_in6`, потому что при этом не возвращается никакой дополнительной информации — эти адреса уже возвращены в структурах `sockaddr_in`.

■ Это утверждение POSIX также подразумевает, что если флаг `AI_PASSIVE` задан без имени узла, то должен быть возвращен универсальный адрес IPv6 (`IN6ADDR_ANY_INIT` или `0::0`) в структуре `sockaddr_in6` вместе с универсальным адресом IPv4 (`INADDR_ANY` или `0.0.0.0`) в структуре `sockaddr_in`. Также нет смысла возвращать сначала универсальный адрес IPv4, поскольку мы увидим в разделе 12.2, что на узле с двойным стеком сокет сервера IPv6 может обрабатывать и клиенты IPv4, и клиенты IPv6.

■ Семейство адресов, указанное в поле `ai_family` структуры `hint` вместе с флагами `AI_V4MAPPED` и `AI_ALL` поля `ai_flags`, задают тип записей, поиск которых ведется в DNS (тип A или тип AAAA), и тип возвращаемых адресов (IPv4, IPv6 или IPv4, преобразованные к виду IPv6). Мы обобщили это в табл. 11.3.

■ Имя узла может также быть либо шестнадцатеричной строкой IPv6, либо строкой в точечно-десятичной записи IPv4. Допустимость этой строки зависит от семейства адресов, заданного вызывающим процессом. Шестнадцатеричная строка IPv6 неприемлема, если задано семейство `AF_INET`, а строка в точечно-десятичной записи IPv4 неприемлема, если задано семейство `AF_INET6`. Но если задано семейство `AF_UNSPEC`, то допустимы оба варианта, и при этом возвращается соответствующий тип структуры адреса сокета.

ПРИМЕЧАНИЕ

Можно возразить, что если в качестве семейства протоколов задано `AF_INET6`, строка в точечно-десятичной записи должна возвращаться как адрес IPv4, преобразованный к виду IPv6 в структуре `sockaddr_in6`. Но другим способом получения этого результата является установка префикса строки с десятичной точкой `0::ffff:`.

В табл. 11.3 показано, как будут обрабатываться адреса IPv4 и IPv6 функцией `getaddrinfo`. Колонка «Результат» отражает то, что мы хотим возвратить вызывающему процессу, если входные переменные таковы, как показано в первых трех колонках. Колонка «Действия» — то, каким образом мы получаем этот результат.

Таблица 11.3. Функция `getaddrinfo`: ее действия и результаты

Имя узла, указанное вызывающим процессом	Семейство адресов, указанное вызывающим процессом	Строка с именем узла содержит	Результат	Действия
Ненулевая строка с именем узла; активное или пассивное открытие	<code>AF_UNSPEC</code>	Имя узла	Все записи AAAA возвращаются как структуры <code>sockaddr_in6{}</code> и все записи A возвращаются как структуры <code>sockaddr_in{}</code>	Поиск по записям AAAA и поиск по записям A
		Шестнадцатеричная строка	Одна структура <code>sockaddr_in6{}</code> <code>inet_pton(AF_INET6)</code>	
		Строка в точечно- десятичной записи	Одна структура <code>sockaddr_in{}</code> <code>inet_pton(AF_INET)</code>	

	AF_INET6	Имя узла	Все записи AAAA возвращаются как структуры sockaddr_in6{} либо все записи А возвращаются как структуры sockaddr_in6{} с адресами IPv4, преобразованными к виду IPv6	Поиск по записям AAAA
		Шестнадцатеричная строка	Одна структура sockaddr_in6{} inet_nton(AF_INET6)	
		Строка в точечно- десятичной записи	Ищется как имя узла	
		Имя узла	Все записи А возвращаются как структуры sockaddr_in{} типа А	Поиск по записям типа А
	AF_INET	Шестнадцатеричная строка	Ошибка: EAI_ADDRFAMILY	
		Строка в точечно- десятичной записи	Одна структура sockaddr_in{} inet_nton(AF_INET)	
		Неявный адрес 0::0	Одна структура sockaddr_in6{} inet_nton(AF_INET6)	
Пустая строка с AF_UNSPEC именем узла;		Неявный адрес 0.0.0	и одна структура sockaddr_in{} inet_nton(AF_INET)	
пассивное открытие	AF_INET6	Неявный адрес 0::0	Одна структура sockaddr_in6{} inet_nton(AF_INET6)	
	AF_INET	Неявный адрес 0.0.0.0	Одна структура sockaddr_in{} inet_nton(AF_INET)	
		Неявный адрес 0::1	Одна структура sockaddr_in6{} inet_nton(AF_INET6)	
Пустая строка с AF_UNSPEC именем узла;		Неявный адрес 127.0.0.1	и одна структура sockaddr_in{} inet_nton(AF_INET)	
активное открытие	AF_INET6	Неявный адрес 0::1	Одна структура sockaddr_in6{} inet_nton(AF_INET6)	
	AF_INET	Неявный адрес 127.0.0.1	Одна структура sockaddr_in{} inet_nton(AF_INET)	

Обратите внимание, что табл. 11.3 описывает только обработку адресов IPv4 и IPv6 функцией `getaddrinfo`, то есть количество и тип адресов, возвращаемых процессу в различных ситуациях. Реальное количество структур `addrinfo` зависит также от типа сокета и имени службы, о чем уже говорилось в связи с табл. 11.1.

11.10. Функция `getaddrinfo`: примеры

Теперь мы покажем некоторые примеры работы функции `getaddrinfo`, используя тестовую программу, которая позволяет нам вводить все параметры: имя узла, имя службы, семейство адресов, тип сокета и флаги `AI_CANONNAME` и `AI_PASSIVE`. (Мы не показываем эту тестовую программу, поскольку она содержит около 350 строк малоинтересного кода. Ее можно получить тем же способом, что и прочие исходные коды для этой книги.) Тестовая программа выдает информацию о переменном числе возвращаемых структур `addrinfo`, показывая аргументы вызова функции `socket` и адрес в каждой структуре адреса сокета. Сначала показываем тот же пример, что и на рис. 11.3:

```
freebsd % testga -f inet -c -h freebsd4 -s domain
```

```
socket(AF_INET, SOCK_DGRAM, 17) ai_canonname = freebsd4.unpbook.com
      address: 135.197.17.100:53
socket(AF_INET, SOCK_DGRAM, 17)
      address: 172.24.37.94:53
socket(AF_INET, SOCK_STREAM, 6) ai_canonname = freebsd4.unpbook.com
      address: 135.197.17.100:53
socket(AF_INET, SOCK_STREAM, 6)
      address: 172.24.37.94:53
```

Параметр `-f inet` задает семейство адресов, `-c` указывает, что нужно возвратить каноническое имя, `-h freebsd4` задает имя узла, `-s domain` задает имя службы.

Типичный сценарий клиента — задать семейство адресов, тип сокета (параметр `-t`), имя узла и имя службы. Следующий пример показывает это для узла с несколькими сетевыми интерфейсами с шестью адресами IPv4:

```
freebsd % testga -f inet -t stream -h gateway.tuc.noao.edu -s daytime
socket(AF_INET, SOCK_STREAM, 6)
address: 140.252.108.1:13

socket(AF_INET, SOCK_STREAM, 6)
address: 140.252.1.4:13

socket(AF_INET, SOCK_STREAM, 6)
address: 140.252.104.1:13

socket(AF_INET, SOCK_STREAM, 0)
address: 140.252.3.6.13

socket(AF_INET, SOCK_STREAM, 0)
address: 140.252.4.100.13

socket(AF_INET, SOCK_STREAM, 0)
address: 140.252.1.4.13
```

Затем мы задаем наш узел `aix`, у которого имеется и запись типа AAAA, и запись типа A, не указывая семейства адресов. Имя службы — `ftp`, которая предоставляет только TCP.

```
freebsd % testga -h aix -s ftp -t stream

socket(AF_NET6, SOCK_STREAM, 6)
address: [3ffe:b80:1f8d:2:204:acff:fe17:bf38]:21

socket(AF_INET, SOCK_STREAM, 6)
address: 192.168.42.2:21
```

Поскольку мы не задали семейство адресов и запустили этот пример на узле, который поддерживает и IPv4, и IPv6, возвращаются две структуры: одна для IPv6 и одна для IPv4.

Затем мы задаем флаг `AI_PASSIVE` (параметр `-p`), не указываем ни семейства адресов, ни имени узла (подразумевая универсальный адрес), задаем номер порта 8888 и не указываем тип сокета.

```
freebsd % testga -p -s 8888 -t stream
```

```
socket(AF_INET6, SOCK_STREAM, 6)
address: [::]:8888

socket(AF_INET, SOCK_STREAM, 6)
address: 0.0.0.0:8888
```

Возвращаются две структуры. Поскольку мы запустили эту программу на узле, поддерживающем и IPv4, и IPv6, не задав семейства адресов, функция `getaddrinfo` возвращает универсальный адрес IPv6 и универсальный адрес IPv4. Структура IPv6 возвращается перед структурой IPv4, поскольку, как мы увидим в главе 12, клиент или сервер IPv6 на узле с двойным стеком может взаимодействовать с собеседниками по IPv6 и по IPv4.

11.11. Функция host_serv

Наш первый интерфейс функции `getaddrinfo` не требует от вызывающего процесса размещать в памяти структуру рекомендаций и заполнять ее. Вместо этого аргументами нашей функции `host_serv` будут интересующие нас поля — семейство адресов и тип сокета.

```
#include "unp.h"
```

```
    struct addrinfo *host_serv(const char *hostname, const char *service, int family, int
socktype);
```

Возвращает: в случае успешного выполнения указатель на структуру addrinfo. NULL в случае ошибки

В листинге 11.3 показан исходный код этой функции.

Листинг 11.3. Функция host_serv

```
//lib/host_serv.c
1 #include "unp.h"

2 struct addrinfo*
3 host_serv(const char *host, const char *serv, int family, int socktype)
4 {
5     int n;
6     struct addrinfo hints, *res;

7     bzero(&hints, sizeof(struct addrinfo));
8     hints.ai_flags = AI_CANONNAME; /* всегда возвращает каноническое имя */
9     hints.ai_family = family; /* AF_UNSPEC, AF_INET, AF_INET6, ... */
10    hints.ai_socktype = socktype; /* 0, SOCK_STREAM, SOCK_DGRAM, ... */

11    if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
12        return (NULL);

13    return (res); /* возвращает указатель на первый элемент в связном
списке */
14 }
```

7-13 Функция инициализирует структуру рекомендаций (hints), вызывает функцию getaddrinfo и возвращает пустой указатель, если происходит ошибка.

Мы вызываем эту функцию в листинге 16.11, когда нам нужно использовать getaddrinfo для получения информации об узле и о службе и при этом мы хотим установить соединение самостоятельно.

11.12. Функция tcp_connect

Теперь мы напишем две функции, использующие функцию getaddrinfo для обработки большинства сценариев клиентов и серверов TCP, которые мы создаем. Первая из этих функций, tcp_connect, выполняет обычные шаги клиента: создание сокета TCP и соединение с сервером.

```
#include "unp.h"
```

```
int tcp_connect(const char *hostname, const char *service);
```

Возвращает: в случае успешного соединения - дескриптор присоединенного сокета, в случае ошибки не возвращается ничего

В листинге 11.4 показан исходный код.

Листинг 11.4. Функция tcp_connect: выполнение обычных шагов клиента

```
//lib/tcp_connect.c
```

```
1 #include "unp.h"

2 int
3 tcp_connect(const char *host, const char *serv)
4 {
5     int sockfd, n;
6     struct addrinfo hints, *res, *ressave;

7     bzero(&hints, sizeof(struct addrinfo));
8     hints.ai_family = AF_UNSPEC;
9     hints.ai_socktype = SOCK_STREAM;
```

```

10  if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
11    err_quit("tcp_connect error for %s, %s: %s",
12    host, serv, gai_strerror(n));
13  ressave = res;

14  do {
15    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16    if (sockfd < 0)
17      continue; /* игнорируем этот адрес */
18    if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
19      break; /* успех */

20  Close(sockfd); /* игнорируем этот адрес */
21 } while ((res = res->ai_next) != NULL);

22 if (res == NULL) /* значение errno устанавливается при
                     последней попытке connect() */
23   err_sys("tcp_connect error for %s, %s", host, serv);

24 freeaddrinfo(ressave);

25 return (sockfd);
26 }

```

Вызов функции `getaddrinfo`

7-13 функция `getaddrinfo` вызывается один раз, когда мы задаем семейство адресов `AF_UNSPEC` и тип сокета `SOCK_STREAM`.

Перебор всех структур `addrinfo` до успешного выполнения или до окончания списка

14-25 Затем пробуется каждый IP-адрес: вызываются функции `socket` и `connect`. Если выполнение функции `socket` неудачно, это не фатальная ошибка, так как такое может случиться, если был возвращен адрес IPv6, а ядро узла не поддерживает IPv6. Если выполнение функции `connect` успешно, выполняется функция `break` для выхода из цикла. В противном случае, после того как перепробованы все адреса, цикл также завершается. Функция `freeaddrinfo` освобождает всю динамически выделенную память.

Эта функция (как и другие наши функции, предоставляющие более простой интерфейс для функции `getaddrinfo` в следующих разделах) завершается, если либо оказывается неудачным вызов функции `getaddrinfo`, либо вызов функции `connect` не выполняется успешно. Возвращение из нашей функции возможно лишь в случае успешного выполнения. Было бы сложно возвратить код ошибки (одну из констант `EAI_xxx`), не добавляя еще одного аргумента. Это значит, что наша функция-обертка тривиальна:

```

Tcp_connect(const char *host, const char *serv) {
    return(tcp_connect(host, serv));
}

```

Тем не менее мы по-прежнему вызываем функцию-обертку вместо функции `tcp_connect` ради сохранения единства в оставшейся части книги.

ПРИМЕЧАНИЕ

Проблема с возвращаемым значением заключается в том, что дескрипторы неотрицательные, но мы не знаем, положительны или отрицательны значения `EAI_xxx`. Если бы эти значения были положительными, мы могли бы возвратить равные им по абсолютной величине отрицательные значения, когда вызов функции `getaddrinfo` окажется неудачным. Но мы также должны возвратить

некое другое отрицательное значение, чтобы указать, что все структуры были перепробованы безуспешно.

Пример: клиент времени и даты

В листинге 11.5 показан наш клиент времени и даты из листинга 1.1, переписанный с использованием функции `tcp_connect`.

Листинг 11.5. Клиент времени и даты, переписанный с использованием функции `tcp_connect`

```
//names/daytimetcpccli.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd, n;
6     char recvline[MAXLINE + 1];
7     socklen_t len;
8     struct sockaddr_storage *ss;

9     if (argc != 3)
10    err_quit
11    ("usage, daytimetcpccli <hostname/IPaddress> <service/port#>");

12    sockfd = Tcp_connect(argv[1], argv[2]);

13    len = sizeof(ss);
14    Getpeername(sockfd, (SA*)&ss, &len);
15    printf("connected to %s\n", Sock_ntop_host((SA*)&ss, len));

16    while ((n = Read(sockfd, recvline, MAXLINE)) > 0) {
17        recvline[n] = 0; /* завершающий нуль */
18        Fputs(recvline, stdout);
19    }
20    exit(0);
21 }
```

Аргументы командной строки

9-11 Теперь нам требуется второй аргумент командной строки для задания либо имени службы, либо номера порта, что позволит нашей программе соединяться с другими портами.

Соединение с сервером

12 Теперь весь код сокета для этого клиента выполняется функцией `tcp_connect`.

Вывод ответа сервера

13-15 Мы вызываем функцию `getpeername`, чтобы получить адрес протокола сервера и вывести его. Мы делаем это для проверки протокола, используемого в примерах, которые скоро покажем.

Обратите внимание, что функция `tcp_connect` не возвращает размера структуры адреса сокета, который использовался для функции `connect`. Мы могли добавить еще один аргумент-указатель, чтобы получить это значение, но при создании этой функции мы стремились добиться меньшего числа

аргументов, чем у функции `getaddrinfo`. Поэтому мы определяем константу `MAXSOCKADDR` в нашем заголовке `inpr.h` так, чтобы ее размер равнялся размеру наибольшей структуры адреса сокета. Обычно это размер структуры адреса доменного сокета Unix (см. раздел 14.2), немного более 100 байт. Мы выделяем в памяти пространство для структуры указанного размера и заполняем ее с помощью функции `getpeername`.

Эта версия нашего клиента работает и с IPv4, и с IPv6, тогда как версия, представленная в листинге 1.1, работала только с IPv4, а версия из листинга 1.2 — только с IPv6. Сравните нашу новую версию с представленной в листинге Д.6, которую мы написали, чтобы использовать функции `gethostbyname` и `getservbyname` для поддержки IPv4, и IPv6.

Сначала мы задаем имя узла, поддерживающего только IPv4:

```
freebsd % daytimetcpccli linux daytime
connected to 206.168.112.96
Sun Jul 27 23:06:24 2003
```

Затем мы задаем имя узла, поддерживающего и IPv4, и IPv6:

```
freebsd % daytimetcpccli aix daytime
connected to 3ffe:b80:1f8d:2:204:acff:fe17:bf38
Sun Jul 27 23:17:13 2003
```

Используется адрес IPv6, поскольку у узла имеется и запись типа AAAA, и запись типа A. Кроме того, функция `tcp_connect` устанавливает семейство адресов `AF_UNSPEC`, поэтому, как было отмечено в табл. 11.3, сначала идет поиск записей типа AAAA, и только если этот поиск неудачен, выполняется поиск записей типа A.

В следующем примере мы указываем на необходимость использования именно адреса IPv4, задавая имя узла с суффиксом `-4`, что, как мы отмечали в разделе 11.2, в соответствии с принятым нами соглашением означает имя узла, который поддерживает только записи типа A:

```
freebsd % daytimetcpccli aix-4 daytime
connected to 192.168.42.2
Sun Jul 27 23:17:48 2003
```

11.13. Функция `tcp_listen`

Наша следующая функция, `tcp_listen`, выполняет обычные шаги сервера TCP: создание сокета TCP, связывание его с заранее известным портом с помощью функции `bind` и разрешение приема входящих запросов через соединение. В листинге 11.6 представлен исходный код.

```
#include "unp.h"
```

```
int tcp_listen(const char *hostname, const char *service, socklen_t *lenptr);
```

В случае успешного выполнения возвращает дескриптор присоединенного сокета, в случае ошибки не возвращает ничего

Листинг 11.6. Функция `tcp_listen`: выполнение обычных шагов сервера TCP

```
//lib/tcp_listen.c
1 #include "unp.h"

2 int
3 tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
4 {
5     int listenfd, n;
6     const int on = 1;
7     struct addrinfo hints, *res, *ressave;

8     bzero(&hints, sizeof(struct addrinfo));
9     hints.ai_flags = AI_PASSIVE;
10    hints.ai_family = AF_UNSPEC;
11    hints.ai_socktype = SOCK_STREAM;

12    if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
13        err_quit("tcp_listen error for %s, %s: %s",
14            host, serv, gai_strerror(n));
```

```

15  ressave = res;

16  do {
17      listenfd =
18      socket(res->ai_family, res->ai_socktype, res->ai_protocol);
19      if (listenfd < 0)
20          continue; /* ошибка, пробуем следующий адрес */
21      Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
22      if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
23          break; /* успех */

24  Close(listenfd); /* ошибка при вызове функции bind, закрываем
                     * сокет и пробуем следующий адрес*/
25 } while ((res = res->ai_next) != NULL);

26 if (res == NULL) /* значение errno устанавливается при последнем
                     * вызове функции socket() или bind() */
27     err_sys("tcp_listen error for %s, %s", host, serv);

28 Listen(listenfd, LISTENQ);

29 if (addrlenp)
30     *addrlenp = res->ai_addrlen; /* возвращает размер адреса протокола */

31 freeaddrinfo(ressave);

32 return (listenfd);
33 }

```

Вызов функции getaddrinfo

8-15 Мы инициализируем структуру `addrinfo` с учетом следующих рекомендаций (элементов структуры `hints`): `AI_PASSIVE`, поскольку это функция для сервера, `AF_UNSPEC` для семейства адресов и `SOCK_STREAM`. Вспомните табл. 11.3: если имя узла не задано (что вполне нормально для сервера, который хочет связать с дескриптором универсальный адрес), то наличие значений `AI_PASSIVE` и `AF_UNSPEC` вызовет возвращение двух структур адреса сокета: первой для IPv6 и второй для IPv4 (в предположении, что это узел с двойным стеком).

Создание сокета и связывание с адресом

16-24 Вызываются функции `socket` и `bind`. Если любой из вызовов окажется неудачным, мы просто игнорируем данную структуру `addrinfo` и переходим к следующей. Как было сказано в разделе 7.5, для сервера TCP мы всегда устанавливаем параметр сокета `SO_REUSEADDR`.

Проверка на наличие ошибки

25-26 Если все вызовы функций `socket` и `bind` окажутся неудачными, мы сообщаем об ошибке и завершаем выполнение. Как и в случае с нашей функцией `tcp_connect` из предыдущего раздела, мы не пытаемся возвратить ошибку из этой функции.

27 Сокет превращается в прослушиваемый сокет с помощью функции `listen`.

Возвращение размера структуры адреса

28-31 Если аргумент addrlenp является непустым указателем, мы возвращаем размер адресов протокола через этот указатель. Это позволяет вызывающему процессу выделять память для структуры адреса сокета, чтобы получить адрес протокола клиента из функции accept (см. также упражнение 11.7).

Пример: сервер времени и даты

В листинге 11.7 показан наш сервер времени и даты из листинга 4.2, переписанный с использованием функции `tcp_listen`.

Листинг 11.7. Сервер времени и даты, переписанный с использованием функции `tcp_listen`

```
//names/daytimetcpsrv1.c
1 #include "unp.h"
2 #include <time.h>

3 int
4 main(int argc, char **argv)
5 {
6     int listenfd, connfd;
7     socklen_t addrlen, len;
8     char = buff[MAXLINE];
9     time_t ticks;
10    struct sockaddr_storage cliaddr;

11    if (argc != 2)
12        err_quit("usage: daytimetcpsrv1 <service or port#>");

13    listenfd = Tcp_listen(NULL, argv[1], &addrlen);

14    for (;;) {
15        len = sizeof(cliaddr);
16        connfd = Accept(listenfd, (SA*)&cliaddr, &len);
17        printf("connection from %s\n", Sock_ntop((SA*)&cliaddr, len));

18        ticks = time(NULL);
19        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
20        Write(connfd, buff, strlen(buff));

21        Close(connfd);
22    }
23 }
```

Ввод имени службы или номера порта в качестве аргумента командной строки

11-12 Нам нужно использовать аргумент командной строки, чтобы задать либо имя службы, либо номер порта. Это упрощает проверку нашего сервера, поскольку связывание с портом 13 для сервера времени и даты требует прав привилегированного пользователя.

Создание прослушиваемого сокета

13 Функция `tcp_listen` создает прослушиваемый сокет. В качестве третьего аргумента мы передаем нулевой указатель, потому что нам безразличен размер структуры адреса, используемого данным семейством: мы будем работать со структурой `sockaddr_storage`.

Цикл сервера

14-22 Функция `accept` ждет соединения с клиентом. Мы выводим адрес клиента, вызывая функцию `sock_ntop`. В случае IPv4 или IPv6 эта функция выводит IP-адрес и номер порта. Мы могли бы использовать функцию `getnameinfo` (описанную в разделе 11.17), чтобы попытаться получить имя узла клиента, но это подразумевает запрос PTR в DNS, что может занять некоторое время, особенно если запрос PTR окажется неудачным. В разделе 14.8 [112] упоминается, что на занятом веб-сервере почти у 25% всех клиентов, соединяющихся с этим сервером, в DNS нет записей типа PTR. Поскольку мы не хотим, чтобы наш сервер (особенно последовательный сервер) в течение нескольких секунд ждал запрос PTR, мы просто выводим IP-адрес и порт.

Пример: сервер времени и даты с указанием протокола

В листинге 11.7 есть небольшая проблема: первый аргумент функции `tcp_listen` — пустой указатель, объединенный с семейством адресов `AF_UNSPEC`, который задает функция `tcp_listen`, — может заставить функцию `getaddrinfo` возвратить структуру адреса сокета с семейством адресов, отличным от желаемого. Например, первой на узле с двойным стеком будет возвращена структура адреса сокета для IPv6 (см. табл. 11.3), но, возможно, нам требуется, чтобы наш сервер обрабатывал только IPv4.

У клиентов такой проблемы нет, поскольку клиент должен всегда задавать либо IP-адрес, либо имя узла. Клиентские приложения обычно позволяют пользователю вводить этот параметр как аргумент командной строки. Это дает нам возможность задавать имя узла, связанное с определенным типом IP-адреса (вспомните наши имена узлов -4 и -6 в разделе 11.2), или же задавать либо строку в точечно-десятичной записи (для IPv4), либо шестнадцатеричную строку (для IPv6).

И для серверов существует простая методика, позволяющая нам указать, какой именно протокол следует использовать — IPv4 или IPv6. Для этого нужно позволить пользователю ввести либо IP-адрес, либо имя узла в качестве аргумента командной строки и передать его функции `getaddrinfo`. В случае IP-адреса строка точечно-десятичной записи IPv4 отличается от шестнадцатеричной строки IPv6. Следующие вызовы функции `inet_pton` оказываются либо успешными либо нет, как это показано в данном случае:

```
inet_pton(AF_INET, "0.0.0.0", &foo); /* успешно */
inet_pton(AF_INET, "0::0", &foo); /* неудачно*/
inet_pton(AF_INET6, "0.0.0.0", &foo); /* неудачно */
inet_pton(AF_INET6, "0::0", &foo); /* успешно */
```

Следовательно, если мы изменим наши серверы таким образом, чтобы они получали дополнительный аргумент, то при вводе

```
% server
по умолчанию мы получим IPv6 на узле с двойным стеком, но при вводе
% server 0.0.0.0
явно задается IPv4, а при вводе
% server 0::0
явно задается IPv6.
```

В листинге 11.8 показана окончательная версия нашего сервера времени и даты.

Листинг 11.8. Не зависящий от протокола сервер времени и даты, использующий функцию `tcp_listen`

```
names/daytimetcpsrv2.c
1 #include "unp.h"
2 #include <time.h>

3 int
4 main(int argc, char **argv)
5 {
6     int listenfd, connfd;
7     socklen_t addrlen, len;
8     struct sockaddr_storage cliaddr;
9     char buff[MAXLINE];
10    time_t ticks;

11    if (argc == 2)
12        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
```

```

13 else if (argc == 3)
14     listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
15 else
16     err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");

17 for (;;) {
18     len = sizeof(cliaddr);
19     connfd = Accept(listenfd, (SA*)&cliaddr, &len);
20     printf("connection from %s\n", Sock_ntop((SA*)&cliaddr, len));

21     ticks = time(NULL);
22     snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
23     Write(connfd, buff, strlen(buff));

24     Close(connfd);
25 }
26 }
```

Обработка аргументов командной строки

11-16 Единственное изменение по сравнению с листингом 11.6 — это обработка аргументов командной строки, позволяющая пользователю в дополнение к имени службы или порту задавать либо имя узла, либо IP-адрес для связывания с сервером.

Сначала мы запускаем этот сервер с сокетом IPv4 и затем соединяемся с сервером от клиентов на двух различных узлах, расположенных в локальной подсети:

```

freebsd % daytimetcpsrv2 0.0.0.0 9999
connection from 192.168.42.2:32961
connection from 192.168.42.2:1389
A теперь мы запустим сервер с сокетом IPv6:
solaris % daytimetcpsrv2 0::0 9999
connection from [3ffe:b80:1f8d:2:204:acff:fe17:bf38]:32964
connection from [3ffe:b80:1f8d:2:230:65ff:fe15:caa7]:49601
connection from [::ffff:192:168:42:3]:32967
connection from [::ffff:192:168:42:3]:49602
```

Первое соединение — от узла aix, использующего IPv6, а второе — от узла macosx, использующего IPv6. Два следующих соединения — от узлов aix и macosx, но они используют IPv4, а не IPv6. Мы можем определить это, потому что оба адреса клиента, возвращаемые функцией accept, являются адресами IPv4, преобразованными к виду IPv6.

Мы только что показали, что сервер IPv6, работающий на узле с двойным стеком, может обрабатывать как клиенты IPv4, так и клиенты IPv6. Адреса IPv4-клиента передаются серверу IPv6 как адреса IPv4, преобразованные к виду IPv6, что мы рассматривали в разделе 12.2.

11.14. Функция udp_client

Наши функции, предоставляющие более простой интерфейс для функции getaddrinfo, в случае UDP изменяются: в этом разделе мы представляем клиентскую функцию, создающую неприсоединенный сокет UDP, а в следующем — другую функцию, создающую присоединенный сокет UDP.

```
#include "unp.h"
```

```

int udp_client(const char *hostname, const char *service,
    void **saptr, socklen_t *lencp);
```

Возращает: дескриптор неприсоединенного сокета в случае успешного выполнения, в случае ошибки не возвращает ничего

Эта функция создает неприсоединенный сокет UDP, возвращая три элемента. Во-первых, возвращаемое значение функции — это дескриптор сокета. Во-вторых, saptr — это адрес указателя

(объявляемого вызывающим процессом) на структуру адреса сокета (которая динамически размещается в памяти функцией `udp_client`), и в этой структуре функция хранит IP-адрес получателя и номер порта для будущих вызовов функции `sendto`. Размер этой структуры адреса сокета возвращается как значение переменной, на которую указывает `lenp`. Последний аргумент не может быть пустым указателем (как это допустимо для последнего аргумента функции `tcp_listen`), поскольку длина структуры адреса сокета требуется в любых вызовах функций `sendto` и `recvfrom`.

В листинге 11.9 показан исходный код для этой функции.

Листинг 11.9. Функция `udp_client`: создание неприсоединенного сокета UDP

```
//lib/udp_client.c
1 #include "unp.h"

2 int
3 udp_client(const char *host, const char *serv, void **saptr, socklen_t *lenp)
4 {
5     int sockfd, n;
6     struct addrinfo hints, *res, *ressave;

7     bzero(&hints, sizeof(struct addrinfo));
8     hints.ai_family = AF_UNSPEC;
9     hints.ai_socktype = SOCK_DGRAM;

10    if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
11        err_quit("udp_client error for %s, %s: %s",
12                host, serv, gai_strerror(n));
13    ressave = res;

14    do {
15        sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16        if (sockfd >= 0)
17            break; /* успех */
18    } while ((res = res->ai_next) != NULL);

19    if (res == NULL) /* значение errno устанавливается при последнем
                      вызове функции socket() */
20        err_sys("udp_client error for %s, %s", host, serv);

21    *saptr = Malloc(res->ai_addrlen);
22    memcpy(*saptr, res->ai_addr, res->ai_addrlen);
23    *lenp = res->ai_addrlen;

24    freeaddrinfo(ressave);

25    return (sockfd);
26 }
```

Функция `getaddrinfo` преобразует аргументы `hostname` и `service`. Создается дейтаграммный сокет. Выделяется память для одной структуры адреса сокета, и структура адреса сокета, соответствующая созданному сокету, копируется в память.

Пример: не зависящий от протокола UDP-клиент времени и даты

Теперь мы перепишем наш клиент времени и даты, показанный в листинге 11.3, так, чтобы в нем использовалась наша функция `udp_client`. В листинге 11.10 представлен не зависящий от протокола исходный код.

Листинг 11.10. UDP-клиент времени и даты, использующий нашу функцию `udp_client`

```
//names/daytimeudpcl1.c
```

```

1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd, n;
6     char recvline[MAXLINE + 1];
7     socklen_t salen;
8     struct sockaddr *sa;

9     if (argc != 3)
10    err_quit
11    ("usage: daytimeudpcl1 <hostname/IPaddress> <service/port#>");

12    sockfd = Udp_client(argv[1], argv[2], (void**)&sa, &salen);

13    printf("sending to %s\n", Sock_ntop_host(sa, salen));

14    Sendto(sockfd, "", 1, 0, sa, salen); /* посылается 1-байтовая
                                         дейтаграмма */

15    n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
16    recvline[n] = 0; /* завершающий пустой байт */
17    Fputs(recvline, stdout);

18    exit(0);
19 }

```

12-17 Мы вызываем нашу функцию `udp_client` и затем выводим IP-адрес и порт сервера, которому мы отправим нашу дейтаграмму UDP. Мы посылаем однобайтовую дейтаграмму и затем читаем и выводим ответ сервера.

ПРИМЕЧАНИЕ

Нам достаточно отправить дейтаграмму, содержащую 0 байт, поскольку ответ сервера времени и даты инициируется самим получением дейтаграммы от клиента, независимо от ее длины и содержания. Но многие реализации SVR4 не допускают нулевой длины дейтаграмм UDP.

Мы запускаем наш клиент, задавая имя узла с записью типа AAAA и типа A. Поскольку функция `getaddrinfo` в первую очередь возвращает структуру с записью типа AAAA, создается сокет IPv6:

```
freebsd % daytimeudpcl1 aix daytime
sending to 3ffe:b80:1f8d:2:204:acff:fe17:bf38
Sun Jul 23:21:12 2003
```

Затем мы задаем адрес того же узла в точечно-десятичной записи, в результате чего создается сокет IPv4:

```
freebsd % daytimeudpcl1 192.168.42.2 daytime
sending to 192.168.42.2
Sun Jul 23:21:40 2003
```

11.15. Функция `udp_connect`

Наша функция `udp_connect` создает присоединенный сокет UDP.

```
#include "unp.h"
```

```
int udp_connect(const char *hostname, const char *service);
```

Возвращает; дескриптор присоединенного сокета в случае успешного выполнения, в случае ошибки ничего не возвращает

В случае присоединенного сокета UDP два последних аргумента, которые требовались в функции `udp_client`, больше не нужны. Вызывающий процесс может вызвать функцию `write` вместо `sendto`, таким образом нашей функции не нужно возвращать структуру адреса сокета и ее длину. В листинге 11.11 представлен исходный код.

Листинг 11.11. Функция `udp_connect`: создание присоединенного сокета UDP

```
//lib/udp_connect.c
1 #include "unp.h"

2 int
3 udp_connect(const char *host, const char *serv)
4 {
5     int sockfd, n;
6     struct addrinfo hints, *res, *ressave;

7     bzero(&hints, sizeof(struct addrinfo));
8     hints.ai_family = AF_UNSPEC;
9     hints.ai_socktype = SOCK_DGRAM;

10    if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
11        err_quit("udp_connect error for %s, %s: %s",
12                host, serv, gai_strerror(n));
13    ressave = res;

14    do {
15        sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16        if (sockfd < 0)
17            continue; /* игнорируем этот адрес */
18
19        if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
20            break; /* успех */

21    } while ((res = res->ai_next) != NULL);

22    if (res == NULL) /* значение errno устанавливается при
                       последнем вызове функции connect() */
23    err_sys("udp_connect error for %s, %s", host, serv);

24    freeaddrinfo(ressave);

25    return (sockfd);
26 }
```

Эта функция почти идентична функции `tcp_connect`. Однако отличие в том, что при вызове функции `connect` для сокета UDP ничего не отправляется собеседнику. Если что-то не в порядке (собеседник недоступен или на заданном порте не запущен сервер), вызывающий процесс не обнаружит этого, пока не пошлет собеседнику дейтаграмму.

11.16. Функция `udp_server`

Наша последняя функция, предоставляющая более простой интерфейс для функции `getaddrinfo`, — это функция `udp_server`.

```
#include "unp.h"
```

```
int udp_server(const char *hostname, const char *service, socklen_t *lenptr);
```

Возвращает; дескриптор неприсоединенного сокета в случае успешного выполнения, в случае ошибки не возвращает ничего

Аргументы функции те же, что и для функции `tcp_listen`: необязательный `hostname`, обязательный `service` (для связывания номера порта) и необязательный указатель на переменную, в которой возвращается размер структуры адреса сокета. В листинге 11.12 представлен исходный код.

Листинг 11.12. Функция `udp_server`: создание неприсоединенного сокета для сервера UDP

```
//lib/udp_server.c
1 #include "unp.h"

2 int
3 udp_server(const char *host, const char *serv, socklen_t *addrlenp)
4 {
5     int sockfd, n;
6     struct addrinfo hints, *res, *ressave;

7     bzero(&hints, sizeof(struct addrinfo));
8     hints.ai_flags = AI_PASSIVE;
9     hints.ai_family = AF_UNSPEC;
10    hints.ai_socktype = SOCK_DGRAM;

11    if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
12        err_quit("udp_server error for %s, %s: %s",
13                 host, serv, gai_strerror(n));
14    ressave = res;

15    do {
16        sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
17        if (sockfd < 0)
18            continue; /* ошибка, пробуем следующий адрес */

19        if (bind(sockfd, res->ai_addr, res->ai_addrlen) == 0)
20            break; /* успех */

21        Close(sockfd); /* ошибка при вызове функции bind, закрываем
                           сокет и пробуем следующий адрес */
22    } while ((res = res->ai_next) != NULL);

23    if (res == NULL) /* значение errno устанавливается при
                       последнем вызове функции socket() or bind() */
24        err_sys("udp_server error for %s, %s", host, serv);

25    if (addrlenp)
26        *addrlenp = res->ai_addrlen; /* возвращается размер адреса
                                         протокола */
27    freeaddrinfo(ressave);

28    return (sockfd);
29 }
```

Эта функция практически идентична функции `tcp_listen`, в ней нет только вызова функции `listen`. Мы устанавливаем семейство адресов `AF_UNSPEC`, но вызывающий процесс может использовать ту же технологию, которую мы описали при рассмотрении листинга 11.6, чтобы потребовать использование определенного протокола (IPv4 или IPv6).

Мы не устанавливаем параметр сокета `SO_REUSEADDR` для сокета UDP, поскольку этот параметр сокета может допустить связывание множества сокетов с одним и тем же портом UDP на узлах, поддерживающих многоадресную передачу, как мы говорили в разделе 7.5. Поскольку у сокета UDP нет аналога состояния

TIME_WAIT, свойственного сокетам TCP, нет необходимости устанавливать этот параметр при запуске сервера.

Пример: не зависящий от протокола UDP-сервер времени и даты

В листинге 11.13 представлен наш сервер времени и даты, полученный путем модификации листинга 11.8 и предназначенный для использования UDP.

Листинг 11.13. Не зависящий от протокола UDP-сервер времени и даты

```
//names/daytimeudpsrv2.c
1 #include "unp.h"
2 #include <time.h>

3 int
4 main(int argc, char **argv)
5 {
6     int sockfd;
7     ssize_t n;
8     char buff[MAXLINE];
9     time_t ticks;
10    socklen_t addrlen, len;
11    struct sockaddr_storage cliaddr;

12    if (argc == 2)
13        sockfd = Udp_server(NULL, argv[1], &addrlen);
14    else if (argc == 3)
15        sockfd = Udp_server(argv[1], argv[2], &addrlen);
16    else
17        err_quit("usage: daytimeudpsrv [ <host> ] <service or port>");

18    for (;;) {
19        len = sizeof(cliaddr);
20        n = Recvfrom(sockfd, buff, MAXLINE, 0, (SA*)&cliaddr, &len);
21        printf("datagram from %s\n", Sock_ntop((SA*)&cliaddr, len));

22        ticks = time(NULL);
23        snprintf(buff, sizeof(buff), "% 24s\r\n", ctime(&ticks));
24        Sendto(sockfd, buff, strlen(buff), 0, (SA*)&cliaddr, len);
25    }
26 }
```

11.17. Функция getnameinfo

Эта функция дополняет функцию getaddrinfo: она получает адрес сокета и возвращает одну символьную строку с описанием узла и другую символьную строку с описанием службы. Эта функция предоставляет указанную информацию в не зависящем от протокола виде, то есть вызывающему процессу неважно, какой тип адреса протокола содержится в структуре адреса сокета, поскольку эти подробности обрабатываются функцией.

```
#include <netdb.h>
```

```
int getnameinfo(const struct sockaddr *sockaddr, socklen_t addrlen, char *host,
    size_t hostlen, char *serv, size_t servlen, int flags);
```

Возвращает 0 в случае успешного выполнения, -1 в случае ошибки

Аргумент sockaddr указывает на структуру адреса сокета, содержащую адрес протокола, преобразуемый в строку, удобную для человеческого восприятия, а аргумент addrlen содержит длину этой

структурой. Эта структура и ее длина обычно возвращаются любой из следующих функций: `accept`, `recvfrom`, `getsockname` или `getpeername`.

Вызывающий процесс выделяет в памяти пространство для двух строк, удобных для человеческого восприятия: аргументы `host` и `hostlen` определяют строку, описывающую узел, а аргументы `serv` и `servlen` определяют строку, которая описывает службы. Если вызывающему процессу не нужна возвращаемая строка с описанием узла, задается нулевая длина этой строки (`hostlen`). Аналогично, нулевое значение аргумента `servlen` означает, что не нужно возвращать информацию о службе.

Разница между функциями `sock_ntop` и `getnameinfo` состоит в том, что первая не задействует DNS, а только возвращает IP-адрес и номер порта. Последняя же обычно пытается получить имя и для узла, и для службы.

В табл. 11.4 показаны шесть флагов, которые можно задать для изменения действия, выполняемого функцией `getnameinfo`.

Таблица 11.4. Флаги функции `getnameinfo`

Константа	Описание
<code>NI_DGRAM</code>	Дейтаграммный сокет
<code>NI_NAMEREQD</code>	Возвращать ошибку, если невозможно получить имя узла по его адресу
<code>NI_NOFQDN</code>	Возвращать только ту часть FQDN, которая содержит имя узла
<code>NI_NUMERICHOST</code>	Возвращать численное значение адреса вместо имени узла
<code>NI_NUMERICSCOPE</code>	Возвращать численное значение идентификатора области
<code>NI_NUMERICSERV</code>	Возвращать номер порта вместо имени службы

■ Флаг `NI_DGRAM` должен быть задан, когда вызывающий процесс знает, что работает с дейтаграммным сокетом. Причина в том, что если функции `getnameinfo` задать только IP-адрес и номер порта в структуре адреса сокета, она не сможет определить протокол (TCP или UDP). Существует несколько номеров портов, которые в случае TCP задействованы для одной службы, а в случае UDP для совершенно другой. Примером может служить порт 514, используемый службой `rsh` в TCP и службой `syslog` в UDP.

■ Флаг `NI_NAMEREQD` приводит к возвращению ошибки, если имя узла не может быть разрешено при использовании DNS. Этот флаг может использоваться серверами, которым требуется, чтобы IP-адресу клиента было сопоставлено имя узла. Затем эти серверы получают возвращаемое имя узла, вызывают функцию `gethostbyname` и проверяют, совпадают ли результаты вызова этих двух функций хотя бы частично.

■ Флаг `NI_NOFQDN` вызывает сокращение имени узла, отбрасывая все, что идет после первой точки. Например, если в структуре адреса сокета содержится IP-адрес 192.168.42.2, то функция `gethostbyaddr` возвратит имя `aix.unpbook.com`. Но если в функции `getnameinfo` задан флаг `NI_NOFQDN`, она возвратит в имени узла только `aix`.

■ Флаг `NI_NUMERICHOST` сообщает функции `getnameinfo`, что не нужно вызывать DNS (поскольку это занимает некоторое время). Вместо этого возвращается численное представление IP-адреса, вероятно, при помощи вызова функции `inet_ntop`. Аналогично, флаг `NI_NUMERICSERV` определяет, что вместо имени службы должен быть возвращен десятичный номер порта. Обычно серверы должны задавать этот флаг, поскольку номера портов клиента, как правило, не имеют соответствующего имени службы — это динамически назначаемые порты. `NI_NUMERICSCOPE` указывает на необходимость возвращения идентификатора области в численном, а не в текстовом виде.

Можно объединять несколько флагов путем логического сложения, если их сочетание имеет смысл, например `NI_DGRAM` и `NI_NUMERICHOST`.

11.18. Функции, допускающие повторное вхождение

Функция `gethostbyname` из раздела 11.3 имеет интересную особенность, которую мы еще не рассматривали: она не допускает повторное вхождение (nonreentrant). Мы еще столкнемся с этой проблемой в главе 23, когда будем обсуждать потоки, но не менее интересно найти решение этой проблемы сейчас, без необходимости обращаться к понятию потоков.

Сначала посмотрим, как эта функция работает. Если мы изучим ее исходный код (это несложно, поскольку исходный код для всей реализации BIND свободно доступен), то увидим, что обе функции — и `gethostbyname`, и `gethostbyaddr` — содержатся в одном файле, который имеет следующий вид:

```

static struct hostent host; /* здесь хранится результат */

struct hostent*
gethostbyname(const char *hostname) {
    return(gethostbyname2(hostname, family));
}

struct hostent*
gethostbyname2(const char *hostname, int family) {
    /* вызов функций DNS для запроса A или AAAA */

    /* заполнение структуры адреса узла */
    return(&host);
}

struct hostent*
gethostbyaddr(const char *addr, size_t len, int family) {
    /* вызов функций DNS для запроса PTR в домене in-addr.arpa */

    /* заполнение структуры адреса узла */
    return(&host);
}

```

Мы выделили полужирным шрифтом спецификатор класса памяти `static` итоговой структуры, потому что основная проблема в нем. Тот факт, что эти три функции используют общую переменную `host`, представляет другую проблему, которую мы обсудим в упражнении 11.1. (Вспомните табл. 11.4.) Функция `gethostbyname2` появилась в BIND 4.9.4 с добавлением поддержки IPv6. Мы будем игнорировать тот факт, что когда мы вызываем функцию `gethostbyname`, задействуется функция `gethostbyname2`, поскольку это не относится к предмету обсуждения.

Проблема повторного вхождения может возникнуть в нормальном процессе Unix, вызывающем функцию `gethostbyname` или `gethostbyaddr` и из управляющего элемента главного потока, и из обработчика сигнала. Когда вызывается обработчик сигнала (допустим, это сигнал `SIGALRM`, который генерируется раз в секунду), главный поток управляющего элемента процесса временно останавливается и вызывается функция обработки сигнала. Рассмотрим следующую ситуацию:

```

main() {
    struct hostent *hptr;
    ...
    signal(SIGALRM, sig_alrm);
    ...
    hptr = gethostbyname( ... );
    ...
}

void
sig_alrm(int signo) {
    struct hostent *hptr;
    ...
    hptr = gethostbyname( ... );
    ...
}

```

Если главный поток управления в момент остановки находится в середине выполнения функции `gethostbyname` (допустим, функция заполнила переменную `host` и должна сейчас возвратить управление), а затем обработчик сигналов вызывает функцию `gethostbyname`, то поскольку в процессе существует только один экземпляр переменной `host`, эта переменная используется снова. При этом значения переменных, вычисленные при вызове из главного потока управления, заменяются значениями, вычисленными при вызове из обработчика сигнала.

Если мы посмотрим на функции преобразования имен и адресов, представленные в этой главе и в главе 9, вместе с функциями `inet_XXX` из главы 4, мы заметим следующее:

- Функции `gethostbyname`, `gethostbyname2`, `gethostbyaddr`, `getservbyname` и `getservbyport` традиционно не допускают повторного вхождения, поскольку все они возвращают указатель на статическую структуру.

Некоторые реализации, поддерживающие программные потоки (Solaris 2.x), предоставляют версии этих четырех функций, допускающие повторное вхождение, с именами, оканчивающимися суффиксом `_r`. О них рассказывается в следующем разделе.

В качестве альтернативы некоторые реализации с поддержкой программных потоков (Digital Unix 4.0 и HP_UX 10.30) предоставляют версии этих функций, допускающие повторное вхождение за счет использования собственных данных программных потоков.

- Функции `inet_pton` и `inet_ntop` всегда допускают повторное вхождение.
- Исторически функция `inet_ntoa` не допускает повторное вхождение, но некоторые реализации с поддержкой потоков предоставляют версию, допускающую повторное вхождение, которая строится на основе собственных данных потоков.
- Функция `getaddrinfo` допускает повторное вхождение, только если она сама вызывает функции, допускающие повторное вхождение, то есть если она вызывает соответствующую версию функции `gethostbyname` или `getservbyname` для имени узла или имени службы. Одной из причин, по которым вся память для результатов ее выполнения выделяется динамически, является возможность повторного вхождения.

■ Функция `getnameinfo` допускает повторное вхождение, только если она сама вызывает такие функции, то есть если она вызывает соответствующую версию функции `gethostbyaddr` для получения имени узла или функции `getservbyport` для получения имени службы. Обратите внимание, что обе результирующих строки (для имени узла и для имени службы) размещаются в памяти вызывающим процессом, чтобы обеспечить возможность повторного вхождения.

Похожая проблема возникает с переменной `errno`. Исторически существовало по одной копии этой целочисленной переменной для каждого процесса. Если процесс выполняет системный вызов, возвращающий ошибку, то в этой переменной хранится целочисленный код ошибки. Например, функция `close` из стандартной библиотеки языка С может выполнить примерно такую последовательность действий:

- поместить аргумент системного вызова (целочисленный дескриптор) в регистр;
- поместить значение в другой регистр, указывая, что был сделан системный вызов функции `close`;
- активизировать системный вызов (переключиться на ядро со специальной инструкцией);
- проверить значение регистра, чтобы увидеть, что произошла ошибка;
- если ошибки нет, возвратить (0);
- сохранить значение какого-то другого регистра в переменной `errno`;
- возвратить (-1).

Прежде всего заметим, что если ошибки не происходит, значение переменной `errno` не изменяется. Поэтому мы не можем посмотреть значение этой переменной, пока мы не узнаем, что произошла ошибка (обычно на это указывает возвращаемое функцией значение -1).

Будем считать, что программа проверяет возвращаемое значение функции `close` и затем выводит значение переменной `errno`, если произошла ошибка, как в следующем примере:

```
if (close(fd) < 0) {  
    fprintf(stderr, "close error, errno = %d\n", errno);  
    exit(1);  
}
```

Существует небольшой промежуток времени между сохранением кода ошибки в переменной `errno` в тот момент, когда системный вызов возвращает управление, и выводом этого значения программой. В течение этого промежутка другой программный поток внутри процесса (то есть обработчик сигналов) может изменить значение переменной `errno`. Если, например, при вызове обработчика сигналов главный поток управления находится между `close` и `fprintf` и обработчик сигналов делает какой-то другой системный вызов, возвращающий ошибку (допустим, вызывается функция `write`), то значение переменной `errno`, записанное при вызове функции `close`, заменяется на значение, записанное при вызове функции `write`.

При рассмотрении этих двух проблем в связи с обработчиками сигналов одним из решений проблемы с функцией `gethostbyname` (возвращающей указатель на статическую переменную) будет не вызывать из

обработчика сигнала функции, которые не допускают повторное вхождение. Проблемы с переменной `errno` (одна глобальная переменная, которая может быть изменена обработчиком сигнала) можно избежать, перекодировав обработчик сигнала так, чтобы он сохранял и восстанавливал значение переменной `errno` следующим образом:

```
void sig_alrm(int signo) {
    int errno_save;

    errno_save = errno; /* сохраняем значение этой переменной
                         при вхождении */

    if (write( ... ) != nbytes)
        fprintf(stderr, "write error, errno = %d\n", errno);
    errno = errno_save; /* восстанавливаем значение этой переменной
                         при завершении */
}
```

В этом коде мы также вызываем функцию `fprintf`, стандартную функцию ввода-вывода, из обработчика сигнала. Это еще одна проблема повторного вхождения, поскольку многие версии функций стандартной библиотеки ввода-вывода не допускают повторного вхождения: стандартные функции ввода-вывода не должны вызываться из обработчиков сигналов.

Мы вернемся к проблеме повторного вхождения в главе 26 и увидим, как проблема с переменной `errno` решается с помощью потоков. В следующем разделе описываются некоторые версии функций имен узлов, допускающие повторное вхождение.

11.19. Функции `gethostbyname_r` и `gethostbyaddr_r`

Чтобы превратить функцию, не допускающую повторное вхождение, такую как `gethostbyname`, в повторно входимую, можно воспользоваться двумя способами.

1. Вместо заполнения и возвращения статической структуры вызывающий процесс размещает структуру в памяти, и функция, допускающая повторное вхождение, заполняет эту структуру. Эта технология используется для перехода от функции `gethostbyname` (которая не допускает повторное вхождение) к функции `gethostbyname_r` (которая допускает повторное вхождение). Но это решение усложняется, поскольку помимо того, что вызывающий процесс должен предоставить структуру `hostent` для заполнения, эта структура также указывает на другую информацию: каноническое имя, массив указателей на псевдонимы, строки псевдонимов, массив указателей на адреса и сами адреса (см., например, рис. 11.2). Вызывающий процесс должен предоставить один большой буфер, используемый для дополнительной информации, и заполняемая структура `hostent` будет содержать различные указатели на этот буфер. При этом добавляется как минимум три аргумента функции: указатель на заполняемую структуру `hostent`, указатель на буфер, используемый для всей прочей информации, и размер этого буфера. Требуется также четвертый дополнительный аргумент — указатель на целое число, в котором будет храниться код ошибки, поскольку глобальная целочисленная переменная `h_errno` больше не может использоваться. (Глобальная целочисленная переменная `h_errno` создает ту же проблему повторного вхождения, которая описана нами для переменной `errno`.)

Эта технология также используется функциями `getnameinfo` и `inet_ntop`.

2. Входящая функция вызывает функцию `malloc` и динамически выделяет память. Это технология, используемая функцией `getaddrinfo`. Проблема при таком подходе заключается в том, что приложение,зывающее эту функцию, должно вызвать также функцию `freeaddrinfo`, чтобы освободить динамическую память. Если эта функция не вызывается, происходит утечка памяти: каждый раз, когда процесс вызывает функцию, выделяющую память, объем памяти, задействованной процессом, возрастает. Если процесс выполняется в течение длительного времени (что свойственно сетевым серверам), то потребление памяти этим процессом с течением времени неуклонно растет.

Обсудим функции Solaris 2.x, допускающие повторное вхождение, не используемые для сопоставления имен с адресами, и наоборот (то есть для разрешения имен).

```
#include <netdb.h>

struct hostent *gethostbyname_r(const char *hostname,
                                struct hostent *result, char *buf, int buflen, int *h_errnop);
struct hostent *gethostbyaddr_r(const char *addr, int len,
```

```
int type, struct hostent *result, char *buf, int buflen,
int *h_errnop);
```

Обе функции возвращают: непустой указатель в случае успешного выполнения, `NULL` в случае ошибки

Для каждой функции требуется четыре дополнительных аргумента. Аргумент `result` — это структура `hostent`, размещенная в памяти вызывающим процессом и заполняемая данной функцией. При успешном выполнении функции этот указатель также является возвращаемым значением.

Аргумент `buf` — это буфер, размещенный в памяти вызывающим процессом, а `buflen` — его размер. Буфер будет содержать каноническое имя, массив указателей на псевдонимы, строки псевдонимов, массив указателей на адреса и сами адреса. Все указатели в структуре `hostent`, на которую указывает `result`, указывают на этот буфер. Насколько большим должен быть этот буфер? К сожалению, все, что сказано в большинстве руководств, это что-то неопределенное вроде «Буфер должен быть достаточно большим, чтобы содержать все данные, связанные с записью узла». Текущие реализации функции `gethostbyname` могут возвращать до 35 указателей на альтернативные имена (псевдонимы), до 35 указателей на адреса и использовать буфер размером 8192 байт для хранения альтернативных имен (псевдонимов) и адресов. Поэтому буфер размером 8192 байт можно считать подходящим.

Если происходит ошибка, код ошибки возвращается через указатель `h_errnop`, а не через глобальную переменную `h_errno`.

ПРИМЕЧАНИЕ

К сожалению, проблема повторного вхождения гораздо серьезнее, чем может показаться. Во-первых, не существует стандарта относительно повторного вхождения и функций `gethostbyname` и `gethostbyaddr`. POSIX утверждает, что эти две функции не обязаны быть безопасными в многопоточной среде.

Во-вторых, не существует стандарта для функций `_g`. В этом разделе (в качестве примера) мы привели две функции `_g`, предоставляемые Solaris 2.x. В Linux присутствуют аналогичные функции, возвращающие `hostent` в качестве аргумента типа значение-результат. В Digital Unix и HP-UX имеются версии этих функций с другими аргументами. Первые два аргумента функции `gethostbyname_g` такие же, как и в версии Solaris, но оставшиеся три аргумента версии Solaris объединены в новую структуру `hostent_data` (которая должна быть размещена в памяти вызывающим процессом), а указатель на эту структуру — это третий и последний аргумент. Обычные функции `gethostbyname` и `gethostbyaddr` в Digital Unix 4.0 и в HP-UX 10.30 допускают повторное вхождение при использовании собственных данных потоков (см. раздел 23.5). Интересный рассказ о разработке функций `_g` Solaris 2.x содержится в [70].

Наконец, хотя версия функции `gethostbyname`, допускающая повторное вхождение, может обеспечить безопасность, когда ее одновременно вызывают несколько различных потоков, это ничего не говорит нам о возможности повторного вхождения для лежащих в ее основе функций распознавателя.

11.20. Устаревшие функции поиска адресов IPv6

В процессе разработки IPv6 интерфейс поиска адресов IPv6 много раз претерпевал серьезные изменения. В какой-то момент интерфейс был сочен усложненным и недостаточно гибким, так что от него полностью отказались в RFC 2553 [38]. Документ RFC 2553 предлагал собственные функции, которые в RFC 3493 [36] были попросту заменены `getaddrinfo` и `getnameinfo`. В этом разделе мы вкратце рассмотрим старые интерфейсы на тот случай, если вам придется переписывать программы, использующие их.

Константа RES_USE_INET6

Поскольку функция `gethostbyname` не имеет аргумента для указания нужного семейства адресов (подобного `hints.ai_family` для `getaddrinfo`), в первом варианте API использовалась константа `RES_USE_INET6`, которая должна была добавляться к флагам распознавателя посредством внутреннего

интерфейса. Этот API был недостаточно переносимым, поскольку системам, использовавшим альтернативные внутренние интерфейсы распознавателя, приходилось имитировать интерфейс BIND.

Включение `RES_USE_INET6` приводило к тому, что функция `gethostbyname` начинала поиск с записей AAAA, а записи A возвращались только в случае отсутствия первых. Поскольку в структуре `hostent` есть только одно поле длины адреса, функция `gethostbyname` могла возвращать адреса только одного типа (либо IPv6, либо IPv4).

Кроме того, включение `RES_USE_INET6` приводило к тому, что функция `gethostbyname2` начинала возвращать адреса IPv4 в преобразованном к IPv6 виде.

Функция `gethostbyname2`

Функция `gethostbyname2` имеет добавочный аргумент, позволяющий задать семейство адресов.

```
#include <netdb.h>
```

```
struct hostent *gethostbyname2(const char *hostname, int family);
```

Возвращает: непустой указатель в случае успешного выполнения, в случае ошибки возвращает NULL и задает значение переменной h_errno

Возвращаемое значение то же, что и у функции `gethostbyname` — указатель на структуру `hostent`, и сама эта структура устроена так же. Логика функции зависит от аргумента `family` и параметра распознавателя `RES_USE_INET6` (который мы упомянули в конце предыдущего раздела).

Функция `getipnodebyname`

Документ RFC 2553 [38] запретил использование `RES_USE_INET6` и `gethostbyname2` из-за глобальности флага `RES_USE_INET6` и желания предоставить больше возможностей по управлению возвращаемыми сведениями. Для решения перечисленных проблем была предложена функция `getipnodebyname`.

```
#include <sys/socket.h>
#include <netdb.h>
```

```
struct hostent *getipnodebyname(const char *name, int af,
    int flags, int *error_num);
```

Возвращает: ненулевой указатель в случае успешного завершения, нулевой в случае ошибки

Функция возвращает указатель на ту же структуру `hostent`, которая использовалась `gethostbyname`.

Аргументы `af` и `flags` непосредственно соответствуют полям `hints.ai_family` и `hints.ai_flags`. Для обеспечения безопасности в многопоточной среде возвращаемое значение выделяется динамически, поэтому его приходится освобождать вызовом `freehostent`.

```
#include <netdb.h>
```

```
void freehostent(struct hostent *ptr);
```

Функции `getipnodebyname` и `getipnodebyaddr` были отменены в RFC 3493 [36], а вместо них было предложено использовать `getaddrinfo` и `getnameinfo`.

11.21. Другая информация о сетях

В этой главе мы сфокусировали внимание на именах узлов, IP-адресах, именах и номерах портов служб. Если же обобщить полученную информацию, мы увидим, что существует четыре типа данных (имеющих отношение к сетям), которые могут понадобиться приложению: узлы, сети, протоколы и службы. В большинстве случаев происходит поиск данных, относящихся к узлам (функции `gethostbyname` и `gethostbyaddr`), реже — к службам (функции `getservbyname` и `getservbyaddr`) и еще реже — к сетям и протоколам.

Все четыре типа данных могут храниться в файле, и для каждого из четырех типов определены три функции:

1. Функция `getXXXent`, читающая следующую запись в файле, при необходимости открывая файл.
2. Функция `setXXXent`, которая открывает файл (если он еще не открыт) и переходит к началу файла.
3. Функция `endXXXent`, закрывающая файл.

Для каждого из четырех типов данных определяется его собственная структура (соответственно, структуры `hostent`, `netent`, `protoent` и `servent`), что требует включения заголовка `<netdb.h>`.

В дополнение к трем функциям `get`, `set` и `end`, которые допускают последовательную обработку файла, для каждого из четырех типов данных предоставляются функции *ключевого поиска*, или *поиска по ключу* (*keyed lookup*). Эти функции последовательно проходят файл (вызывая функцию `getXXXent` для чтения каждой строки файла), но вместо того чтобы возвращать каждую строку вызывающему процессу, эти функции ищут элемент, совпадающий с аргументом. Имена функций поиска по ключу имеют вид `getXXXbyYYY`. Например, две функции *ключевого поиска* для информации об узле — это функции `gethostbyname` (ищет элемент, совпадающий с именем узла) и `gethostbyaddr` (ищет элемент, совпадающий с IP-адресом). Таблица 11.5 обобщает эту информацию.

Таблица 11.5. Четыре типа данных, относящихся к сетям

Тип данных	Файл	Структура	Функции поиска по ключу
Узлы	<code>/etc/hosts</code>	<code>Hostent</code>	<code>gethostbyaddr</code> , <code>gethostbyname</code>
Сети	<code>/etc/networks</code>	<code>Netent</code>	<code>getnetbyaddr</code> , <code>getnetbyname</code>
Протоколы	<code>/etc/protocols</code>	<code>Protoent</code>	<code>getprotobynumber</code> , <code>getprotobyname</code>
Службы	<code>/etc/services</code>	<code>Servent</code>	<code>getservbyname</code> , <code>getservbyport</code>

Как это применяется, если используется DNS? Прежде всего, с помощью DNS возможен доступ только к информации об узле и о сети. Информация о протоколе и службах всегда считывается из соответствующего файла. Ранее в этой главе мы отмечали (см. подраздел «Альтернативы DNS»), что в разных реализациях отличаются способы, с помощью которых администратор определяет, что именно использовать для получения информации об узле и сети — DNS или файл.

Далее, если DNS используется для получения информации об узле и о сети, имеют смысл только функции поиска по ключу. Используя, например, функцию `gethostent`, не стоит надеяться, что она выполнит последовательный перебор всех записей DNS! Если вызывается функция `gethostent`, она считывает только информацию об узлах и не использует DNS.

ПРИМЕЧАНИЕ

Хотя информацию о сети можно сделать доступной с помощью DNS, очень немногие пользуются этим. На с. 347-348 [1] рассказывается об этой возможности. Однако обычно администраторы создают и обслуживают файл `/etc/networks`, используемый вместо DNS. Программа `netstat` с параметром `-i` использует этот файл, если он есть, и выводит имя каждой сети. Однако бесклассовая адресация (см. раздел А.4) делает эти функции бесполезными, а поскольку они не поддерживают IPv6, новые приложения не должны использовать их.

11.22. Резюме

Набор функций, вызываемых приложением для преобразования имени узла в IP-адрес и обратно, называется распознавателем. Две функции, `gethostbyname` и `gethostbyaddr`, являются типичными точками входа. С переходом на IPv6 и многопоточное программирование полезными становятся `getaddrinfo` и `getnameinfo`, способные работать с адресами IPv6 и безопасные в многопоточной среде.

Для работы с именами служб и номерами портов широко используется функция `getservbyname`, принимающая имя службы и возвращающая структуру, содержащую номер порта. Преобразование чаще всего осуществляется на основании данных, содержащихся в некотором текстовом файле. Существует возможность сопоставления имен и номеров протоколов, а также имен и номеров сетей, но используется она реже.

Альтернативой DNS, которую мы не упомянули, является непосредственный вызов функций распознавателя вместо использования функций `gethostbyname` и `gethostbyaddr`. Таким способом пользуется, например, программа `sendmail`, предназначенная для поиска записи типа MX, чего не может сделать функция `gethostbyXXX`. У функций распознавателя имена начинаются с `res_`. Примером такой функции является функция `res_init`, которую мы описали в разделе 11.4. Описание этих функций и пример вызывающей их программы находятся в главе 15 книги [1]. При вводе в командной строке `man resolver` должны отобразиться страницы руководства для этих функций.

Упражнения

1. Измените программу, представленную в листинге 11.1, так, чтобы для каждого возвращаемого адреса вызывалась функция `gethostbyaddr`, а затем выведите возвращаемое имя `h_name`. Сначала запустите программу, задав имя узла только с одним IP-адресом, а затем — с несколькими IP-адресами. Что происходит?
 2. Устраните проблему, показанную в предыдущем упражнении.
 3. Запустите программу, показанную в листинге 11.4, задав имя службы `chargen`.
4. Запустите программу, показанную в листинге 11.4, задав IP-адрес в точечно-десятичной записи в качестве имени узла. Допускает ли это ваш распознаватель? Измените листинг 11.4, чтобы разрешить IP-адрес в виде строки десятичных чисел с точками в качестве имени узла и строку с десятичным номером порта в качестве имени службы. В каком порядке должно выполняться тестирование IP-адреса для строки в точечно-десятичной записи и для имени?
5. Измените программу в листинге 11.4 так, чтобы можно было работать либо с IPv4, либо с IPv6.
6. Измените программу в листинге 8.5 так, чтобы сделать запрос DNS, и сравните возвращаемый IP-адрес со всеми IP-адресами узла получателя, то есть вызовите функцию `gethostbyaddr`, используя IP-адрес, возвращаемый функцией `recvfrom`, а затем вызовите `gethostbyname` для поиска всех IP-адресов для узла.
7. Измените листинг 11.6, чтобы вызывать функцию `getnameinfo` вместо функции `sock_ntop`. Какие флаги вы должны передать функции `getnameinfo`?
8. В разделе 7.5 мы обсуждали завладение портом с помощью параметра сокета `SO_REUSEADDR`. Чтобы увидеть, как это происходит, создайте не зависящий от протокола сервер времени и даты UDP, показанный в листинге 11.13. Запустите один экземпляр сервера в одном окне, свяжите его с универсальным адресом и некоторым портом, который вы выберете. Запустите в другом окне клиент и убедитесь, что этот сервер выполняет обработку клиента (отметьте вызов функции `printf` на узле сервера). Затем запустите другой экземпляр сервера в другом окне, и на этот раз свяжите его с одним из адресов направленной передачи узла и тем же портом, что и первый сервер. С какой проблемой вы сразу же столкнетесь? Устраните эту проблему и перезапустите второй сервер. Запустите клиент, отправьте дейтаграмму и проверьте, что второй сервер захватил порт первого сервера. Если возможно, запустите второй сервер снова с учетной записью, отличной от учетной записи первого сервера, чтобы проверить, происходит ли по-прежнему захват порта, поскольку некоторые производители не допускают второго связывания, если идентификатор пользователя отличен от идентификатора процесса, уже связанного с портом.
9. В конце раздела 2.12 мы показали два примера Telnet: сервер времени и даты и эхо-сервер. Зная, что клиент проходит через два этапа — функцию `gethostbyname` и функцию `connect`, определите, к каким этапам относятся строки вывода клиента.
10. Функции `getnameinfo` может потребоваться длительное время (до 80 с) на возвращение ошибки, если для IP-адреса не может быть найдено имя узла. Напишите новую функцию `getnameinfo_timeo`, которая получает дополнительный целочисленный аргумент, задающий максимальную длительность ожидания ответа в секундах. Если время таймера истекает и флаг `NI_NAMEREQD` не задан, вызовите функцию `inet_ntop` и возвратите строку адреса.

Часть 3

Дополнительные возможности сокетов

Глава 12

Совместимость IPv4 и IPv6

12.1. Введение

В течение ближайших лет, возможно, произойдет постепенный переход Интернета с IPv4 на IPv6. Во время этого переходного периода важно, чтобы существующие приложения IPv4 продолжали работать с более новыми приложениями IPv6. Например, производитель не может предложить клиент Telnet, работающий только с серверами IPv6, — он должен предоставить и клиент для серверов IPv4, и клиент для серверов IPv6. Мы бы предпочли обойтись одним Telnet-клиентом IPv6, способным работать с серверами и IPv4, и IPv6, и одним сервером Telnet, который работал бы с клиентами и IPv4, и IPv6. В этой главе мы увидим, как это сделать.

В этой главе мы предполагаем, что на узлах работают *двойные стеки протоколов (dual stacks)*, то есть набор протоколов IPv4 и набор протоколов IPv6. На рис. 2.1 представлен узел с двойным стеком. Возможно, узлы и маршрутизаторы будут работать подобным образом в течение многих лет в процессе перехода к IPv6. В какой-то момент многие системы смогут отключить свои стеки IPv4, но только с течением времени можно будет сказать, когда это произойдет, да и произойдет ли вообще.

В этой главе мы обсудим, каким образом приложения IPv4 и IPv6 могут взаимодействовать друг с другом. Существует четыре комбинации клиентов и серверов, использующих либо IPv4, либо IPv6, что показано в табл. 12.1.

Таблица 12.1. Сочетания клиентов и серверов, использующих IPv4 или IPv6

	Сервер IPv4	Сервер IPv6
Клиент IPv4 и существующие серверы клиенты	Почти все	Обсуждается в разделе 12.2
Клиент IPv6	Обсуждается в разделе 12.3	Простые модификации большинства существующих клиентов (например, клиент из листинга 1.1 модифицируется к виду, представленному в листинге 1.2)

Мы не будем подробно рассматривать два сценария, когда клиент и сервер используют один и тот же протокол. Более интересны случаи, когда клиент и сервер используют разные протоколы.

12.2. Клиент IPv4, сервер IPv6

Общим свойством узла с двойным стеком является то, что серверы IPv6 могут выполнять обслуживание клиентов IPv4 и IPv6. Это достигается за счет преобразования адресов IPv4 к виду IPv6 (см. рис. A.6). Пример такого преобразования приведен на рис. 12.1.

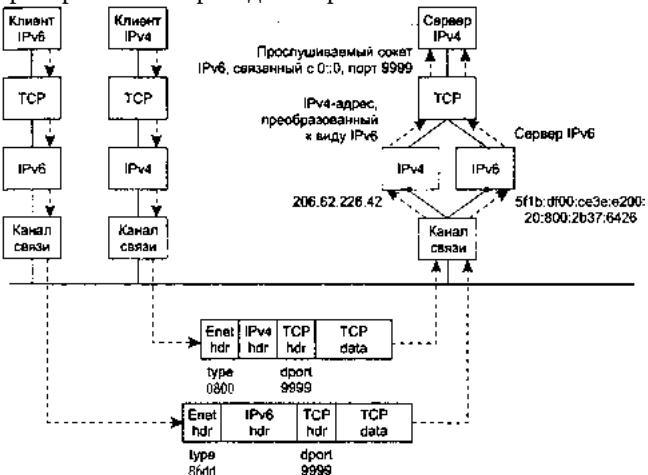


Рис. 12.1. Сервер IPv6 на узле с двойным стеком, обслуживающий клиенты IPv4 и IPv6

Слева у нас находятся клиент IPv4 и клиент IPv6. Сервер (справа) написан с использованием IPv6 и запущен на узле с двойным стеком. Сервер создал прослушиваемый TCP-сокет IPv6, связанный с универсальным адресом IPv6, и порт TCP 9999.

Мы считаем, что клиент и сервер находятся в одной сети Ethernet. Они могут быть соединены и через маршрутизаторы, поскольку все маршрутизаторы поддерживают и IPv4, и IPv6, но в данном случае это ничего не меняет. В разделе Б.3 описывается другой случай, когда клиенты и серверы IPv6 соединяются через маршрутизаторы, поддерживающие только IPv4.

Мы считаем, что оба клиента посылают сегменты SYN для установления соединения с сервером. Узел клиента IPv4 посылает сегмент SYN и дейтаграмму IPv4, а клиент IPv6 посылает сегмент SYN и дейтаграмму IPv6. Сегмент TCP от клиента IPv4 выглядит в сети как заголовок Ethernet, за которым идет заголовок IPv4, заголовок TCP и данные TCP. Заголовок Ethernet содержит поле типа 0x0800, которое идентифицирует кадр как кадр IPv4. Заголовок TCP содержит порт получателя 9999 (в приложении A рассказывается более подробно о форматах и содержании этих заголовков). IP-адрес получателя в заголовке IPv4, который мы не показываем, — это 206.62.226.42.

Сегмент TCP от клиента IPv6 выглядит в сети как заголовок Ethernet, за которым следует заголовок IPv6, заголовок TCP и данные TCP. Заголовок Ethernet содержит поле типа 0x86dd, которое идентифицирует кадр как кадр IPv6. Заголовок TCP имеет тот же формат, что и заголовок TCP в пакете IPv4, и содержит порт получателя 9999. IP-адрес получателя в заголовке IPv6, который мы не показываем, будет таким: 5f1b:df00:ce3e:a200:20:800:2b37:6426.

Принимающий канальный уровень просматривает поле типа Ethernet и передает каждый кадр соответствующему модулю IP. Модуль IPv4 (возможно, вместе с модулем TCP) определяет, что сокетом получателя является сокет IPv6, и IPv4-адрес отправителя в заголовке IPv4 заменяется на эквивалентный ему адрес IPv4, преобразованный к виду IPv6. Этот преобразованный адрес возвращается сокету IPv6 как IPv6-адрес клиента, когда функция accept сервера соединяется с клиентом IPv4. Все оставшиеся дейтаграммы для этого соединения являются дейтаграммами IPv4.

Когда функция сервера accept соединяется с клиентом IPv6, клиентский адрес IPv6 остается таким же, каким был адрес отправителя в заголовке IPv6. Все оставшиеся дейтаграммы для этого соединения являются дейтаграммами IPv6.

Теперь мы можем свести воедино шаги, позволяющие TCP-клиенту IPv4 соединяться с сервером IPv6.

1. Сервер IPv6 запускается, создает прослушиваемый сокет IPv6, и мы считаем, что с помощью функции bind он связывает с сокетом универсальный адрес.

2. Клиент IPv4 вызывает функцию gethostbyname и находит запись типа A для сервера. У узла сервера будут записи и типа A, и типа AAAA, поскольку он поддерживает оба протокола, но клиент IPv4 запрашивает только запись типа A.

3. Клиент вызывает функцию connect, и клиентский узел отправляет серверу сегмент SYN IPv4.

4. Узел сервера получает сегмент SYN IPv4, направленный прослушиваемому сокету IPv6, устанавливает флаг, указывающий, что это соединение использует адреса IPv4, преобразованные к виду IPv6, и отвечает сегментом IPv4 SYN/ACK. Когда соединение установлено, адрес, возвращаемый серверу функцией accept, является адресом IPv4, преобразованным к виду IPv6.

5. Все взаимодействие между клиентом и сервером происходит с использованием дейтаграмм IPv4.

6. Пока сервер не определит при помощи явного запроса, является ли данный IPv6-адрес адресом IPv4, преобразованным к виду IPv6 (с использованием макроопределения IN6_IS_ADDR_V4MAPPED, описанного в разделе 10.4), он не будет знать, что взаимодействует с клиентом IPv4. Двойной стек протоколов решает эту проблему. Аналогично, клиент IPv4 не знает, что он взаимодействует с сервером IPv6.

Главное в данном сценарии то, что узел сервера с двойным стеком имеет и адрес IPv4, и адрес IPv6. Этот сценарий будет работать, пока используются адреса IPv4.

Сценарий работы UDP-сервера IPv6 аналогичен, но формат адреса может меняться для каждой дейтаграммы. Например, если сервер IPv6 получает дейтаграмму от клиента IPv4, адрес, возвращаемый функцией recvfrom, будет адресом IPv4, преобразованным к виду IPv6. Сервер отвечает на запрос клиента, вызывая функцию sendto с адресом IPv4, преобразованным к виду IPv6, в качестве адреса получателя. Формат адреса сообщает ядру, что нужно отправить клиенту дейтаграмму IPv4. Но следующей дейтаграммой, полученной сервером, может быть дейтаграмма IPv6, и функция recvfrom вернет адрес IPv6. Если сервер отвечает, ядро генерирует дейтаграмму IPv6.

На рис. 12.2 показано, как обрабатывается полученная дейтаграмма IPv4 или IPv6 в зависимости от типа принимающего сокета для TCP и UDP. Предполагается, что это узел с двойным стеком.

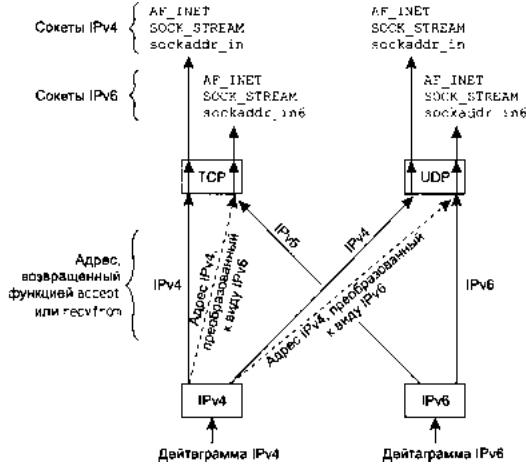


Рис. 12.2. Обработка полученных дейтаграмм IPv4 или IPv6 в зависимости от типа принимающего сокета

- Если дейтаграмма IPv4 приходит на сокет IPv4, ничего особенного не происходит. На рисунке изображены две стрелки, помеченные «IPv4»: одна для TCP, другая для UDP. Между клиентом и сервером происходит обмен дейтаграммами IPv4.
- Если дейтаграмма IPv6 приходит на сокет IPv6, ничего особенного не происходит. На рисунке изображены две стрелки, помеченные «IPv6»: одна для TCP, другая для UDP. Между клиентом и сервером происходит обмен дейтаграммами IPv6.
- Когда дейтаграмма IPv4 приходит на сокет IPv6, ядро возвращает соответствующий адрес IPv4, преобразованный к виду IPv6, в качестве адреса, возвращаемого функцией accept (TCP) или recvfrom (UDP). На рисунке это показано двумя штриховыми стрелками. Такое сопоставление возможно, поскольку адрес IPv4 можно всегда представить как адрес IPv6. Между клиентом и сервером происходит обмен дейтаграммами IPv4.
- Обратное неверно: поскольку, вообще говоря, адрес IPv6 нельзя представить как адрес IPv4, на рисунке отсутствуют стрелки от протокола IPv6 к двум сокетам IPv4.

Большинство узлов с двойным стеком должны использовать следующие правила обращения с прослушиваемыми сокетами:

1. Прослушиваемый сокет IPv4 может принимать соединения только от клиентов IPv4.
2. Если у сервера есть прослушиваемый сокет IPv6, связанный с универсальным адресом, и параметр сокета IPV6_V6ONLY (см. раздел 7.8) не установлен, этот сокет может принимать исходящие соединения как от клиентов IPv4, так и от клиентов IPv6. Для соединения с клиентом IPv4 локальный адрес сервера для соединения будет соответствующим адресом IPv4, преобразованным к виду IPv6.
3. Если у сервера есть прослушиваемый сокет IPv6, связанный с адресом IPv6, не являющимся адресом IPv4, преобразованным к виду IPv6, или его сокет связан с универсальным адресом при установленном параметре сокета IPV6_V6ONLY (раздел 7.8), этот сокет может принимать исходящие соединения только от клиентов IPv6.

12.3. Клиент IPv6, сервер IPv4

Теперь мы поменяем протоколы, используемые клиентом и сервером в примере из предыдущего раздела. Сначала рассмотрим TCP-клиент IPv6, запущенный на узле с двойным стеком протоколов.

1. Сервер IPv4 запускается на узле, поддерживающем только IPv4, и создает прослушиваемый сокет IPv4.
2. Запускается клиент IPv6 и вызывает функцию `gethostbyname`, запрашивая только адреса IPv6 (запрашивает семейство AF_INET6 и устанавливает флаг AI_V4MAPPED в структуре hints). Поскольку у сервера, поддерживающего только IPv4, есть лишь записи типа A, мы видим, согласно табл. 11.3, что клиенту возвращается адрес IPv4, преобразованный к виду IPv6.
3. Клиент IPv6 вызывает функцию `connect` с адресом IPv4, преобразованным к виду IPv6, в структуре адреса сокета IPv6. Ядро обнаруживает преобразованный адрес и автоматически посыпает серверу сегмент SYN IPv4.

4. Сервер отвечает сегментом SYN/ACK IPv4, и устанавливается соединение, по которому происходит обмен дейтаграммами IPv4. Этот сценарий мы схематически изображаем на рис. 12.3.

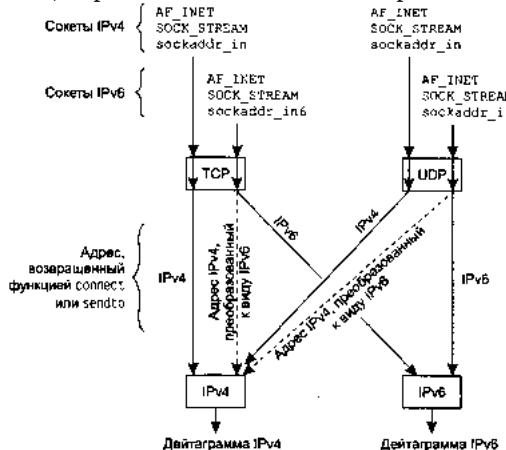


Рис. 12.3. Обработка клиентских запросов в зависимости от типа адреса и типа сокета

■ Если TCP-клиент IPv4 вызывает функцию `connect`, задавая адрес IPv4, или если UDP-клиент IPv4 вызывает функцию `sendto`, задавая адрес IPv4, ничего особенного не происходит. На рисунке это изображено двумя стрелками, помеченными «IPv4».

■ Если TCP-клиент IPv6 вызывает функцию `connect`, задавая адрес IPv6, или если UDP-клиент IPv6 вызывает функцию `sendto`, задавая адрес IPv6, тоже ничего особенного не происходит. На рисунке это показано двумя стрелками, помеченными «IPv6».

■ Если TCP-клиент IPv6 вызывает функцию `connect`, задавая адрес IPv4, преобразованный к виду IPv6, или если UDP-клиент вызывает функцию `sendto`, задавая адрес IPv4, преобразованный к виду IPv6, ядро обнаруживает сопоставленный адрес и инициирует отправку дейтаграммы IPv4 вместо дейтаграммы IPv6. На рисунке это показано двумя штриховыми стрелками.

■ Клиент IPv4 не может задать адрес IPv6 ни функции `connect`, ни функции `sendto`, поскольку 16-байтовый адрес IPv6 не соответствует 4-байтовой структуре `in_addr` в структуре IPv4 `sockaddr_in`. Следовательно, на рисунке нет стрелок от сокетов IPv4 к протоколу IPv6.

В предыдущем разделе (дейтаграмма IPv4, приходящая для сокета сервера IPv6) преобразование полученного адреса IPv4 к виду IPv6 выполняется ядром и результат прозрачно (то есть незаметно для приложения) возвращается приложению функцией `accept` или `recvfrom`. В этом разделе (если необходимо отправить дейтаграмму IPv4 на сокете IPv6) преобразование адреса IPv4 к виду IPv6 выполняется распознавателем в соответствии с правилами, представленными в табл. 11.3, и затем преобразованный адрес прозрачно передается приложению функцией `connect` или `sendto`.

Резюме: совместимость IPv4 и IPv6

Таблица 12.2, содержащая сочетания клиентов и серверов, подводит итог обсуждению, проведенному в данном и предыдущем разделах.

Таблица 12.2. Обобщение совместимости клиентов и серверов IPv4 и IPv6

Сервер IPv4, узел с только IPv4 (только A)	Сервер IPv4, узел с только IPv6 (только AAAA)	Сервер IPv4, узел с двойным стеком (A и AAAA)	Сервер IPv6, узел с двойным стеком (A и AAAA)
Клиент IPv4, узел только IPv4	IPv4	Нет	IPv4
Клиент IPv6, узел только IPv6	Нет	IPv6	Нет
Клиент IPv4, узел с двойным стеком	IPv4	Нет	IPv4
Клиент IPv6,	IPv4	IPv6	Нет*
			IPv6

узел с двойным
стеком

Каждая ячейка этой таблицы содержит поля «IPv4» или «IPv6» с указанием используемого протокола, если данное сочетание работает, либо «нет», если комбинация недопустима. Ячейка в последней строке третьей колонки отмечена звездочкой, поскольку совместимость зависит от адреса, выбранного клиентом. При выборе записи типа AAAA отправка дейтаграммы IPv6 будет невозможна. Но выбор записи типа A, которая возвращается клиенту как адрес IPv4, преобразованный к виду IPv6, приведет к отправке дейтаграммы IPv4. Перебрав все адреса, возвращаемые `getaddrinfo`, мы обязательно доберемся до адреса IPv4, преобразованного к виду IPv6, пусть даже и потратив некоторое время на безуспешное ожидание.

Хотя четверть из представленных в таблице сочетаний недопустима, в обозримом будущем большинство реализаций IPv6 будут использоваться на узлах с двойным стеком протоколов и поддерживать не только IPv6. Если мы удалим из таблицы вторую строку и вторую колонку, все записи «Нет» исчезнут и единственной проблемой останется запись, помеченная звездочкой.

12.4. Макроопределения проверки адреса IPv6

Существует небольшой класс приложений IPv6, которые должны знать, с каким собеседником они взаимодействуют (IPv4 или IPv6). Эти приложения должны знать, является ли адрес собеседника адресом IPv4, преобразованным к виду IPv6. Определены двенадцать макросов, проверяющих некоторые свойства адреса IPv6.

```
#include <netinet/in.h>
```

```
int IN6_IS_ADDR_UNSPECIFIED(const struct in6_addr *aptr);
int IN6_IS_ADDR_LOOPBACK(const struct in6_addr *aptr);
int IN6_IS_ADDR_MULTICAST(const struct in6_addr *aptr);
int IN6_IS_ADDR_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4MAPPED(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4COMPAT(const struct in6_addr *aptr);

int IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_ORGLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_GLOBAL(const struct in6_addr *aptr);
```

Все возвращают: ненулевое значение, если адрес IPv6 имеет указанный тип, 0 в противном случае

Первые семь макросов проверяют базовый тип адреса IPv6. Мы покажем различные типы адресов в разделе A.5. Последние пять макросов проверяют область действия адреса многоадресной передачи IPv6 (см. раздел 19.2).

Клиент IPv6 может вызвать макрос `IN6_IS_ADDR_V4MAPPED` для проверки адреса IPv6, возвращенного распознавателем. Сервер IPv6 может вызвать этот макрос для проверки адреса IPv6, возвращенного функцией `accept` или `recvfrom`.

Как пример приложения, которому нужен этот макрос, можно привести FTP и его команду `PORT`. Если мы запустим FTP-клиент, зарегистрируемся на FTP-сервере и выполним команду `FTP dir`, FTP-клиент пошлет команду `PORT` FTP-серверу через управляющее соединение. Она сообщит серверу IP-адрес и порт клиента, с которым затем сервер создаст соединение. (В главе 27 [111] содержатся подробные сведения о протоколе приложения FTP.) Но FTP-клиент IPv6 должен знать, с каким сервером имеет дело — IPv4 или IPv6, поскольку сервер IPv4 требует команду в формате `PORT a1, a2, a3, a4, p1, p2` (где первые четыре числа, каждое от 0 до 255, формируют 4-байтовый адрес IPv4, а два последних — 2-байтовый номер порта), а серверу IPv6 необходима команда `EPRT` (RFC 2428 [3]), содержащая семейство адреса, адрес в текстовом формате и порт в текстовом формате. В упражнении 12.1 приводятся примеры использования обеих команд.

12.5. Переносимость исходного кода

Большинство существующих сетевых приложений написаны для IPv4. Структуры `sockaddr_in` размещаются в памяти и заполняются, а функция `socket` задает `AF_INET` в качестве первого аргумента. При переходе от листинга 1.1 к листингу 1.2 мы видели, что эти приложения IPv4 можно преобразовать в приложения IPv6 без особых усилий. Многие показанные нами изменения можно выполнить автоматически, используя некоторые сценарии редактирования. Программы, более зависящие от IPv4, использующие такие свойства, как многоадресная передача, параметры IP или символьные (неструктурированные) сокеты, потребуют больших усилий при преобразовании.

Если мы преобразуем приложение для работы с IPv6 и распространим его исходный код, нам придется думать о том, поддерживает ли принимающая система протокол IPv6. Типичный способ решения этой проблемы — применять в коде `#ifdef`, используя по возможности IPv6 (поскольку мы видели в этой главе, что клиент IPv6 может взаимодействовать с серверами IPv4 и наоборот). Проблема такого подхода в том, что код очень быстро засоряется директивами `#ifdef`, и его становится сложнее отслеживать и обслуживать.

Наилучшим подходом будет рассмотрение перехода на IPv6 как возможности сделать программу не зависящей от протокола. Первым шагом здесь будет удаление вызовов функций `gethostbyname` и `gethostbyaddr` и использование функций `getaddrinfo` и `getnameinfo`, описанных в предыдущей главе. Это позволит нам обращаться со структурами адресов сокетов как с непрозрачными объектами, ссылаясь на которые можно с помощью указателя и размера, что как раз и выполняют основные функции сокетов: `bind`, `connect`, `recvfrom` и т.д. Наши функции `sock_XXX` из раздела 3.8 помогут работать с ними независимо от IPv4 и IPv6. Очевидно, эти функции содержат `#ifdef` для работы с IPv4 и IPv6, но если мы скроем эту зависимость от протокола в нескольких библиотечных функциях, наш код станет проще. В разделе 21.7 мы разработаем ряд функций `mcast_XXX`, которые помогут сделать приложения многоадресной передачи не зависящими от версии протокола IP.

Другой момент, который нужно учесть, — что произойдет, если мы откомпилируем наш исходный код в системе, поддерживающей IPv4, и IPv6, затем распространим либо исполняемый код, либо объектные файлы (но не исходный код) и кто-то запустит наше приложение в системе, не поддерживающей IPv6. Есть вероятность, что сервер локальных имен поддерживает записи типа AAAA и возвращает как записи типа AAAA, так и записи типа A некоему собеседнику, с которым пытается соединиться наше приложение. Если наше приложение, работающее с IPv6, вызовет функцию `socket` для создания сокета IPv6, она не будет работать, если узел не поддерживает IPv6. Мы решаем этот вопрос с помощью функций, описанных в следующей главе, игнорируя ошибку функции `socket` и пытаясь использовать следующий адрес в списке, возвращаемом сервером имен. Если предположить, что у собеседника имеется запись типа A и что сервер имен возвращает запись типа A в дополнение к любой записи типа AAAA, то сокет IPv4 успешно создастся. Этот тип функциональности имеется в библиотечной функции, но не в исходном коде каждого приложения.

Чтобы получить возможность передавать дескрипторы сокетов, программам, работающим только с одним из протоколов, в стандарте RFC 2133 [37] предлагается использовать параметр сокета `IPv6_ADDRFORM`, позволяющий получить или изменить семейство сокета. Однако семантика параметра не была описана полностью, да и использоваться он мог только в очень специфических ситуациях, поэтому в следующей версии интерфейса сокетов данный параметр был отменен.

12.6. Резюме

Сервер IPv6 на узле с двойным стеком протоколов может предоставлять сервис как клиентам IPv4, так и клиентам IPv6. Клиент IPv4 посылает серверу дейтаграммы IPv4, но стек протоколов сервера преобразует адрес клиента к виду IPv6, поскольку сервер IPv6 работает со структурами адресов сокетов IPv6.

Аналогично, клиент IPv6 на узле с двойным стеком протоколов может взаимодействовать с сервером IPv4. Распознаватель клиента возвращает адреса IPv4, преобразованные к виду IPv6, для всех записей сервера типа A, и вызов функции `connect` для одного из этих адресов приводит к тому, что двойной стек посыпает сегмент SYN IPv4. Только отдельным специальным клиентам и серверам необходимо знать протокол, используемый собеседником (например, FTP), и чтобы определить, что собеседник использует IPv4, можно использовать макрос `IN6_IS_ADDR_V4MAPPED`.

Упражнения

1. Запустите FTP-клиент IPv6 на узле с двойным стеком протоколов. Соединитесь с FTP-сервером IPv4, запустите команду `debug`, а затем команду `dir`. Далее выполните те же операции, но для сервера IPv6, и сравните команды `PORT`, являющиеся результатом выполнения команд `dir`.

2. Напишите программу, требующую ввода одного аргумента командной строки, который является адресом IPv4 в точечно-десятичной записи. Создайте TCP-сокет IPv4 и свяжите этот адрес и некоторый порт, например 8888, с сокетом при помощи функции `bind`. Вызовите функцию `listen`, а затем `pause`. Напишите аналогичную программу, которая в качестве аргумента командной строки принимает шестнадцатеричную строку IPv6 и создает прослушиваемый TCP-сокет IPv6. Запустите программу IPv4, задав в качестве аргумента универсальный адрес. Затем перейдите в другое окно и запустите программу IPv6, задав в качестве аргумента универсальный адрес IPv6. Можете ли вы запустить программу IPv6, если программа IPv4 уже связана с этим портом? Появляется ли разница при использовании параметра сокета `SO_REUSEADDR`? Что будет, если вы сначала запустите программу IPv6, а затем попытаетесь запустить программу IPv4?

Глава 13

Процессы-демоны и суперсервер inetd

13.1. Введение

Демон (*daemon*) — это процесс, выполняющийся в фоновом режиме и не связанный с управляющим терминалом. Системы Unix обычно имеют множество процессов (от 20 до 50), которые являются демонами, работают в фоновом режиме и выполняют различные административные задачи.

Независимость от терминала обычно является побочным эффектом запуска из системного сценария инициализации (например, в процессе загрузки компьютера). Если же демон запускается командой интерпретатора, он должен самостоятельно отключиться от терминала во избежание нежелательного взаимодействия с системами управления задачами, сессиями терминалов, а также вывода на терминал при работе в фоновом режиме.

Существует несколько способов запустить демон:

1. Во время запуска системы многие демоны запускаются сценариями инициализации системы. Эти сценарии часто находятся в каталоге /etc или в каталоге, имя которого начинается с /etc/rc, но их расположение и содержание зависят от реализации. Такие демоны запускаются с правами привилегированного пользователя.

Некоторые сетевые серверы часто запускаются из сценариев инициализации: суперсервер inetd (следующий пункт, который мы рассмотрим), веб-сервер и почтовый сервер (обычно это программа sendmail). Демон syslogd, обсуждаемый в разделе 13.2, тоже обычно запускается одним из этих сценариев.

2. Многие сетевые серверы запускаются суперсервером inetd, который мы опишем далее в этой главе. Сам inetd запускается в одном из сценариев на этапе 1. Суперсервер inetd прослушивает сетевые порты (Telnet, FTP и т.д.), и когда приходит запрос, активизирует требуемый сервер (сервер Telnet, сервер FTP и т.д.).

3. За периодические процессы в системе отвечает демон cron, и программы, которые он активизирует, выполняются как демоны. Сам демон cron запускается на этапе 1 во время загрузки системы.

4. Если программа должна быть выполнена однократно в определенный момент времени в будущем, применяется команда at. Демон cron обычно инициирует эти программы, когда приходит время их выполнения, поэтому они выполняются как демоны.

5. Демоны можно запускать с пользовательских терминалов, как в основном, так и в фоновом режимах. Это часто осуществляется при тестировании демона или перезапуске демона, завершенного по некоей причине.

Поскольку у демона нет управляющего терминала, ему необходимы средства для вывода сообщений о некоторых событиях — это могут быть обычные информационные сообщения или экстренные сообщения об аварийных ситуациях, которые должен обрабатывать администратор. Использование функции syslog — стандартный способ вывода таких сообщений. Эта функция посылает сообщения демону syslogd.

13.2. Демон syslogd

Системы Unix обычно запускают демон syslogd в одном из сценариев инициализации системы, и он функционирует, пока система работает. Реализации syslogd, происходящие от Беркли, выполняют при запуске следующие действия:

1. Считывается файл конфигурации, обычно /etc/syslog.conf, в котором указано, что делать с каждым типом сообщений, получаемых демоном. Эти сообщения могут добавляться в файл (особой разновидностью такого файла является /dev/console, который записывает сообщение на консоль), передаваться определенному пользователю (если этот пользователь вошел в систему) или передаваться демону syslogd на другом узле.

2. Создается доменный сокет Unix и связывается с полным именем /var/run/log (в некоторых системах /dev/log).

3. Создается сокет UDP и связывается с портом 514 (служба syslog).

4. Открывается файл (устройство) /dev/klog. Любые сообщения об ошибках внутри ядра появляются как входные данные на этом устройстве.

Демон `syslogd` выполняется в бесконечном цикле, в котором вызывается функция `select`, ожидающая, когда один из трех его дескрипторов (из п. 2, 3 и 4) станет готов для чтения. Этот демон считывает сообщение и выполняет то, что предписывает делать с этим сообщением файл конфигурации. Если демон получает сигнал `SIGHUP`, он заново считывает файл конфигурации.

ПРИМЕЧАНИЕ

Более новые реализации отключают возможность создания сокета UDP, если она не задана администратором, поскольку если позволить кому угодно отправлять дейтаграммы UDP на этот порт (возможно, заполняя приемный буфер его сокета), это может привести к тому, что законные сообщения не будут получены (атака типа отказ в обслуживании) или переполнится файловая система из-за неограниченного роста журналов.

Между реализациями демона `syslogd` существуют различия. Например, доменные сокеты Unix используются Беркли-реализациями, а реализации System V используют потоковый драйвер (*streams log driver*). Различные реализации, происходящие от Беркли, используют для доменных сокетов Unix различные полные имена. Мы можем игнорировать все эти тонкости, если используем функцию `syslog`.

Мы можем отправлять сообщения о событиях для записи в журнал (*log messages*) демону `syslogd` из наших демонов, создав дейтаграммный доменный сокет Unix и указывая при отправке полное имя, с которым связан демон, но более простым интерфейсом является функция `syslog`, которую мы описываем в следующем разделе. В качестве альтернативы мы можем создать сокет UDP и отправлять наши сообщения на адрес закольцовки и порт 514.

13.3. Функция `syslog`

Поскольку у демона нет управляющего терминала, он не может просто вызывать функцию `fprintf` для вывода в стандартный поток сообщений об ошибках (`stderr`). Обычная техника записи в журнал сообщений для демона — это вызов функции `syslog`.

```
#include <syslog.h>

void syslog(int priority, const char *message, ...);
```

Хотя эта функция изначально разрабатывалась для BSD, в настоящее время она предоставляется большинством производителей систем Unix. Описание `syslog` в POSIX соответствует тому, что мы пишем здесь. RFC 3164 содержит документацию, касающуюся протокола `syslog` BSD.

Аргумент `priority` — это комбинация аргументов `level` и `facility`, которые мы показываем в табл. 13.1 и 13.2. Дополнительные сведения об этом аргументе можно найти в RFC 3164. Аргумент `message` аналогичен строке формата `printf` с добавлением спецификации `%m`, которая заменяется сообщением об ошибке, соответствующим текущему значению переменной `errno`. Символ перевода строки может появиться в конце строки `message`, но он не является обязательным.

Сообщения для журнала имеют значение `level` (уровень) от 0 до 7, что мы показываем в табл. 13.1. Это упорядоченные значения. Если отправитель не задает значение `level`, используется значение по умолчанию `LOG_NOTICE`.

Таблица 13.1. Аргумент `level` журнальных сообщений

Level	Значение	Описание
<code>LOG_EMERG</code>	0	Система не может функционировать, экстренная ситуация (наивысший приоритет)
<code>LOG_ALERT</code>	1	Следует немедленно принять меры, срочная ситуация
<code>LOG_CRIT</code>	2	Критическая ситуация
<code>LOG_ERR</code>	3	Состояние ошибки
<code>LOG_WARNING</code>	4	Предупреждение
<code>LOG_NOTICE</code>	5	Необычное, хотя и не ошибочное состояние (значение аргумента <code>level</code> по умолчанию)

LOG_INFO	6	Информационное сообщение
LOG_DEBUG	7	Отладочные сообщения (низший приоритет)

Сообщения также содержат аргумент `facility` для идентификации типа процесса, посылающего сообщение. Мы показываем его различные значения в табл. 13.2. Если не задано значение аргумента `facility`, используется его значение по умолчанию — `LOG_USER`.

Таблица 13.2. Аргумент `facility` журнальных сообщений

facility	Описание
LOG_AUTH	Сообщения по безопасности/авторизации
LOG_AUTHPRIV	Сообщения по безопасности/авторизации (частные)
LOG_CRON	Демон cron
LOG_DAEMON	Системные демоны
LOG_FTP	Демон FTP
LOG_KERN	Сообщения ядра
LOG_LOCAL0	Локальное использование
LOG_LOCAL1	Локальное использование
LOG_LOCAL2	Локальное использование
LOG_LOCAL3	Локальное использование
LOG_LOCAL4	Локальное использование
LOG_LOCAL5	Локальное использование
LOG_LOCAL6	Локальное использование
LOG_LOCAL7	Локальное использование
LOG_LPR	Демон принтера
LOG_MAIL	Почтовая система
LOG_NEWS	Система телеконференций
LOG_SYSLOG	Внутренние сообщения системы syslog
LOG_USER	Сообщения пользовательского уровня (значение аргумента <code>facility</code> по умолчанию)
LOG_UUCP	Система UUCP

Например, демон может сделать следующий вызов, когда вызов функции `rename` неожиданно оказывается неудачным:

```
syslog(LOG_INFO|LOG_LOCAL2, "rename(%s, %s): %m", file1, file2);
```

Назначение аргументов `facility` и `level` в том, чтобы все сообщения, которые посылаются процессами определенного типа (то есть с одним значением аргумента `facility`), могли обрабатываться одинаково в файле `/etc/syslog.conf` или чтобы все сообщения одного уровня (с одинаковым значением аргумента `level`) обрабатывались одинаково. Например, файл конфигурации может содержать строки

```
kern.* /dev/console
local7.debug /var/log/cisco.log
```

для указания, что все сообщения ядра направляются на консоль, а сообщения относительно отладки со значением аргумента `facility`, равным `local7`, добавляются в файл `/var/log/cisco.log`.

Когда приложение впервые вызывает функцию `syslog`, она создает дейтаграммный доменный сокет Unix и затем вызывает функцию `connect` для сокета с заранее известным полным именем, которое создано демоном `syslogd` (например, `/var/run/log`). Этот сокет остается открытым, пока процесс не завершится. Другим вариантом является вызов процессом функций `openlog` и `closelog`.

```
#include <syslog.h>
```

```
void openlog(const char *ident, int options, int facility);
void closelog(void);
```

Функция `openlog` может быть вызвана перед первым вызовом функции `syslog`, а функция `closelog` — когда приложение закончит отправлять сообщения в журнал.

Аргумент `ident` — это строка, которая будет добавлена в начало каждого журнального сообщения функцией `syslog`. Часто это имя программы.

Обычно аргумент options формируется путем применения операции логического ИЛИ к константам из табл. 13.3.

Таблица 13.3. Аргумент options (параметр) для функции openlog

Параметр	Описание
LOG_CONS	Выводить журнал на консоль, если невозможно послать сообщение демону syslogd
LOG_NDELAY	Не откладывать создание сокета, открыть его сейчас
LOG_PERROR	Записывать сообщение в stderr, а также посыпать его демону syslogd
LOG_PID	Включать идентификатор процесса (PID) в каждую запись журнала
	Обычно доменный сокет Unix не создается при вызове функции openlog. Вместо этого сокет открывается при первом вызове функции syslog. Параметр LOG_NDELAY указывает, что сокет должен создаваться при вызове функции openlog.
	Аргумент facility функции openlog задает значение facility, используемое по умолчанию для любого последующего вызова функции syslog, при котором не задается аргумент facility. Некоторые демоны вызывают функцию openlog и задают значение аргумента facility (которое обычно не изменяется для данного демона) и затем в каждом вызове функции syslog задают только аргумент level (поскольку level может изменяться в зависимости от ошибки).

Сообщения для записи в журнал могут также генерироваться командой logger. Это может использоваться в сценариях интерпретатора команд, например для отправки сообщений демону syslogd.

13.4. Функция daemon_init

В листинге 13.1^[1] показана функция, называемая daemon_init, которую мы можем вызвать (обычно с сервера), чтобы придать процессу свойства демона.

Листинг 13.1. Функция daemon_init: приданье процессу свойств демона

```
//daemon_init.c
1 #include "unp.h"
2 #include <syslog.h>

3 #define MAXFD 64

4 extern int daemon_proc; /* определен в error.c */

5 int
6 daemon_init(const char *pname, int facility)
7 {
8     int i;
9     pid_t pid;

10    if ((pid = Fork()) < 0)
11        return (-1);
12    else if (pid)
13        _exit(0); /* родитель завершается */

14    /* 1-й дочерний процесс продолжает работу... */

15    if (setsid() < 0) /* становится главным процессом сеанса */
16        return (-1);

17    Signal(SIGHUP, SIG_IGN);
18    if ((pid = Fork()) < 0)
19        return (-1);
20    else if (pid)
21        _exit(0); /* 1-й дочерний процесс завершается */
```

```
22 /* 2-й дочерний процесс продолжает работу */
23 daemon_proc = 1; /* для функций err_XXX() */
24 chdir("/");
25 /* закрытие дескрипторов файлов*/
26 for (i = 0; i < MAXFD; i++)
27   close(i);
28 /* перенаправление stdin, stdout и stderr в /dev/null */
29 open("/dev/null", O_RDONLY);
30 open("/dev/null", O_RDWR);
31 open("/dev/null", O_RDWR);
32 openlog(fname, LOG_PID, facility);
33 return (0); /* успешное завершение */
34 }
```

Вызов функции *fork*

10-13 Сначала мы вызываем функцию *fork*, после чего родительский процесс завершается, а дочерний продолжается. Если процесс был запущен из интерпретатора команд в фоновом режиме, то, когда родительский процесс завершается, оболочка считает, что команда выполнена. Это автоматически запускает дочерний процесс в фоновом режиме. Дочерний процесс наследует идентификатор группы процессов от родительского процесса, но получает свой собственный идентификатор процесса. Это гарантирует, что дочерний процесс не является главным в группе процессов, что требуется для следующего вызова функции *setsid*.

Вызов функции *setsid*

15-16 Функция *setsid* — это функция POSIX, создающая новый сеанс. (В главе 9 [110] подробно рассказывается о взаимоотношениях процессов.) Процесс становится главным в новом сеансе, становится главным в новой группе процессов и не имеет управляющего терминала.

Игнорирование сигнала *SIGHUP* и новый вызов функции *fork*

17-21 Мы игнорируем сигнал *SIGHUP* и снова вызываем функцию *fork*. Когда эта функция завершается, родительский процесс на самом деле является первым дочерним процессом, и он завершается, оставляя выполняться второй дочерний процесс. Назначение второй функции *fork* — гарантировать, что демон не сможет автоматически получить управляющий терминал, если потом он откроет устройство терминала. В SVR4, когда главный процесс сеанса без управляющего терминала открывает устройство терминала (которое в этот момент не является управляющим терминалом для другого сеанса), терминал становится управляющим терминалом главного процесса сеанса. Но вызывая второй раз функцию *fork*, мы гарантируем, что второй дочерний процесс больше не является главным в сеансе, поэтому он не может получить управляющий терминал. Сигнал *SIGHUP* приходится игнорировать, поскольку, когда главный процесс сеанса завершает работу (первый дочерний процесс), всем процессам в сеансе (нашему второму дочернему процессу) посыпается сигнал *SIGHUP*.

Установка флага для функций ошибок

23 Мы присваиваем глобальной переменной `daemon_proc` ненулевое значение. Эта внешняя переменная задается нашими функциями `err_XXX` (см. раздел Г.4), и ее ненулевое значение сообщает этим функциям, что нужно вызвать функцию `syslog` вместо функции `fprintf` (которая выводит сообщение об ошибке в стандартный поток сообщений об ошибках). Это спасает нас от необходимости проходить через весь наш код и вызывать одну из наших функций ошибок, если сервер не работает как демон (то есть когда мы проверяем сервер), а при работе в режиме демона заменять все вызовы на вызовы `syslog`.

Изменение рабочего каталога и сброс всех битов в маске режима создания файла

24 Мы изменяем рабочий каталог на корневой каталог, хотя у некоторых демонов могут быть причины изменить рабочий каталог на какой-либо другой. Например, демон печати может изменить его на каталог, в котором накапливается содержимое заданий для принтера и происходит вся работа по выводу данных на печать. Если демоном сбрасывается дамп (файл `core`), он появляется в текущем рабочем каталоге. Другой причиной для изменения рабочего каталога является то, что демон мог быть запущен в любой файловой системе, и если он там останется, эту систему нельзя будет размонтировать, во всяком случае, без жестких мер.

Закрытие всех открытых дескрипторов

25-27 Мы закрываем все открытые дескрипторы, которые наследуются от процесса, запустившего демон (обычно этим процессом бывает интерпретатор команд). Проблема состоит в определении наибольшего используемого дескриптора: в Unix нет ни одной функции, предоставляющей это значение. Есть способы определения максимального числа дескрипторов, которое может открыть процесс, но даже это достаточно сложно [110, с. 43], поскольку предел может быть бесконечным. Наше решение — закрыть первые 64 дескриптора, даже если большинство из них, возможно, не было открыто.

ПРИМЕЧАНИЕ

Solaris предоставляет функцию `closefrom`, позволяющую демонам решать эту проблему.

Перенаправление `stdin`, `stdout` и `stderr` в `/dev/null`

29-31 Некоторые демоны открывают `/dev/null` для чтения и записи и подключают к нему дескрипторы стандартных потоков ввода, вывода и сообщений об ошибках. Это гарантирует, что наиболее типичные дескрипторы открыты и операция чтения из любого из них возвращает 0 (конец файла), а ядро игнорирует все, что записано в любой из этих трех дескрипторов. Причина, по которой требуется открыть эти дескрипторы, заключается в том, что любая библиотечная функция, вызываемая демоном и считающая, что она может читать из стандартного потока ввода или записывать либо в стандартный поток вывода, либо в стандартный поток сообщений об ошибках, не должна завершиться с ошибкой. Отказ был бы потенциально опасен: если демон открывает сокет для связи с клиентом, дескриптор сокета воспринимается как стандартный поток вывода, поэтому ошибочный вызов какой-нибудь функции типа `perror` может привести к отправке клиенту нежелательных данных.

Использование демона `syslogd` для вывода сообщений об ошибках

32 Вызывается функция `openlog`. Первый ее аргумент берется из вызывающего процесса и обычно является именем программы (например, `argv[0]`). Мы указываем, что идентификатор процесса должен добавляться к каждому сообщению. Аргумент `facility` также задается вызывающим процессом, и его значением может быть константа из табл. 13.2 либо, если приемлемо значение по умолчанию `LOG_USER`, нулевое значение.

Отметим, что поскольку демон выполняется без управляющего терминала, он никогда не должен получать сигнал SIGHUP от ядра. Следовательно, многие демоны используют этот сигнал в качестве уведомления от администратора, что файл конфигурации демона изменился и демон должен еще раз считать файл. Два других сигнала, которые демон никогда не должен получать, — это сигналы SIGINT и SIGWINCH, и они также могут использоваться для уведомления демона о некоторых изменениях.

Пример: сервер времени и даты в качестве демона

В листинге 13.2 представлено изменение нашего сервера времени и даты, не зависящего от протокола. В отличие от сервера, показанного в листинге 11.8, в нем вызывается функция `daemon_init`, чтобы этот сервер мог выполняться в качестве демона.

Листинг 13.2. Не зависящий от протокола сервер времени и даты, работающий в качестве демона

```
//inetd/daytimetcpsrv2.c
1 #include "unp.h"
2 #include <time.h>

3 int
4 main(int argc, char **argv)
5 {
6     int listenfd, connfd;
7     socklen_t addrlen, len;
8     struct sockaddr *cliaddr;
9     char buff[MAXLINE];
10    time_t ticks;

11    daemon_init(argv[0], 0);

12    if (argc == 2)
13        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
14    else if (argc == 3)
15        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
16    else
17        err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");

18    cliaddr = Malloc(addrlen);

19    for (;;) {
20        len = addrlen;
21        connfd = Accept(listenfd, cliaddr, &len);
22        err_msg("connection from %s", Sock_ntop(cliaddr, len));

23        ticks = time(NULL);
24        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
25        Write(connfd, buff, strlen(buff));

26        Close(connfd);
27    }
28 }
```

Изменений всего два: мы вызываем нашу функцию `daemon_init`, как только программа запускается, а затем вызываем нашу функцию `err_msg` вместо `printf`, чтобы вывести IP-адрес и порт клиента. На самом деле, если мы хотим, чтобы наши программы могли выполняться как демоны, мы должны исключить вызов функций `printf` и `fprintf` и вместо них использовать нашу функцию `err_msg`.

Обратите внимание, что мы проверяем `argc` и выводим соответствующее сообщение до вызова `daemon_init`. Таким образом пользователь, запустивший демона, получает немедленное уведомление о

недопустимом количестве аргументов. После вызова `daemon_init` все сообщения направляются в системный журнал.

Если мы запустим эту программу на нашем узле `linux` и затем проверим файл `/var/log/messages` (куда мы отправляем все сообщения `LOG_USER`) после соединения с тем же узлом, мы получим:

```
Jul 10 09:54:37 linux daytimetcpsrv2[24288]: connection from 127.0.0.1.55862
```

Дата, время и имя узла автоматически ставятся в начале сообщения демоном `syslogd`.

13.5. Демон `inetd`

В типичной системе Unix может существовать много серверов, ожидающих запроса клиента. Примерами являются FTP, Telnet, Rlogin, TFTP и т.д. В системах, предшествующих 4.3BSD, каждая из этих служб имела связанный с ней процесс. Этот процесс запускался во время загрузки из файла `/etc/rc`, и каждый процесс выполнял практически идентичные задачи запуска: создание сокета, связывание при помощи функции `bind` заранее известного порта с сокетом, ожидание соединения (TCP) или получения дейтаграммы (UDP) и последующее выполнение функции `fork`. Дочерний процесс выполнял обслуживание клиента, а родительский процесс ждал, когда поступит следующий запрос клиента. Эта модель характеризуется двумя недостатками.

1. Все демоны содержали практически идентичный код запуска, направленный сначала на создание сокета, а затем на превращение процесса в процесс демона (аналогично нашей функции `daemon_init`).

2. Каждый демон занимал некоторое место в таблице процессов, но при этом большую часть времени находился в состоянии ожидания.

Реализация 4.3BSD упростила ситуацию, предоставив *суперсервер* (*superserver*) Интернета — демон `inetd`. Этот демон может применяться серверами, использующими TCP или UDP, и не поддерживает других протоколов, таких как доменные сокеты Unix. Демон `inetd` решает две вышеупомянутые проблемы.

1. Он упрощает написание процессов демонов, поскольку обрабатывает большинство подробностей запуска. Таким образом устраняется необходимость вызова нашей функции `daemon_init` для каждого сервера.

2. Этот демон позволяет одиночному процессу (`inetd`) ждать входящие клиентские запросы ко множеству служб (вместо одного процесса для каждой службы). Это сокращает общее число процессов в системе.

Процесс `inetd` сам становится демоном, используя технологии, которые мы изложили при описании функции `daemon_init`. Затем он считывает и обрабатывает файл конфигурации, обычно файл `/etc/inetd.conf`. Этот файл задает, какие службы должен обрабатывать суперсервер, а также что нужно делать, когда приходит запрос к одной из этих служб. Каждая строка содержит поля, показанные в табл. 13.4. Вот несколько строк в качестве примера:

```
ftp    stream  tcp  nowait  root   /usr/bin/ftpd  ftpd -l
telnet stream  tcp  nowait  root   /usr/bin/telnetd telnetd
login  stream  tcp  nowait  root   /usr/bin/rlogind rlogind -s
tftp   dgram   udp   wait    nobody /usr/bin/tftpd tftpd -s /tftpboot
```

Действительное имя сервера всегда передается в качестве первого аргумента программы, выполняемой с помощью функции `exec`.

Таблица 13.4. Поля файла `inetd.conf`

Поле	Описание
<code>service-name</code>	Должен быть в <code>/etc/services</code>
<code>socket-type</code>	<code>stream</code> (TCP) или <code>dgram</code> (UDP)
<code>Protocol</code>	Должен быть в <code>/etc/protocols</code> ; либо <code>tcp</code> , либо <code>udp</code>
<code>wait-flag</code>	Обычно <code>nowait</code> для TCP и <code>wait</code> для UDP
<code>login-name</code>	Из <code>/etc/password</code> ; обычно <code>root</code>
<code>server-program</code>	Полное имя программы для вызова <code>exec</code>
<code>server-program-arguments</code>	Аргументы программы для вызова <code>exec</code>

ПРИМЕЧАНИЕ

Таблица и приведенные строки — это только пример. Большинство производителей добавили демону inetd свои собственные функции. Примером может служить возможность обрабатывать серверы вызовов удаленных процедур (RPC) в дополнение к серверам TCP и UDP, а также возможность обрабатывать другие протоколы, отличные от TCP и UDP. Полное имя для функции exec и аргументы командной строки сервера, очевидно, зависят от приложения.

Флаг wait-flag может быть достаточно труден для понимания. Он указывает, собирается ли демон, запускаемый inetd, взять на себя работу с прослушиваемым сокетом. Сервисы UDP лишены деления на прослушиваемые и принятые сокеты, и потому практически всегда создаются с флагом wait-flag, равным wait. Сервисы TCP могут вести себя по-разному, но чаще всего для них указывается флаг wait-flag со значением nowait.

Взаимодействие IPv6 с файлом /etc/inetd.conf зависит от производителя. Иногда в качестве поля protocol указывается tcp6 или udp6, чтобы подчеркнуть, что для сервера должен быть создан сокет IPv6. Некоторые разрешают использовать значения protocol, равные tcp46 и udp46, если сервер готов принимать соединения по обоим протоколам. Специальные названия протоколов обычно не включаются в файл /etc/protocols.

Иллюстрация действий, выполняемых демоном inetd, представлена на рис. 13.1.

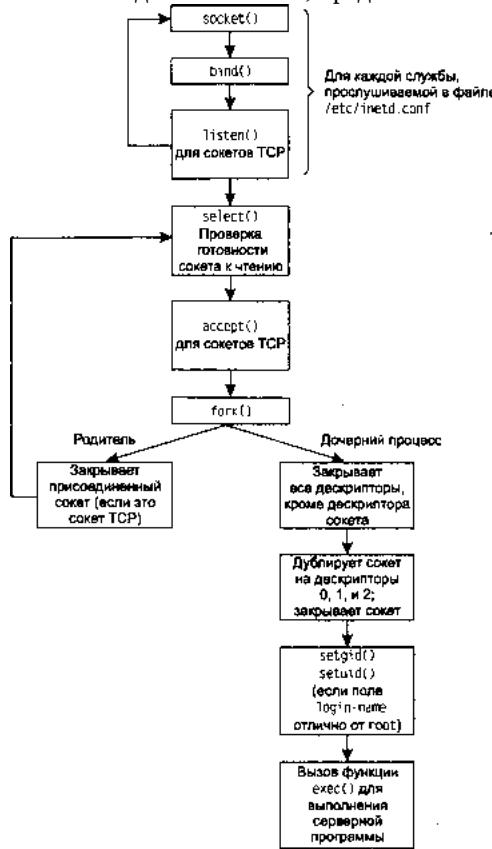


Рис. 13.1. Действия, выполняемые демоном inetd

- При запуске демон читает файл /etc/inetd.conf и создает сокет соответствующего типа (потоковый или дейтаграммный сокет) для всех служб, заданных в файле. Максимальное число серверов, которые может обрабатывать демон inetd, зависит от максимального числа дескрипторов, которые он может создать. Каждый новый сокет добавляется к набору дескрипторов, который будет использован при вызове функции select.

- Для каждого сокета вызывается функция bind, задающая заранее известный порт для сервера и универсальный IP-адрес. Этот номер порта TCP или UDP получается при вызове функции getservbyname с полями service-name и protocol из файла конфигурации в качестве аргументов.

- Для сокетов TCP вызывается функция listen, так что принимаются входящие запросы на соединение. Этот шаг не выполняется для дейтаграммных сокетов.

4. После того как созданы все сокеты, вызывается функция `select`, ожидающая, когда какой-либо из сокетов станет готов для чтения. Вспомните (раздел 6.3), что прослушиваемый сокет TCP становится готов для чтения, когда новое соединение готово быть принятым с помощью функции `accept`, а сокет UDP становится готов для чтения, когда приходит дейтаграмма. Демон `inetd` большую часть времени блокирован в вызове функции `select`, ожидая, когда сокет станет готов для чтения.

5. При указании флага `nowait` для сокетов TCP вызывается функция `accept` сразу же, как только дескриптор сокета становится готов для чтения.

6. Демон `inetd` запускает функцию `fork`, и дочерний процесс обрабатывает запрос клиента. Это аналогично стандартному параллельному серверу (см. раздел 4.8).

Дочерний процесс закрывает все дескрипторы, кроме дескриптора, который он обрабатывает: новый присоединенный сокет, возвращаемый функцией `accept` для сервера TCP, или исходный сокет UDP. Дочерний процесс трижды вызывает функцию `dup2`, подключая сокет к дескрипторам 0, 1 и 2 (стандартные потоки ввода, вывода и сообщений об ошибках). Исходный дескриптор сокета затем закрывается. При этом в дочернем процессе открытыми остаются только дескрипторы 0, 1 и 2. Если дочерний процесс читает из стандартного потока ввода, он читает из сокета, и все, что он записывает в стандартный поток вывода или стандартный поток сообщений об ошибках, записывается в сокет. Дочерний процесс вызывает функцию `getpwname`, чтобы получить значение поля `login-name`, заданного в файле конфигурации. Если это не поле `root`, дочерний процесс становится указанным пользователем при помощи функций `setgid` и `setuid`. (Поскольку процесс `inetd` выполняется с идентификатором пользователя, равным 0, дочерний процесс наследует этот идентификатор пользователя при выполнении функции `fork`, поэтому он имеет возможность стать любым пользователем по своему выбору.)

Теперь дочерний процесс вызывает функцию `exec`, чтобы выполнить соответствующую *программу сервера* (поле `server-program`) для обработки запроса, передавая аргументы, указанные в файле конфигурации.

7. Если сокет является потоковым сокетом, родительский процесс должен закрыть присоединенный сокет (как наш стандартный параллельный сервер). Родительский процесс снова вызывает функцию `select`, ожидая, когда следующий сокет станет готов для чтения.

Чтобы рассмотреть более подробно, что происходит с дескрипторами, на рис. 13.2 показаны дескрипторы демона `inetd` в момент прихода нового запроса на соединение от клиента FTP.

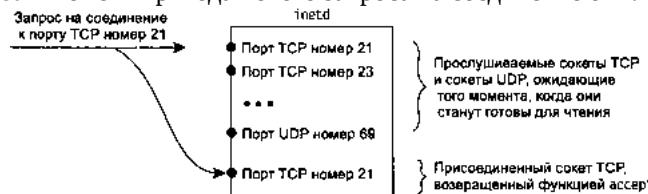


Рис. 13.2. Дескрипторы демона `inetd` в тот момент, когда приходит запрос на порт 21 TCP

Запрос на соединение направляется на порт 21 TCP; новый присоединенный сокет создается функцией `accept`.

На рис. 13.3 показаны дескрипторы в дочернем процессе после вызова функции `fork`, после того как дочерний процесс закрывает все остальные дескрипторы, кроме дескрипторов присоединенного сокета.

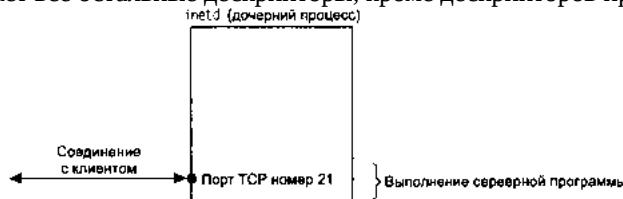


Рис. 13.3. Дескрипторы демона `inetd` в дочернем процессе

Следующий шаг для дочернего процесса — подключение присоединенного сокета к дескрипторам 0, 1 и 2 и последующее закрытие присоединенного сокета. При этом мы получаем дескрипторы, изображенные на рис. 13.4.

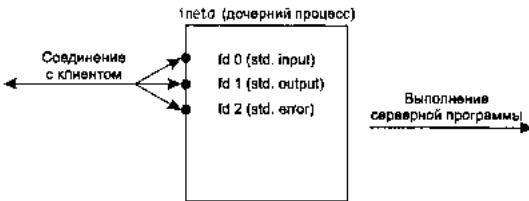


Рис. 13.4. Дескрипторы демона `inetd` после выполнения функции `dup2`

Затем дочерний процесс вызывает функцию `exec`, и, как сказано в разделе 4.7, во время выполнения функции `exec` все дескрипторы обычно остаются открытыми, поэтому реальный сервер, на котором выполняется функция `exec`, использует любой из дескрипторов 0, 1 и 2 для взаимодействия с клиентом. Эти дескрипторы должны быть единственными открытыми на стороне сервера дескрипторами.

Описанный нами сценарий относится к ситуации, при которой файл конфигурации задает в поле `wait-flag` значение `nowait` для сервера. Это типично для всех служб TCP и означает, что демону `inetd` не нужно ждать завершения его дочернего процесса, перед тем как он примет другое соединение для данной службы. Если приходит другой запрос на соединение для той же службы, он возвращается родительскому процессу, как только тот снова вызовет функцию `select`. Шаги 4, 5 и 6, перечисленные выше, выполняются снова, и новый запрос обрабатывается другим дочерним процессом.

Задание флага `wait` для дейтаграммного сервиса изменяет шаги, выполняемые родительским процессом. Флаг указывает на то, что демон `inetd` должен ждать завершения своего дочернего процесса, прежде чем снова вызвать функцию `select` для определения готовности этого сокета UDP для чтения. Происходят следующие изменения:

1. После выполнения функции `fork` в родительском процессе сохраняется идентификатор дочернего процесса. Это дает возможность родительскому процессу узнать, когда завершается определенный дочерний процесс, анализируя значение, возвращаемое функцией `waitpid`.

2. Родительский процесс отключает способность сокета выполнять последующие функции `select`, сбрасывая соответствующий бит в наборе дескрипторов с помощью макроса `FD_CLR`. Это значит, что дочерний процесс завладевает сокетом до своего завершения.

3. Когда завершается дочерний процесс, родительский процесс уведомляется об этом с помощью сигнала `SIGCHLD`, и обработчик сигналов родительского процесса получает идентификатор завершающегося дочернего процесса. Он снова включает функцию `select` для соответствующего сокета, устанавливая бит для этого сокета в своем наборе дескрипторов.

Причина, по которой дейтаграммный сервер должен завладевать сокетом, пока он не завершил работу, лишая тем самым демон `inetd` возможности выполнять функцию `select` на этом сокете для проверки готовности его для чтения (в ожидании другой дейтаграммы клиента), в том, что для сервера дейтаграмм существует только один сокет, в отличие от сервера TCP, у которого имеется прослушиваемый сокет и по одному присоединенному сокету для каждого клиента. Если демон `inetd` не отключил чтение на сокете дейтаграмм и, допустим, родительский процесс (`inetd`) завершил выполнение перед дочерним, дейтаграмма от клиента все еще будет находиться в приемном буфере сокета. Это приводит к тому, что функция `select` снова сообщает, что сокет готов для чтения, и демон `inetd` снова выполняет функцию `fork`, порождая другой (ненужный) дочерний процесс. Демон `inetd` должен игнорировать дейтаграммный сокет до тех пор, пока он не узнает, что дочерний процесс прочитал дейтаграмму из приемного буфера сокета. Демон `inetd` узнает, что дочерний процесс закончил работу с сокетом, путем получения сигнала `SIGCHLD`, указывающего на то, что дочерний процесс завершился. Подобный пример мы показываем в разделе 22.7.

Пять стандартных служб Интернета, описанных в табл. 2.1, обеспечиваются самим демоном `inetd` (см. упражнение 13.2).

Поскольку функцию `accept` для сервера TCP вызывает демон `inetd` (а не сам сервер), реальный сервер, запускаемый демоном `inetd`, обычно вызывает функцию `getpeername` для получения IP-адреса и номера порта клиента. Вспомните рис. 4.9, где мы показывали, что после выполнения вызовов `fork` и `exec` (что выполняет демон `inetd`) у реального сервера есть единственный способ получить идентификацию клиента — вызвать функцию `getpeername`.

Демон `inetd` обычно не используется для серверов, работающих с большими объемами данных, в особенности почтовыми серверами и веб-серверами. Например, функция `sendmail` обычно запускается как стандартный параллельный сервер, как мы отмечали в разделе 4.8. В этом режиме стоимость порождения процесса для каждого клиентского соединения равна стоимости функции `fork`, тогда как в случае сервера TCP, активизированного демоном `inetd`, — стоимости функций `fork` и `exec`. Веб-серверы используют

множество технологий для минимизации накладных расходов при порождении процессов для обслуживания клиентов, как мы покажем в главе 30.

13.6. Функция daemon_inetd

В листинге 13.3 показана функция `daemon_inetd`, которую мы можем вызвать с сервера, запущенного демоном `inetd`.

Листинг 13.3. Функция `daemon_inetd` для придания свойств демона процессу, запущенному демоном `inetd`

```
//daemon_inetd.c
1 #include "unp.h"
2 #include <syslog.h>

3 extern int daemon_proc; /* определено в error.c */

4 void
5 daemon_inetd(const char *pname, int facility)
6 {
7     daemon_proc = 1; /* для наших функций err_XXX() */
8     openlog(pname, LOG_PID, facility);
9 }
```

Эта функция тривиальна по сравнению с `daemon_init`, потому что все шаги выполняются демоном `inetd` при запуске. Все, что мы делаем, — устанавливаем флаг `daemon_proc` для наших функций ошибок (см. табл. Г.1) и вызываем функцию `openlog` с теми же аргументами, что и при вызове функции `daemon_init`, представленной в листинге 13.1.

Пример: сервер времени и даты, активизированный демоном `inetd`

Листинг 13.4 представляет собой модификацию нашего сервера времени и даты, показанного в листинге 13.2, который может быть активизирован демоном `inetd`.

Листинг 13.4. Не зависящий от протокола сервер времени и даты, который может быть активизирован демоном `inetd`

```
//inetd/daytimetcpsrv3.c
1 #include "unp.h"
2 #include <time.h>

3 int
4 main(int argc, char **argv)
5 {
6     socklen_t len;
7     struct sockaddr *cliaddr;
8     char buff[MAXLINE];
9     time_t ticks;

10    daemon_inetd(argv[0], 0);

11    cliaddr = Malloc(MAXSOCKADDR);
12    len = MAXSOCKADDR;
13    Getpeername(0, cliaddr, &len);
14    err_msg("connection from %s", Sock_ntop(cliaddr, len));

15    ticks = time(NULL);
16    snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
17    Write(0, buff, strlen(buff));
```

```
18 Close(0); /* закрываем соединение TCP */
19 exit(0);
20 }
```

В программе сделано два важных изменения. Во-первых, исчез весь код создания сокета: вызовы функций `tcp_listen` и `accept`. Эти шаги выполняются демоном `inetd`, и мы ссылаемся на соединение TCP, используя нулевой дескриптор (стандартный поток ввода). Во-вторых, исчез бесконечный цикл `for`, поскольку сервер активизируется по одному разу для каждого клиентского соединения. После предоставления сервиса клиенту сервер завершает свою работу.

Вызов функции `getpeername`

11-14 Поскольку мы не вызываем функцию `tcp_listen`, мы не знаем размера структуры адреса сокета, которую она возвращает, а поскольку мы не вызываем функцию `accept`, то не знаем и адреса протокола клиента. Следовательно, мы выделяем буфер для структуры адреса сокета, используя нашу константу `MAXSOCKADDR` и вызываем функцию `getpeername` с нулевым дескриптором в качестве первого аргумента.

Чтобы выполнить этот пример в нашей системе Solaris, сначала мы присваиваем службе имя и порт, добавляя следующую строку в `/etc/services`:

```
mydaytime 9999/tcp
```

Затем добавляем строку в `/etc/inetd.conf`:

```
mydaytime stream tcp nowait andy
/home/andy/daytimetcpsrv3 daytimetcpsrv3
```

(Мы разбили длинную строку на более короткие.) Мы помещаем выполняемый код в заданный файл и отправляем демону `inetd` сигнал `SIGHUP`, сообщающий ему, что нужно заново считать файл конфигурации. Следующий шаг — выполнить программу `netstat`, чтобы проверить, что на порте TCP 9999 создан прослушиваемый сокет:

```
solaris % netstat -na | grep 9999
```

```
*.9999 *.* 0 0 49152 0 LISTEN
```

Затем мы запускаем сервер с другого узла:

```
linux % telnet solaris 9999
```

```
Trying 192.168.1.20...
```

```
Connected to solaris.
```

```
Escape character is '^]'.
```

```
Tue Jun 10 11:04:02 2003
```

```
Connection closed by foreign host.
```

Файл `/var/ amd/messages` (в который, как указано в нашем файле `/etc/syslog.conf`, должны направляться наши сообщения с аргументом `facility=LOG_USER`) содержит запись:

```
Jun 10 11:04:02 solaris daytimetcpsrv3[28724]: connection from 192.168.1.10.58145
```

13.7. Резюме

Демоны — это процессы, выполняемые в фоновом режиме независимо от управления с терминалов. Многие сетевые серверы работают как демоны. Все выходные данные демона обычно отправляются демону `syslogd` при помощи вызова функции `syslog`. Администратор полностью контролирует все, что происходит с этими сообщениями, основываясь на том, какой демон отправил данное сообщение и насколько оно серьезно.

Чтобы запустить произвольную программу и выполнять ее в качестве демона, требуется пройти несколько шагов: вызвать функцию `fork` для запуска в фоновом режиме, вызвать функцию `setsid` для того, чтобы создать новый сеанс POSIX и стать главным процессом сеанса, снова вызвать функцию `fork`, чтобы избежать перехода в режим управления с терминала, изменить рабочий каталог и маску режима создания файла и закрыть все ненужные файлы. Наша функция `daemon_init` выполняет все эти шаги.

Многие серверы Unix запускаются демоном `inetd`. Он осуществляет все необходимые шаги по превращению процесса в демон, и при запуске действительного сервера открывается сокет для стандартных потоков ввода, вывода и сообщений об ошибках. Это позволяет нам опустить вызовы функций `socket`, `bind`, `listen` и `accept`, поскольку все эти шаги выполняются демоном `inetd`.

Упражнения

1. Что произойдет в листинге 13.2, если мы отложим вызов функции `daemon_init` до завершения обработки аргументов командной строки и функция `err_quit` будет вызвана до того, как программа станет демоном?

2. Как вы думаете, какие из 10 серверов, перечисленных в табл. 2.1 (учитываются версии TCP и UDP для каждой из пяти служб, управляемых демоном `inetd`), реализуются с помощью вызова функции `fork`, а какие не требуют этой функции?

3. Что произойдет, если мы создадим сокет UDP, свяжем порт 7 с сокетом (стандартный эхо-сервер в табл. 2.1) и отправим дейтаграмму UDP-серверу `chargen`?

4. В руководстве Solaris 2.x для демона `inetd` описывается флаг `-t`, заставляющий демон `inetd` вызывать функцию `syslog` (с аргументами `facility=LOG_DAEMON` и `level=LOG_NOTICE`) для протоколирования клиентского IP-адреса и порта любой службы TCP, которые обрабатывает демон `inetd`. Как демон `inetd` получает эту информацию?

В этом же руководстве сказано, что демон `inetd` не может выполнить это для сокета UDP. Почему?

Есть ли способ обойти эти ограничения для служб UDP?

Глава 14

Дополнительные функции ввода-вывода

14.1. Введение

Эта глава охватывает разнообразные функции и технологии, которые мы помещаем в общую категорию «расширенного ввода-вывода». Сначала мы описываем установку тайм-аута для операции ввода-вывода, которую можно выполнить тремя различными способами. Затем мы рассматриваем три варианта функций `read` и `write`: `recv` и `send`, допускающие четвертый аргумент, содержащий флаги, передаваемые от процесса к ядру; `readv` и `writev`, позволяющие нам задавать массив буферов для ввода или вывода; `recvmsg` и `sendmsg`, объединяющие все свойства других функций ввода-вывода и обладающие новой возможностью получения и отправки вспомогательных данных.

Мы также рассказываем о том, как определить, сколько данных находится в приемном буфере сокета и как использовать с сокетами стандартную библиотеку ввода-вывода C, и обсуждаем более совершенные способы ожидания событий.

14.2. Тайм-ауты сокета

Существует три способа установки тайм-аута для операции ввода-вывода через сокет.

1. Вызов функции `alarm`, которая генерирует сигнал `SIGALRM`, когда истекает заданное время. Это подразумевает обработку сигналов, которая может варьироваться от одной реализации к другой. К тому же такой подход может стать помехой другим существующим вызовам функции `alarm` в данном процессе.

2. Блокирование при ожидании ввода-вывода в функции `select`, имеющей встроенное ограничение времени, вместо блокирования в вызове функции `read` или `write`.

3. Использование более новых параметров сокета — `SO_RCVTIMEO` и `SO_SNDTIMEO`. Проблема при использовании этого подхода заключается в том, что не все реализации поддерживают новые параметры сокетов.

Все три технологии работают с функциями ввода и вывода (такими как `read`, `write` и их вариациями, например `recvfrom` и `sendto`), но нам также хотелось бы иметь технологию, работающую с функцией `connect`, поскольку процесс соединения TCP может занять длительное время (обычно 75 с). Функцию `select` можно использовать для установки тайм-аута функции `connect`, только когда сокет находится в неблокируемом режиме (который мы рассматриваем в разделе 16.3), а параметры сокетов, устанавливающие тайм-аут, не работают с функцией `connect`. Мы также должны отметить, что первые две технологии работают с любым дескриптором, в то время как третья технология только с дескрипторами сокетов.

Теперь мы представим примеры применения всех трех технологий.

Тайм-аут для функции `connect` (сигнал `SIGALRM`)

В листинге 14.1^[1] показана наша функция `connect_timeo`, вызывающая функцию `connect` с ограничением по времени, заданным вызывающим процессом. Первые три аргумента — это аргументы, которых требует функция `connect`, а четвертый — это длительность ожидания в секундах.

Листинг 14.1. Функция `connect` с тайм-аутом

```
//lib/connect_timeo.c
1 #include "unp.h"

2 static void connect_alarm(int);

3 int
4 connect_timeo(int sockfd, const SA *saptr, socklen_t salen, int nsec)
5 {
6     Sigfunc *sigfunc;
7     int n;
```

```

8  sigfunc = Signal(SIGALRM, connect_alarm);
9  if (alarm(nsec) != 0)
10   err_msg("connect_timeo: alarm was already set");

11 if ((n = connect(sockfd, saptr, salen)) < 0) {
12   close(sockfd);
13   if (errno == EINTR)
14     errno = ETIMEDOUT;
15 }
16 alarm(0); /* отключение alarm */
17 Signal(SIGALRM, sigfunc); /* восстанавливаем прежний обработчик
18   сигнала */
19 }

20 static void
21 connect_alarm(int signo)
22 {
23   return; /* просто прерываем connect() */
24 }
```

Установка обработчика сигналов

8 Для SIGALRM устанавливается обработчик сигнала. Текущий обработчик сигнала (если таковой имеется) сохраняется, и таким образом мы можем восстановить его в конце функции.

Установка таймера

9-10 Таймер для процесса устанавливается на время (число секунд), заданное вызывающим процессом. Возвращаемое значение функции `alarm` — это число секунд, остающихся в таймере для процесса (если он уже установлен для процесса) в настоящий момент или 0 (если таймер не был установлен прежде). В первом случае мы выводим сообщение с предупреждением, поскольку мы стираем предыдущую установку таймера (см. упражнение 14.2).

Вызов функции `connect`

11-15 Вызывается функция `connect`, и если функция прерывается (`EINTR`), мы присваиваем переменной `errno` значение `ETIMEDOUT`. Сокет закрывается, чтобы не допустить продолжения трехэтапного рукопожатия.

Выключение таймера и восстановление предыдущего обработчика сигнала

16-18 Таймер при обнулении выключается, и восстанавливается предыдущий обработчик сигналов (если таковой имеется).

Обработка сигнала SIGALRM

20-24 Обработчик сигнала просто возвращает управление. Предполагается, что это прервет ожидание функции `connect`, заставив ее возвратить ошибку `EINTR`. Вспомните нашу функцию `signal` (см. листинг 5.5), которая не устанавливает флага `SA_RESTART`, когда перехватываемый сигнал — это сигнал SIGALRM.

Одним из важных моментов в этом примере является то, что мы всегда можем сократить период ожидания для функции `connect`, используя эту технологию, но мы не можем увеличить период, заданный для ядра. В Беркли-ядре тайм-аут для функции `connect` обычно равен 75 с. Мы можем задать меньшее значение для нашей функции, допустим 10, но если мы задаем большее значение, скажем 80, тайм-аут самой функции `connect` все равно составит 75 с.

Другой важный момент в данном примере — то, что мы используем возможность прерывания системного вызова (`connect`) для того, чтобы возвратить управление, прежде чем истечет время ожидания ядра. Такой подход допустим, когда мы выполняем системный вызов и можем обработать возвращение ошибки `EINTR`. Но в разделе 29.7 мы встретимся с библиотечной функцией, выполняющей системный вызов, которая сама выполняет заново системный вызов при возвращении ошибки `EINTR`. Мы можем продолжать работать с сигналом `SIGALRM` и в этом случае, но в листинге 29.6 мы увидим, что нам придется воспользоваться функциями `sigsetjmp` и `siglongjmp`, поскольку библиотечная функция игнорирует ошибку `EINTR`.

Тайм-аут для функции `recvfrom` (сигнал `SIGALRM`)

В листинге 14.2 показана новая версия функции `dg_cli`, приведенной в листинге 8.4, в которую добавлен вызов функции `alarm` для прерывания функции `recvfrom` при отсутствии ответа в течение 5 с.

Листинг 14.2. Функция `dg_cli`, в которой при установке тайм-аута для функции `recvfrom` используется функция `alarm`

```
//advio/dgclitimeo3.c
1 #include "unp.h"

2 static void signalrm(int);

3 void
4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
5 {
6     int n;
7     char sendline[MAXLINE], recvline[MAXLINE + 1];

8     Signal(SIGALRM, signalrm);

9     while (Fgets(sendline, MAXLINE, fp) != NULL) {

10        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

11        alarm(5);
12        if ((n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL)) < 0) {
13            if (errno == EINTR)
14                fprintf(stderr, "socket timeout\n");
15            else
16                err_sys("recvfrom error");
17        } else {
18            alarm(0);
19            recvline[n] = 0; /* завершающий нуль */
20            Fputs(recvline, stdout);
21        }
22    }
23 }

24 static void
25 sig_alrm(int signo)
26 {
27     return; /* просто прерываем recvfrom() */
```

```
28 }
```

Обработка тайм-аута из функции recvfrom

8-22 Мы устанавливаем обработчик для сигнала SIGALRM и затем вызываем функцию `alarm` для 5-секундного тайм-аута при каждом вызове функции `recvfrom`. Если функция `recvfrom` прерывается нашим обработчиком сигнала, мы выводим сообщение об ошибке и продолжаем работу. Если получена строка от сервера, мы отключаем функцию `alarm` и выводим ответ.

Обработчик сигнала SIGALRM

24-28 Наш обработчик сигналов возвращает управление, прерывая блокированную функцию `recvfrom`.

Этот пример работает корректно, потому что каждый раз, когда мы устанавливаем функцию `alarm`, мы читаем только один ответ. В разделе 20.4 мы попытаемся использовать ту же технологию, но поскольку мы будем считывать множество ответов для данной функции `alarm`, возникнет ситуация гонок, которую нам придется разрешить.

Тайм-аут для функции recvfrom (функция select)

Мы демонстрируем вторую технологию для установки тайм-аута (использование функции `select`) в листинге 14.3. Здесь показана наша функция `readable_timeo`, которая ждет, когда дескриптор станет готов для чтения, но не более заданного числа секунд.

Листинг 14.3. Функция `readable_timeo`: ожидание, когда дескриптор станет готов для чтения

```
//lib/readable_timeo.c
1 #include "unp.h"

2 int
3 readable_timeo(int fd, int sec)
4 {
5     fd_set rset;
6     struct timeval tv;

7     FD_ZERO(&rset);
8     FD_SET(fd, &rset);

9     tv.tv_sec = sec;
10    tv.tv_usec = 0;

11    return (select(fd + 1, &rset, NULL, NULL, &tv));
12    /* > если дескриптор готов для чтения */
13 }
```

Подготовка аргументов для функции select

7-10 В наборе дескрипторов для чтения включается бит, соответствующий данному дескриптору. В структуре `timeval` устанавливается время (число секунд), в течение которого вызывающий процесс готов ждать.

Блокирование в функции select

11-12 Функция `select` ждет, когда дескриптор станет готов для чтения или истечет заданное время ожидания. Возвращаемое значение этой функции — это возвращаемое значение функции `select`: -1 в случае ошибки, 0, если истекло время ожидания, и положительное значение, задающее число готовых дескрипторов, если таковые появились.

Эта функция не выполняет операции чтения — она просто ждет, когда дескриптор будет готов к чтению. Следовательно, эту функцию можно использовать с любым типом сокета — TCP или UDP.

Создание аналогичной функции, называемой `writable_timeo`, тривиально. Эта функция ждет, когда дескриптор будет готов для записи.

Мы используем эту функцию в листинге 14.4, где показана еще одна версия нашей функции `dg_cli`, приведенной в листинге 8.4. Эта новая версия вызывает функцию `recvfrom`, только когда наша функция `readable_timeo` возвращает положительное значение.

Мы не вызываем функцию `recvfrom`, пока функция `readable_timeo` не сообщит нам, что дескриптор готов для чтения. Тем самым мы гарантируем, что функция `recvfrom` не заблокируется.

Листинг 14.4. Функция `dg_cli`, вызывающая функцию `readable_timeo` для установки тайм-аута

```
//advio/dgclitimeo1.c
1 #include "unp.h"

2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int n;
6     char sendline[MAXLINE], recvline[MAXLINE + 1];

7     while (Fgets(sendline, MAXLINE, fp) != NULL) {

8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

9         if (Readable_timeo(sockfd, 5) == 0) {
10             fprintf(stderr, "socket timeout\n");
11         } else {
12             n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
13             recvline[n] = 0; /* завершающий нуль */
14             Fputs(recvline, stdout);
15         }
16     }
17 }
```

Тайм-аут для функции `recvfrom` (параметр сокета `SO_RCVTIMEO`)

В нашем последнем примере демонстрируется применение параметра сокета `SO_RCVTIMEO`. Мы устанавливаем этот параметр один раз для дескриптора, задавая значение тайм-аута, и этот тайм-аут затем применяется ко всем операциям чтения этого дескриптора. Одна из замечательных особенностей этого метода состоит в том, что мы устанавливаем данный параметр только один раз, тогда как предыдущие два метода требовали выполнения некоторых действий перед каждой операцией, для которой мы хотели задать временной предел. Но этот параметр сокета применяется только к операциям чтения. Аналогичный параметр `SO_SNDFTIMEO` применяется только к операциям записи, и ни один параметр сокета не может использоваться для установки тайм-аута для функции `connect`.

Листинг 14.5. Функция `dg_cli`, использующая параметр сокета `SO_RCVTIMEO` для установки тайм-аута

```
//advio/dgclitimeo2.c
1 #include "unp.h"

2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
```

```

5 int n;
6 char sendline[MAXLINE], recvline[MAXLINE + 1];
7 struct timeval tv;

8 tv.tv_sec = 5;
9 tv.tv_usec = 0;
10 Setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));

11 while (Fgets(sendline, MAXLINE, fp) != NULL) {

12     Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

13     n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
14     if (n < 0) {
15         if (errno == EWOULDBLOCK) {
16             fprintf(stderr, "socket timeout\n");
17             continue;
18         } else
19             err_sys("recvfrom error");
20     }
21     recvline[n] = 0; /* завершающий нуль */
22     Fputs(recvline, stdout);
23 }
24 }
```

Установка параметра сокета

8-10 Четвертый аргумент функции `setsockopt` — это указатель на структуру `timeval`, в которую записывается желательное значение тайм-аута.

Проверка тайм-аута

15-17 Если тайм-аут операции ввода-вывода истекает, функция (в данном случае `recvfrom`) возвращает ошибку `EWOULDBLOCK`.

14.3. Функции `recv` и `send`

Эти две функции аналогичны стандартным функциям `read` и `write`, но для них требуется дополнительный аргумент.

```
#include <sys/socket.h>
```

```
ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);
ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags);
```

Обе функции возвращают: количество прочитанных или записанных байтов в случае успешного выполнения, -1 в случае ошибки

Первые три аргумента функций `recv` и `send` совпадают с тремя первыми аргументами функций `read` и `write`. Аргумент `flags` либо имеет нулевое значение, либо формируется в результате применения операции логического ИЛИ к константам, представленным в табл. 14.1.

Таблица 14.1. Аргумент `flags` для функций ввода-вывода

flags	Описание	recv	send
<code>MSG_DONTROUTE</code>	Неискать в таблице маршрутизации	•	
<code>MSG_DONTWAIT</code>	Только эта операция является неблокируемой	•	•

MSG_OOB	Отправка или получение внеполосных данных •
MSG_PEEK	Просмотр приходящих сообщений •
MSG_WAITALL	Ожидание всех данных •

■ **MSG_DONTROUTE.** Этот флаг сообщает ядру, что получатель находится в нашей сети, и поэтому не нужно выполнять поиск в таблице маршрутизации. Дополнительную информацию об этом свойстве мы приводим при описании параметра сокета `SO_DONTROUTE` (см. раздел 7.5). Это свойство можно включить для одной операции вывода с флагом `MSG_DONTROUTE` или для всех операций вывода данного сокета, используя указанный параметр сокета.

■ **MSG_DONTWAIT.** Этот флаг указывает, что отдельная операция ввода-вывода является неблокируемой. Таким образом, отпадает необходимость включать флаг отсутствия блокировки для сокета, выполнять операцию ввода-вывода и затем выключать флаг отсутствия блокировки. Неблокируемый ввод-вывод мы опишем в главе 15 вместе с включением и выключением флага отсутствия блокировки для всех операций ввода-вывода через сокет.

ПРИМЕЧАНИЕ

Этот флаг введен в Net/3 и может не поддерживаться в некоторых системах.

■ **MSG_OOB.** С функцией `send` этот флаг указывает, что отправляются внеполосные данные. В случае TCP в качестве внеполосных данных должен быть отправлен только 1 байт, как показано в главе 21. С функцией `recv` этот флаг указывает на то, что вместо обычных данных должны читаться внеполосные данные.

■ **MSG_PEEK.** Этот флаг позволяет нам просмотреть пришедшие данные, готовые для чтения, при этом после выполнения функции `recv` или `recvfrom` данные не сбрасываются (при повторном вызове этих функций снова возвращаются уже просмотренные данные). Подробнее мы поговорим об этом в разделе 14.7.

■ **MSG_WAITALL.** Этот флаг был впервые введен в 4.3BSD Reno. Он сообщает ядру, что операция чтения должна выполняться до тех пор, пока не будет прочитано запрашиваемое количество байтов. Если система поддерживает этот флаг, мы можем опустить функцию `readn` (см. листинг 3.9) и заменить ее макроопределением

```
#define readn(fd, ptr, n) recv(fd, ptr, n, MSG_WAITALL)
```

Даже если мы задаем флаг `MSG_WAITALL`, функция может возвратить количество байтов меньше запрашиваемого в том случае, если или перехватывается сигнал, или соединение завершается, или есть ошибка сокета, требующая обработки.

Существуют дополнительные флаги, используемые протоколами, отличными от TCP/IP. Например, транспортный уровень OSI основан на записях (а не на потоке байтов, как TCP), и для операций вывода поддерживает флаг `MSG_EOR`, задающий конец логической записи.

С аргументом `flags` связана одна фундаментальная проблема: он передается по значению и не является аргументом типа «значение-результат». Следовательно, он может использоваться только для передачи флагов от процесса к ядру. Ядро не может передать флаги обратно процессу. Это не представляет проблемы с TCP/IP, поскольку очень редко бывает необходимо передавать флаги обратно процессу от ядра. Но когда к 4.3BSD Reno были добавлены протоколы OSI, появилась необходимость возвращать процессу флаг `MSG_EOR` при операции ввода. В 4.3BSD Reno было принято решение оставить аргументы для общепотребительных функций (`recv` и `recvfrom`) как есть и изменить структуру `msghdr`, которая используется с функциями `recvmmsg` и `sendmsg`. В разделе 14.5 мы увидим, что в эту структуру был добавлен целочисленный элемент `msg_flags`, и поскольку структура передается по ссылке, ядро может изменить флаги, содержащиеся в этом элементе, по завершении функции. Это значит также, что если процессу необходимо, чтобы флаги изменились ядром, процесс должен вызвать функцию `recvmmsg` вместо вызова функций `recv` или `recvfrom`.

14.4. Функции `readv` и `writev`

Эти две функции аналогичны функциям `read` и `write`, но `readv` и `writev` позволяют использовать для чтения или записи один или более буферов с помощью одного вызова функции. Эти операции называются

операциями *распределяющего чтения* (*scatter read*) (поскольку вводимые данные распределяются по нескольким буферам приложения) и *объединяющей записи* (*gather write*) (поскольку данные из нескольких буферов объединяется для одной операции вывода).

```
#include <sys/uio.h>

ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);
ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);
```

Обе функции возвращают: количество считанных или записанных байтов, -1 в случае ошибки

Второй аргумент обеих функций — это указатель на массив структур iovec, для определения которого требуется включить заголовочный файл <sys/uio.h>:

```
struct iovec {
    void *iov_base; /* начальный адрес буфера */
    size_t iov_len; /* размер буфера */
};
```

ПРИМЕЧАНИЕ

Типы данных элементов структуры iovec определяются POSIX. Вам могут встретиться реализации, определяющие iov_base как char*, а iov_len как int.

Существует некоторый предел числа элементов в массиве структур iovec, зависящий от реализации. Linux позволяет использовать до 1024 элементов, а HP-UD — до 2100. POSIX требует, чтобы константа IOV_MAX определялась включением заголовочного файла <sys/uio.h> и чтобы ее значение было не менее 16.

Функции readv и writev могут использоваться с любым дескриптором, а не только с сокетами. Кроме того, writev является атомарной операцией. Для протокола, основанного на записях, такого как UDP, один вызов функции writev генерирует однудейтаграмму UDP.

Мы отметили одно использование функции writev с параметром сокета TCP_NODELAY в разделе 7.9. Мы сказали, что при записи с помощью функции write 4 байт и затем 396 байт может активизироваться алгоритм Нагла, и предпочтительное решение в данном случае заключается в вызове функции writev для двух буферов.

14.5. Функции recvmsg и sendmsg

Эти две функции являются наиболее общими для всех операций ввода-вывода. Действительно, мы можем заменить все вызовы функций ввода read, readv, recv и recvfrom вызовами функции recvmsg. Аналогично, все вызовы различных функций вывода можно заменить вызовами функции sendmsg.

```
#include <sys/socket.h>
```

```
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
ssize_t sendmsg(int sockfd, struct msghdr *msg, int flags);
```

Обе функции возвращают: количество прочитанных или записанных байтов в случае успешного выполнения, -1 в случае ошибки

Большинство аргументов обеих функций скрыто в структуре msghdr:

```
struct msghdr {
    void *msg_name;      /* адрес протокола */
    socklen_t msg_namelen; /* размер адреса протокола */
    struct iovec *msg_iov; /* массив буферов */
    size_t msg_iovlen;   /* количество элементов в массиве msg_iov */
    void *msg_control;  /* вспомогательные данные: должны быть
                           выровнены для структуры cmsghdr */
    socklen_t msg_controllen; /* размер вспомогательных данных */
    int msg_flags;        /* флаги, возвращенные функцией recvmsg() */
};
```

ПРИМЕЧАНИЕ

Показанная нами структура `msghdr` восходит к 4.3BSD Reno и определяется POSIX. Некоторые системы (например, Solaris 2.5) используют более раннюю структуру `msghdr`, которая появилась в 4.2BSD. У более ранней структуры нет элемента `msg_flags`, а элементы `msg_control` и `msg_controllen` называются `msg_accrights` и `msg_accrightslen`. В этой системе поддерживается только одна форма вспомогательных данных — передача дескрипторов файлов (так называемые права доступа). При появлении протоколов OSI в 4.3BSD Reno были добавлены новые формы вспомогательных данных, вследствие чего были обобщены имена элементов структуры.

Элементы `msg_name` и `msg_namelen` используются, когда сокет не является присоединенным (например, неприсоединенный сокет UDP). Они аналогичны пятому и шестому аргументам функций `recvfrom` и `sendto`: `msg_name` указывает на структуру адреса сокета, в которой вызывающий процесс хранит адрес протокола получателя для функции `sendmsg` или функция `recvmsg` хранит адрес протокола отправителя. Если нет необходимости задавать адрес протокола (например, сокет TCP или присоединенный сокет UDP), элемент `msg_name` должен быть пустым указателем. Элемент `msg_namelen` является аргументом типа «значение» для функции `sendmsg`, но для функции `recvmsg` это аргумент типа «значение-результат».

Элементы `msg iov` и `msg iovlen` задают массив буферов ввода и вывода (массив структур `iovec`), аналогичный второму и третьему аргументам функций `readv` и `writev`.

Элементы `msg_control` и `msg_controllen` задают расположение и размер необязательных вспомогательных данных. Элемент `msg_controllen` — это аргумент типа «значение-результат» функции `recvmsg`. Вспомогательные данные мы рассматриваем в разделе 14.6.

Работая с функциями `recvmsg` и `sendmsg`, следует учитывать различие между двумя флаговыми переменными: это аргумент `flags`, который передается по значению, и элемент `msg_flags` структуры `msghdr`, который передается по ссылке (поскольку функции передается адрес этой структуры).

■ Элемент `msg_flags` используется только функцией `recvmsg`. Когда вызывается функция `recvmsg`, аргумент `flags` копируется в элемент `msg_flags` [128, с. 502], и это значение используется ядром для управления приемом данных. Затем это значение обновляется в зависимости от результата функции `recvmsg`.

■ Элемент `msg_flags` игнорируется функцией `sendmsg`, поскольку эта функция использует аргумент `flags` для управления выводом данных. Это значит, что если мы хотим установить флаг `MSG_DONTWAIT` при вызове функции `sendmsg`, то мы должны присвоить это значение аргументу `flags`, а присваивание значения `MSG_DONTWAIT` элементу `msg_flags` не имеет никакого эффекта.

В табл. 14.2 показано, какие флаги проверяются ядром для функций ввода и вывода и какие элементы `msg_flags` может возвращать функция `recvmsg`. Для элемента `sendmsg.msg_flags` нет колонки, потому что, как мы отмечали, он не используется.

Таблица 14.2. Флаги для различных функций ввода-вывода

Флаг	Проверяются функциями <code>send flags</code> <code>sendto flags</code> <code>sendmsg flags</code>	Проверяются функциями <code>recv flags</code> <code>recvfrom flags</code> <code>recvmsg flags</code>	Возвращаются функцией <code>recvmsg msg_flags</code>
<code>MSG_DONTRROUTE</code>	•		
<code>MSG_DONTWAIT</code>	•	•	
<code>MSG_PEEK</code>		•	
<code>MSG_WAITALL</code>		•	
<code>MSG_EOR</code>	•		
<code>MSG_OOB</code>	•	•	•
<code>MSG_BCAST</code>			•
<code>MSG_MCAST</code>			•
<code>MSG_TRUNC</code>			•
<code>MSG_CTRUNC</code>			•

Первые четыре флага только проверяются и никогда не возвращаются, вторые два проверяются и возвращаются, а последние четыре флага только возвращаются. Следующие ниже комментарии относятся

к шести флагам, возвращаемым функцией `recvmsg`.

- `MSG_BCAST`. Этот флаг введен в BSD/OS и возвращается, если дейтаграмма была получена как широковещательная дейтаграмма канального уровня или если ее IP-адрес получателя является широковещательным адресом. Этот флаг предоставляет более удачную возможность определить, что дейтаграмма UDP была отправлена на широковещательный адрес, чем параметр сокета `IP_RECVSTAADDR`.

- `MSG_MCAST`. Этот флаг введен в BSD/OS и возвращается, если дейтаграмма была получена как дейтаграмма многоадресной передачи канального уровня.

- `MSG_TRUNC`. Этот флаг возвращается, если дейтаграмма была усечена: у ядра имеется больше данных для возвращения, чем позволяет пространство в памяти, выделенное для них процессом (сумма всех элементов `iov_len`). Более подробно мы рассмотрим это в разделе 22.3.

- `MSG_CTRUNC`. Этот флаг возвращается, если были усечены вспомогательные данные: у ядра имеется больше вспомогательных данных для возвращения, чем позволяет выделенное для них процессом пространство в памяти (`msg_controllen`).

- `MSG_EOR`. Этот флаг означает конец записи. Он сбрасывается, если возвращаемые данные не заканчиваются записью. Если же возвращаемые данные заканчиваются логической записью, этот флаг устанавливается. TCP не использует этот флаг, поскольку это потоковый протокол.

- `MSG_OOB`. Этот флаг никогда не возвращается для внеполосных данных TCP. Он возвращается другими наборами протоколов (например, протоколами OSI).

Реализации могут возвращать некоторые из входных аргументов `flags` в элементе `msg_flags`, поэтому мы должны проверять только те значения флагов, которые нас интересуют (например, последние шесть в табл. 14.2).

На рис. 14.1 представлена структура `msghdr` и информация, на которую она указывает. На этом рисунке отражена ситуация, предшествующая вызову функции `recvmsg` для сокета UDP.

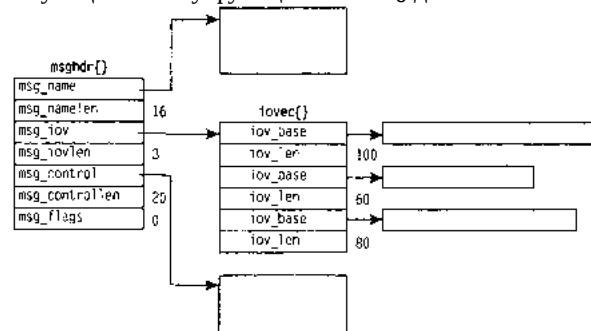


Рис. 14.1. Структуры данных в тот момент, когда функция `recvmsg` вызывается для сокета UDP

Для адреса протокола в памяти выделяется 16 байт, а для вспомогательных данных — 20 байт. Инициализируется массив из трех структур `iovec`: первая задает 100-байтовый буфер, вторая — 60-байтовый буфер, третья — 80-байтовый буфер. Мы также предполагаем, что был установлен параметр сокета `IP_RECVSTAADDR` для получения IP-адреса получателя из дейтаграммы UDP.

Затем будем считать, что с адреса 198.6.38.100, порт 2000, приходит 170-байтовая дейтаграмма UDP, предназначенная для нашего сокета UDP с IP-адресом получателя 206.168.112.96. На рис. 14.2 показана вся информация, содержащаяся в структуре `msghdr` в момент завершения функции `recvmsg`.

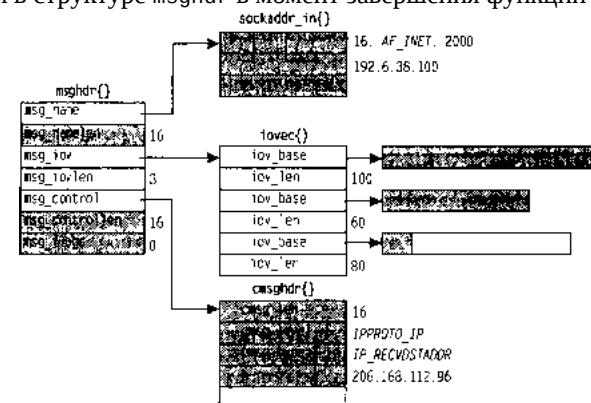


Рис. 14.2. Изменение рис. 14.1 при завершении функции

Затемненными показаны поля, изменяемые функцией `recvmsg`. По сравнению с рис. 14.1 на рис. 14.2 изменяется следующее:

- В буфер, на который указывает элемент `msg_name`, записывается структура адреса сокета Интернета, содержащая IP-адрес и UDP-порт отправителя, определенные по полученной дейтаграмме.
- Обновляется аргумент `msg_namelen`, имеющий тип «значение-результат». Его новым значением становится количество данных, хранящихся в `msg_name`. Но на самом деле его значение как перед вызовом функции `recvmsg`, так и при ее завершении равно 16.
- Первые 100 байт данных записываются в первый буфер, следующие 60 байт — во второй буфер и последние 10 байт — в третий буфер. Последние 70 байт третьего буфера не изменяются. Возвращаемое значение функции `recvmsg` — это размер дейтаграммы (170).
- Буфер, на который указывает `msg_control`, заполняется как структура `cmsg_hdr`. (Более подробно о вспомогательных данных мы поговорим в разделе 14.6, а об этом параметре сокета — в разделе 22.2.) Значение `cmsg_len` равно 16, `cmsg_level` — `IPPROTO_IP`, `cmsg_type` — `IP_RECVSTADDR`, а следующие 4 байта 20-байтового буфера содержат IP-адрес получателя из полученной дейтаграммы UDP. Последние 4 байта 20-байтового буфера, которые мы предоставили для хранения вспомогательных данных, не изменяются.
- Обновляется элемент `msg_controllen` — его новым значением становится фактический размер записанных вспомогательных данных. Этот аргумент также является аргументом типа «значение-результат», и его результат по завершении функции равен 16.
- Элемент `msg_flags` изменяется функцией `recvmsg`, но процессу никакие флаги не возвращаются.

В табл. 14.3 показаны различия между рассмотренными пятью группами функций ввода-вывода.

Таблица 14.3. Сравнение пяти групп функций ввода-вывода

Функция	Произвольный дескриптор	Только дескриптор сокета	Один буфер для чтения и записи	Распределяющее чтение, объединяющая запись	Наличие флагов	Указание адреса собеседника	Управляющая информация
read,
write
readv,
writev
recv, send
recvfrom,
sendto
recvmsg,
sendsg

14.6. Вспомогательные данные

Вспомогательные данные (ancillary data) можно отправлять и получать, используя элементы `msg_control` и `msg_controllen` структуры `msghdr` с функциями `sendmsg` и `recvmsg`. Другой термин, используемый для обозначения вспомогательных данных, — *управляющая информация (control information)*. В этом разделе мы рассматриваем данное понятие и показываем структуру и макросы, используемые для создания и обработки вспомогательных данных. Примеры программ мы откладываем до следующих глав, в которых рассказывается о применении вспомогательных данных.

В табл. 14.4 приводится обобщение различных вариантов применения вспомогательных данных, рассматриваемых в этой книге.

Таблица 14.4. Использование вспомогательных данных

Протокол	<code>cmsg_level</code>	<code>cmsg_type</code>	Описание
IPv4	IPPROTO_IP	IP_RECVSTADDR	Получает адрес получателя с дейтаграммой UDP
		IP_RECVIF	Получает индекс интерфейса с дейтаграммой UDP

	IPV6_DSTOPTS	Задает/получает параметры получателя
	IPV6_HOPLIMIT	Задает/получает предел количества транзитных узлов
IPv6	IPPROTO_IPV6	Задает/получает параметры для транзитных узлов
	IPV6_HOPOPTS	Задает следующий транзитный адрес
	IPV6_NEXTHOP	Задает/получает информацию о пакете
	IPV6_PKTINFO	Задает/получает информацию о пакете
	IPV6_RTHDR	Посыпает/получает дескрипторы
Домен Unix	SOL_SOCKET	Посыпает/получает данные, идентифицирующие пользователя
	SCM_CREDS	

Набор протоколов OSI также использует вспомогательные данные для различных целей, которые мы не рассматриваем в этой книге.

Вспомогательные данные состоят из одного или более *объектов вспомогательных данных* (*ancillary data objects*), каждый из которых начинается со структуры cmsghdr, определяемой подключением заголовочного файла <sys/socket.h>:

```
struct cmsghdr {
    socklen_t cmsg_len; /* длина структуры в байтах */
    int cmsg_level; /* исходящий протокол */
    int cmsg_type; /* тип данных, специфичный для протокола */
    /* далее следует массив символов без знака cmsg_data[] */
};
```

Мы уже видели эту структуру на рис. 14.2, когда она использовалась с параметром сокета IP_RECVSOCKNAME для возвращения IP-адреса получателя полученной дейтаграммы UDP. Вспомогательные данные, на которые указывает элемент msg_control, должны быть соответствующим образом выровнены для структуры cmsghdr. Один из способов выравнивания мы показываем в листинге 15.7.

На рис. 14.3 приводится пример двух объектов вспомогательных данных, содержащихся в буфере управляющей информации.

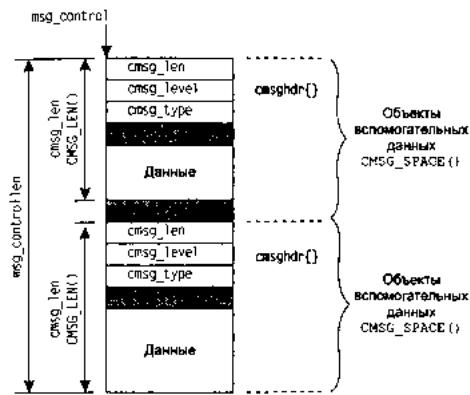


Рис. 14.3. Два объекта вспомогательных данных

Элемент msg_control указывает на первый объект вспомогательных данных, а общая длина вспомогательных данных задается элементом msg_controllen. Каждому объекту предшествует структура cmsghdr, которая описывает объект. Между элементом cmsg_type и фактическими данными может существовать заполнение, а также заполнение может быть в конце данных, перед следующим объектом вспомогательных данных. Пять макросов CMSG_xxx, которые мы описываем далее, учитывают это возможное заполнение.

ПРИМЕЧАНИЕ

Не все реализации поддерживают наличие нескольких объектов вспомогательных данных в буфере управляющей информации.

На рис. 14.4 приводится формат структуры cmsghdr при ее использовании с доменным сокетом Unix для передачи дескрипторов (см. раздел 15.7) или передачи данных, идентифицирующих пользователя (см. раздел 15.8).

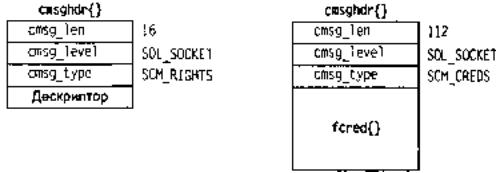


Рис. 14.4. Структура cmsghdr при использовании с доменными сокетами Unix

Предполагается, что каждый из трех элементов структуры cmsghdr занимает 4 байта и между структурой cmsghdr и данными нет заполнения. При передаче дескрипторов содержимое массива cmsg_data — это фактические значения дескрипторов. На этом рисунке мы показываем только один передаваемый дескриптор, но в общем может передаваться и более одного дескриптора (тогда значение элемента cmsg_len будет равно 12 плюс число дескрипторов, умноженное на 4, если считать, что каждый дескриптор занимает 4 байта).

Вспомогательные данные, возвращаемые функцией recvmsg, могут содержать любое число объектов вспомогательных данных. Чтобы скрыть возможное заполнение от приложения, для упрощения обработки вспомогательных данных определены следующие пять макросов (что требует включения заголовочного файла <sys/socket.h>).

```
#include <sys/socket.h>
#include <sys/param.h> /* для макрона ALIGN во многих реализациях */

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mhdrptr);
Возвращает: указатель на первую структуру cmsghdr или NULL, если нет вспомогательных данных

struct cmsghdr *CMSG_NXTHDR(struct msghdr *mhdrptr, struct cmsghdr *cmsgptr);
Возвращает: указатель на структуру cmsghdr или NULL, если нет больше объектов вспомогательных данных

unsigned char *CMSG_DATA(struct cmsghdr *cmsgptr);
Возвращает: указатель на первый байт данных, связанных со структурой cmsghdr

unsigned int CMSG_LEN(unsigned int length);
Возвращает: значение, которое записывается в cmsg_len

unsigned int CMSG_SPACE(unsigned int length);
Возвращает: общий размер объекта вспомогательных данных
```

ПРИМЕЧАНИЕ

В POSIX определены первые пять макросов, а в [113] определены последние два.

Эти макросы могли бы быть использованы в следующем псевдокоде:

```
struct msghdr msg;
struct cmsghdr *cmsgptr;

/* заполнение структуры msg */

/* вызов recvmsg() */

for (cmsgptr = CMSG_FIRSTHDR(&msg); cmsgptr != NULL;
     cmsgptr = CMSG_NXTHDR(&msg, cmsgptr)) {
    if (cmsgptr->cmsg_level == ... &&
        cmsgptr->cmsg_type == ...) {
```

```

u_char *ptr;

ptr = CMSG_DATA(cmsgptr);
/* обработка данных, на которые указывает ptr */
}
}

```

Макрос `CMSG_FIRSTHDR` возвращает указатель на первый объект вспомогательных данных или пустой указатель, если в структуре `msghdr` нет вспомогательных данных (или `msg_control` является пустым указателем, или `cmsg_len` меньше размера структуры `cmsghdr`). Макрос `CMSG_NXTHDR` возвращает пустой указатель, когда в буфере управления нет другого объекта вспомогательных данных.

ПРИМЕЧАНИЕ

Многие существующие реализации макрона `CMSG_FIRSTHDR` никогда не используют элемент `msg_controllen` и просто возвращают значение `cmsg_control`. В листинге 22.2 мы проверяем значение `msg_controllen` перед вызовом макроопределения.

Разница между макросами `CMSG_LEN` и `CMSG_SPACE` заключается в том, что первый возвращает длину объекта вместе с дополняющими нулями (это значение хранится в `cmsg_len`), а последний возвращает длину собственно объекта (это значение может использоваться для динамического выделения памяти под объект).

14.7. Сколько данных находится в очереди?

Иногда требуется узнать, сколько данных находится в очереди для чтения данного сокета, не считывая эти данные. Для этого имеется три способа.

1. Если нашей целью не является блокирование в ядре (поскольку мы можем выполнять другие задачи, пока данные для чтения еще не готовы), может использоваться неблокируемый ввод-вывод. Мы обсуждаем его в главе 16.

2. Если мы хотим проверить данные, но при этом оставить их в приемном буфере для считывания какой-либо другой частью процесса, мы можем использовать флаг `MSG_PEEK` (см. табл. 14.1). Если мы не уверены, что какие-либо данные готовы для чтения, мы можем объединить этот флаг с отключением блокировки для сокета или с флагом `MSG_DONTWAIT`.

Помните о том, что для потокового сокета количество данных в приемном буфере может изменяться между двумя последовательными вызовами функции `recv`. Например, предположим, что мы вызываем `recv` для сокета TCP, задавая буфер длиной 1024 и флаг `MSG_PEEK`, и возвращаемое значение равно 100. Если затем мы снова вызовем функцию `recv`, возможно, возвратится более 100 байт (мы задаем длину буфера больше 100), поскольку в промежутке между двумя нашими вызовами `recv` могли быть получены дополнительные данные.

А что произойдет в случае сокета UDP, когда в приемном буфере имеется дейтаграмма? При вызове `recvfrom` с флагом `MSG_PEEK`, за которым последует другой вызов без задания `MSG_PEEK`, возвращаемые значения обоих вызовов (размер дейтаграммы, ее содержимое и адрес отправителя) будут совпадать, даже если в приемный буфер сокета между двумя вызовами добавляются дополнительные дейтаграммы. (Мы считаем, конечно, что никакой другой процесс не использует тот же дескриптор и не осуществляет чтение из данного сокета в это же время.)

3. Некоторые реализации поддерживают команду `FIONREAD` функции `ioctl`. Третий аргумент функции `ioctl` — это указатель на целое число, а возвращаемое в этом целом числе значение — это текущее число байтов в приемном буфере сокета [128, с. 553]. Это значение является общим числом установленных в очередь байтов, которое для сокета UDP включает все дейтаграммы, установленные в очередь. Также помните о том, что значение, возвращаемое для сокета UDP, в Беркли-реализациях включает пространство, требуемое для структуры адреса сокета, содержащей IP-адрес отправителя и порт для каждой дейтаграммы (16 байт для IP4, 24 байта для IP6).

14.8. Сокеты и стандартный ввод-вывод

Во всех наших примерах мы применяли то, что иногда называется *вводом-выводом Unix*, вызывали функции `read` и `write` и их разновидности (`recv`, `send` и т.д.). Эти функции работают с дескрипторами и обычно реализуются как системные вызовы внутри ядра Unix.

Другой метод выполнения ввода-вывода заключается в использовании *стандартной библиотеки ввода-вывода*. Она задается стандартом ANSI C и была задумана как библиотека, совместимая с не-Unix системами, поддерживающими ANSI C. Стандартная библиотека ввода-вывода обрабатывает некоторые моменты, о которых мы должны заботиться сами при использовании функций ввода-вывода Unix, таких как автоматическая буферизация потоков ввода и вывода. К сожалению, ее обработка буферизации потока может представить новый ряд проблем, о которых следует помнить. Глава 5 [110] подробно описывает стандартную библиотеку ввода-вывода, а в [92] представлена полная реализация стандартной библиотеки ввода-вывода и ее обсуждение.

ПРИМЕЧАНИЕ

При обсуждении стандартной библиотеки ввода-вывода используется термин «поток» в выражениях типа «мы открываем поток ввода» или «мы очищаем поток вывода». Не путайте это с подсистемой потоков STREAMS, которую мы обсуждаем в главе 31.

Стандартная библиотека ввода-вывода может использоваться с сокетами, но есть несколько моментов, которые необходимо при этом учитывать.

- Стандартный поток ввода-вывода может быть создан из любого дескриптора при помощи вызова функции `fdopen`. Аналогично, имея стандартный поток ввода-вывода, мы можем получить соответствующий дескриптор, вызывая функцию `fileno`. С функцией `fileno` мы впервые встретились в листинге 6.1, когда мы хотели вызвать функцию `select` для стандартного потока ввода-вывода. Функция `select` работает только с дескрипторами, поэтому нам необходимо было получить дескриптор для стандартного потока ввода-вывода.

- Сокеты TCP и UDP являются двусторонними. Стандартные потоки ввода-вывода также могут быть двусторонними: мы просто открываем поток типа `r+`, что означает чтение-запись. Но в таком потоке за функцией вывода не может следовать функция ввода, если между ними нет вызова функции `fflush`, `fseek`, `fsetpots` или `rewind`. Аналогично, за функцией вывода не может следовать функция ввода, если между ними нет вызова функции `fseek`, `fsetpots`, `rewind`, в том случае, когда при вводе не получен признак конца файла. Проблема с последними тремя функциями состоит в том, что все они вызывают функцию `lseek`, которая не работает с сокетами.

- Простейший способ обработки подобной проблемы чтения-записи — это открытие двух стандартных потоков ввода-вывода для данного сокета: одного для чтения и другого для записи.

Пример: функция str_echo, использующая стандартный ввод-вывод

Сейчас мы модифицируем наш эхо-сервер TCP (см. листинг 5.2) для использования стандартного ввода-вывода вместо функций `readline` и `writen`. В листинге 14.6 представлена версия нашей функции `str_echo`, использующая стандартный ввод-вывод. (С этой версией связана проблема, которую мы вскоре опишем.)

Листинг 14.6. Функция `str_echo`, переписанная с использованием стандартного ввода-вывода

```
//advio/str_echo_stdio02.c
1 #include "unp.h"

2 void
3 str_echo(int sockfd)
4 {
5     char line[MAXLINE];
6     FILE *fpin, *fpout;

7     fpin = Fdopen(sockfd, "r");
8     fpout = Fdopen(sockfd, "w");
```

```
9 while (Fgets(line, MAXLINE, fpin) != NULL)
10 Fputs(line, fpout);
11 }
```

Преобразование дескриптора в поток ввода и поток вывода

7-10 Функцией `fdopen` создаются два стандартных потока ввода-вывода: один для ввода и другой для вывода. Вызовы функций `readline` и `written` заменены вызовами функций `fgets` и `fputs`.

Если мы запустим наш сервер с этой версией функции `str_echo` и затем запустим наш клиент, мы увидим следующее:

```
hpx % tcpcli02 206.168.112.96
hello, world мы набираем эту строку, но не получаем отражения
and hi      и на эту строку нет ответа
hello??     и на эту строку нет ответа
^D          наш символ конца файла
hello, world затем выводятся три отраженные строки
and hi
hello??
```

Здесь возникает проблема буферизации, поскольку сервер ничего не отражает, пока мы не введем наш символ конца файла. Выполняются следующие шаги:

- Мы набираем первую строку ввода, и она отправляется серверу.
- Сервер читает строку с помощью функции `fgets` и отражает ее с помощью функции `fputs`.
- Но стандартный поток ввода-вывода сервера *полностью буферизован* стандартной библиотекой ввода-вывода. Это значит, что библиотека копирует отраженную строку в свой стандартный буфер ввода-вывода для этого потока, но не выдает содержимое буфера в дескриптор, поскольку буфер не заполнен.
- Мы набираем вторую строку ввода, и она отправляется серверу.
- Сервер читает строку с помощью функции `fgets` и отражает ее с помощью функции `fputs`.
- Снова стандартная библиотека ввода-вывода сервера только копирует строку в свой буфер, но не выдает содержимое буфера в дескриптор, поскольку он не заполнен.
- По тому же сценарию вводится третья строка.
- Мы набираем наш символ конца файла, и функция `str_cli` (см. листинг 6.2) вызывает функцию `shutdown`, посылая серверу сегмент FIN.
- TCP сервера получает сегмент FIN, который читает функция `fgets`, в результате чего функция `fgets` возвращает пустой указатель.
- Функция `str_echo` возвращает серверу функцию `main` (см. листинг 5.9), и дочерний процесс завершается при вызове функции `exit`.
- Библиотечная функция `exit` языка C вызывает стандартную функцию очистки ввода-вывода [110, с. 162-164], и буфер вывода, который был частично заполнен нашими вызовами функции `fputs`, теперь выводит скопившиеся в нем данные.
- Дочерний процесс сервера завершается, в результате чего закрывается его присоединенный сокет, клиенту отсылается сегмент FIN и заканчивается последовательность завершения соединения TCP.
- Наша функция `str_cli` получает и выводит три отраженных строки.
- Затем функция `str_cli` получает символ конца файла на своем сокете, и клиент завершает свою работу.

Проблема здесь заключается в том, что буферизация на стороне сервера выполняется автоматически стандартной библиотекой ввода-вывода. Существует три типа буферизации, выполняемой стандартной библиотекой ввода-вывода.

1. *Полная буферизация (fully buffered)* означает, что ввод-вывод имеет место, только когда буфер заполнен, процесс явно вызывает функцию `fflush` или процесс завершается посредством вызова функции `exit`. Обычный размер стандартного буфера ввода-вывода — 8192 байта.

2. *Буферизация по строкам (line buffered)* означает, что ввод-вывод имеет место, только когда встречается символ перевода строки, процесс вызывает функцию `fflush` или процесс завершается вызовом функции `exit`.

3. *Отсутствие буферизации (unbuffered)* означает, что ввод-вывод имеет место каждый раз, когда вызывается функция стандартного ввода-вывода.

Большинство реализаций Unix стандартной библиотеки ввода-вывода используют следующие правила:

- Стандартный поток ошибок никогда не буферизуется.
- Стандартные потоки ввода и вывода буферизованы полностью, если они не подключены к терминальному устройству, в противном случае они буферизуются по строкам.
- Все остальные потоки тоже буферизованы полностью, если они не подключены к терминалу, в случае чего они буферизованы по строкам.

Поскольку сокет не является терминальным устройством, проблема, отмеченная с нашей функцией `str_echo` в листинге 14.6, заключается в том, что поток вывода (`frot`) полностью буферизован. Есть два решения: мы можем сделать поток вывода буферизованным по строкам при помощи вызова функции `setvbuf` либо заставить каждую отраженную строку выводиться при помощи вызова функции `fflush` после каждого вызова функции `fputs`. Применение любого из этих изменений скорректирует поведение нашей функции `str_echo`. На практике оба варианта чреваты ошибками и могут плохо взаимодействовать с алгоритмом Нагла. В большинстве случаев оптимальным решением будет отказаться от использования стандартной библиотеки ввода-вывода для сокетов и работать с буферами, а не со строками (см. раздел 3.9). Использование стандартных функций ввода-вывода имеет смысл в тех случаях, когда потенциальный выигрыш перевешивает затруднения.

ПРИМЕЧАНИЕ

Будьте осторожны — некоторые реализации стандартной библиотеки ввода-вывода все еще вызывают проблемы при работе с дескрипторами, большими 255. Эта проблема может возникнуть с сетевыми серверами, обрабатывающими множество дескрипторов. Проверьте определение структуры FILE в вашем заголовочном файле `<stdio.h>`, чтобы увидеть, к какому типу переменных относится дескриптор.

14.9. Расширенный опрос

В начале этой главы мы рассказывали о способах установки таймеров для операций с сокетами. Во многих операционных системах для этого существуют функции `poll` и `select`, которые были описаны в главе 6. Ни один из этих методов еще не стандартизован POSIX, поэтому между реализациями существуют определенные различия. Код, использующий подобные механизмы, должен считаться непереносимым. Мы рассмотрим два механизма, прочие весьма похожи на них.

Интерфейс `/dev/poll`

В Solaris имеется специальный файл `/dev/poll`, с помощью которого можно опрашивать большое количество дескрипторов файлов. Проблема `select` и `poll` состоит в том, что список дескрипторов приходится передавать при каждом вызове. Устройство опроса поддерживает информацию о состоянии между вызовами, так что программа может подготовить список подлежащих опросу дескрипторов, а потом спокойно зациклиться в опросе и не заполнять список каждый раз.

После открытия `/dev/poll` программа должна инициализировать массив структур `pollfd` (тех же, которые используются функцией `poll`, но в этом случае поле `revents` не используется). Затем массив передается ядру вызовом `write` (структура записывается непосредственно в `/dev/poll`). После этого программа может вызывать `ioctl DP_POLL` и ждать событий. При вызове `ioctl` передается следующая структура:

```
struct dvpoll {  
    struct pollfd* dp_fds;  
    int           dp_nfds;  
    int           dp_timeout;  
};
```

Поле `dp_fds` указывает на буфер, используемый для хранения массива структур `pollfd`, возвращаемых вызовом `ioctl`. Поле `dp_nfds` задает размер буфера. Вызов `ioctl` блокируется до появления интересующих программу событий на любом из опрашиваемых дескрипторов, или до прохождения `dp_timeout`.

миллисекунд. При нулевом значении тайм-аута функция ioctl возвращается немедленно (то есть данный способ может использоваться для реализации неблокируемых сокетов). Тайм-аут, равный -1, означает неопределенное долгое ожидание.

Измененный код функции str_cli, переписанной из листинга 6.2 с использованием /dev/poll, приведен в листинге 14.7.

Листинг 14.7. Функция str_cli, использующая /dev/poll

```
//advio/str_cli_poll03.c
1 #include "unp.h"
2 #include <sys/devpoll.h>

3 void
4 str_cli(FILE *fp, int sockfd)
5 {
6     int stdineof;
7     char buf[MAXLINE];
8     int n;
9     int wfd;
10    struct pollfd pollfd[2];
11    struct dvpoll dopoll;
12    int i;
13    int result;

14    wfd = Open("/dev/poll", O_RDWR, 0);

15    pollfd[0].fd = fileno(fp);
16    pollfd[0].events = POLLIN;
17    pollfd[0].revents = 0;

18    pollfd[1].fd = sockfd;
19    pollfd[1].events = POLLIN;
20    pollfd[1].revents = 0;

21    Write(wfd, pollfd, sizeof(struct pollfd) * 2);

22    stdineof = 0;
23    for (;;) {
24        /* блокирование до готовности сокета */
25        dopoll.dp_timeout = -1;
26        dopoll.dp_nfds = 2;
27        dopoll.dp_fds = pollfd;
28        result = Iioctl(wfd, DP_POLL, &dopoll);

29        /* цикл по готовым дескрипторам */
30        for (i = 0; i < result; i++) {
31            if (dopoll.dp_fds[i].fd == sockfd) {
32                /* сокет готов к чтению */
33                if ((n = Read(sockfd, buf, MAXLINE)) == 0) {
34                    if (stdineof == 1)
35                        return; /* нормальное завершение */
36                    else
37                        err_quit("str_cli: server terminated prematurely");
38                }
39                Write(fileno(stdout), buf, n);
40            } else {
41                /* дескриптор готов к чтению */
```

```

42     if ((n = Read(fileno(fp), buf, MAXLINE)) == 0) {
43         stdineof = 1;
44         Shutdown(sockfd, SHUT_WR); /* отправка FIN */
45         continue;
46     }

47     Writen(sockfd, buf, n);
48 }
49 }
50 }
51 }

```

Составление списка дескрипторов для /dev/poll

14-21 Заполнив массив структур pollfd, мы передаем его в /dev/poll. В нашем примере используются только два файловых дескриптора, так что мы помещаем их в статический массив. На практике программы, использующие /dev/poll, обычно следят за сотнями или даже тысячами дескрипторов одновременно, поэтому массив выделяется динамически.

Ожидание данных

24-28 Программа не вызывает select, а блокируется в вызове ioctl в ожидании поступления данных. Возвращаемое значение представляет собой количество готовых к чтению дескрипторов файлов.

Цикл по дескрипторам

30-49 Наша программа относительно проста, потому что мы знаем, что дескрипторов всего два. В большой программе цикл будет более сложным. Возможно даже разделение программы на потоки для обработки данных, полученных по разным дескрипторам.

Интерфейс kqueue

Система FreeBSD версии 4.1 предложила сетевым программистам новый интерфейс, получивший название kqueue. Этот интерфейс позволяет процессу зарегистрировать фильтр событий, описывающий интересующие данный процесс события kqueue. К событиям этого типа относятся операции ввода-вывода с файлами и тайм-ауты, а также асинхронный ввод-вывод, уведомление об изменении файлов и обработка сигналов.

```

#include <sys/types.h>
#include <sys/event.h>
#include <sys/time.h>

int kqueue(void);
int kevent(int kq, const struct kevent *changelist, int nchanges,
           struct kevent *eventlist, int nevents, const struct timespec *timeout);
void EV_SET(struct kevent *kev, uintptr_t ident, short filter,
           u_short flags, u_int fflags, intptr_t data, void *udata);

```

Функция kqueue возвращает новый дескриптор kqueue, который может использоваться в последующих вызовах kevent. Функция kevent применяется для регистрации интересующих событий, а также для получения уведомлений об этих событиях. Параметры changelist и nchanges описывают изменения в предыдущем варианте списка событий. Если nchanges отлично от нуля, выполняются все запрошенные в структуре changelist изменения. Функция kevent возвращает количество событий или нуль, если произошел выход по тайм-ауту. В аргументе timeout хранится значение тайм-аута, обрабатываемое

подобно тому, как при вызове `select` (`NULL` для блокирования, ненулевое значение для задания конкретного таймаута, а нулевое значение трактуется как необходимость неблокирующего вызова). Обратите внимание, что параметр `timeout` имеет тип `struct timespec`, отличающийся от `struct timeval` в вызове `select` тем, что первый имеет наносекундное разрешение, а второй — микросекундное.

Структура `kevent` определяется в заголовочном файле `<sys/event.h>`:

```
struct kevent {
    uintptr_t ident; /* идентификатор (например, дескриптор файла) */
    short filter; /* тип фильтра (например, EVFILT_READ) */
    u_short flags; /* флаги действий (например, EV_ADD); */
    u_int fflags; /* флаги, относящиеся к конкретным фильтрам */
    intptr_t data; /* данные, относящиеся к конкретным фильтрам */
    void *uidata; /* непрозрачные пользовательские данные */
};
```

Действия по смене фильтра и флаговые возвращаемые значения приведены в табл. 14.5.

Таблица 14.5. Флаги для операций `kevent`

Значение <code>flags</code>	Описание	Изменяется	Возвращается
<code>EV_ADD</code>	Добавить новое событие, подразумевается по умолчанию, если не указан флаг <code>EV_DISABLE</code>	•	
<code>EV_CLEAR</code>	Сброс состояния события после считывания его пользователем	•	
<code>EV_DELETE</code>	Удаление события из фильтра	•	
<code>EV_DISABLE</code>	Отключение события без удаления его из фильтра	•	
<code>EV_ENABLE</code>	Включение отключенного перед этим события	•	
<code>EV_ONESHOT</code>	Удаление события после его однократного срабатывания	•	
<code>EV_EOF</code>	Достигнут конец файла	•	
<code>EV_ERROR</code>	Произошла ошибка, код ошибки записан в поле <code>data</code>	•	

Типы фильтров приведены в табл. 14.6.

Таблица 14.6. Типы фильтров

Значение <code>filter</code>	Описание
<code>EVFILT_AIO</code>	События асинхронного ввода-вывода
<code>EVFILT_PROC</code>	События <code>exit</code> , <code>fork</code> , <code>exec</code> для процесса
<code>EVFILT_READ</code>	Дескриптор готов для чтения (аналогично <code>select</code>)
<code>EVFILT_SIGNAL</code>	Описание сигнала
<code>EVFILT_TIMER</code>	Периодические или одноразовые таймеры
<code>EVFILT_VNODE</code>	Изменение и удаление файлов
<code>EVFILT_WRITE</code>	Дескриптор готов для записи (аналогично <code>select</code>)

Перепишем функцию `str_cli` из листинга 6.2 так, чтобы она использовала `kqueue`. Результат представлен в листинге 14.8.

Листинг 14.8. Функция `str_cli`, использующая `kqueue`

```
//advio/str_cli_kqueue04.c
1 #include "unp.h"

2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     int kq, i, n, nev, stdineof = 0, isfile;
6     char buf[MAXLINE];
7     struct kevent kev[2];
8     struct timespec ts;
9     struct stat st;
```

```

10  isfile = ((fstat(fileno(fp), &st) == 0) &&
11    (st.st_mode & S_IFMT) == S_IFREG);

12  EV_SET(&kev[0], fileno(fp), EVFILT_READ, EV_ADD, 0, 0, NULL);
13  EV_SET(&kev[1], sockfd, EVFILT_READ, EV_ADD, 0, 0, NULL);

14  kq = Kqueue();
15  ts.tv_sec = ts.tv_nsec = 0;
16  Kevent(kq, kev, 2, NULL, 0, &ts);

17  for (;;) {
18    nev = Kevent(kq, NULL, 0, kev, 2, NULL);

19    for (i = 0; i < nev; i++) {
20      if (kev[i].ident == sockfd) { /* сокет готов для чтения */
21        if ((n = Read(sockfd, buf, MAXLINE)) == 0) {
22          if (stdineof == 1)
23            return; /* нормальное завершение*/
24          else
25            err_quit("str_cli: server terminated prematurely");
26        }

27        Write(fileno(stdout), buf, n);
28      }

29      if (kev[i].ident == fileno(fp)) { /* входной поток готов к чтению */
30        n = Read(fileno(fp), buf, MAXLINE);
31        if (n > 0)
32          Writen(sockfd, buf, n);

33        if (n == 0 || (.isfile && n == kev[i].data)) {
34          stdineof = 1;
35          Shutdown(sockfd, SHUT_WR); /* отправка FIN */
36          kev[i].flags = EV_DELETE;
37          Kevent(kq, &kev[i], 1, NULL, 0, &ts); /* удаление
38                                         kevent */
39        }
40      }
41    }
42  }
43 }

```

Проверка, указывает ли дескриптор на файл

10-11 Поведение `kqueue` при достижении конца файла зависит от того, связан ли данный дескриптор с файлом, каналом или терминалом, поэтому мы вызываем `fstat`, чтобы убедиться, что мы работаем с файлом. Эти сведения понадобятся позже.

Настройка структур `kevent` для `kqueue`

12-13 При помощи макроса `EV_SET` мы настраиваем две структуры `kevent`. Обе содержат фильтр событий готовности к чтению (`EVFILT_READ`) и запрос на добавление этого события к фильтру (`EV_ADD`).

Создание кьюе и добавление фильтров

14-16 Мы вызываем `kqueue`, чтобы получить дескриптор `kqueue`, устанавливаем тайм-аут равным нулю, чтобы сделать вызов `kevent` неблокируемым, и наконец, вызываем `kevent` с массивом `kevent` на месте соответствующего аргумента.

Бесконечный цикл с блокированием в kevent

17-18 Мы входим в бесконечный цикл и блокируемся в `kevent`. Функции передается пустой список изменений, потому что все интересующие нас события уже зарегистрированы, и нулевой тайм-аут, что позволяет заблокироваться навечно.

Перебор возвращаемых событий в цикле

19 Мы проверяем все возвращаемые события и обрабатываем их последовательно.

Сокет готов для чтения

20-28 Эта часть кода ничем не отличается от листинга 6.2.

Вход готов для чтения

29-40 Код практически аналогичен листингу 6.2 за тем отличием, что нам приходится обрабатывать конец файла, возвращаемый `kqueue`. Для каналов и терминалов `kqueue` возвращает событие готовности дескриптора к чтению, подобно `select`, так что мы можем считать из этого дескриптора символ конца файла. Для файлов `kqueue` возвращает количество байтов, оставшихся в поле `data` структуры `struct kevent` и предполагает, что приложение само определит, когда оно доберется до конца этих данных. Поэтому мы переписываем цикл таким образом, чтобы отправлять данные по сети, если они были считаны из дескриптора. Затем проверяется достижение конца файла: если мы считали нуль байтов или если мы считали все оставшиеся байты из дескриптора файла, значит, это и есть `EOF`. Еще одно изменение состоит в том, что вместо `FD_CLR` для удаления дескриптора из набора файлов мы используем флаг `EV_DELETE` и вызываем `kevent` для удаления события из фильтра в ядре.

Рекомендации

Новыми интерфейсами следует пользоваться аккуратно. Читайте свежую документацию, относящуюся к конкретному выпуску операционной системы. Интерфейсы часто меняются от одного выпуска к другому, причем таким образом, что заметно это далеко не сразу. Все это продолжается до тех пор, пока поставщики не проработают все детали функционирования новых интерфейсов.

В целом, лучше избегать написания непереносимых программ. Однако для оптимизации ресурсоемких приложений годятся все средства.

14.10. Резюме

Существует три способа установить ограничение времени для операции с сокетом:

- Использовать функцию `alarm` и сигнал `SIGALRM`.
- Задать предел времени в функции `select`.
- Использовать более новые параметры сокета `SO_RCVTIMEO` и `SO_SNDFTIMEO`.

Первый способ легко использовать, но он включает обработку сигналов и, как показано в разделе 20.5, может привести к ситуации гонок. Использование функции `select` означает, что блокирование происходит в этой функции (с заданным в ней пределом времени) вместо блокирования в вызове функций `read`, `write`

или `connect`. Другая альтернатива — использование новых параметров сокета — также проста в использовании, но предоставляется не всеми реализациями.

Функции `recvmsg` и `sendmsg` являются наиболее общими из пяти групп предоставляемых функций ввода-вывода. Они объединяют целый ряд возможностей, свойственных другим функциям ввода-вывода, позволяя задавать флаг `MSG_XXX` (как функции `recv` и `send`), возвращать или задавать адрес протокола собеседника (как функции `recvfrom` и `sendto`), использовать множество буферов (как функции `readv` и `writev`). Кроме того, они обеспечивают две новых возможности: возвращение флагов приложению и получение или отправку вспомогательных данных.

В тексте книги мы описываем десять различных форм вспомогательных данных, шесть из которых появились в IPv6. Вспомогательные данные состоят из объектов вспомогательных данных. Перед каждым объектом идет структура `cmsghdr`, задающая его длину, уровень протокола и тип данных. Пять макросов, начинающихся с префикса `CMSG_`, используются для создания и анализа вспомогательных данных.

Сокеты могут использоваться со стандартной библиотекой ввода-вывода C, но это добавляет еще один уровень буферизации к уже имеющемуся в TCP. На самом деле недостаток понимания буферизации, выполняемой стандартной библиотекой ввода-вывода, является наиболее общей проблемой при работе с этой библиотекой. Поскольку сокет не является терминальным устройством, общим решением этой потенциальной проблемы будет отключение буферизации стандартного потока ввода-вывода.

Многие производители предоставляют усовершенствованные средства опроса событий без накладных расходов на `select` и `poll`. Не стоит увлекаться написанием непереносимого кода, однако иногда преимущества перевешивают риск непереносимости.

Упражнения

1. Что происходит в листинге 14.1, когда мы переустанавливаем обработчик сигналов, если процесс не установил обработчик для сигнала `SIGALRM`?

2. В листинге 14.1 мы выводим предупреждение, если у процесса уже установлен таймер `alarm`. Измените эту функцию так, чтобы новое значение `alarm` для процесса задавалось после выполнения `connect` до завершения функции.

3. Измените листинг 11.5 следующим образом: перед вызовом функции `read` вызовите функцию `recv` с флагом `MSG_PEEK`. Когда она завершится, вызовите функцию `ioctl` с командой `FIONREAD` и выведите число байтов, установленных в очередь в буфере приема сокета. Затем вызовите функцию `read` для фактического чтения данных.

4. Что происходит с оставшимися в стандартном буфере ввода-вывода данными, если процесс, дойдя до конца функции `main`, не обнаруживает там функции `exit`?

5. Примените каждое из двух изменений, описанных после листинга 14.6, и убедитесь в том, что каждое из них решает проблему буферизации.

Глава 15

Доменные протоколы Unix

15.1. Введение

Доменные протоколы Unix — это не набор протоколов, а способ связи клиентов и серверов на отдельном узле, использующий тот же API, который используется для клиентов и серверов на различных узлах, — сокеты или XTI. Доменные протоколы Unix представляют альтернативу методам IPC (Interprocess Communications — взаимодействие процессов), которым посвящен второй том^[2] этой серии, применяемым, когда клиент и сервер находятся на одном узле. Подробности действительной реализации доменных сокетов Unix в ядре, происходящем от Беркли, приводятся в третьей части [112].

В домене Unix предоставляются два типа сокетов: потоковые (аналогичные сокетам TCP) и дейтаграммные (аналогичные сокетам UDP). Хотя предоставляется также и символьный сокет, но его семантика никогда не документировалась, он не используется никакой из известных автору программ и не определяется в POSIX.

Доменные сокеты Unix используются по трем причинам.

1. В реализациях, происходящих от Беркли, доменные сокеты Unix часто вдвое быстрее сокетов TCP, когда оба собеседника находятся на одном и том же узле [112, с. 223–224]. Есть приложение, которое использует это преимущество: X Window System. Когда клиент X11 запускается и открывает соединение с сервером X11, клиент проверяет значение переменной окружения DISPLAY, которая задает имя узла сервера, окно и экран. Если сервер находится на том же узле, что и клиент, клиент открывает потоковое соединение с сервером через доменный сокет Unix, в противном случае клиент открывает соединение TCP.

2. Доменные сокеты Unix используются при передаче дескрипторов между процессами на одном и том же узле. Пример мы приводим в разделе 15.7.

3. Более новые реализации доменных сокетов Unix предоставляют регистрационные данные клиента (идентификатор пользователя и идентификаторы группы) серверу, что может служить дополнительной проверкой безопасности. Мы покажем это в разделе 15.8.

Адреса протоколов, используемые для идентификации клиентов и серверов в домене Unix, — это полные имена в обычной файловой системе. Вспомните, что IPv4 использует комбинацию 32-разрядных адресов и 16-разрядных номеров портов для своих адресов протоколов, а IPv6 для своих адресов протоколов использует комбинацию 128-разрядных адресов и 16-разрядных номеров портов. Эти полные имена не являются обычными именами файлов Unix: в общем случае мы не можем читать из этих файлов или записывать в них. Это может делать только программа, связывающая полное имя с доменным сокетом Unix.

15.2. Структура адреса доменного сокета Unix

В листинге 15.1^[1] показана структура адреса доменного сокета Unix, задаваемая включением заголовочного файла <sys/un.h>.

Листинг 15.1. Структура адреса доменного сокета Unix: sockaddr_un

```
struct sockaddr_un {  
    uint8_t      sun_len;  
    sa_family_t  sun_family; /* AF_LOCAL */  
    char        sun_path[104]; /* полное имя, оканчивающееся нулем */  
};
```

ПРИМЕЧАНИЕ

POSIX не задает длину массива sun_path и предупреждает, что разработчику приложения не следует делать предположений об этой длине. Воспользуйтесь оператором sizeof для определения длины массива во время выполнения программы. Убедитесь, что полное имя помещается в этот массив. Длина, скорее всего, будет где-то между 92 и 108. Причина этих ограничений — в

артефакте реализации, возникшем еще в 4.2BSD, где требовалось, чтобы структура помещалась в 128-байтовый буфер памяти ядра mbuf.

Полное имя, хранимое в символьном массиве sun_path, должно завершаться нулем. Имеется макрос SUN_LEN, который получает указатель на структуру sockaddr_un и возвращает длину структуры, включая число непустых байтов в полном имени. Неопределенный адрес обозначается пустой строкой, то есть элемент sun_path[0] должен быть равен нулю. Это эквивалент константы INADDR_ANY протокола IPv4 и константы IN6ADDR_ANY_INIT протокола IPv6 для домена Unix.

ПРИМЕЧАНИЕ

В POSIX доменным протоколом Unix дали название «локального IPC», чтобы не подчеркивать зависимость от операционной системы Unix. Историческая константа AF_UNIX становится константой AF_LOCAL. Тем не менее мы будем продолжать использовать термин «домен Unix», так как он стал именем де-факто, независимо от соответствующей операционной системы. Кроме того, несмотря на попытку POSIX исключить зависимость от операционной системы, структура адреса сокета сохраняет суффикс _un!

Пример: функция bind и доменный сокет Unix

Программа, показанная в листинге 15.2, создает доменный сокет Unix, с помощью функции bind связывает с ним полное имя и затем вызывает функцию getsockname и выводит это полное имя.

Листинг 15.2. Связывание полного имени с доменным сокетом Unix

```
unixdomain/unixbind.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;
6     socklen_t len;
7     struct sockaddr_un addr1, addr2;

8     if (argc != 2)
9         err_quit("usage: unixbind <pathname>");

10    sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);

11    unlink(argv[1]); /* игнорируем возможную ошибку */

12    bzero(&addr1, sizeof(addr1));
13    addr1.sun_family = AF_LOCAL;
14    strncpy(addr1.sun_path, argv[1], sizeof(addr1.sun_path) - 1);
15    Bind(sockfd, (SA*)&addr1, SUN_LEN(&addr1));

16    len = sizeof(addr2);
17    Getsockname(sockfd, (SA*)&addr2, &len);
18    printf("bound name = %s, returned len = %d\n", addr2.sun_path, len);

19    exit(0);
20 }
```

Удаление файла

11 Полное имя, которое функция `bind` должна связать с сокетом, — это аргумент командной строки. Если это полное имя уже существует в файловой системе, при выполнении функции `bind` возникает ошибка. Следовательно, мы вызываем функцию `unlink`, чтобы удалить файл в том случае, если он уже существует. Если его не существует, функция `unlink` возвращает ошибку, которую мы игнорируем.

Вызов функций `bind` и `getsockname`

12-18 Мы копируем аргумент командной строки, используя функцию `strncpy`, чтобы избежать переполнения структуры, если полное имя слишком длинное. Поскольку мы инициализируем структуру нулем и затем вычитаем единицу из размера массива `sun_path`, мы знаем, что полное имя оканчивается нулем. Далее вызывается функция `bind` и мы используем макрос `SUN_LEN` для вычисления длины аргумента функции. Затем мы вызываем функцию `getsockname`, чтобы получить имя, которое было только что связано с сокетом, и выводим результат.

Если мы запустим программу в Solaris, то получим следующие результаты:

```
solaris % umask сначала выводим наше значение umask  
022           оно отображается в восьмеричной системе  
solaris % unixbind /tmp/moose  
bound name = /tmp/moose, returned len = 13  
solaris % unixbind /tmp/moose снова запускаем программу  
bound name = /tmp/moose, returned len = 13  
solaris % ls -l /tmp/moose  
srwxr-xr-x 1 andy staff 0 Aug 10 13:13 /tmp/moose  
solaris % ls -lF /tmp/foo.bar  
srwxr-xr-x 1 andy staff 0 Aug 10 13:13 /tmp/moose=
```

Сначала мы выводим наше значение `umask`, поскольку в POSIX указано, что права доступа к создаваемому объекту определяются этим значением. Наше значение 022 выключает биты, разрешающие запись в файл для пользователей из группы (group-write) и прочих пользователей (other-write). Затем мы запускаем программу и видим, что длина, возвращаемая функцией `getsockname`, равна 13: один байт для элемента `sun_len`, один байт для элемента `sun_family` и 11 байт для полного имени (исключая завершающий нуль). Это пример аргумента типа «значение-результат», значение которого при завершении функции отличается от значения при вызове функции. Мы можем вывести полное имя, используя спецификатор формата `%s` функции `printf`, поскольку полное имя, хранящееся в `sun_path`, представляет собой завершающуюся нулем строку. Затем мы снова запускаем программу, чтобы проверить, что вызов функции `unlink` удаляет соответствующий файл.

Мы запускаем команду `ls -l`, чтобы увидеть биты разрешения для файла и тип файла. В Solaris (и большинстве версий Unix) тип файла — это сокет, что обозначается символом `s`. Мы также замечаем, что все девять битов разрешения включены, так как Solaris не изменяет принятые по умолчанию биты разрешения на наше значение `umask`. Наконец, мы снова запускаем `ls` с параметром `-F`, что заставляет Solaris добавить знак равенства (соответствующий типу «сокет») к полному имени.

ПРИМЕЧАНИЕ

Изначально значение `umask` не действовало на создаваемые процессами доменные сокеты Unix, но с течением времени производители исправили это упущение, чтобы устанавливаемые разрешения соответствовали ожиданиям разработчиков. Тем не менее все еще существуют системы, в которых разрешения доменного сокета могут не зависеть от значения `umask`. В других системах сокеты могут отображаться как каналы (символ `p`), а значок равенства при вызове `ls -F` может не отображаться вовсе. Однако поведение, демонстрируемое в нашем примере, является наиболее типичным.

15.3. Функция `socketpair`

Функция `socketpair` создает два сокета, которые затем соединяются друг с другом. Эта функция применяется только к доменным сокетам Unix.

```
#include <sys/socket.h>

int socketpair(int family, int type, int protocol, int sockfd[2]);
```

Возвращает: ненулевое значение в случае успешного выполнения, -1 в случае ошибки

Аргумент *family* должен быть равен *AF_LOCAL*, а аргумент *protocol* должен быть нулевым. Однако аргумент *type* может быть равен как *SOCK_STREAM*, так и *SOCK_DGRAM*. Два дескриптора сокета создаются и возвращаются как *sockfd[0]* и *sockfd[1]*.

ПРИМЕЧАНИЕ

Эта функция аналогична функции Unix *pipe*: при ее вызове возвращаются два дескриптора, причем каждый дескриптор соединен с другим. Действительно, в Беркли-реализации внутреннее устройство функции *pipe* полностью аналогично функции *socketpair* [112, с. 253-254].

Два созданных сокета не имеют имен. Это значит, что не было неявного вызова функции *bind*.

Результат выполнения функции *socketpair* с аргументом *type*, равным *SOCK_STREAM*, называется потоковым каналом (*stream pipe*). Потоковый канал является аналогом обычного канала Unix (который создается функцией *pipe*), но он двусторонний, что позволяет использовать оба дескриптора и для чтения, и для записи. Потоковый канал, созданный функцией *socketpair*, изображен на рис. 15.1.

ПРИМЕЧАНИЕ

POSIX не требует поддержки двусторонних каналов. В SVR4 функция *pipe* возвращает два двусторонних дескриптора, в то время как ядра, происходящие от Беркли, традиционно возвращают односторонние дескрипторы (см. рис. 17.31 [112]).

15.4. Функции сокетов

Функции сокетов применяются к доменным сокетам Unix с учетом некоторых особенностей и ограничений. Далее мы перечисляем требования POSIX, указывая, где они применимы. Отметим, что на сегодняшний день не все реализации соответствуют этим требованиям.

1. Права доступа к файлу по умолчанию для полного имени, созданного функцией *bind*, задаются значением 0777 (чтение, запись и выполнение данного файла разрешены владельцу файла, группе пользователей, в которую он входит, и всем остальным пользователям) и могут быть изменены в соответствии с текущим значением *umask*.

2. Имя, связанное с доменным сокетом Unix, должно быть абсолютным, а не относительным именем. Причина, по которой нужно избегать относительного имени, в том, что в таком случае разрешение имени зависит от текущего рабочего каталога вызывающего процесса. То есть если сервер связывается с относительным именем, клиент должен находиться в том же каталоге, что и сервер (или должен знать этот каталог), для того чтобы вызов клиентом функции *connect* или *sendto* был успешным.

ПРИМЕЧАНИЕ

В POSIX сказано, что связывание относительного имени с доменным сокетом Unix приводит к непредсказуемым результатам.

3. Полное имя, заданное в вызове функции *connect*, должно быть именем, в настоящий момент связанным с открытym доменным сокетом Unix того же типа (потоковым или дейтаграммным). Ошибка происходит в следующих случаях: если имя существует, но не является сокетом; если имя существует и является сокетом, но ни один открытый дескриптор с ним не связан; если имя существует и является открытym сокетом, но имеет неверный тип (то есть потоковый доменный сокет Unix не может соединиться с именем, связанным с дейтаграммным доменным сокетом Unix, и наоборот).

4. С функцией `connect` доменного сокета Unix связана такая же проверка прав доступа, какая имеет место при вызове функции `open` для доступа к файлу только на запись.

5. Потоковые доменные сокеты Unix аналогичны сокетам TCP: они предоставляют интерфейс байтового потока без границ записей.

6. Если при вызове функции `connect` для потокового доменного сокета Unix обнаруживается, что очередь прослушиваемого сокета переполнена (см. раздел 4.5), немедленно возвращается ошибка `ECONNREFUSED`. В этом отличие от сокета TCP: прослушиваемый сокет TCP игнорирует приходящий сегмент SYN, если очередь сокета заполнена, благодаря чему стеком клиента выполняется несколько попыток отправки сегмента SYN.

7. Дейтаграммные доменные сокеты Unix аналогичны сокетам UDP: они предоставляют ненадежный сервис дейтаграмм, сохраняющий границы записей.

8. В отличие от сокетов UDP, при отправке дейтаграммы на неприсоединенный дейтаграммный доменный сокет Unix с сокетом не связывается полное имя. (Вспомните, что отправка дейтаграммы UDP на неприсоединенный сокет UDP заставляет динамически назначаемый порт связываться с сокетом.) Это означает, что получатель дейтаграммы не будет иметь возможности отправить ответ, если отправитель не связал со своим сокетом полное имя. Аналогично, в отличие от TCP и UDP, при вызове функции `connect` для дейтаграммного доменного сокета Unix с сокетом не связывается полное имя.

15.5. Клиент и сервер потокового доменного протокола Unix

Теперь мы перепишем наш эхо-клиент и эхо-сервер TCP из главы 5 с использованием доменных сокетов Unix. В листинге 15.3 показан сервер, который является модификацией сервера из листинга 5.9 и использует потоковый доменный протокол Unix вместо протокола TCP.

Листинг 15.3. Эхо-сервер потокового доменного протокола Unix

```
//unixdomain/unixstrserv01.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int listenfd, connfd;
6     pid_t childpid;
7     socklen_t clilen;
8     struct sockaddr_un cliaddr, servaddr;
9     void sig_chld(int);

10    listenfd = Socket(AF_LOCAL, SOCK_STREAM, 0);

11    unlink(UNIXSTR_PATH);
12    bzero(&servaddr, sizeof(servaddr));
13    servaddr.sun_family = AF_LOCAL;
14    strcpy(servaddr.sun_path, UNIXSTR_PATH);

15    Bind(listenfd, (SA*)&servaddr, sizeof(servaddr));

16    Listen(listenfd, LISTENQ);
17    Signal(SIGCHLD, sig_chld);

18    for (;;) {
19        clilen = sizeof(cliaddr);
20        if ((connfd = accept(listenfd, (SA*)&cliaddr, &clilen)) < 0) {
21            if (errno == EINTR)
22                continue; /* назад в for() */
23            else
24                err_sys("accept error");
25        }
```

```

26     if ((childpid = Fork()) == 0) { /* дочерний процесс */
27         Close(listenfd); /* закрывается прослушиваемый сокет */
28         str_echo(connfd); /* обработка запроса */
29         exit(0);
30     }
31     Close(connfd); /* родитель закрывает присоединенный сокет */
32 }
33 }

```

8 Теперь две структуры адреса сокета относятся к типу `sockaddr_un`.

10 Для создания потокового доменного сокета Unix первый аргумент функции `socket` должен иметь значение `AF_LOCAL`.

11-15 Константа `UNIXSTR_PATH` определяется в файле `unp.h` как `/tmp/unix/str`. Сначала мы вызываем функцию `unlink`, чтобы удалить полное имя в случае, если оно сохранилось после предыдущего запуска сервера, а затем инициализируем структуру адреса сокета перед вызовом функции `bind`. Ошибка при выполнении функции `unlink` не является аварийной ситуацией.

Обратите внимание, что этот вызов функции `bind` отличается от вызова, показанного в листинге 15.2. Здесь мы задаем размер структуры адреса сокета (третий аргумент) как общий размер структуры `sockaddr_un`, а не просто число байтов, занимаемое полным именем. Оба значения длины приемлемы, поскольку полное имя должно оканчиваться нулем.

Оставшаяся часть функции такая же, как и в листинге 5.9. Используется та же функция `str_echo` (см. листинг 5.2).

В листинге 15.4 представлен эхо-клиент потокового доменного протокола Unix. Это модификация листинга 5.3.

Листинг 15.4. Эхо-клиент потокового доменного протокола Unix

```

//unixdomain/umxstrcli01.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;
6     struct sockaddr_un servaddr;

7     sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);

8     bzero(&servaddr, sizeof(servaddr));
9     servaddr.sun_family = AF_LOCAL;
10    strcpy(servaddr.sun_path, UNIXSTR_PATH);
11    Connect(sockfd, (SA*)&servaddr, sizeof(servaddr));
12    str_cli(stdin, sockfd); /* выполняет всю работу */

13    exit(0);
14 }

```

6 Теперь структурой адреса сокета, которая должна содержать адрес сервера, будет структура `sockaddr_un`.

7 Первый аргумент функции `socket` — `AF_LOCAL`.

8-10 Код для заполнения структуры адреса сокета идентичен коду, показанному для сервера: инициализация структуры нулем, установка семейства протоколов `AF_LOCAL` и копирование полного имени в элемент `sun_path`.

12 Функция `str_cli` — та же, что и раньше (в листинге 6.2 представлена последняя разработанная нами версия).

15.6. Клиент и сервер дейтаграммного доменного протокола Unix

Теперь мы перепишем наши клиент и сервер UDP из разделов 8.3 и 8.5 с использованием сокетов. В листинге 15.5 показан сервер, который является модификацией листинга 8.1.

Листинг 15.5. Эхо-сервер дейтаграммного доменного протокола Unix

```
//unixdomain/unixdgserver01.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;
6     struct sockaddr_un servaddr, cliaddr;

7     sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);

8     unlink(UNIXDG_PATH);
9     bzero(&servaddr, sizeof(servaddr));
10    servaddr.sun_family = AF_LOCAL;
11    strcpy(servaddr.sun_path, UNIXDG_PATH);

12    Bind(sockfd, (SA*)&servaddr, sizeof(servaddr));

13    dg_echo(sockfd, (SA*)&cliaddr, sizeof(cliaddr));
14 }
```

6 Две структуры адреса сокета относятся теперь к типу `sockaddr_un`.

7 Для создания дейтаграммного доменного сокета Unix первый аргумент функции `socket` должен иметь значение `AF_LOCAL`.

8-12 Константа `UNIXDG_PATH` определяется в заголовочном файле `unp.h` как `/tmp/unix.dg`. Сначала мы вызываем функцию `unlink`, чтобы удалить полное имя в случае, если оно сохранилось после предыдущего запуска сервера, а затем инициализируем структуру адреса сокета перед вызовом функции `bind`. Ошибка при выполнении функции `unlink` — это нормальное явление.

13 Используется та же функция `dg_echo` (см. листинг 8.2).

В листинге 15.6 представлен эхо-клиент дейтаграммного доменного протокола Unix. Это модификация листинга 8.3.

Листинг 15.6. Эхо-клиент дейтаграммного доменного протокола Unix

```
//unixdomain/unixdgclient01.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;
6     struct sockaddr_un cliaddr, servaddr;

7     sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);

8     bzero(&cliaddr, sizeof(cliaddr)); /* связывание сокета с адресом */
9     cliaddr.sun_family = AF_LOCAL;
10    strcpy(cliaddr.sun_path, tmpnam(NULL));

11    Bind(sockfd, (SA*)&cliaddr, sizeof(cliaddr));

12    bzero(&servaddr, sizeof(servaddr)); /* заполняем структуру адреса */
                                         /* сокета сервера */

13    servaddr.sun_family = AF_LOCAL;
14    strcpy(servaddr.sun_path, UNIXDG_PATH);

15    dg_cli(stdin, sockfd, (SA*)&servaddr, sizeof(servaddr));
```

```
16 exit(0);
17 }
```

6 Структурой адреса сокета, содержащей адрес сервера, теперь будет структура `sockaddr_un`. Мы также размещаем в памяти одну из этих структур, чтобы она содержала адрес клиента, о чём мы расскажем далее.

7 Первый аргумент функции `socket` — это `AF_LOCAL`.

8-11 В отличие от клиента UDP при использовании дейтаграммного доменного протокола Unix требуется явно связать с помощью функции `bind` полное имя с нашим сокетом, чтобы сервер имел полное имя, на которое он мог бы отправить свой ответ. Мы вызываем функцию `tmpnam`, чтобы получить уникальное полное имя, с которым затем при помощи функции `bind` связем наш сокет. Вспомните из раздела 15.4, что при отправке дейтаграммы на неприсоединенный дейтаграммный доменный сокет Unix не происходит неявного связывания полного имени с сокетом. Следовательно, если мы опустим этот шаг, вызов сервером функции `recvfrom` в функции `dg_echo` возвращает пустое полное имя, что затем приведёт к ошибке, когда сервер вызовет функцию `sendto`.

12-14 Код для заполнения структуры адреса сокета заранее известным полным именем идентичен коду, представленному ранее для сервера.

15 Функция `dg_cli` остается той же, что и раньше (см. листинг 8.4).

15.7. Передача дескрипторов

Когда нам требуется передать дескриптор от одного процесса другому, обычно мы выбираем одно из двух решений:

1. Дочерний процесс использует все открытые дескрипторы совместно с родительским процессом после вызова функции `fork`.

2. Все дескрипторы обычно остаются открытыми при вызове функции `exec`.

В первом случае процесс открывает дескриптор, вызывает функцию `fork`, а затем родительский процесс закрывает дескриптор, позволяя дочернему процессу с ним работать. При этом открытый дескриптор передается от родительского процесса дочернему. Но нам также хотелось бы, чтобы у дочернего процесса была возможность открывать дескриптор и передавать его обратно родительскому процессу.

Современные системы Unix предоставляют способ передавать любой открытый дескриптор от одного процесса любому другому процессу. При этом вовсе не обязательно, чтобы процессы были родственными, как родительский и дочерний. Эта технология требует, чтобы мы сначала создали между двумя процессами доменный сокет Unix и затем использовали функцию `sendmsg` для отправки специального сообщения через этот доменный сокет. Ядро обрабатывает это сообщение специальным образом, передавая открытый дескриптор от отправителя получателю.

ПРИМЕЧАНИЕ

Передача ядром 4.4BSD открытого дескриптора через доменный сокет Unix описывается в главе 18 [112].

SVR4 использует другую технологию внутри ядра для передачи открытого дескриптора: команды `I_SENDFD` и `I_RECVFD` функции `ioctl`, описанные в разделе 15.5.1 [110]. Но процесс все же имеет возможность доступа к указанному свойству ядра за счет доменного сокета Unix. В этой книге мы описываем применение доменных сокетов Unix для передачи открытых дескрипторов, поскольку это наиболее переносимая технология программирования: она работает как с Беркли-ядрами, так и с SVR4, в то время как команды `I_SENDFD` и `I_RECVFD` функции `ioctl` работают только в SVR4.

Технология 4.4BSD позволяет передавать множество дескрипторов с помощью одиночной функции `sendmsg`, в то время как технология SVR4 передает за один раз только один дескриптор. Во всех наших примерах за один раз передается один дескриптор.

Шаги при передаче дескриптора между процессами будут такими:

1. Создание доменного сокета Unix, или потокового сокета, или дейтаграммного сокета.

Если целью является породить с помощью функции `fork` дочерний процесс, с тем чтобы дочерний процесс открыл дескриптор и передал его обратно родительскому процессу, родительский процесс может вызвать функцию `socketpair` для создания потокового канала, который может использоваться для передачи дескриптора.

Если процессы не являются родственными, сервер должен создать потоковый доменный сокет Unix, связать его при помощи функции `bind` с полным именем, тем самым позволяя клиенту соединиться с этим сокетом при помощи функции `connect`. Затем клиент может отправить запрос серверу для открытия некоторого дескриптора, а сервер может передать дескриптор обратно через доменный сокет Unix. Как альтернатива между клиентом и сервером может также использоваться дейтаграммный доменный сокет Unix, однако преимущества этого способа невелики, к тому же существует возможность игнорирования дейтаграммы. Далее в примерах этой главы мы будем использовать потоковый сокет между клиентом и сервером.

2. Один процесс открывает дескриптор при помощи вызова любой из функций Unix, возвращающей дескриптор, например `open`, `pipe`, `mkfifo`, `socket` или `accept`. От одного процесса к другому можно передать дескриптор **любого** типа, поэтому мы называем эту технологию «передачей дескриптора», а не «передачей дескриптора файла».

3. Отправляющий процесс строит структуру `msghdr` (см. раздел 14.5), содержащую дескриптор, который нужно передать. В POSIX определено, что дескриптор должен отправляться как вспомогательные данные (элемент `msg_control` структуры `msghdr`, см. раздел 14.6), но более старые реализации используют элемент `msg_accrights`. Отправляющий процесс вызывает функцию `sendmsg` для отправки дескриптора через доменный сокет Unix, созданный на шаге 1. На этом этапе мы говорим, что дескриптор находится «в полете». Даже если отправляющий процесс закроет дескриптор после вызова функции `sendmsg`, но до вызова принимающим процессом функции `recvmsg`, дескриптор останется открытым для принимающего процесса. Отправка дескриптора увеличивает счетчик ссылок дескриптора на единицу.

4. Принимающий процесс вызывает функцию `recvmsg` для получения дескриптора через доменный сокет Unix, созданный на шаге 1. Номер дескриптора в принимающем процессе может отличаться от номера дескриптора в отправляющем процессе. Передача дескриптора — это не передача номера дескриптора. Этот процесс включает создание нового дескриптора в принимающем процессе, который ссылается на ту же запись таблицы файлов в ядре, что и дескриптор, отправленный отправляющим процессом.

Клиент и сервер должны располагать некоторым протоколом уровня приложения, с тем чтобы получатель дескриптора имел информацию о времени его появления. Если получатель вызывает функцию `recvmsg`, не выделив места в памяти для получения дескриптора, и дескриптор передается как готовый для чтения, то передаваемый дескриптор закрывается [128, с. 518]. Кроме того, нужно избегать установки флага `MSG_PEEK` в функции `recvmsg`, если предполагается получение дескриптора, поскольку в этом случае результат непредсказуем.

Пример передачи дескриптора

Теперь мы представим пример передачи дескриптора. Мы напишем программу под названием `mysat`, которой в качестве аргумента командной строки передается полное имя файла. Эта программа открывает файл и копирует его в стандартный поток вывода. Но вместо вызова обычной функции Unix `open` мы вызываем нашу собственную функцию `my_open`. Эта функция создает потоковый канал и вызывает функции `fork` и `exec` для запуска другой программы, открывающей нужный файл. Эта программа должна затем передать дескриптор обратно родительскому процессу по потоковому каналу.

На рис. 15.1 показан первый шаг: наша программа `mysat` после создания потокового канала при помощи вызова функции `socketpair`. Мы обозначили два дескриптора, возвращаемых функцией `socketpair`, как [0] и [1].



Рис. 15.1. Программа `mysat` после создания потокового канала при использовании функции `socketpair`

Затем процесс вызывает функцию `fork`, и дочерний процесс вызывает функцию `exec` для выполнения программы `openfile`. Родительский процесс закрывает дескриптор [1], а дочерний процесс закрывает

дескриптор [0]. (Нет разницы, на каком конце потокового канала происходит закрытие. Дочерний процесс мог бы закрыть [1], а родительский — [0].) При этом получается схема, показанная на рис. 15.2.



Рис. 15.2. Программа mycat после запуска программы openfile

Родительский процесс должен передать программе openfile три фрагмента информации: полное имя открываемого файла, режим открытия (только чтение чтение и запись или только запись) и номер дескриптора, соответствующий его концу потокового канала (который мы обозначили [1]). Мы выбрали такой способ передачи этих трех элементов, как ввод аргументов командной строки при вызове функции exec. Альтернативным способом будет отправка этих элементов в качестве данных по потоковому каналу. Программа отправляет обратно открытый дескриптор по потоковому каналу и завершается. Статус выхода программы сообщает родительскому процессу, смог ли файл открыться, и если нет, то какого типа ошибка произошла.

Преимущество выполнения дополнительной программы для открытия файла заключается в том, что за счет приравнивания привилегий пользователя к привилегиям владельца файла мы получаем возможность открывать те файлы, которые не имеем права открывать в обычной ситуации. Эта программа позволяет расширить концепцию обычных прав доступа Unix (пользователь, группа и все остальные) и включить любые формы проверки прав доступа. Мы начнем с программы mycat, показанной в листинге 15.7.

Листинг 15.7. Программа mycat: копирование файла в стандартный поток вывода

```

//unixdomain/mycat.c
1 #include "unp.h"

2 int my_open(const char*, int);

3 int
4 main(int argc, char **argv)
5 {
6     int fd, n;
7     char buff[BUFFSIZE];

8     if (argc != 2)
9         err_quit("usage: mycat <pathname>");

10    if ((fd = my_open(argv[1], O_RDONLY)) < 0)
11        err_sys("cannot open %s", argv[1]);

12    while ((n = Read(fd, buff, BUFFSIZE)) > 0)
13        Write(STDOUT_FILENO, buff, n);

14    exit(0);
15 }
  
```

Если мы заменим вызов функции my_open вызовом функции open, эта простая программа всего лишь скопирует файл в стандартный поток вывода.

Функция my_open, показанная в листинге 15.8, должна выглядеть для вызывающего процесса как обычная функция Unix open. Она получает два аргумента — полное имя и режим открытия (например, O_RDONLY обозначает, что файл доступен только для чтения), открывает файл и возвращает дескриптор.

Листинг 15.8. Функция my_open: открытие файла и возвращение дескриптора

```

//unixdomain/myopen.c
1 #include "unp.h"

2 int
  
```

```

3 my_open(const char *pathname, int mode)
4 {
5     int fd, sockfd[2], status;
6     pid_t childpid;
7     char c, argsockfd[10], argmode[10];
8
9     Socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd);
10
11    if ((childpid = Fork()) == 0) { /* дочерний процесс */
12        Close(sockfd[0]);
13        sprintf(argsockfd, sizeof(argsockfd), "%d", sockfd[1]);
14        sprintf(argmode, sizeof(argmode), "%d", mode);
15        execl("./openfile", "openfile", argsockfd, pathname, argmode,
16              (char*)NULL);
17        err_sys("execl error");
18    }
19    /* родительский процесс - ожидание завершения дочернего процесса */
20    Close(sockfd[1]); /* закрываем конец, который мы не используем */
21
22    Waitpid(childpid, &status, 0);
23    if (WIFEXITED(status) == 0)
24        err_quit("child did not terminate");
25    if ((status = WEXITSTATUS(status)) == 0)
26        Read_fd(sockfd[0], &c, 1, &fd);
27    else {
28        errno = status; /* установка значения errno в статус дочернего
                           процесса */
29        fd = -1;
30    }
31
32    Close(sockfd[0]);
33    return (fd);
34 }
```

Создание потокового канала

8 Функция `socketpair` создает потоковый канал. Возвращаются два дескриптора: `sockfd[0]` и `sockfd[1]`. Это состояние, которое мы показали на рис. 15.1.

Функции `fork` и `exec`

9-16 Вызывается функция `fork`, после чего дочерний процесс закрывает один конец потокового канала. Номер дескриптора другого конца потокового канала помещается в массив `argsockfd`, а режим открытия помещается в массив `argmode`. Мы вызываем функцию `sprintf`, поскольку аргументы функции `exec` должны быть символьными строками. Выполняется программа `openfile`. Функция `execl` возвращает управление только в том случае, если она встретит ошибку. При удачном выполнении начинает выполняться функция `main` программы `openfile`.

Родительский процесс в ожидании завершения дочернего процесса

17-22 Родительский процесс закрывает другой конец потокового канала и вызывает функцию `waitpid` для ожидания завершения дочернего процесса. Статус завершения дочернего процесса возвращается в переменной `status`, и сначала мы проверяем, что программа завершилась нормально (то есть не была

завершена из-за возникновения какого-либо сигнала). Затем макрос `WEXITSTATUS` преобразует статус завершения в статус выхода, значение которого должно быть между 0 и 255. Мы вскоре увидим, что если при открытии необходимого файла программой `openfile` происходит ошибка, то эта программа завершается, причем статус ее завершения равен соответствующему значению переменной `errno`.

Получение дескриптора

23 Наша функция `read_fd`, которую мы показываем в следующем листинге, получает дескриптор потокового канала. Кроме получения дескриптора мы считываем 1 байт данных, но ничего с этими данными не делаем.

ПРИМЕЧАНИЕ

При отправке и получении дескриптора по потоковому каналу мы всегда отправляем как минимум 1 байт данных, даже если получатель никак эти данные не обрабатывает. Иначе получатель не сможет распознать, что значит нулевое возвращаемое значение из функции `read_fd`: отсутствие данных (но, возможно, есть дескриптор) или конец файла.

В листинге 15.9 показана функция `readfd`, вызывающая функцию `recvmsg` для получения данных и дескриптора через доменный сокет Unix. Первые три аргумента этой функции те же, что и для функции `read`, а четвертый (`recvfd`) является указателем на целое число. После выполнения этой функции `recvfd` будет указывать на полученный дескриптор.

Листинг 15.9. Функция `read_fd`: получение данных и дескриптора

```
//lib/read_fd.c
1 #include "unp.h"

2 ssize_t
3 read_fd(int fd, void *ptr, size_t nbytes, int *recvfd)
4 {
5     struct msghdr msg;
6     struct iovec iov[1];
7     ssize_t n;
8     int newfd;

9 #ifdef HAVE_MSGHDR_MSG_CONTROL
10    union {
11        struct cmsghdr cm;
12        char control[CMSG_SPACE(sizeof(int))];
13    } control_un;
14    struct cmsghdr *cmprtr;

15    msg.msg_control = control_un.control;
16    msg.msg_controllen = sizeof(control_un.control);
17 #else
18    msg.msg_accrights = (caddr_t)&newfd;
19    msg.msg_accrightslen = sizeof(int);
20 #endif

21    msg.msg_name = NULL;
22    msg.msg_namelen = 0;

23    iov[0].iov_base = ptr;
24    iov[0].iov_len = nbytes;
25    msg.msg_iov = iov;
```

```

26 msg.msg iovlen = 1;

27 if ((n = recvmsg(fd, &msg, 0)) <= 0)
28     return (n);

29 #ifdef HAVE_MSGHDR_MSG_CONTROL
30     if ((cmptr = CMSG_FIRSTHDR(&msg)) != NULL &&
31         mptr->cmsg_len == CMSG_LEN(sizeof(int))) {
32         if (cmptr->cmsg_level != SOL_SOCKET)
33             err_quit("control level != SOL_SOCKET");
34         if (cmptr->cmsg_type != SCM_RIGHTS)
35             err_quit("control type != SCM_RIGHTS");
36         *recvfd = *((int*)CMSG_DATA(cmptr));
37     } else
38         *recvfd = -1; /* дескриптор не был передан */
39 #else
40     if (msg.msg_accrightslen == sizeof(int))
41         *recvfd = newfd;
42     else
43         *recvfd = -1; /* дескриптор не был передан */
44 #endif

45     return (n);
46 }

```

8-26 Эта функция должна работать с обеими версиями функции `recvmsg`: с элементом `msg_control` и с элементом `msg_accrights`. Наш заголовочный файл `config.h` (см. листинг Г.2) определяет константу `HAVE_MSGHDR_MSG_CONTROL`, если поддерживается версия функции `recvmsg` с `msg_control`.

Проверка выравнивания буфера `msg_control`

10-13 Буфер `msg_control` должен быть выровнен в соответствии со структурой `msghdr`. Просто выделить в памяти массив типа `char` недостаточно. Здесь мы объявляем объединение, состоящее из структуры `cmsghdr` и символьного массива, что гарантирует необходимое выравнивание массива. Возможно и другое решение — вызвать функцию `malloc`, но это потребует освобождения памяти перед завершением функции.

27-45 Вызывается функция `recvmsg`. Если возвращаются вспомогательные данные, их формат будет таким, как показано на рис. 14.4. Мы проверяем, верны ли длина, уровень и тип, затем получаем вновь созданный дескриптор и возвращаем его через указатель вызывающего процесса `recvfd`. Макрос `CMSG_DATA` возвращает указатель на элемент `cmsg_data` объекта вспомогательных данных как указатель на элемент типа `unsigned char`. Мы преобразуем его к указателю на элемент типа `int` и получаем целочисленный дескриптор, на который указывает этот указатель.

Если поддерживается более старый элемент `msg_accrights`, то длина должна быть равна размеру целого числа, а вновь созданный дескриптор возвращается через указатель `recvfd` вызывающего процесса.

В листинге 15.10 показана программа `openfile`. Она получает три аргумента командной строки, которые должны быть переданы, и вызывает обычную функцию `open`.

Листинг 15.10. Программа `openfile`: открытие файла и передача дескриптора обратно

```

//unixdomain/openfile.c

1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int fd;
6     ssize_t n;

```

```

7 if (argc != 4)
8 err_quit("openfile <sockfd#> <filename> <mode>");

9 if ((fd = open(argv[2], atoi(argv[3]))) < 0)
10 exit((errno > 0) ? errno : 255);

11 if ((n = write_fd(atoi(argv[1]), "", 1, fd)) < 0)
12 exit((errno > 0) ? errno : 255);

13 exit(0);
14 }

```

Аргументы командной строки

6-7 Поскольку два из трех аргументов командной строки были превращены в символьные строки функцией `my_open`, они преобразуются обратно в целые числа при помощи функции `atoi`.

Открытие файла

9-10 Файл открывается с помощью функции `open`. Если встречается ошибка, статус завершения этого процесса содержит значение переменной `errno`, соответствующее ошибке функции `open`.

Передача дескриптора обратно

11-12 Дескриптор передается обратно функцией `write_fd`, которую мы покажем в следующем листинге. Затем этот процесс завершается, но ранее в этой главе мы сказали, что отправляющий процесс может закрыть переданный дескриптор (это происходит, когда мы вызываем функцию `exit`), поскольку ядро знает, что дескриптор находится в состоянии передачи («в полете»), и оставляет его открытым для принимающего процесса.

ПРИМЕЧАНИЕ

Статус выхода должен лежать в пределах от 0 до 255. Максимальное значение переменной `errno` — около 150. Альтернативный способ, при котором не требуется, чтобы значение переменной `errno` было меньше 256, заключается в том, чтобы передать обратно указание на ошибку в виде обычных данных при вызове функции `sendmsg`.

В листинге 15.11 показана последняя функция, `write_fd`, вызывающая функцию `sendmsg` для отправки дескриптора (и, возможно, еще каких-либо данных, которые мы не используем) через доменный сокет Unix.

Листинг 15.11. Функция `write_fd`: передача дескриптора при помощи вызова функции `sendmsg`

```

//lib/write_fd.c
1 #include "unp.h"

2 ssize_t
3 write_fd(int fd, void *ptr, size_t nbytes, int sendfd)
4 {
5     struct msghdr msg;
6     struct iovec iov[1];

7 #ifdef HAVE_MSGHDR_MSG_CONTROL

```

```

8 union {
9     struct cmsghdr cm;
10    char control[CMSG_SPACE(sizeof(int))];
11 } control_un;
12 struct cmsghdr *cmptr;

13 msg.msg_control = control_un.control;
14 msg.msg_controllen = sizeof(control_un.control);

15 cmptr = CMSG_FIRSTHDR(&msg);
16 cmptr->cmsg_len = CMSG_LEN(sizeof(int));
17 cmptr->cmsg_level = SOL_SOCKET;
18 cmptr->cmsg_type = SCM_RIGHTS;
19 *((int*)CMSG_DATA(cmptr)) = sendfd;
20 #else
21 msg.msg_accrights = (caddr_t)&sendfd;
22 msg.msg_accrightslen = sizeof(int);
23 #endif

24 msg.msg_name = NULL;
25 msg.msg_namelen = 0;

26 iov[0].iov_base = ptr;
27 iov[0].iov_len = nbytes;
28 msg.msg_iov = iov;
29 msg.msg iovlen = 1;

30 return (sendmsg(fd, &msg, 0));
31 }

```

Как и в случае функции `read_fg`, эта функция обрабатывает либо вспомогательные данные, либо права доступа, которые предшествовали вспомогательным данным в более ранних реализациях. В любом случае инициализируется структура `msghdr` и затем вызывается функция `sendmsg`.

В разделе 28.7 мы приводим пример передачи дескриптора, в котором участвуют неродственные (*unrelated*) процессы, а в разделе 30.9 — пример, где задействованы родственные процессы. В них мы будем использовать функции `read_fd` и `write_fd`, которые только что описали.

15.8. Получение информации об отправителе

На рис. 14.4 мы показали другой тип информации, передаваемой через доменный сокет Unix в виде вспомогательных данных: информацию об отправителе, которая передается с помощью структуры `cmsgcred`, определяемой путем включения заголовочного файла `<sys/socket.h>`. Упаковка и формат данных зависят от операционной системы. Такая возможность появилась только в BSD/OS 2.1. Мы описываем FreeBSD, а прочие варианты Unix во многом подобны ей (проблема обычно состоит в выборе структуры, которую следует использовать для передачи данных). Рассказ об этой возможности мы считаем необходимым, поскольку это важное, хотя и простое дополнение доменных протоколов Unix. Когда клиент и сервер связываются с помощью этих протоколов, серверу часто бывает необходим способ точно узнать, кто является клиентом, чтобы убедиться, что клиент имеет право запрашивать определенный сервис.

```

struct fcred {
    uid_t fc_ruid;           /* действующий идентификатор пользователя */
    gid_t fc_rgid;           /* действующий групповой идентификатор */
    char fc_login[MAXLOGNAME]; /* имя setlogin() */
    uid_t fc_uid;             /* идентификатор пользователя */
    short fc_ngroups;         /* количество групп */
    gid_t fc_groups[NGROUPS]; /* дополнительные групповые идентификаторы */
};

#define fc_gid fc_groups[0] /* групповой идентификатор */

```

Обычно MAXLONGNAME и NGROUPS имеют значение 16. Значение fc_ngroups равно как минимум 1, а первым элементом массива является идентификатор группы.

Эта информация всегда доступна через доменный сокет Unix, хотя отправителю часто приходится принимать дополнительные меры для обеспечения ее отправки вместе с данными, и получателю также приходится выполнять некоторые действия (например, устанавливать параметры сокета). В системе FreeBSD получатель может обойтись вызовом recvmsg с достаточно большим буфером для вспомогательных данных, чтобы туда поместились идентифицирующие данные (листинг 15.12). Однако отправитель обязан включить структуру cmsgcred при отправке данных посредством sendmsg. Хотя включение структуры осуществляется отправителем, заполняется она ядром. Благодаря этому передача идентифицирующих данных через доменный сокет Unix является надежным способом проверки клиента.

Пример

В качестве примера передачи идентифицирующих данных мы изменим наш потоковый доменный сервер Unix, так чтобы он запрашивал идентифицирующие данные клиента. В листинге 15.12 показана новая функция, read_cred, аналогичная функции read, но возвращающая также структуру fcred, содержащую идентифицирующие данные отправителя.

Листинг 15.12. Функция read_cred: чтение и возвращение идентифицирующих данных отправителя

```
//unixdomain/readcred.c
1 #include "unp.h"

2 #define CONTROL_LEN (sizeof(struct cmsghdr) + sizeof(struct cmsgcred))

3 ssize_t
4 read_cred(int fd, void *ptr, size_t nbytes, struct cmsgcred *cmsgcredptr)
5 {
6     struct msghdr msg;
7     struct iovec iov[1];
8     char control[CONTROL_LEN];
9     int n;

10    msg.msg_name = NULL;
11    msg.msg_namelen = 0;
12    iov[0].iov_base = ptr;
13    iov[0].iov_len = nbytes;
14    msg.msg_iov = iov;
15    msg.msg_iovlen = 1;
16    msg.msg_control = control;
17    msg.msg_controllen = sizeof(control);
18    msg.msg_flags = 0;

19    if ((n = recvmsg(fd, &msg, 0)) < 0)
20        return(n);

21    cmsgcredptr->cmcred_ngroups = 0; /* идентифицирующие данные не получены */
22    if (cmsgcredptr && msg.msg_controllen > 0) {
23        struct cmsghdr *cmptr = (struct cmsghdr*)control;

24        if (cmptr->cmsg_len < CONTROL_LEN)
25            err_quit("control length = %d", cmptr->cmsg_len);
26        if (cmptr->cmsg_level != SOL_SOCKET)
27            err_quit("control level != SOL_SOCKET");
28        if (cmptr->cmsg_type != SCM_CREDS)
29            err_quit("control type != SCM_CREDS");
30        memcpy(cmsgcredptr, CMSG_DATA(cmptr), sizeof(struct cmsgcred));
```

```
31 }
32 return(n);
33 }
```

3-4 Первые три аргумента идентичны аргументам функции `read`, а четвертый аргумент — это указатель на структуру `cmmsgcred`, которая будет заполнена.

22-31 Если данные были переданы, проверяются длина, уровень и тип вспомогательных данных, и результирующая структура копируется обратно вызывающему процессу. Если никаких идентифицирующих данных не было передано, мы обнуляем структуру. Поскольку число групп (`cmcred_ngroups`) всегда равно 1 или больше, нулевое значение указывает вызывающему процессу, что ядро не возвратило никаких идентифицирующих данных.

Функция `main` для нашего эхо-сервера (см. листинг 15.3) остается неизменной. В листинге 15.13 показана новая версия функции `str_echo`, полученная путем модификации листинга 5.2. Эта функция вызывается дочерним процессом после того, как родительский процесс принял новое клиентское соединение и вызвал функцию `fork`.

Листинг 15.13. Функция `str_echo`, запрашивающая идентифицирующие данные клиента

```
//unixdomain/strecho.c
1 #include "unp.h"

2 ssize_t read_cred(int, void*, size_t, struct cmmsgcred*);

3 void
4 str_echo(int sockfd)
5 {
6     ssize_t n;
7     int i;
8     char buf[MAXLINE];
9     struct cmmsgcred cred;
10    again:
11    while ((n = read_cred(sockfd, buf, MAXLINE, &cred)) > 0) {
12        if (cred.cmcred_ngroups == 0) {
13            printf("(no credentials returned)\n");
14        } else {
15            printf("PID of sender = %d\n", cred.cmcred_pid);
16            printf("real user ID = %d\n", cred.cmcred_uid);
17            printf("real group ID = %d\n", cred.cmcred_gid);
18            printf("effective user ID = %d\n", cred.cmcred_euid);
19            printf("%d groups:", cred.cmcred_ngroups - 1);
20            for (i = 1; i < cred.cmcred_ngroups; i++)
21                printf(" %d", cred.cmcred_groups[i]);
22            printf("\n");
23        }
24        Writen(sockfd, buf, n);
25    }

26    if (n < 0 && errno == EINTR)
27        goto again;
28    else if (n < 0)
29        err_sys("str_echo: read error");
30 }
```

11-23 Если идентифицирующие данные возвращаются, они выводятся.

24-25 Оставшаяся часть цикла не меняется. Этот код считывает строки от клиента и затем отправляет их обратно клиенту.

Наш клиент, представленный в листинге 15.4, остается практически неизменным. Мы добавляем передачу пустой структуры `cmmsgcred` при вызове `sendmsg`, которая заполняется ядром.

Перед запуском клиента определим свои личные данные командой `id`:

```
freebsd % id
uid=1007(andy) gid=1007(andy) groups=1007(andy), 0(wheel)
```

Если мы запустим сервер в одном окне, а клиент в другом, то для сервера после однократного выполнения клиента получим представленный ниже вывод.

```
freebsd % unixstrserv02
PID of sender = 26881
real user ID = 1007
real group ID = 1007
effective user ID = 1007
2 groups: 1007 0
```

Информация выводится только после отправки клиентом данных серверу. Мы видим, что сведения соответствуют тем, которые были получены командой **id**.

15.9. Резюме

Доменные сокеты Unix являются альтернативой IPC, когда клиент и сервер находятся на одном узле. Преимущество использования доменных сокетов Unix перед некоторой формой IPC состоит в том, что используемый API практически идентичен клиент-серверному сетевому соединению. Преимущество использования доменных сокетов Unix перед TCP, когда клиент и сервер находятся на одном узле, заключается в повышенной производительности доменных сокетов Unix относительно TCP во многих реализациях.

Мы изменили наш эхо-сервер и эхо-клиент TCP и UDP для использования доменных протоколов Unix, и единственным главным отличием оказалась необходимость при помощи функции **bind** связывать полное имя с клиентским сокетом UDP так, чтобы серверу UDP было куда отправлять ответы.

Передача дескрипторов между клиентами и серверами, находящимися на одном узле, — это мощная технология, которая используется при работе с доменными сокетами Unix. Мы показали пример передачи дескриптора от дочернего процесса обратно родительскому процессу в разделе 15.7. В разделе 28.7 мы покажем пример, в котором клиент и сервер не будут родственными, а в разделе 30.9 — другой пример, когда дескриптор передается от родительского процесса дочернему.

Упражнения

1. Что произойдет, если доменный сервер Unix вызовет функцию **unlink** после вызова функции **bind**?
2. Что произойдет, если доменный сервер Unix при завершении не отсоединит с помощью функции **unlink** свое известное полное имя, а клиент будет пытаться с помощью функции **connect** соединиться с сервером через некоторое время после того, как тот завершил работу?

3. Измените листинг 11.5 так, чтобы после того как будет выведен адрес протокола собеседника, вызывалась бы функция **sleep(5)**, а также чтобы вывести число байтов, возвращаемых функцией **read** всякий раз, когда она возвращает положительное значение. Измените листинг 11.8 так, чтобы для каждого байта результата, отправляемого клиенту, вызывалась функция **write**. (Мы обсуждаем подобные изменения в решении упражнения 1.5.) Запустите клиент и сервер на одном узле, используя TCP. Сколько байтов считывает клиент с помощью функции **read**?

Запустите клиент и сервер на одном узле, используя доменный сокет Unix. Изменилось ли что-нибудь? Теперь для сервера вместо функции **write** вызовите функцию **send** и задайте флаг **MSG_EOR** (чтобы выполнить это упражнение, вам нужно использовать Беркли-реализацию). Запустите клиент и сервер на одном узле, используя доменный сокет Unix. Изменилось ли что-нибудь?

4. Напишите программу, определяющую значения, показанные в табл. 4.6. Один из подходов — создать потоковый канал и затем с помощью функции **fork** разветвить родительский и дочерний процессы. Родительский процесс входит в цикл **for**, увеличивая на каждом шаге значение **backlog** от 0 до 14. Каждый раз при прохождении цикла родительский процесс сначала записывает значение **backlog** в потоковый канал. Дочерний процесс читает это значение, создает прослушиваемый сокет, связанный с адресом закольцовки, и присваивает **backlog** считанное значение. Затем дочерний процесс делает запись в потоковый канал просто для того, чтобы сообщить родительскому процессу о своей готовности. Затем родительский процесс пытается установить как можно больше соединений, задав предварительно аргумент функции **alarm** равным 2 с, поскольку при достижении предельного значения **backlog** вызов функции

`connect` заблокируется, и отправляет еще раз сегмент SYN. Дочерний процесс никогда не вызывает функцию `accept`, что позволяет ядру установить в очередь все соединения с родительским процессом. Когда истекает время ожидания родительского процесса (аргумент функции `alarm`, в данном случае 2 с), по счетчику цикла он может определить, какая по счету функция `connect` соответствует предельному значению `backlog`. Затем родительский процесс закрывает свои сокеты и пишет следующее новое значение в потоковый канал для дочернего процесса. Когда дочерний процесс считывает новое значение, он закрывает прежний прослушиваемый сокет и создает новый, заново начиная процедуру.

5. Проверьте, вызывает ли пропуск вызова функции `bind` в листинге 15.6 ошибку сервера.

Глава 16

Неблокируемый ввод-вывод

16.1. Введение

По умолчанию сокеты блокируют выполнение процесса. Это означает, что, когда мы вызываем на сокете функцию, которая не может выполниться немедленно, наш процесс переходит в «спящее» состояние и ждет, когда будет выполнено определенное условие. Мы можем разделить функции сокетов, способные вызвать блокирование, на четыре категории.

1. Операции ввода: функции `read`, `readv`, `recv`, `recvfrom` и `recvmsg`. Если мы вызываем одну из этих функций ввода для блокируемого сокета TCP (а по умолчанию такой сокет является блокируемым) и в приемном буфере сокета отсутствуют данные, то сокет вызывает переход в спящее состояние на то время, пока не придут какие-нибудь данные. Поскольку TCP является протоколом байтового потока, из этого состояния мы выйдем, когда придет «хоть сколько-нибудь» данных: это может быть одиночный байт, а может быть и целый сегмент данных TCP. Если мы хотим ждать до тех пор, пока не будет доступно определенное фиксированное количество данных, мы вызываем нашу функцию `readn` (см. листинг 3.9) или задаем флаг `MSG_WAITALL` (см. табл. 14.1). Поскольку UDP является протоколом дейтаграмм, то если приемный буфер блокируемого сокета UDP пуст, мы переходим в состояние ожидания и находимся в нем до тех пор, пока не придет дейтаграмма UDP.

В случае неблокируемого сокета при невозможности удовлетворить условию операции ввода (как минимум 1 байт данных для сокета TCP или целая дейтаграмма для сокета UDP) возврат происходит немедленно с ошибкой `EWOULDBLOCK`.

2. Операции вывода: функции `write`, `writev`, `send`, `sendto`, и `sendmsg`. В отношении сокета TCP в разделе 2.9 мы сказали, что ядро копирует данные из буфера приложения в буфер отправки сокета. Если для блокируемого сокета недостаточно места в буфере отправки, процесс переходит в состояние ожидания до тех пор, пока место не освободится.

В случае неблокируемого сокета TCP при недостатке места в буфере отправки завершение происходит немедленно с ошибкой `EWOULDBLOCK`. Если в буфере отправки сокета есть место, возвращаемое значение будет представлять количество байтов, которое ядро смогло скопировать в буфер (это называется *частичным копированием* — *short count*).

В разделе 2.9 мы также сказали, что на самом деле буфера отправки UDP не существует. Ядро только копирует данные приложения и перемещает их вниз по стеку, добавляя к данным заголовки UDP и IP. Следовательно, операция вывода на блокируемом сокете UDP (каким он является по умолчанию) никогда не заблокируется.

3. Прием входящих соединений: функция `accept`. Если функция `accept` вызывается для блокируемого сокета и новое соединение недоступно, процесс переводится в состояние ожидания.

Если функция `accept` вызывается для неблокируемого сокета и новое соединение недоступно, возвращается ошибка `EWOULDBLOCK`.

4. Инициирование исходящих соединений: функция `connect` для TCP. (Вспомните, что функция `connect` может использоваться с UDP, но она не вызывает создания «реального» соединения — она лишь заставляет ядро сохранить IP-адрес и номер порта собеседника.) В разделе 2.5 мы показали, что установление соединения TCP включает трехэтапное рукопожатие и что функция `connect` не возвращает управление, пока клиент не получит сегмент ACK или SYN. Это значит, что функция TCP `connect` всегда блокирует вызывающий процесс как минимум на время обращения (RTT) к серверу.

Если функция `connect` вызывается для неблокируемого сокета TCP и соединение не может быть установлено немедленно, инициируется установление соединения (например, отправляется первый пакет трехэтапного рукопожатия TCP), но возвращается ошибка `EINPROGRESS`. Обратите внимание, что эта ошибка отличается от ошибки, возвращаемой в первых трех сценариях. Также отметим, что некоторые соединения могут быть установлены немедленно, когда сервер находится на том же узле, что и клиент, поэтому даже в случае неблокируемого вызова функции `connect` мы должны быть готовы к тому, что она успешно выполнится. Пример неблокируемой функции `connect` мы покажем в разделе 16.3.

ПРИМЕЧАНИЕ

Традиционно System V возвращала для неблокируемой операции ввода-вывода, которую невозможно выполнить, ошибку EAGAIN, в то время как Беркли-реализации возвращали ошибку EWOULDBLOCK. Еще больше дело запутывается тем, что согласно POSIX.1 используется EAGAIN, в то время как в POSIX.1g определено, что используется EWOULDBLOCK. К счастью, большинство систем (включая SVR4 и 4.4BSD) определяют один и тот же код для этих двух ошибок (проверьте свой системный заголовочный файл <sys/errno.h>), поэтому не важно, какой из них использовать. В нашем тексте мы используем ошибку EWOULDBLOCK, как определяется в POSIX.

В разделе 6.2 мы представили различные модели ввода-вывода и сравнили неблокируемый ввод-вывод с другими моделями. В этой главе мы покажем примеры четырех типов операций и разработаем новый тип клиента, аналогичный веб-клиенту, инициирующий одновременно множество соединений TCP при помощи неблокируемой функции connect.

16.2. Неблокируемые чтение и запись: функция str_cli (продолжение)

Мы снова возвращаемся к нашей функции str_cli, которую мы обсуждали в разделах 5.5 и 6.4. Последняя ее версия, задействующая функцию select, продолжает использовать блокируемый ввод-вывод. Например, если в стандартном устройстве ввода имеется некоторая строка, мы читаем ее с помощью функции fgets и затем отправляем серверу с помощью функции writen. Но вызов функции writen может вызвать блокирование процесса, если буфер отправки сокета полон. В то время как мы заблокированы в вызове функции writen, данные могут быть доступны для чтения из приемного буфера сокета. Аналогично, когда строка ввода доступна из сокета, мы можем заблокироваться в последующем вызове функции fputs, если стандартный поток вывода работает медленнее, чем сеть. Наша цель в данном разделе — создать версию этой функции, использующую неблокируемый ввод-вывод. Блокирование будет предотвращено, благодаря чему в это время мы сможем сделать еще что-то полезное.

К сожалению, добавление неблокируемого ввода-вывода значительно усложняет управление буфером функции, поэтому мы будем представлять функцию частями. Мы также заменим вызовы функций из стандартной библиотеки ввода-вывода на обычные read и write. Это даст возможность отказаться от функций стандартной библиотеки ввода-вывода с неблокируемыми дескрипторами, так как их применение может привести к катастрофическим последствиям.

Мы работаем с двумя буферами: буфер to содержит данные, направляющиеся из стандартного потока ввода к серверу, а буфер fr — данные, приходящие от сервера в стандартный поток вывода. На рис. 16.1 представлена организация буфера to и указателей в буфере.

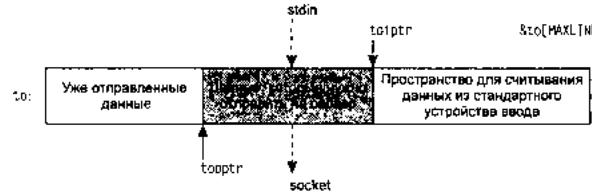


Рис. 16.1. Буфер, содержащий данные из стандартного потока ввода, идущие к сокету

Указатель toiptr указывает на следующий байт, в который данные могут быть считаны из стандартного потока ввода. Указатель tooptr указывает на следующий байт, который должен быть записан в сокет. Число байтов, которое может быть считано из стандартного потока ввода, равно &to[MAXLINE] минус toiptr. Как только значение tooptr достигает toiptr, оба указателя переустанавливаются на начало буфера.

На рис. 16.2 показана соответствующая организация буфера fr. В листинге 16.1^[1] представлена первая часть функции.

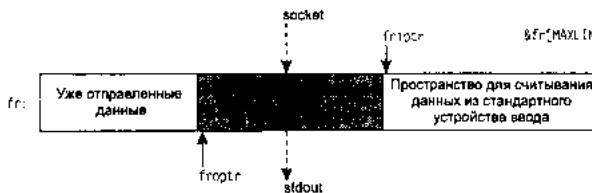


Рис. 16.2. Буфер, содержащий данные из сокета, идущие к стандартному устройству вывода

Листинг 16.1. Функция str_cli: первая часть, инициализация и вызов функции

```
//nonblock/strclintonb.c
1 #include "unp.h"

2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     int maxfdp1, val, stdineof;
6     ssize_t n, nwritten;
7     fd_set rset, wset;
8     char to[MAXLINE], fr[MAXLINE];
9     char *toiptr, *tooptr, *friptr, *froptr;

10    val = Fcntl(sockfd, F_GETFL, 0);
11    Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);

12    val = Fcntl(STDIN_FILENO, F_SETFL, 0);
13    Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);

14    val = Fcntl(STDOUT_FILENO, F_SETFL, 0);
15    Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);

16    toiptr = tooptr = to; /* инициализация указателей буфера */
17    friptr = froptr = fr;
18    stdineof = 0;

19    maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
20    for (;;) {
21        FD_ZERO(&rset);
22        FD_ZERO(&wset);
23        if (stdineof == 0 && toiptr < &to[MAXLINE])
24            FD_SET(STDIN_FILENO, &rset); /* чтение из стандартного потока
25                                         ввода */
26        if (friptr < &fr[MAXLINE])
27            FD_SET(sockfd, &rset); /* чтение из сокета */
28        if (tooptr != toiptr)
29            FD_SET(sockfd, &wset); /* данные для записи в сокет */
30        if (froptr != friptr)
31            FD_SET(STDOUT_FILENO, &wset); /* данные для записи в стандартный
32                                         поток вывода */
31    Select(maxfdp1, &rset, &wset, NULL, NULL);
```

Установка неблокируемых дескрипторов

10-15 Все три дескриптора делаются неблокируемыми при помощи функции fcntl: сокет в направлении к серверу и от сервера, стандартный поток ввода и стандартный поток вывода.

Инициализация указателей буфера

16-19 Инициализируются указатели в двух буферах и вычисляется максимальный дескриптор. Это значение, увеличенное на единицу, будет использоваться в качестве первого аргумента функции select.

Основной цикл: подготовка к вызову функции select

20 Как и в случае первой версии этой функции, показанной в листинге 6.2, основной цикл функции содержит вызов функции `select`, за которой следуют отдельные проверки различных интересующих нас условий.

Подготовка интересующих нас дескрипторов

21-30 Оба набора дескрипторов обнуляются и затем в каждом наборе включается не более двух битов. Если мы еще не прочитали конец файла из стандартного потока ввода и есть место как минимум для 1 байта данных в буфере `to`, то в наборе флагов чтения включается бит, соответствующий стандартному потоку ввода. Если есть место как минимум для 1 байта данных в буфере `fr`, то в наборе флагов чтения включается бит, соответствующий сокету. Если есть данные для записи в сокет в буфере `to`, то в наборе флагов записи включается бит, соответствующий сокету. Наконец если в буфере `fr` есть данные для отправки в стандартный поток вывода, то в наборе флагов записи включается бит, соответствующий этому стандартному потоку.

Вызов функции `select`

31 Вызывается функция `select`, ожидающая, когда одно из четырех условий станет истинным. Для этой функции мы не задаем тайм-аута.

Следующая часть нашей функции показана в листинге 16.2. Этот код содержит первые две проверки (из четырех возможных), выполняемые после завершения функции `select`.

Листинг 16.2. Функция `str_cli`: вторая часть, чтение из стандартного потока ввода или сокета

```
//nonblock/strcllinonb.c
32  if (FD_ISSET(STDIN_FILENO, &rset)) {
33      if ((n = read(STDIN_FILENO, toiptr, &to[MAXLINE] - toiptr)) < 0) {
34          if (errno != EWOULDBLOCK)
35              err_sys("read error on stdin");
36      } else if (n == 0) {
37          fprintf(stderr, "%s: EOF on stdin\n", gf_time());
38          stdineof = 1; /* c stdin все сделано */
39          if (toiptr == toiptr)
40              Shutdown(sockfd, SHUT_WR); /* отсылаем FIN */
41
42      } else {
43          fprintf(stderr, "%s: read %d bytes from stdin\n", gf_time(),
44                  n);
45          toiptr += n; /* только что полученное из функции read число */
46          FD_SET(sockfd, &wset); /* включаем бит в наборе чтения */
47      }
48  if (FD_ISSET(sockfd, &rset)) {
49      if ((n = read(sockfd, friptr, &fr[MAXLINE] - friptr)) < 0) {
50          if (errno != EWOULDBLOCK)
51              err_sys("read error on socket");
52      } else if (n == 0) {
53          fprintf(stderr, "%s: EOF on socket\n", gf_time());
54          if (stdineof)
55              return; /* нормальное завершение */
56          else
57              err_quit("str_cli: server terminated prematurely");
58  } else {
59      fprintf(stderr, "%s: read %d bytes from socket\n",
```

```
60     gf_time(), n);
61     friptr += n; /* только что полученное из функции read число */
62     FD_SET(STDOUT_FILENO, &wset); /* включаем бит в наборе
63     чтения */
64 }
```

Чтение из стандартного потока ввода с помощью функции read

32-33 Если стандартный поток ввода готов для чтения, мы вызываем функцию `read`. Третий ее аргумент — это количество свободного места в буфере `to`.

Обработка ошибки

34-35 Если происходит ошибка `EWOULDBLOCK`, мы ничего не предпринимаем. Обычно эта ситуация — когда функция `select` сообщает нам о том, что дескриптор готов для чтения, а функция `read` возвращает ошибку `EWOULDBLOCK` — не должна возникать, но тем не менее мы ее обрабатываем.

Возвращение конца файла функцией read

36-40 Если функция `read` возвращает нуль, мы закончили со стандартным потоком ввода. Флаг `stdineof` установлен. Если в буфере `to` больше нет данных для отправки (`tooptr` равно `toiptr`), функция `shutdown` отправляет серверу сегмент FIN. Если в буфере `to` еще есть данные для отправки, сегмент FIN не может быть отправлен до тех пор, пока содержимое буфера не будет записано в сокет.

ПРИМЕЧАНИЕ

Мы выводим в стандартный поток сообщений об ошибках строку, отмечающую конец файла, вместе с текущим временем. Мы покажем, как мы используем этот вывод, после описания функции. Аналогичные вызовы функции `fprinf` выполняются неоднократно в процессе выполнения нашей функции.

Возвращение данных функцией read

41-45 Когда функция `read` возвращает данные, мы увеличиваем на единицу `toiptr`. Мы также включаем бит, соответствующий сокету, в наборе флагов записи, чтобы позже при проверке этого бита в цикле он был включен и тем самым инициировалась бы попытка записи в сокет с помощью функции `write`.

ПРИМЕЧАНИЕ

Это одно из непростых конструктивных решений, которые приходится принимать при написании кода. У нас есть несколько альтернатив. Вместо установки бита в наборе записи мы можем ничего не делать, и в этом случае функция `select` будет проверять возможность записи в сокет, когда она будет вызвана в следующий раз. Но это требует дополнительного прохода цикла и вызова функции `select`, когда мы уже знаем, что у нас есть данные для записи в сокет. Другой вариант — дублировать код, который записывает в сокет, но это кажется расточительным, к тому же это возможный источник ошибки (в случае, если в этой части дублируемого кода есть ошибка и мы обнаруживаем и устранием ее только в одном месте). Наконец, мы можем создать функцию, записывающую в сокет, и вызывать эту функцию вместо дублирования кода, но эта функция должна использовать три локальные переменные совместно с функцией `str_cli`, что может

привести к необходимости сделать эти переменные глобальными. Выбор, сделанный в нашем случае, — это результат субъективного мнения автора относительно того, какой из описанных трех вариантов предпочтительнее.

Чтение из сокета с помощью функции read

48-64 Эти строки кода аналогичны выражению `if`, только что описанному для случая, когда стандартный поток ввода готов для чтения. Если функция `read` возвращает ошибку `EWOULDBLOCK`, ничего не происходит. Если мы встречаем признак конца файла, присланный сервером, это нормально, когда мы уже получили признак конца файла в стандартном потоке ввода. Но иначе это будет ошибкой, означающей преждевременное завершение работы сервера (*Server terminated prematurely*). Если функция `read` возвращает некоторые данные, `friptr` увеличивается на единицу и в наборе флагов записи включается бит для стандартного потока вывода, с тем чтобы попытаться записать туда данные в следующей части функции.

В листинге 16.3 показана последняя часть нашей функции.

Листинг 16.3. Функция `str_cli`: третья часть, запись в стандартный поток вывода или сокет

```
//nonblock/strclintonb.c
65  if (FD_ISSET(STDOUT_FILENO, &wset) && ((n = friptr - froptr) > 0)) {
66      if ((nwritten = write(STDOUT_FILENO, froptr, n)) < 0) {
67          if (errno != EWOULDBLOCK)
68              err_sys("write error to stdout");
69      } else {
70          fprintf(stderr, "%s: wrote %d bytes to stdout\n",
71                  gf_time(), nwritten);
72          froptr += nwritten; /* только что полученное из функции write
                           число */
73          if (froptr == friptr)
74              froptr = friptr - fr; /* назад к началу буфера */
75      }
76  }

77  if (FD_ISSET(sockfd, &wset) && ((n - toiptr - tooptr) > 0)) {
78      if ((nwritten = write(sockfd, tooptr, n)) < 0) {
79          if (errno != EWOULDBLOCK)
80              err_sys("write error to socket");

81      } else {
82          fprintf(stderr, "%s: wrote %d bytes to socket\n",
83                  gf_time(), nwritten);
84          tooptr += nwritten; /* только что полученное из функции write
                           число */
85          if (tooptr == toiptr) {
86              toiptr - tooptr = to; /* назад к началу буфера */
87              if (stdineof)
88                  Shutdown(sockfd, SHUT_WR); /* посыпаем FIN */
89          }
90      }
91  }
92 }
93 }
```

Запись в стандартный поток вывода с помощью функции write

65-68 Если есть возможность записи в стандартный поток вывода и число байтов для записи больше нуля, вызывается функция `write`. Если возвращается ошибка `EWOULDBLOCK`, ничего не происходит. Обратите внимание, что это условие возможно, поскольку код в конце предыдущей части функции включает бит в наборе флагов записи для стандартного потока вывода, когда не известно, успешно выполнилась функция `write` или нет.

Успешное выполнение функции `write`

68-74 Если функция `write` выполняется успешно, `froptr` увеличивается на число записанных байтов. Если указатель вывода стал равен указателю ввода, оба указателя переустанавливаются на начало буфера.

Запись в сокет с помощью функции `write`

76-90 Эта часть кода аналогична коду, только что описанному для записи в стандартный поток вывода. Единственное отличие состоит в том, что когда указатель вывода доходит до указателя ввода, не только оба указателя переустанавливаются в начало буфера, но и появляется возможность отправить серверу сегмент FIN.

Теперь мы проверим работу этой функции и операций неблокируемого ввода-вывода. В листинге 16.4 показана наша функция `gf_time`, вызываемая из функции `str_cli`.

Листинг 16.4. Функция `gf_time`: возвращение указателя на строку времени

```
//lib/gf_time.c
1 #include "unp.h"
2 #include <time.h>

3 char*
4 gf_time(void)
5 {
6     struct timeval tv;
7     static char str[30];
8     char *ptr;

9     if (gettimeofday(&tv, NULL) < 0)
10        err_sys("gettimeofday error");

11    ptr = ctime(&tv.tv_sec);
12    strcpy(str, &ptr[11]);
13    /* Fri Sep 13 00:00:00 1986\n\0 */
14    /* 0123456789012345678901234 5 */
15    sprintf(str + 8, sizeof(str) - 8, ".%06ld", tv.tv_usec);
16    return (str);
17 }
```

Эта функция возвращает строку, содержащую текущее время с точностью до микросекунд, в таком формате:

12:34:56.123456

Здесь специально используется тот же формат, что и для отметок времени, которые выводятся программой `tcpdump`. Обратите внимание, что все вызовы функции `fprintf` в нашей функции `str_cli` записывают данные в стандартный поток сообщений об ошибках, позволяя нам отделить данные стандартного потока вывода (строки, отраженные сервером) от наших диагностических данных. Затем мы можем запустить наш клиент и функцию `tcpdump`, получить эти диагностические данные вместе с результатом функции `tcpdump` и отсортировать вместе два вида выходных данных в порядке их получения. Это позволит нам увидеть, что происходит в нашей программе, и соотнести это с действиями TCP.

Например, сначала мы запускаем функцию `tcpdump` на нашем узле `solaris`, собирая только сегменты TCP, идущие к порту 7 или от него (эхо-сервер), и сохраняем выходные данные в файле, который называется `tcpd`:

```
solaris % tcpdump -w tcpd tcp and port 7
```

Затем мы запускаем клиент TCP на этом узле и указываем сервер на узле linux:

```
solaris % tcpccli02 192.168.1.10 < 2000.lines > out 2> diag
```

Стандартный поток ввода — это файл 2000.lines, тот же файл, что мы использовали для листинга 6.2.

Стандартный поток вывода перенаправляется в файл out, а стандартный поток сообщений об ошибках — в файл diag. По завершении мы запускаем:

```
solaris % diff 2000.lines out
```

чтобы убедиться, что отраженные строки идентичны введенным строкам. Наконец, мы прекращаем выполнение функции tcpdump нажатием соответствующей клавиши терминала, после чего выводим записи функции tcpdump, сортируя их по времени получения вместе с данными диагностики, полученными от клиента. В листинге 16.5 показана первая часть этого результата.

Листинг 16.5. Отсортированный вывод функции tcpdump и данных диагностики

```
solaris % tcpdump -r tcpd -N | sort diag -
10:18:34.486392 solaris.33621 > linux.echo: S 1802738644:1802738644(0) win 8760 <mss 1460>
10:18:34.488278 linux.echo > solaris.33621: S 3212986316 3212986316(0) ack 1802738645 win
8760 <mss 1460>
10:18:34.488490 solaris.33621 > linux.echo: . ack 1 win 8760
10:18:34.491482: read 4096 bytes from stdin
10:18:34.518663 solaris.33621 > linux.echo: P 1461(1460) ack 1 win 8760
10:18:34.519016: wrote 4096 bytes to socket
10:18:34.528529 linux echo > solaris.33621: P 1:1461(1460) ack 1461 win 8760
10:18:34.528785 solaris.33621 > linux.echo: . 1461 2921(1460) ack 1461 win 8760
10:18:34.528900 solaris.33621 > linux echo: P 2921:4097(1176) ack 1461 win 8760
10:18:34.528958 solaris 33621 > linux.echo: ack 1461 win 8760
10:18:34.536193 linux echo: > solaris.33621: . 1461:2921(1460) ack 4097 win 8760
10:18:34.536697 linux.echo: > solaris.33621: P 2921.3509(588) ack 4097 win 8760
10:18:34.544636: read 4096 bytes from stdin 10:18:34.568505: read 3508 bytes from socket
10:18:34.580373 solaris 33621 > linux.echo: . ack 3509 win 8760
10:18:34.582244 linux.echo > solaris.33621: P 3509.4097(588) ack 4097 win 8760
10:18:34.593354: wrote 3508 bytes to stdout
10:18:34.617272 solaris.33621 > linux.echo: P 4097.5557(1460) ack 4097 win 8760
10:18:34.617610 solaris 33621 > linux.echo: P 5557:7017(1460) ack 4097 win 8760
10:18:34.617908 solaris.33621 > linux.echo: P 7017.8193(1176) ack 4097 win 8760
10:18:34.618062: wrote 4096 bytes to socket
10:18:34.623310 linux.echo > solaris.33621: . ack 8193 win 8760
10:18:34.626129 linux.echo > solaris.33621: . 4097.5557(1460) ack 8193 win 8760
10:18:34.626339 solaris.33621 > linux.echo: . ack 5557 win 8760
10:18:34.626611 linux.echo > solaris.33621: P 5557:6145(588) ack 8193 win 8760
10:18:34.628396 linux.echo > solaris.33621: 6145:7605(1460) ack 8193 win 8760
10:18:34.643524: read 4096 bytes from stdin 10:18:34.667305: read 2636 bytes from socket
10:18:34.670324 solaris.33621 > linux echo: . ack 7605 win 8760
10:18:34.672221 linux.echo > solaris.33621: P 7605.8193(588) ack 8193 win 8760
10:18:34.691039: wrote 2636 bytes to stdout
```

Мы удалили записи (DF) из сегментов, отправленных Solaris, означающие, что устанавливается бит DF (он используется для определения величины транспортной MTU).

Используя этот вывод, мы можем нарисовать временную диаграмму происходящих событий (рис. 16.3). На этой диаграмме представлены события в различные моменты времени, причем ориентация диаграммы такова, что более поздние события расположены ниже на странице.

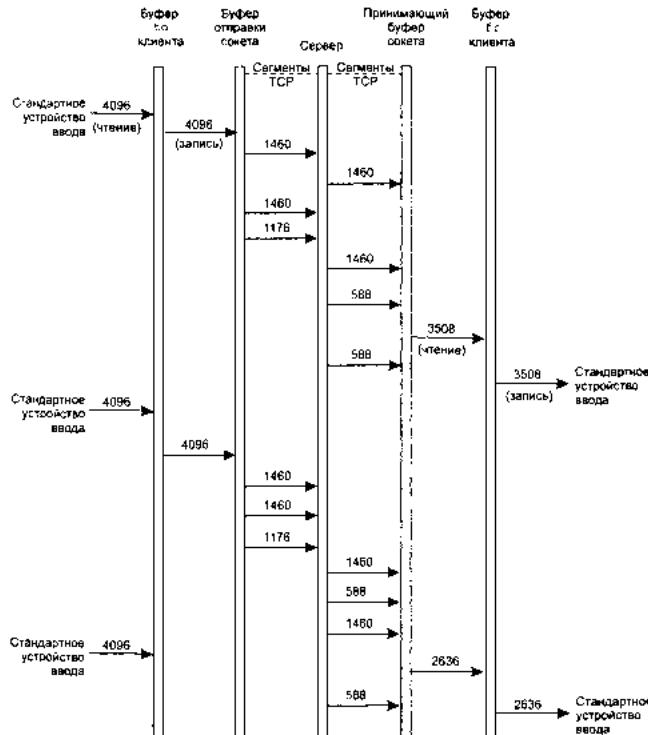


Рис. 16.3. Временная диаграмма событий для примера неблокируемого ввода

На этом рисунке мы не показываем сегменты ACK. Также помните, что если программа выводит сообщение `wrote N bytes to stdout` (записано N байт в стандартное устройство вывода), это означает, что завершилась функция `write`, возможно, заставившая TCP отправить один или более сегментов данных.

По этому рисунку мы можем проследить динамику обмена между клиентом и сервером. Использование неблокируемого ввода-вывода позволяет программе использовать преимущество этой динамики, считывая или записывая данные, когда операция ввода или вывода может иметь место. Ядро сообщает нам, когда может произойти операция ввода-вывода, при помощи функции `select`.

Мы можем рассчитать время выполнения нашей неблокируемой версии, используя тот же файл из 2000 строк и тот же сервер (с периодом RTT, равным 175 мс), что и в разделе 6.7. Теперь время оказалось равным 6,9 с по сравнению с 12,3 с в версии из раздела 6.7. Следовательно, неблокируемый ввод-вывод сокращает общее время выполнения этого примера, в котором файл отправляется серверу.

Более простая версия функции str_cli

Неблокируемая версия функции `str_cli`, которую мы только что показали, нетривиальна: около 135 строк кода по сравнению с 40 строками версии, использующей функцию `select` с блокируемым вводом-выводом (см. листинг 6.2), и 20 строками начальной версии, работающей в режиме остановки и ожидания (см. листинг 5.4). Мы знаем, что эффект от удлинения кода в два раза, с 20 до 40 строк оправдывает затраченные усилия, поскольку в пакетном режиме скорость возрастает почти в 30 раз, а применение функции `select` с блокирующими дескрипторами осуществляется не слишком сложно. Но будут ли оправданы затраченные усилия при написании приложения, использующего неблокируемый ввод-вывод, с учетом усложнения итогового кода? Нет, ответим мы. Если нам необходимо использовать неблокируемый ввод-вывод, обычно бывает проще разделить приложение либо на процессы (при помощи функции `fork`), либо на потоки (см. главу 26).

В листинге 16.6 показана еще одна версия нашей функции `str_cli`, разделяемая на два процесса при помощи функции `fork`.

Эта функция сразу же вызывает функцию `fork` для разделения на родительский и дочерний процессы. Дочерний процесс копирует строки от сервера в стандартный поток вывода, а родительский процесс — из стандартного потока ввода серверу, как показано на рис. 16.4.

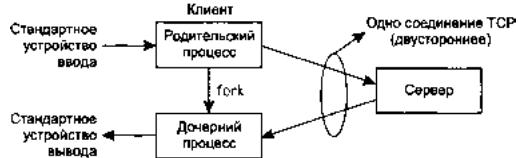


Рис. 16.4. Функция str_cli, использующая два процесса

Мы показываем, что соединения TCP являются двусторонними и что родительский и дочерний процессы совместно используют один и тот же дескриптор сокета: родительский процесс записывает в сокет, а дочерний процесс читает из сокета. Есть только один сокет, один буфер приема сокета и один буфер отправки, но на этот сокет ссылаются два дескриптора: один в родительском процессе и один в дочернем.

Листинг 16.6. Версия функции str_cli, использующая функцию fork

```

//nonblock/strclifork.c
1 #include "unp.h"

2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     pid_t pid;
6     char sendline[MAXLINE], recvline[MAXLINE];

7     if ((pid = Fork()) == 0) { /* дочерний процесс: сервер -> stdout */
8         while (Readline(sockfd, recvline, MAXLINE) > 0)
9             Fputs(recvline, stdout);

10        kill(getppid(), SIGTERM); /* в случае, если родительский процесс
11                               все еще выполняется */
12    }
13    /* родитель: stdin -> сервер */
14    while (Fgets(sendline, MAXLINE, fp) != NULL)
15        Writen(sockfd, sendline, strlen(sendline));
16    Shutdown(sockfd, SHUT_WR); /* конец файла на stdin, посыпаем FIN */
17    pause();
18    return;
19 }

```

Нам нужно снова вспомнить о последовательности завершения соединения. Обычное завершение происходит, когда в стандартном потоке ввода встречается конец файла. Родительский процесс считывает конец файла и вызывает функцию shutdown для отправки сегмента FIN. (Родительский процесс не может вызвать функцию close, см. упражнение 16.1.) Но когда это происходит, дочерний процесс должен продолжать копировать от сервера в стандартный поток вывода, пока он не получит признак конца файла на сокете.

Также возможно, что процесс сервера завершится преждевременно (см. раздел 5.12), и если это происходит, дочерний процесс считывает признак конца файла на сокете. В таком случае дочерний процесс должен сообщить родительскому, что нужно прекратить копирование из стандартного потока ввода в сокет (см. упражнение 16.2). В листинге 16.6 дочерний процесс отправляет родительскому процессу сигнал SIGTERM, в случае, если родительский процесс еще выполняется (см. упражнение 16.3). Другим способом обработки этой ситуации было бы завершение дочернего процесса, и если родительский процесс все еще выполнялся бы к этому моменту, он получил бы сигнал SIGCHLD.

Родительский процесс вызывает функцию pause, когда заканчивает копирование, что переводит его в состояние ожидания того момента, когда будет получен сигнал. Даже если родительский процесс не перехватывает никаких сигналов, он все равно переходит в состояние ожидания до получения сигнала SIGTERM от дочернего процесса. По умолчанию действие этого сигнала — завершение процесса, что вполне устраивает нас в этом примере. Родительский процесс ждет завершения дочернего процесса, чтобы измерить точное время для этой версии функции str_cli. Обычно дочерний процесс завершается после

родительского, но поскольку мы измеряем время, используя команду оболочки `time`, измерение заканчивается, когда завершается родительский процесс.

Отметим простоту этой версии по сравнению с неблокируемым вводом-выводом, представленным ранее в этом разделе. Наша неблокируемая версия управляла четырьмя различными потоками ввода-вывода одновременно, и поскольку все четыре были неблокируемыми, нам пришлось иметь дело с частичным чтением и частичной записью для всех четырех потоков. Но в версии с функцией `fork` каждый процесс обрабатывает только два потока ввода-вывода, копируя из одного в другой. В применении неблокируемого ввода-вывода не возникает необходимости, поскольку если нет данных для чтения из потока ввода, то и в соответствующий поток вывода записывать нечего.

Сравнение времени выполнения различных версий функции `str_cli`

Итак, мы продемонстрировали четыре различных версии функции `str_cli`. Для каждой версии мы покажем время, которое потребовалось для ее выполнения, в том числе и для версии, использующей программные потоки (см. листинг 26.1). В каждом случае было скопировано 2000 строк от клиента Solaris к серверу с периодом RTT, равным 175 мс:

- 354,0 с, режим остановки и ожидания (см. листинг 5.4);
- 12,3 с, функция `select` и блокируемый ввод-вывод (см. листинг 6.2);
- 6,9 с, неблокируемый ввод-вывод (см. листинг 16.1);
- 8,7 с, функция `fork` (см. листинг 16.6);
- 8,5 с, версия с потоками (см. листинг 26.1).

Наша версия с неблокируемым вводом-выводом почти вдвое быстрее версии, использующей блокируемый ввод-вывод с функцией `select`. Наша простая версия с применением функции `fork` медленнее версии с неблокируемым вводом-выводом. Тем не менее, учитывая сложность кода неблокируемого ввода-вывода по сравнению с кодом функции `fork`, мы рекомендуем более простой подход.

16.3. Неблокируемая функция `connect`

Когда сокет TCP устанавливается как неблокируемый, а затем вызывается функция `connect`, она немедленно возвращает ошибку `EINPROGRESS`, однако трехэтапное рукопожатие TCP продолжается. Далее мы с помощью функции `select` проверяем, успешно или нет завершилось установление соединения. Неблокируемая функция `connect` находит применение в трех случаях:

1. Трехэтапное рукопожатие может наложиться на какой-либо другой процесс. Для выполнения функции `connect` требуется один период обращения RTT (см. раздел 2.5), и это может занять от нескольких миллисекунд в локальной сети до сотен миллисекунд или нескольких секунд в глобальной сети. Это время мы можем провести с пользой, выполняя какой-либо другой процесс.

2. Мы можем установить множество соединений одновременно, используя эту технологию. Этот способ уже стал популярен в применении к веб-браузерам, и такой пример мы приводим в разделе 16.5.

3. Поскольку мы ждем завершения установления соединения с помощью функции `select`, мы можем задать предел времени для функции `select`, что позволит нам сократить тайм-аут для функции `connect`. Во многих реализациях тайм-аут функции `connect` лежит в пределах от 75 с до нескольких минут. Бывают случаи, когда приложению нужен более короткий тайм-аут, и одним из решений может стать использование неблокируемой функции `connect`. В разделе 14.2 рассматриваются другие способы помещения тайм-аута в операции с сокетами.

Как бы просто ни выглядела неблокируемая функция `connect`, есть ряд моментов, которые следует учитывать.

- Даже если сокет является неблокируемым, то когда сервер, с которым мы соединяемся, находится на том же узле, обычно установление соединения происходит немедленно при вызове функции `connect`.
- В Беркли-реализациях (а также POSIX) имеются два следующих правила, относящихся к функции `select` и неблокируемой функции `connect`: во-первых, когда соединение устанавливается успешно, дескриптор становится готовым для записи [128, с. 531], и во-вторых, когда при установлении соединения встречается ошибка, дескриптор становится готовым как для чтения, так и для записи [128, с. 530].

ПРИМЕЧАНИЕ

Эти два правила в отношении функции select выпадают из общего ряда наших правил из раздела 6.3 относительно условий, при которых дескриптор становится готовым для чтения или записи. В сокет TCP можно записывать, если достаточно места в буфере отправки (что всегда будет выполнено в случае присоединенного сокета, поскольку мы еще ничего не записали в сокет) и сокет является присоединенным (что выполняется, только когда завершено трехэтапное рукопожатие). При наличии ошибки, ожидающей обработки, появляется возможность читать из сокета и записывать в сокет.

С неблокируемыми функциями connect связано множество проблем переносимости, которые мы отметим в последующих примерах.

16.4. Неблокируемая функция connect: клиент времени и даты

В листинге 16.7 показана наша функция connect_nonb, вызывающая неблокируемую функцию connect. Мы заменяем вызов функции connect, имеющейся в листинге 1.1, следующим фрагментом кода:

```
if (connect_nonb(sockfd, (SA*)&servaddr, sizeof(servaddr), 0) < 0)
err_sys("connect error");
```

Первые три аргумента являются обычными аргументами функции connect, а четвертый аргумент — это число секунд, в течение которых мы ждем завершения установления соединения. Нулевое значение подразумевает отсутствие тайм-аута для функции select; следовательно, для установления соединения TCP ядро будет использовать свой обычный тайм-аут.

Листинг 16.7. Неблокируемая функция connect

```
//lib/connect_nonb.c
1 #include "unp.h"

2 int
3 connect_nonb(int sockfd, const SA *saptr, socklen_t salen, int nsec)
4 {
5     int flags, n, error;
6     socklen_t len;
7     fd_set rset, wset;
8     struct timeval tval;

9     flags = Fcntl(sockfd, F_GETFL, 0);
10    Fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);

11    error = 0;
12    if ((n = connect(sockfd, saptr, salen)) < 0)
13        if (errno != EINPROGRESS)
14            return (-1);

15    /* Пока соединение устанавливается, мы можем заняться чем-то другим */
16    if (n == 0)
17        goto done; /* функция connect завершилась немедленно */

18    FD_ZERO(&rset);
19    FDSET(sockfd, &rset);
20    wset = rset;
21    tval.tv_sec = nsec;
22    tval.tv_usec = 0;

23    if ((n = Select(sockfd + 1, &rset, &wset, NULL,
24    nsec ? &tval : NULL)) == 0) {
25        close(sockfd); /* тайм-аут */
26        errno = ETIMEDOUT;
```

```

27     return (-1);
28 }
29 if (FD_ISSET(sockfd, &rset) || FD_ISSET(sockfd, &wset)) {
30     len = sizeof(error);
31     if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &len) < 0)
32         return (-1); /* в Solaris ошибка, ожидающая обработки */
33 } else
34     err_quit("select error: sockfd not set");

35 done:
36 Fcntl(sockfd, F_SETFL, flags); /* восстанавливаем флаги, задающие статус файла */

37 if (error) {
38     close(sockfd); /* на всякий случай */
39     errno = error;
40     return (-1);
41 }
42 return (0);
43 }

```

Задание неблокируемого сокета

9-10 Мы вызываем функцию `fcntl`, которая делает сокет неблокируемым.

11-14 Мы вызываем неблокируемую функцию `connect`. Ошибка, которую мы ожидаем (`EINPROGRESS`), указывает на то, что установление соединения началось, но еще не завершилось [128, с. 466]. Любая другая ошибка возвращается вызывающему процессу.

Выполнение других процессов во время установления соединения

15 На этом этапе мы можем делать все, что захотим, ожидая завершения установления соединения.

Проверка немедленного завершения

16-17 Если неблокируемая функция `connect` вернула нуль, установление соединения завершилось. Как мы сказали, это может произойти, когда сервер находится на том же узле, что и клиент.

Вызов функции select

18-24 Мы вызываем функцию `select` и ждем, когда сокет будет готов либо для чтения, либо для записи. Мы обнуляем `rset`, включаем бит, соответствующий `sockfd` в этом наборе дескрипторов и затем копируем `rset` в `wset`. Это присваивание, возможно, является структурным присваиванием, поскольку обычно наборы дескрипторов представляются как структуры. Далее мы инициализируем структуру `timeval` и затем вызываем функцию `select`. Если вызывающий процесс задает четвертый аргумент нулевым (что соответствует использованию тайм-аута по умолчанию), следует задать в качестве последнего аргумента функции `select` пустой указатель, а не структуру `timeval` с нулевым значением (означающим, что мы не ждем вообще).

Обработка тайм-аутов

25-28 Если функция `select` возвращает нуль, это означает, что время таймера истекло, и мы возвращаем вызывающему процессу ошибку `ETIMEDOUT`. Мы также закрываем сокет, чтобы трехэтапное

рукопожатие не продолжалось.

Проверка возможности чтения или записи

29-34 Если дескриптор готов для чтения или для записи, мы вызываем функцию `getsockopt`, чтобы получить ошибку сокета (`SO_ERROR`), ожидающую обработки. Если соединение завершилось успешно, это значение будет нулевым. Если при установлении соединения произошла ошибка, это значение является значением переменной `errno`, соответствующей ошибке соединения (например, `ECONNREFUSED`, `ETIMEDOUT` и т.д.). Мы также сталкиваемся с нашей первой проблемой переносимости. Если происходит ошибка, Беркли-реализации функции `getsockopt` возвращают нуль, а ошибка, ожидающая обработки, возвращается в нашей переменной `error`. Но в системе Solaris сама функция `getsockopt` возвращает -1, а переменная `errno` при этом принимает значение, соответствующее ошибке, ожидающей обработки. В нашем коде обрабатываются оба сценария.

Восстановление возможности блокировки сокета и завершение

36-42 Мы восстанавливаем флаги, задающие статус файла, и возвращаемся. Если наша переменная `egtpo` имеет ненулевое значение в результате выполнения функции `getsockopt`, это значение хранится в переменной `errno`, и функция возвращает -1.

Как мы сказали ранее, проблемы переносимости для функции `connect` связаны с различными реализациями сокетов и отключения блокировки. Во-первых, возможно, что установление соединения завершится и придут данные для собеседника до того, как будет вызвана функция `select`. В этом случае сокет будет готов для чтения и для записи при успешном выполнении функции, как и при неудачном установленном соединении. В нашем коде, показанном в листинге 16.7, этот сценарий обрабатывается при помощи вызова функции `getsockopt` и проверки на наличие ошибки, ожидающей обработки, для сокета.

Во-вторых, проблема в том, как определить, успешно завершилось установление соединения или нет, если мы не можем считать возможность записи единственным указанием на успешное установление соединения. В Usenet предлагалось множество решений этой проблемы, которые заменяют наш вызов функции `getsockopt` в листинге 16.7:

1. Вызвать функцию `getpeername` вместо функции `getsockopt`. Если этот вызов окажется неудачным и возвратится ошибка `ENOTCONN`, значит, соединение не было установлено, и чтобы получить ошибку, ожидающую обработки, следует вызвать для сокета функцию `getsockopt` с `SO_ERROR`.

2. Вызвать функцию `read` с нулевым значением аргумента `length`. Если выполнение функции `read` окажется неудачным, функция `connect` выполнилась неудачно, и переменная `egtpo` из функции `read` при этом указывает на причину неудачной попытки установления соединения. Если соединение успешно установлено, функция `read` возвращает нуль.

3. Снова вызвать функцию `connect`. Этот вызов окажется неудачным, и если ошибка — `EISCONN`, сокет уже присоединен, а значит, первое соединение завершилось успешно.

К сожалению, неблокируемая функция `connect` — это одна из самых сложных областей сетевого программирования с точки зрения переносимости. Будьте готовы к проблемам совместимости, особенно с более ранними реализациями. Более простой технологией является создание потока (см. главу 26) для обработки соединения.

Прерванная функция `connect`

Что происходит, если наш вызов функции `connect` на обычном блокируемом сокете прерывается, скажем, перехваченным сигналом, прежде чем завершится трехэтапное рукопожатие TCP? Если предположить, что функция `connect` не перезапускается автоматически, то она возвращает ошибку `EINTR`. Но мы не можем снова вызвать функцию `connect`, чтобы добиться завершения установления соединения. Это приведет к ошибке `EADDRINUSE`.

Все, что требуется сделать в этом сценарии, — вызвать функцию `select`, так, как мы делали в этом разделе для неблокируемой функции `connect`. Тогда функция `select` завершится, если соединение успешно

устанавливается (делая сокет доступным для записи) или если попытка соединения неудачна (сокет становится доступен для чтения и для записи).

16.5. Неблокируемая функция connect: веб-клиент

Первое практическое использование неблокируемой функции `connect` относится к веб-клиенту Netscape (см. раздел 13.4 [112]). Клиент устанавливает соединение HTTP с веб-сервером и попадает на домашнюю страницу. На этой странице часто присутствуют ссылки на другие веб-страницы. Вместо того чтобы получать последовательно по одной странице за один раз, клиент может получить сразу несколько страниц, используя неблокируемые функции `connect`. На рис. 16.5 показан пример установления множества параллельных соединений. Сценарий, изображенный слева, показывает все три соединения, устанавливаемые одно за другим. Мы считаем, что первое соединение занимает 10 единиц времени, второе — 15, а третье — 4, что в сумме дает 29 единиц времени.



Рис. 16.5. Установление множества параллельных соединений

В центре рисунка показан сценарий, при котором мы выполняем два параллельных соединения. В момент времени 0 запускаются первые два соединения, а когда первое из них устанавливается, мы запускаем третье. Общее время сократилось почти вдвое и равно 15, а не 29 единицам времени, но учтите, что это идеальный случай. Если параллельные соединения совместно используют общий канал связи (допустим, клиент использует модем для соединения с Интернетом), то каждое из этих соединений конкурирует с другими за обладание ограниченными ресурсами этого канала связи, и время установления каждого соединения может возрасти. Например, время 10 может дойти до 15, 15 — до 20, а время 4 может превратиться в 6. Тем не менее общее время будет равно 21 единице, то есть все равно меньше, чем в последовательном сценарии.

В третьем сценарии мы выполняем три параллельных соединения и снова считаем, что эти три соединения не мешают друг другу (идеальный случай). Но общее время при этом такое же (15 единиц), как и во втором сценарии.

При работе с веб-клиентами первое соединение устанавливается само по себе, за ним следуют соединения по ссылкам, обнаруженным в данных от первого соединения. Мы показываем это на рис. 16.6.

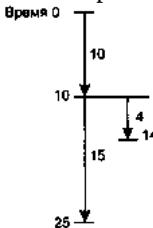


Рис. 16.6. Установление первого соединения, а затем множества параллельных соединений

Для дальнейшей оптимизации клиент может начать обработку данных, возвращаемых по первому соединению, до того, как установление первого соединения завершится, и инициировать дополнительные соединения, как только ему станет известно, что они нужны.

Поскольку мы выполняем несколько неблокируемых функций `connect` одновременно, мы не можем использовать нашу функцию `connect_nonblock`, показанную в листинге 16.7, так как она не завершается, пока соединение не установлено. Вместо этого мы отслеживаем множество соединений самостоятельно.

Наша программа считывает около 20 строк с веб-сервера. Мы задаем в качестве аргументов командной строки максимальное число параллельных соединений, имя узла сервера, а затем каждое из имен файлов, получаемых с сервера. Типичное выполнение нашей программы выглядит так:

```
solaris % web % www.foobar.com / image1.gif image2.gif \
image3.gif image4.gif image5.gif \
image6.gif image7.gif
```

Аргументы командной строки задают три одновременных соединения, имя узла сервера, имя файла домашней страницы (/ обозначает корневой каталог сервера) и семь файлов, которые затем нужно прочитать (в нашем примере это файлы с изображениями в формате GIF). Обычно на эти семь файлов имеются ссылки с домашней страницы, и чтобы получить их имена, веб-клиент читает домашнюю страницу и обрабатывает код HTML. Чтобы не усложнять этот пример разбором кода HTML, мы просто задаем имена файлов в командной строке.

Это большой пример, поэтому мы будем показывать его частями. В листинге 16.8 представлен наш заголовочный файл web.h, который включен во все файлы.

Листинг 16.8. Заголовок web.h

```
//nonblock/web.h
1 #include "unp.h"

2 #define MAXFILES 20
3 #define SERV "80" /* номер порта или имя службы */

4 struct file {
5     char *f_name; /* имя файла */
6     char *f_host; /* имя узла или адрес IPv4/IPv6 */
7     int f_fd;      /* дескриптор */
8     int f_flags;   /* F_xxx определены ниже */
9 } file[MAXFILES];

10 #define F_CONNECTING 1 /* connect() в процессе выполнения */
11 #define F_READING 2 /* соединение установлено; происходит считывание */
12 #define F_DONE 4 /* все сделано */

13 #define GET_CMD "GET %s HTTP/1.0\r\n\r\n"

14 /* глобальные переменные */
15 int nconn, nfiles, nlefttoconn, nlefttoread, maxfd;
16 fd_set rset, wset;

17 /* прототипы функций */
18 void home_page(const char*, const char*);
19 void start_connect (struct file*);
20 void write_get_cmd(struct file*);
```

Задание структуры file

2-13 Программа считывает некоторое количество (не более MAXFILES) файлов с веб-сервера. Структура file содержит информацию о каждом файле: его имя (копируется из аргумента командной строки), имя узла или IP-адрес сервера, с которого читается файл, дескриптор сокета, используемый для этого файла, и набор флагов, которые указывают, что мы делаем с этим файлом (устанавливаем соединение для получения файла или считываем файл).

Определение глобальных переменных и прототипов функций

14-20 Мы определяем глобальные переменные и прототипы для наших функций, которые мы вскоре опишем.

Листинг 16.9. Первая часть программы одновременного выполнения функций connect: глобальные переменные и начало функции main

```
//nonblock/web.c
1 #include "web.h"

2 int
3 main(int argc, char **argv)
4 {
5     int i, fd, n, maxnconn, flags, error;
6     char buf[MAXLINE];
7     fd_set rs, ws;

8     if (argc < 5)
9         err_quit("usage: web <#conns> <hostname> <homepage> <file1> ...");
10    maxnconn = atoi(argv[1]);

11    nfiles = min(argc - 4, MAXFILES);
12    for (i = 0; i < nfiles; i++) {
13        file[i].f_name = argv[i + 4];
14        file[i].f_host = argv[2];
15        file[i].f_flags = 0;
16    }
17    printf("nfiles = %d\n", nfiles);

18    home_page(argv[2], argv[3]);

19    FD_ZERO(&rset);
20    FD_ZERO(&wset);
21    maxfd = -1;
22    nlefttoread = nlefttoconn = nfiles;
23    nconn = 0;
```

Обработка аргументов командной строки

11-17 Структуры `file` заполняются соответствующей информацией из аргументов командной строки.

Чтение домашней страницы

18 Функция `home_page`, которую мы показываем в следующем листинге, создает соединение TCP, посыпает команду серверу и затем читает домашнюю страницу. Это первое соединение, которое выполняется самостоятельно, до того как мы начнем устанавливать параллельные соединения.

Инициализация глобальных переменных

19-23 Инициализируются два набора дескрипторов, по одному для чтения и для записи. `maxfd` — это максимальный дескриптор для функции `select` (который мы инициализируем значением `-1`, поскольку дескрипторы неотрицательны), `nlefttoread` — число файлов, которые осталось прочитать (когда это значение становится нулевым, чтение заканчивается), `nlefttoconn` — это количество файлов, для которых пока еще требуется соединение TCP, а `nconn` — это число соединений, открытых в настоящий момент (оно никогда не может превышать первый аргумент командной строки).

В листинге 16.10 показана функция `home_page`, вызываемая один раз, когда начинается выполнение функции `main`.

Листинг 16.10. Функция `home_page`

```
//nonblock/home_page.c
1 #include "web.h"

2 void
3 home_page(const char *host, const char *fname)
4 {
5     int fd, n;
6     char line[MAXLINE];

7     fd = Tcp_connect(host, SERV); /* блокируемая функция connect() */

8     n = sprintf(line, sizeof(line), GET_CMD, fname);
9     Writen(fd, line, n);

10    for (;;) {
11        if ((n = Read(fd, line, MAXLINE)) == 0)
12            break; /* сервер закрыл соединение */

13        printf("read %d bytes of home page\n", n);
14        /* обрабатываем полученные данные */
15    }
16    printf("end-of-file on home page\n");
17    Close(fd);
18 }
```

Установление соединения с сервером

7 Наша функция `tcp_connect` устанавливает соединение с сервером.

Отправка команды HTTP серверу, чтение ответа

8-17 Запускается команда HTTP `GET` для домашней страницы (часто обозначается символом `/`). Читается ответ (с ответом мы в данном случае ничего не делаем), и соединение закрывается.

Следующая функция, `start_connect`, показанная в листинге 16.11, инициирует вызов неблокируемой функции `connect`.

Листинг 16.11. Иницирование неблокируемой функции `connect`

```
//nonblock/start_connect.c
1 #include "web.h"

2 void
3 start_connect(struct file *fptr)
4 {
5     int fd, flags, n;
6     struct addrinfo *ai;

7     ai = Host_serv(fptr->f_host, SERV, 0, SOCK_STREAM);

8     fd = Socket(ai->ai_family; ai->ai_socktype, ai->ai_protocol);
9     fptr->f_fd = fd;
10    printf("start_connect for %s, fd %d\n", fptr->f_name, fd);
```

```

11 /* отключаем блокирование сокета */
12 flags = Fcntl(fd, F_GETFL, 0);
13 Fcntl(fd, F_SETFL, flags | O_NONBLOCK);

14 /* инициируем неблокируемое соединение с сервером */
15 if ((n = connected, ai->ai_addr, ai->ai_addrlen)) < 0) {
16     if (errno != EINPROGRESS)
17         err_sys("nonblocking connect error");
18     fptr->f_flags = F_CONNECTING;
19     FD_SET(fd, &rset); /* включаем дескриптор сокета в наборе чтения
и записи */
20     FD_SET(fd, &wset);
21     if (fd > maxfd)
22         maxfd = fd;

23 } else if (n >= 0) /* соединение уже установлено */
24     write_get_cmd(fptr); /* отправляем команду GET серверу */
25 }

```

Создание сокета, отключение блокировки сокета

7-13 Мы вызываем нашу функцию `host_serv` для поиска и преобразования имени узла и имени службы. Она возвращает указатель на массив структур `addrinfo`. Мы используем только первую структуру. Создается сокет TCP, и он становится неблокируемым.

Вызов неблокируемой функции connect

14-22 Вызывается неблокируемая функция `connect`, и флагу файла присваивается значение `F_CONNECTING`. Включается дескриптор сокета и в наборе чтения, и в наборе записи, поскольку функция `select` будет ожидать любого из этих условий как указания на то, что установление соединения завершилось. При необходимости мы также обновляем значение `maxfd`.

Обработка завершения установления соединения

23-24 Если функция `connect` успешно завершается, значит, соединение уже установлено, и функция `write_get_cmd` (она показана в следующем листинге) посыпает команду серверу.

Мы делаем сокет неблокируемым для функции `connect`, но никогда не переустанавливаем его в блокируемый режим, заданный по умолчанию. Это нормально, поскольку мы записываем в сокет только небольшое количество данных (команда GET следующей функции) и считаем, что эти данные занимают значительно меньше места, чем имеется в буфере отправки сокета. Даже если из-за установленного флага отсутствия блокировки при вызове функции `write` происходит частичное копирование, наша функция `writen` обрабатывает эту ситуацию. Если оставить сокет неблокируемым, это не повлияет на последующее выполнение функций `read`, потому что мы всегда вызываем функцию `select` для определения того момента, когда сокет станет готов для чтения.

В листинге 16.12 показана функция `write_get_cmd`, посыпающая серверу команду HTTP GET.

Листинг 16.12. Отправка команды HTTP GET серверу

```

//nonblock/write_get_cmd.c
1 #include "web.h"

2 void
3 write_get_cmd(struct file *fptr)
4 {
5     int n;

```

```

6 char line[MAXLINE];

7 n = snprintf(line, sizeof(line), GET_CMD, fptr->f_name);
8 Writen(fptr->f_fd, line, n);
9 printf("wrote %d bytes for %s\n", n, fptr->f_name);

10 fptr->f_flags = F_READING; /* сброс F_CONNECTING */
11 FD_SET(fptr->f_fd, &rset); /* прочитаем ответ сервера */
12 if (fptr->f_fd > maxfd)
13 maxfd = fptr->f_fd;
14 }

```

Создание команды и ее отправка

7-9 Команда создается и пишется в сокет.

Установка флагов

10-13 Устанавливается флаг `F_READING`, при этом также сбрасывается флаг `F_CONNECTING` (если он установлен). Это указывает основному циклу, что данный дескриптор готов для ввода. Также включается дескриптор в наборе чтения, и при необходимости обновляется значение `maxfd`.

Теперь мы возвращаемся в функцию `main`, показанную в листинге 16.13, начиная с того места, где закончили в листинге 16.9. Это основной цикл программы: пока имеется ненулевое количество файлов для обработки (значение `nlefttoread` больше нуля), устанавливается, если это возможно, другое соединение и затем вызывается функция `select` для всех активных дескрипторов, обрабатывающая как завершение неблокируемых соединений, так и прием данных.

Можем ли мы инициировать другое соединение?

24-35 Если мы не дошли до заданного предела одновременных соединений и есть дополнительные соединения, которые нужно установить, мы ищем еще не обработанный файл (на него указывает нулевое значение `f_flags`) и вызываем функцию `start_connect` для инициализации соединения. Число активных соединений увеличивается на единицу (`nconn`), а число соединений, которые нужно установить, на единицу уменьшается (`nlefttoconn`).

Функция `select`: ожидание событий

36-37 Функция `select` ожидает готовности сокета либо для чтения, либо для записи. Дескрипторы, для которых в настоящий момент происходит установление соединения (неблокируемая функция `connect` находится в процессе выполнения), будут включены в обоих наборах, в то время как дескрипторы с завершенным соединением, ожидающие данных от сервера, будут включены только в наборе чтения.

Листинг 16.13. Основной цикл функции `main`

```

//nonblock/web.c
24 while (nlefttoread > 0) {
25     while (nconn < maxnconn && nlefttoconn > 0) {
26         /* find a file to read */
27         for (i = 0; i < nfiles; i++)
28             if (file[i].f_flags == 0)
29                 break;
30         if (i == nfiles)
31             err_quit("nlefttoconn = %d but nothing found", nlefttoconn);
32         start_connect(&file[i]);

```

```

33     nconn++;
34     nlefttoconn--;
35 }

36 rs = rset;
37 ws = wset;
38 n = Select(maxfd + 1, &rs, &ws, NULL, NULL);
39 for (i = 0; i < nfiles; i++) {
40     flags = file[i].f_flags;
41     if (flags == 0 || flags & F_DONE)
42         continue;
43     fd = file[i].f_fd;
44     if (flags & F_CONNECTING &&
45         (FD_ISSET(fd, &rs) || FD_ISSET(fd, &ws))) {
46         n = sizeof(error);
47         if (getsockopt(fd, SOL_SOCKET, S0_ERROR, &error, &n) < 0 ||
48             error != 0) {
49             err_ret("nonblocking connect failed
50                 for %s", file[i].f_name);
51         }
52         /* соединение установлено */
53         printf("connection established for %s\n", file[i].f_name);
54         FD_CLR(fd, &wset); /* отключаем запись в этот сокет */
55         write_get_cmd(&file[i]); /* передаем команду GET */

56     } else if (flags & F_READING && FD_ISSET(fd, &rs)) {
57         if ((n = Read(fd, buf, sizeof(buf))) == 0) {
58             printf("end-of-file on %s\n", file[i].f_name);
59             Close(fd);
60             file[i].f_flags = F_DONE; /* сбрасывает флаг F_READING */
61             FD_CLR(fd, &rset);
62             nconn--;
63             nlefttoread--;
64         } else {
65             printf("read %d bytes from %s\n", n, file[i].f_name);
66         }
67     }
68 }
69 }
70 exit(0);
71 }

```

Обработка всех готовых дескрипторов

39-55 Теперь мы анализируем каждый элемент массива структур `file`, чтобы определить, какие дескрипторы нужно обрабатывать. Если установлен флаг `F_CONNECTING` и дескриптор включен либо в наборе чтения, либо в наборе записи, неблокируемая функция `connect` завершается. Как мы говорили при описании листинга 16.7, мы вызываем функцию `getsockopt`, чтобы получить ожидающую обработки ошибку для сокета. Если значение ошибки равно нулю, соединение успешно завершилось. В этом случае мы отключаем дескриптор в наборе флагов записи и вызываем функцию `write_get_cmd` для отправки запроса HTTP серверу.

Проверка, есть ли у дескриптора данные

56-67 Если установлен флаг `F_READING` и дескриптор готов для чтения, мы вызываем функцию `read`. Если соединение было закрыто другим концом, мы закрываем сокет, устанавливаем флаг `F_DONE`, выключаем дескриптор в наборе чтения и уменьшаем число активных соединений и общее число соединений, требующих обработки.

Есть два способа оптимизации, которые мы не используем в этом примере (чтобы не усложнять его еще больше). Во-первых, мы можем завершить цикл `for` в листинге 16.13, когда мы обработали число дескрипторов, которые, по сообщению функции `select`, были готовы. Во-вторых, мы могли, где это возможно, уменьшить значение `maxfd`, чтобы функция `select` не проверяла биты дескрипторов, которые уже сброшены. Поскольку число дескрипторов, используемых в этом коде, в любой момент времени, вероятно, меньше 10, а не порядка тысяч, вряд ли какая-либо из этих оптимизаций стоит дополнительных усложнений.

Эффективность одновременных соединений

Каков выигрыш в эффективности при установлении множества одновременных соединений? В табл. 16.1 показано время, необходимое для выполнения определенной задачи, которая состоит в том, чтобы получить от веб-сервера домашнюю страницу и девять картинок. Время обращения RTT для данного соединения с сервером равно приблизительно 150 мс. Размер домашней страницы — 4017 байт, а средний размер девяти файлов с изображениями составил 1621 байт. Размер сегмента TCP равен 512 байт. Для сравнения мы также представляем в этой таблице значения для многопоточной версии данной программы, которую мы создаем в разделе 26.9.

Таблица 16.1. Время выполнения задания для разного количества одновременных соединений в разных версиях программы

Количество одновременных соединений	Затраченное время (в секундах), отсутствие блокирования	Затраченное время (в секундах), использование потоков
1	6,0	6,3
2	4,1	4,2
3	3,0	3,1
4	2,8	3,0
5	2,5	2,7
6	2,4	2,5
7	2,3	2,3
8	2,2	2,3
9	2,0	2,3

ПРИМЕЧАНИЕ

Мы показали пример использования одновременных соединений, поскольку он служит хорошей иллюстрацией применения неблокируемого ввода-вывода, а также потому, что в данном случае эффективность применения одновременных соединений может быть измерена. Это свойство также используется в популярном приложении — веб-браузере Netscape. В этой технологии могут появиться некоторые «подводные камни», если сеть перегружена. В главе 21 [111] подробно описываются алгоритмы TCP, называемые алгоритмами медленного старта (slow start) и предотвращения перегрузки сети (congestion avoidance). Когда от клиента к серверу устанавливается множество соединений, то взаимодействие между соединениями на уровне TCP отсутствует. То есть если на одном из соединений происходит потеря пакета, другие соединения с тем же сервером не получают соответствующего уведомления, и вполне возможно, что другие соединения вскоре также столкнутся с потерей пакетов, пока не замедлятся. По этим дополнительным соединениям будет продолжаться отправка слишком большого количества пакетов в уже перегруженную сеть. Эта технология также увеличивает нагрузку на сервер.

Максимальное увеличение эффективности происходит при трех одновременных соединениях (время уменьшается вдвое), а при четырех и более одновременных соединениях прирост производительности значительно меньше.

16.6. Неблокируемая функция accept

Как было сказано в главе 6, функция `select` сообщает, что прослушиваемый сокет готов для чтения, когда установленное соединение готово к обработке функцией `accept`. Следовательно, если мы используем функцию `select` для определения готовности входящих соединений, то нам не нужно делать прослушиваемый сокет неблокируемым, потому что когда функция `select` сообщает нам, что соединение установлено, функция `accept` обычно не является блокируемой.

К сожалению, существует определенная проблема, связанная со временем, способная запутать нас [34]. Чтобы увидеть эту проблему, изменим код нашего эхо-клиента TCP (см. листинг 5.3) таким образом, чтобы после установления соединения серверу отсыпался сегмент RST. В листинге 16.14 представлена новая версия.

Листинг 16.14. Эхо-клиент TCP, устанавливающий соединение и посылающий серверу сегмент RST

```
//nonblock/tcpcli03.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;
6     struct linger ling;
7     struct sockaddr_in servaddr;

8     if (argc != 2)
9         err_quit("usage: tcpcli <IPaddress>");
10    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

11    bzero(&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_port = htons(SERV_PORT);
14    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

15    Connect(sockfd, (SA*)&servaddr, sizeof(servaddr));

16    ling.l_onoff = 1; /* для отправки сегмента RST при закрытии соединения */
17    ling.l_linger = 0;
18    Setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &ling, sizeof(ling));
19    Close(sockfd);

20    exit(0);
21 }
```

Установка параметра сокета SOLINGER

16-19 Как только соединение устанавливается, мы задаем параметр сокета `SO_LINGER`, устанавливая флаг `l_onoff` в единицу и обнуляя время `l_linger`. Как утверждалось в разделе 7.5, это вызывает отправку RST на сокете TCP при закрытии соединения. Затем с помощью функции `close` мы закрываем сокет.

Потом мы изменяем наш сервер TCP, приведенный в листингах 6.3 и 6.4, с тем чтобы после сообщения функции `select` о готовности прослушиваемого сокета для чтения, но перед вызовом функции `accept` наступала пауза. В следующем коде, взятом из начала листинга 6.4, две добавленные строки помечены знаком +.

```
if (FD_ISSET(listenfd, &rset)) { /* новое соединение */
```

```

+ printf("listening socket readable\n");
+ sleep(5);
cliлен = sizeof(cliaddr);
connfd = Accept(listenfd, (SA*)&cliaddr, &cliлен);

```

Здесь мы имитируем занятый сервер, который не может вызвать функцию accept сразу же, как только функция select сообщит, что прослушиваемый сокет готов для чтения. Обычно подобное замедление со стороны сервера не вызывает проблем (на самом деле именно для этих ситуаций предусмотрена очередь полностью установленных соединений). Но поскольку после установления соединения от клиента прибыл сегмент RST, у нас возникает проблема.

В разделе 5.11 мы отмечали, что когда клиент разрывает соединение до того, как сервер вызывает функцию accept, в Беркли-реализациях прерванное соединение не возвращается серверу, в то время как другие реализации должны возвращать ошибку ECONNABORTED, но часто вместо нее возвращают ошибку EPROTO. Рассмотрим Беркли-реализацию.

- Клиент устанавливает соединение и затем прерывает его, как показано в листинге 16.14.
- Функция select сообщает процессу сервера, что дескриптор готов для чтения, но у сервера вызов функции accept занимает некоторое, хотя и непродолжительное, время.
- После того, как сервер получил сообщение от функции select, и прежде, чем была вызвана функция accept, прибыл сегмент RST от клиента.
- Установленное соединение удаляется из очереди, и мы предполагаем, что не существует никаких других установленных соединений.
- Сервер вызывает функцию accept, но поскольку установленных соединений нет, он оказывается заблокирован.

Сервер останется блокированным в вызове функции accept до тех пор, пока какой-нибудь другой клиент не установит с ним соединение. Но если сервер аналогичен показанному в листинге 6.4, в это время он заблокирован в вызове функции accept и не может обрабатывать никакие другие готовые дескрипторы.

ПРИМЕЧАНИЕ

Проблема в некоторой степени аналогична проблеме, называемой атакой типа «отказ в обслуживании», описанной в разделе 6.8. Однако в данном случае сервер выходит из состояния блокировки, как только другой клиент установит соединение.

Чтобы решить эту проблему, нужно соблюдать два следующих правила:

1. Всегда делать прослушиваемый сокет неблокируемым, если мы используем функцию select для определения того, готово ли соединение к обработке функцией accept.
2. Игнорировать следующие ошибки, возникающие при повторном вызове функции accept: EWOULDBLOCK (для Беркли-реализаций, когда клиент разрывает соединение), ECONNABORTED (для реализаций POSIX, когда клиент разрывает соединение), EPROTO (для реализаций SVR4, когда клиент разрывает соединение) и EINTR (если перехватываются сигналы).

16.7. Резюме

В примере неблокируемого чтения и записи в разделе 16.2 использовался наш клиент str_cli, который мы изменили для применения неблокируемого ввода-вывода на соединении TCP с сервером. Функция select обычно используется с неблокируемым вводом-выводом для определения того момента, когда дескриптор станет готов для чтения или записи. Эта версия нашего клиента является самой быстродействующей из всех показанных версий, хотя требует нетривиального изменения кода. Затем мы показали, что проще разделить процесс клиента на две части при помощи функции fork. Мы используем ту же технологию при создании потоков в листинге 26.1.

Неблокируемая функция connect позволяет нам во время трехэтапного рукопожатия TCP выполнять другие задачи вместо блокирования в вызове функции connect. К сожалению, с этими функциями также связана проблема совместимости, так как различные реализации по-разному указывают, успешно ли установлено соединение или произошла ошибка. Мы использовали неблокируемые соединения для создания нового клиента, аналогичного веб-клиенту, открывавшему одновременно множество соединений TCP для уменьшения затрат времени при получении нескольких файлов от сервера. Подобное

иницирование множества соединений может сократить временные затраты, но также является «недружественным по отношению к сети», поскольку не позволяет воспользоваться алгоритмом TCP, предназначенным для предотвращения перегрузки (congestion avoidance).

Упражнения

1. Обсуждая листинг 16.6, мы отметили, что родительский процесс должен вызвать функцию `shutdown`, а не функцию `close`. Почему?

2. Что произойдет в листинге 16.6, если процесс сервера завершится преждевременно и дочерний процесс получит признак конца файла, но не уведомит об этом родительский процесс?

3. Что произойдет в листинге 16.6, если родительский процесс непредвиденно завершится до завершения дочернего процесса, и дочерний процесс затем считает конец файла на сокете?

4. Что произойдет в листинге 16.7, если мы удалим следующие две строки:

```
if (n == 0)  
    goto done; /* функция connect завершилась немедленно */
```

5. В разделе 16.3 мы сказали, что возможна ситуация, когда данные для сокета придут раньше, чем завершится функция `connect`. Когда это может случиться?

Глава 17

Операции функции ioctl

17.1. Введение

Функция `ioctl` традиционно являлась системным интерфейсом, используемым для всего, что не входило в какую-либо другую четко определенную категорию. POSIX постепенно избавляется от функции `ioctl`, создавая заменяющие ее функции-обертки и стандартизируя их функциональность. Например, доступ к интерфейсу терминала Unix традиционно осуществлялся с помощью функции `ioctl`, но в POSIX были созданы 12 новых функций для терминалов: `tcgetattr` для получения атрибутов терминала, `tcflush` для опустошения буферов ввода или вывода, и т.д. Аналогичным образом POSIX заменяет одну сетевую функцию `ioctl`: новая функция `socketmark` (см. раздел 24.3) заменяет команду `SIOCATMARK` `ioctl`. Тем не менее прочие сетевые команды `ioctl` остаются не стандартизованными и могут использоваться, например, для получения информации об интерфейсе и обращения к таблице маршрутизации и кэшу ARP (Address Resolution Protocol — протокол разрешения адресов).

В этой главе представлен обзор команд функции `ioctl`, имеющих отношение к сетевому программированию, многие из которых зависят от реализации. Кроме того, некоторые реализации, включая системы, происходящие от 4.4BSD и Solaris 2.6, используют сокеты домена `AF_ROUTE` (маршрутизирующие сокеты) для выполнения многих из этих операций. Маршрутизирующие сокеты мы рассматриваем в главе 18.

Обычно сетевые программы (как правило, серверы) используют функцию `ioctl` для получения информации обо всех интерфейсах узла при запуске программы, с тем чтобы узнать адрес интерфейса, выяснить, поддерживает ли интерфейс широковещательную передачу, многоадресную передачу и т.д. Для возврата этой информации мы разработали нашу собственную функцию. В этой главе мы представляем ее реализацию с применением функции `ioctl`, а в главе 18 — другую реализацию, использующую маршрутизирующие сокеты.

17.2. Функция ioctl

Эта функция работает с открытым файлом, дескриптор которого передается через аргумент `fd`.

```
#include <unistd.h>
```

```
int ioctl(int fd, int request, ... /* void *arg */ );
```

Возвращает: 0 в случае успешного выполнения, -1 в случае ошибки

Третий аргумент всегда является указателем, но тип указателя зависит от аргумента `request`.

ПРИМЕЧАНИЕ

В 4.4BSD второй аргумент имеет тип `unsigned long` вместо `int`, но это не вызывает проблем, поскольку в заголовочных файлах определены константы, используемые для данного аргумента. Пока прототип функции подключен к программе, система будет обеспечивать правильную типизацию.

Некоторые реализации определяют третий аргумент как неопределенный указатель (`void*`), а не так, как он определен в ANSI C.

Не существует единого стандарта заголовочного файла, определяющего прототип функции для `ioctl`, поскольку он не стандартизован в POSIX. Многие системы определяют этот прототип в файле `<unistd.h>`, как это показываем мы, но традиционные системы BSD определяют его в заголовочном файле `<sys/ioctl.h>`.

Мы можем разделить аргументы `request`, имеющие отношение к сети, на шесть категорий:

- операции с сокетами;
- операции с файлами;
- операции с интерфейсами;

- операции с кэшем ARP;
- операции с таблицей маршрутизации;
- операции с потоками (см. главу 31).

Помимо того, что, как показывает табл. 7.9, некоторые операции `ioctl` перекрывают часть операций `fcntl` (например, установка неблокируемого сокета), существуют также некоторые операции, которые с помощью функции `ioctl` можно задать более чем одним способом (например, смена групповой принадлежности сокета).

В табл. 17.1 перечислены аргументы `request` вместе с типами данных, на которые должен указывать адрес `arg`. В последующих разделах эти вызовы рассматриваются более подробно.

Таблица 17.1. Обзор сетевых вызовов `ioctl`

Категория	<code>request</code>	Описание	Тип данных
Сокет	SIOCATMARK	Находится ли указатель чтения сокета на отметке внеполосных данных	int
	SIOCSPGRP	Установка идентификатора процесса или идентификатора группы процессов для сокета	int
	SIOCGPGRP	Получение идентификатора процесса или идентификатора группы процессов для сокета	int
	FIONBIO	Установка/сброс флага отсутствия блокировки	int
Файл	FIOASYNC	Установка/сброс флага асинхронного ввода-вывода	int
	FIONREAD	Получение количества байтов в приемном буфере	int
	FIOSETOWN	Установка идентификатора процесса или идентификатора группы процессов для файла	int
	FIOGETOWN	Получение идентификатора процесса или идентификатора группы процессов для файла	int
Интерфейс	SIOCGIFCONF	Получение списка всех интерфейсов	struct ifconf
	SIOCSIFADDR	Установка адреса интерфейса	struct ifreq
	SIOCGIFADDR	Получение адреса интерфейса	struct ifreq
	SIOCSIFFLAGS	Установка флагов интерфейса	struct ifreq
	SIOCGIFFLAGS	Получение флагов интерфейса	struct ifreq
	SIOCSIFDSTADDR	Установка адреса типа «точка-точка»	struct ifreq
	SIOCGIFDSTADDR	Получение адреса типа «точка-точка»	struct ifreq
	SIOCGIFBRDADDR	Получение широковещательного адреса	struct ifreq
	SIOCSIFBRDADDR	Установка широковещательного адреса	struct ifreq
	SIOCGIFNETMASK	Получение маски подсети	struct ifreq
	SIOCSIFNETMASK	Установка маски подсети	struct ifreq
	SIOCGIFMETRIC	Получение метрики интерфейса	struct ifreq
	SIOCSIFMETRIC	Установка метрики интерфейса	struct ifreq

	SIOCxxx	(Множество вариантов в зависимости от реализации)	
	SIOCSARP	Создание/модификация элемента ARP	struct arpreq
ARP	SIOCGARP	Получение элемента ARP	struct arpreq
	SIOCDARP	Удаление элемента ARP	struct arpreq
	SIOCADDRT	Добавление маршрута	struct rtentry
	SIOCDELRT	Удаление маршрута	struct rtentry
Маршрутизация	I_xxx	(См. раздел 31.5)	
Потоки			

17.3. Операции с сокетами

Существует три типа вызова, или запроса (в зависимости от значения аргумента `request`) функции `ioctl`, предназначенные специально для сокетов [128, с. 551–553]. Все они требуют, чтобы третий аргумент функции `ioctl` был указателем на целое число.

- **SIOCATMARK.** Возвращает указатель на ненулевое значение в качестве третьего аргумента (его тип, как только что было сказано, — указатель на целое число), если указатель чтения сокета в настоящий момент находится на отметке внеполосных данных (out-of-band mark), или указатель на нулевое значение, если указатель чтения сокета не находится на этой отметке. Более подробно внеполосные данные (out-of-band data) рассматриваются в главе 24. POSIX заменяет этот вызов функцией `socketmark`, и мы рассматриваем реализацию этой новой функции с использованием функции `ioctl` в разделе 24.3.

- **SIOCGRP.** Возвращает в качестве третьего аргумента указатель на целое число — идентификатор процесса или группы процессов, которым будут посыпаться сигналы `SIGIO` или `SIGURG` по окончании выполнения асинхронной операции или при появлении срочных данных. Этот вызов идентичен вызову `F_GETOWN` функции `fcntl`, и в табл. 7.9 мы отмечали, что POSIX стандартизирует функцию `fcntl`.

- **SIOCSGRP.** Задает идентификатор процесса или группы процессов для отсылки им сигналов `SIGIO` или `SIGURG` как целое число, на которое указывает третий аргумент. Этот вызов идентичен вызову `F_SETOWN` функции `fcntl`, и в табл. 7.9 мы отмечали, что POSIX стандартизирует функцию `fcntl`.

17.4. Операции с файлами

Следующая группа вызовов начинается с `FIO` и может применяться к определенным типам файлов в дополнение к сокетам. Мы рассматриваем только вызовы, применимые к сокетам [128, с. 553].

Следующие пять вызовов требуют, чтобы третий аргумент функции `ioctl` указывал на целое число.

- **FIONBIO.** Флаг отключения блокировки при выполнении операций ввода-вывода сбрасывается или устанавливается в зависимости от третьего аргумента функции `ioctl`. Если этот аргумент является пустым указателем, то флаг сбрасывается (блокировка разрешена). Если же третий аргумент является указателем на единицу, то включается неблокируемый ввод-вывод. Этот вызов обладает тем же действием, что и команда `F_SETFL` функции `fcntl`, которая позволяет установить или сбросить флаг `O_NONBLOCK`, задающий статус файла.

- **FIOASYNC.** Флаг, управляющий получением сигналов асинхронного ввода-вывода (`SIGIO`), устанавливается или сбрасывается для сокета в зависимости от того, является ли третий аргумент функции `ioctl` пустым указателем. Этот флаг имеет то же действие, что и флаг статуса файла `O_ASYNC`, который можно установить и сбросить с помощью команды `F_SETFL` функции `ioctl`.

- **FIONREAD.** Возвращает число байтов, в настоящий момент находящихся в приемном буфере сокета, как целое число, на которое указывает третий аргумент функции `ioctl`. Это свойство работает также для файлов, каналов и терминалов. Более подробно об этом вызове мы рассказывали в разделе 14.7.

- **FIOSETOWN.** Эквивалент `SIOCSGRP` для сокета.

- **FIOGETOWN.** Эквивалент `SIOCGPGRP` для сокета.

17.5. Конфигурация интерфейса

Один из шагов, выполняемых многими программами, работающими с сетевыми интерфейсами системы, — это получение от ядра списка всех интерфейсов, сконфигурированных в системе. Это делается с помощью вызова SIOCGIFCONF, использующего структуру ifconf, которая, в свою очередь, использует структуру ifreq. Обе эти структуры показаны в листинге 17.1^[1].

Листинг 17.1. Структуры ifconf и ifreq, используемые в различных вызовах функции ioctl, относящихся к интерфейсам

```
//<net/if.h> struct ifconf {
    int ifc_len; /* размер буфера, "значение-результат" */
    union {
        caddr_t ifcu_buf; /* ввод от пользователя к ядру */
        struct ifreq *ifcu_req; /* ядро возвращает пользователю */
    } ifc_ifcu;
};
#define ifc_buf ifc_ifcu.ifcu_buf /* адрес буфера */
#define ifc_req ifc_ifcu.ifcu_req /* массив возвращенных структур */

#define IFNAMSIZ 16

struct ifreq {
    char ifr_name[IFNAMSIZ]; /* имя интерфейса, например "le0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int ifru_metric;
        caddr_t ifru_data;
    } ifr_ifru;
};
#define ifr_addr ifr_ifru.ifru_addr /* адрес */
#define ifr_dstaddr ifr_ifru.ifru_dstaddr /* другой конец линии передачи, называемой
                                         "точка-точка" */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* широковещательный адрес */
#define ifr_flags ifr_ifru.ifru_flags /* флаги */
#define ifr_metric ifr_ifru.ifru_metric /* метрика */
#define ifr_data ifr_ifru.ifru_data /* с использованием интерфейсом */
```

Прежде чем вызвать функцию ioctl, мы выделяем в памяти место для буфера и для структуры ifconf, а затем инициализируем эту структуру. Мы показываем это на рис. 17.1, предполагая, что наш буфер имеет размер 1024 байта. Третий аргумент функции ioctl — это указатель на нашу структуру ifconf.

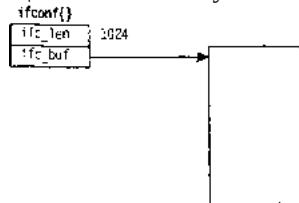


Рис. 17.1. Инициализация структуры ifconf перед вызовом SIOCGIFCONF

Если мы предположим, что ядро возвращает две структуры ifreq, то при завершении функции ioctl мы можем получить ситуацию, представленную на рис. 17.2. Затененные области были изменены функцией ioctl. Буфер заполняется двумя структурами, и элемент ifc_len структуры ifconf обновляется, с тем чтобы соответствовать количеству информации, хранимой в буфере. Предполагается, что на этом рисунке каждая структура ifreq занимает 32 байта.

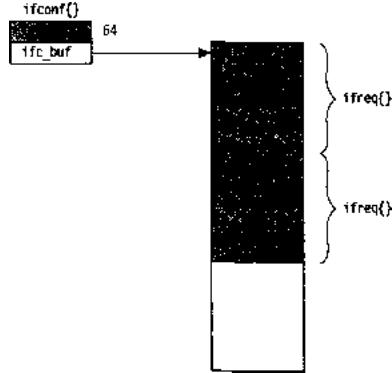


Рис. 17.2. Значения, возвращаемые в результате вызова SIOCGIFCONF

Указатель на структуру `ifreq` также используется в качестве аргумента оставшихся функций `ioctl` интерфейса, показанных в табл. 17.1, которые мы описываем в разделе 17.7. Отметим, что каждая структура `ifreq` содержит объединение (`union`), а директивы компилятора `#define` позволяют непосредственно обращаться к полям объединения по их именам. Помните о том, что в некоторых системах в объединение `ifr_ifru` добавлено много зависящих от реализации элементов.

17.6. Функция get_ifi_info

Поскольку многим программам нужно знать обо всех интерфейсах системы, мы разработаем нашу собственную функцию `get_ifi_info`, возвращающую связный список структур — по одной для каждого активного в настоящий момент интерфейса. В этом разделе мы покажем, как эта функция реализуется с помощью вызова `SIOCGIFCONF` функции `ioctl`, а в главе 18 мы создадим ее другую версию, использующую маршрутизирующие сокеты.

ПРИМЕЧАНИЕ

BSD/OS предоставляет функцию `getifaddrs`, имеющую аналогичную функциональность.

Поиск по всему дереву исходного кода BSD/OS 2.1 показывает, что 12 программ выполняют вызов `SIOCGIFCONF` функции `ioctl` для определения присутствующих интерфейсов.

Сначала мы определяем структуру `ifi_info` в новом заголовочном файле, который называется `unpifi.h`, показанном в листинге 17.2.

Листинг 17.2. Заголовочный файл `unpifi.h`

```

//ioctl/unpifi.h
1 /* Наш собственный заголовочный файл для программ, которым требуется
2 информация о конфигурации интерфейса. Включаем его вместо "unp.h". */
3 #ifndef __unp_ifi_h
4 #define __unp_ifi_h

5 #include "unp.h"
6 #include <net/if.h>

7 #define IFI_NAME 16 /* то же, что и IFNAMSIZ в заголовке <net/if.h> */
8 #define IFI_HADDR 8 /* с учетом 64-битового интерфейса EUI-64 в будущем */

9 struct ifi_info {
10     char ifi_name[IFI_NAME];      /* имя интерфейса, заканчивается
11                                символом конца строки */
12     short ifi_index;            /* индекс интерфейса */
13     short ifi_mtu;              /* MTU для интерфейса */

```

```

13 u_char ifi_haddr[IFI_HADDR]; /* аппаратный адрес */
14 u_short ifi_hlen; /* количество байтов в аппаратном адресе: 0, 6, 8 */
15 short ifi_flags; /* константы IFF_XXX из <net/if.h> */
16 short if_myflags; /* наши флаги IFI_XXX */
17 struct sockaddr *ifi_addr; /* первичный адрес */
18 struct sockaddr *ifi_brdaddr; /* широковещательный адрес */
19 struct sockaddr *ifi_dstaddr; /* адрес получателя */
20 struct ifi_info *ifi_next; /* следующая из этих структур */
21 };

22 #define IFI_ALIAS 1 /* ifi_addr - это псевдоним */

23 /* прототипы функций */
24 struct ifi_info *get_ifi_info(int, int);
25 struct ifi_info *Get_ifi_info(int, int);
26 void free_ifi_info(struct ifi_info*);

27 #endif /* _unp_ifi_h */

```

9-21 Связный список этих структур возвращается нашей функцией. Элемент `ifi_next` каждой структуры указывает на следующую структуру. Мы возвращаем в этой структуре информацию, которая может быть востребована в типичном приложении: имя интерфейса, индекс интерфейса, MTU, аппаратный адрес (например, адрес Ethernet), флаги интерфейса (чтобы позволить приложению определить, поддерживает ли приложение широковещательную или многоадресную передачу и относится ли этот интерфейс к типу «точка-точка»), адрес интерфейса, широковещательный адрес, адрес получателя для связи «точка-точка». Вся память, используемая для хранения структур `ifi_info` вместе со структурами адреса сокета, содержащимися в них, выделяется динамически. Следовательно, мы также предоставляем функцию `free_ifi_info` для освобождения всей этой памяти.

Перед тем как представить реализацию нашей функции `ifi_info`, мы покажем простую программу, которая вызывает эту функцию и затем выводит информацию. Эта программа, представленная в листинге 17.3, является уменьшенной версией программы `ifconfig`.

Листинг 17.3. Программа `prifinfo`, вызывающая нашу функцию `ifi_info`

```

//ioctl/prifinfo.c
1 #include "unpifi.h"

2 int
3 main(int argc, char **argv)
4 {
5     struct ifi_info *ifi, *ifihead;
6     struct sockaddr *sa;
7     u_char *ptr;
8     int i, family, doaliases;

9     if (argc != 3)
10     err_quit("usage: prifinfo <inet4|inet6> <doaliases>");

11    if (strcmp(argv[1], "inet4") == 0)
12        family = AF_INET;
13    else if (strcmp(argv[1], "inet6") == 0)
14        family = AF_INET6;
15    else
16        err_quit("invalid <address-family>");
17    doaliases = atoi(argv[2]);

18    for (ifihead = ifi = Get_ifi_info(family, doaliases);
19         ifi != NULL; ifi = ifi->ifi_next) {
20        printf("%s: <", ifi->ifi_name);

```

```

21  if (ifi->ifi_index != 0)
22    printf("%d", ifi->ifi_index);
23  printf("<");
24  if (ifi->ifi_flags & IFF_UP) printf ("UP ");
25  if (ifi->ifi_flags & IFF_BROADCAST) printf("BCAST ");
26  if (ifi->ifi_flags & IFF_MULTICAST) printf("MCAST ");
27  if (ifi->ifi_flags & IFF_LOOPBACK) printf("LOOP ");
28  if (ifi->ifi_flags & IFF_POINTOPOINT) printf("P2P ");
29  printf(">\n");
30  if ((i = ifi->ifi_hlen) > 0) {
31    ptr = ifi->ifi_haddr;
32    do {
33      printf("%s%lx", (i == ifi->ifi_hlen) ? " " : ":", *ptr++);
34    } while (--i > 0);
35    printf("\n");
36  }
37  if (ifi->ifi_mtu != 0)
38    printf(" MTU: %d\n". ifi->ifi_mtu);
39  if ((sa = ifi->ifi_addr) != NULL)
40    printf(" IP addr: %s\n", Sock_ntop_host(sa, sizeof(*sa)));
41  if ((sa = ifi->ifi_brdaddr) != NULL)
42    printf(" broadcast addr, %s\n",
43           Sock_ntop_host(sa, sizeof(*sa)));
44  if ((sa = ifi->ifi_dstaddr) != NULL)
45    printf(" destination addr %s\n",
46           Sock_ntop_host(sa, sizeof(*sa)));
47 }
48 free_ifi_info(ifihead);
49 exit(0);
50 }

```

18-47 Программа представляет собой цикл `for`, в котором один раз вызывается функция `get_ifi_info`, а затем последовательно перебираются все возвращаемые структуры `ifi_info`.

20-36 Выводятся все имена интерфейсов и флаги. Если длина аппаратного адреса больше нуля, он выводится в виде шестнадцатеричного числа (наша функция `get_ifi_info` возвращает нулевую длину `ifi_hlen`, если адрес недоступен).

37-46 Выводятся MTU и те IP-адреса, которые были возвращены.

Если мы запустим эту программу на нашем узле `macosx` (см. рис. 1.7), то получим следующий результат:

```

macosx % prifinfo inet4 0
lo0: <UP MCAST LOOP >
  MTU: 16384
  IP addr: 127.0.0.1
en1: <UP BCAST MCAST >
  MTU: 1500
  IP addr: 172.24.37.78
  broadcast addr: 172.24.37.95

```

Первый аргумент командной строки `inet4` задает адрес IPv4, а второй, нулевой аргумент указывает, что не должно возвращаться никаких псевдонимов, или альтернативных имен (альтернативные имена IP-адресов мы описываем в разделе A.4). Обратите внимание, что в MacOS X аппаратный адрес интерфейса Ethernet недоступен.

Если мы добавим к интерфейсу Ethernet (`en1`) три альтернативных имени адреса с идентификаторами узла 79, 80 и 81 и изменим второй аргумент командной строки на 1, то получим:

```

macosx % prifinfo inet4 1
lo0: <UP MCAST LOOP >
  MTU: 16384
  IP addr: 127.0.0.1

```

```

en1: <UP BCAST MCAST >
MTU: 1500
IP addr: 172.24.37.78 первичный IP-адрес
broadcast addr: 172.24.37.95
en1: <UP BCAST MCAST >
MTU: 1500
IP addr: 172.24.37.79 первый псевдоним
broadcast addr: 172.24.37.95
en1: <UP BCAST MCAST >
MTU: 1500
IP addr: 172.24.37.80 второй псевдоним
broadcast addr: 172.24.37.95
en1: <UP BCAST MCAST >
MTU: 1500
IP addr: 172.24.37.81 третий псевдоним
broadcast addr: 172.24.37.95

```

Если мы запустим ту же программу под FreeBSD, используя реализацию функции `get_ifi_info`, приведенную в листинге 18.9 (которая может легко получить аппаратный адрес), то получим:

```

freebsd4 % prifinfo inetc 1
de0: <UP BCAST MCAST >
0:80:c8:2b:d9:28
IP addr: 135.197.17.100
broadcast addr: 135.197.17.255
de1: <UP BCAST MCAST >
0:40:5:42:d6:de
IP addr: 172.24.37.94 основной IP-адрес
broadcast addr: 172.24.37.95
ef0: <UP BCAST MCAST >
0:40:5:42:d6:de
IP addr: 172.24.37.93 псевдоним
broadcast addr: 172.24.37.93
lo0: <UP MCAST LOOP >
IP addr: 127.0.0.1

```

В этом примере мы указали программе выводить псевдонимы, и мы видим, что один из псевдонимов определен для второго интерфейса Ethernet (de1) с идентификатором узла 93.

Теперь мы покажем нашу реализацию функции `get_ifi_info`, использующую вызов `SIOCGIFCONF` функции `ioctl`. В листинге 17.4 показана первая часть этой функции, получающая от ядра конфигурацию интерфейса.

Листинг 17.4. Выполнение вызова `SIOCGIFCONF` для получения конфигурации интерфейса

```

//lib/get_if_info.c
1 #include "unpifi.h"

2 struct ifi_info*
3 get_ifi_info(int family, int doaliases)
4 {
5     struct ifi_info *ifi, *ifihead, **ifipnext;
6     int sockfd, len, lastlen, flags, myflags, idx = 0, hlen = 0;
7     char *ptr, *buf, lastname[IFNAMSIZ], *cptr, *haddr, *sdlname;
8     struct ifconf ifc;
9     struct ifreq *ifr, ifrcopy;
10    struct sockaddr_in *sinptr;
11    struct sockaddr_in6 *sin6ptr;

12    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
13    lastlen = 0;

```

```

14 len = 100 * sizeof(struct ifreq); /* начальное приближение к нужному размеру буфера */
15 for (;;) {
16     buf = Malloc(len);
17     ifc.ifc_len = len;
18     ifc.ifc_buf = buf;
19     if (ioctl(sockfd, SIOCGIFCONF, &ifc) < 0) {
20         if (errno != EINVAL || lastlen != 0)
21             err_sys("ioctl error");
22     } else {
23         if (ifc.ifc_len == lastlen)
24             break; /* успех, значение len не изменилось */
25         lastlen = ifc.ifc_len;
26     }
27     len += 10 * sizeof(struct ifreq); /* приращение */
28     free(buf);
29 }
30 ifihead = NULL;
31 ifipnext = &ifihead;
32 lastname[0] = 0;
33 sdlname = NULL;

```

Создание сокета Интернета

11 Мы создаем сокет UDP, который будет использоваться с функциями `ioctl`. Может применяться как сокет TCP, так и сокет UDP [128, с. 163].

Выполнение вызова SIOCGIFCONF в цикле

12-28 Фундаментальной проблемой, связанной с вызовом `SIOCGIFCONF`, является то, что некоторые реализации не возвращают ошибку, если буфер слишком мал для хранения полученного результата [128, с. 118–119]. В этом случае результат просто обрезается так, чтобы поместиться в буфер, и функция `ioctl` возвращает нулевое значение, что соответствует успешному выполнению. Это означает, что единственный способ узнать, достаточно ли велик наш буфер, — сделать вызов, сохранить возвращенную длину, снова сделать вызов с большим размером буфера и сравнить полученную длину со значением, сохраненным из предыдущего вызова. Только если эти две длины одинаковы, наш буфер можно считать достаточно большим.

ПРИМЕЧАНИЕ

Беркли-реализации не возвращают ошибку, если буфер слишком мал [128, с. 118–119], и результат просто обрезается так, чтобы поместиться в существующий буфер. Solaris 2.5 возвращает ошибку `EINVAL`, если возвращаемая длина больше или равна длине буфера. Но мы не можем считать вызов успешным, если возвращаемая длина меньше размера буфера, поскольку Беркли-реализации могут возвращать значение, меньшее размера буфера, если часть структуры в него не помещается.

В некоторых реализациях предоставляется вызов `SIOCGIFNUM`, который возвращает число интерфейсов. Это позволяет приложению перед выполнением вызова `SIOCGIFCONF` выделить в памяти место для буфера достаточного размера, но такой подход не является широко распространенным.

Выделение в памяти места под буфер фиксированного размера для результата вызова `SIOCGIFCONF` стало проблемой с ростом Сети, поскольку большие веб-серверы используют много альтернативных адресов для одного интерфейса. Например, в Solaris 2.5 был предел в 256 альтернативных адресов для интерфейса, но в версии 2.6 этот предел вырос до 8192.

Обнаружилось, что на сайтах с большим числом альтернативных адресов перестают работать программы с буферами фиксированного размера для размещения информации об интерфейсе. Хотя Solaris возвращает ошибку, если буфер слишком мал, эти программы размещают в памяти буфер фиксированного размера, запускают функцию ioctl, но затем перестают работать при возвращении ошибки.

12-15 Мы динамически размещаем в памяти буфер начиная с размера, достаточного для 100 структур ifreq. Мы также отслеживаем длину, возвращаемую последним вызовом SIOCGIFCONF в lastlen, и инициализируем ее нулем.

19-20 Если функция ioctl возвращает ошибку EINVAL и функция еще не возвращалась успешно (то есть lastlen все еще равно нулю), значит, мы еще не выделили буфер достаточного размера, поэтому мы продолжаем выполнять цикл.

22-23 Если функция ioctl завершается успешно и возвращаемая длина равна lastlen, значит, длина не изменилась (наш буфер имеет достаточный размер), и мы с помощью функции break выходим из цикла, так как у нас имеется вся информация.

26-27 В каждом проходе цикла мы увеличиваем размер буфера для хранения еще 10 структур ifreq.

Инициализация указателей связного списка

29-31 Поскольку мы будем возвращать указатель на начало связного списка структур ifi_info, мы используем две переменные ifihead и ifipnext для хранения указателей на список по мере его создания.

Следующая часть нашей функции get_ifi_info, содержащая начало основного цикла, показана в листинге 17.5.

Листинг 17.5. Конфигурация интерфейса процесса

```
//lib/get_ifi_info.c
34 for (ptr = buf; ptr < buf + ifc.ifc_len; ) {
35     ifr = (struct ifreq*)ptr;

36 #ifdef HAVE_SOCKADDR_SA_LEN
37     len = max(sizeof(struct sockaddr), ifr->ifr_addr.sa_len);
38 #else
39     switch (ifr->ifr_addr.sa_family) {
40 #ifdef IPV6
41         case AF_INET6:
42             len = sizeof(struct sockaddr_in6);
43             break;
44 #endif
45         case AF_INET:
46             default:
47             len = sizeof(struct sockaddr);
48             break;
49     }
50 #endif /* HAVE_SOCKADDR_SA_LEN */
51     ptr += sizeof(ifr->ifr_name) + len; /* для следующей строки */

52 #ifdef HAVE_SOCKADDR_DL_STRUCT
53     /* предполагается, что AF_LINK идет перед AF_INET и AF_INET6 */
54     if (ifr->ifr_addr.sa_family == AF_LINK) {
55         struct sockaddr_dl *sdl = (struct sockaddr_dl*)&ifr->ifr_addr;
56         sdlname = ifr->ifr_name;
57         idx = sdl->sdl_index;
58         haddr = sdl->sdl_data + sdl->sdl_nlen;
59         hlen = sdl->sdl_alen;
60     }
61 #endif
```

```

62 if (ifr->ifr_addr.sa_family != family)
63 continue; /* игнорируется, если семейство адреса не то */
64 myflags = 0;
65 if ((cptr = strchr(ifr->ifr_name, ':')) != NULL)
66 *cptr = 0; /* замена двоеточия нулем */
67 if (strncmp(lastname, ifr->ifr_name, IFNAMSIZ) == 0) {
68 if (doaliases == 0)
69 continue; /* этот интерфейс уже обработан */
70 myflags = IFI_ALIAS;
71 }
72 memcpy(lastname, ifr->ifr_name, IFNAMSIZ);

73 ifrcopy = *ifr;
74 Ioctl(sockfd, SIOCGIFFLAGS, &ifrcopy);
75 flags = ifrcopy.ifr_flags;
76 if ((flags & IFF_UP) == 0)
77 continue; /* игнорируется, если интерфейс не используется */

```

Переход к следующей структуре адреса сокета

35-51 При последовательном просмотре всех структур `ifreq` `ifr` указывает на текущую структуру, а мы увеличиваем `ptr` на единицу, чтобы он указывал на следующую. Необходимо предусмотреть особенность более новых систем, предоставляющих поле длины для структур адреса сокета, и вместе с тем учесть, что более старые системы этого поля не предоставляют. Хотя в листинге 17.1 структура адреса сокета, содержащаяся в структуре `ifreq`, объявляется как общая структура адреса сокета, в новых системах она может относиться к произвольному типу. Действительно, в 4.4BSD структура адреса сокета канального уровня также возвращается для каждого интерфейса [128, с. 118]. Следовательно, если поддерживается элемент длины, то мы должны использовать его значение для переустановки нашего указателя на следующую структуру адреса сокета. В противном случае мы определяем длину, исходя из семейства адресов, используя размер общей структуры адреса сокета (16 байт) в качестве значения по умолчанию.

ПРИМЕЧАНИЕ

В системах, поддерживающих IPv6, не оговаривается, возвращается ли адрес IPv6 вызовом `SIOCGIFCONF`. Для более новых систем мы вводим оператор `case`, в котором предусмотрена возможность возвращения адресов IPv6. Проблема состоит в том, что объединение в структуре `ifreq` определяет возвращаемые адреса как общие 16-байтовые структуры `sockaddr`, подходящие для 16-байтовых структур `sockaddr_in` IPv4, но для 24-байтовых структур `sockaddr_in6` IPv6 они слишком малы. В случае возвращения адресов IPv6 возможно некорректное поведение существующего кода, созданного в предположении, что в каждой структуре `ifreq` содержится структура `sockaddr` фиксированного размера. В системах, где структура `sockaddr` имеет поле `sa_len`, никаких проблем не возникает, потому что такие системы легко могут указывать размер структур `sockaddr`.

52-60 Если система возвращает структуры `sockaddr` семейства `AF_LINK` в `SIOCGIFCONF`, мы копируем индекс интерфейса и данные об аппаратном адресе из таких структур.

62-63 Мы игнорируем все адреса из семейств, отличных от указанного вызывающим процессом в аргументе функции `get_ini_info`.

Обработка альтернативных имен

64-72 Нам нужно обнаружить все альтернативные имена (псевдонимы), которые могут существовать для интерфейса, то есть присвоенные этому интерфейсу дополнительные адреса. Обратите внимание в наших примерах, следующих за листингом 17.3, что в Solaris псевдоним содержит двоеточие, в то время как в 4.4BSD имя интерфейса в псевдониме не изменяется. Чтобы обработать оба случая, мы сохраняем последнее имя интерфейса в lastname и сравниваем его только до двоеточия, если оно присутствует. Если двоеточия нет, мы игнорируем этот интерфейс в том случае, когда имя эквивалентно последнему обработанному интерфейсу.

Получение флагов интерфейса

73-77 Мы выполняем вызов SIOCGIFFLAGS функции ioctl (см. раздел 16.5), чтобы получить флаги интерфейса. Третий аргумент функции ioctl — это указатель на структуру ifreq, содержащую имя интерфейса, для которого мы хотим получить флаги. Мы создаем копию структуры ifreq, перед тем как запустить функцию ioctl, поскольку в противном случае этот вызов перезаписал бы IP-адрес интерфейса, потому что оба они являются элементами одного и того же объединения из листинга 17.1. Если интерфейс не активен, мы игнорируем его.

В листинге 17.6 представлена третья часть нашей функции.

Листинг 17.6. Получение и возвращение адресов интерфейса

```
//ioctl/get_ifi_info.c
78  ifi = Calloc(1, sizeof(struct ifi_info));
79  *ifipnext = ifi; /* prev указывает на новую структуру */
80  ifipnext = &ifi->ifi_next; /* сюда указывает указатель на
                           следующую структуру */

81  ifi->ifi_flags = flags; /* значения IFF_XXX */
82  ifi->ifi_myflags = myflags; /* значения IFI_XXX */
83 #if defined(SIOCGIFMTU) && defined(HAVE_STRUCT_IFREQ_IFR_MTU)
84  Iioctl(sockfd, SIOCGIFMTU, &ifrcopy);
85  ifi->ifi_mtu = ifrcopy.ifr_mtu;
86 #else
87  ifi->ifi_mtu = 0;
88#endif
89  memcpy(ifi->ifi_name, ifr->ifr_name, IFI_NAME);
90  ifi->ifi_name[IFI_NAME-1] = '\0';
91  /* если sockaddr_dl относится к другому интерфейсу, он игнорируется */
92  if (sdlname == NULL || strcmp(sdlname, ifr->ifr_name) != 0)
93    idx = hlen = 0;
94  ifi->ifi_index = idx;
95  ifi->ifi_hlen = hlen;
96  if (ifi->ifi_hlen > IFI_HADDR)
97    ifi->ifi_hlen = IFI_HADDR;
98  if (hlen)
99    memcpy(ifi->ifi_haddr, haddr, ifi->ifi_hlen);
```

Выделение памяти и инициализация структуры ifi_info

78-99 На этом этапе мы знаем, что возвратим данный интерфейс вызывающему процессу. Мы выделяем память для нашей структуры ifi_info и добавляем ее в конец связного списка, который мы создаем. Мы копируем флаги и имя интерфейса в эту структуру. Далее мы проверяем, заканчивается ли имя интерфейса нулем, и поскольку функция calloc инициализирует выделенную в памяти область нулями, мы знаем, что ifi_hlen инициализируется нулем, а ifi_next — пустым указателем.

В листинге 17.7 представлена последняя часть нашей функции.

Листинг 17.7. Получение и возврат адреса интерфейса

```
100 switch (ifr->ifr_addr.sa_family) {
```

```

101    case AF_INET:
102        sinptr = (struct sockaddr_in*)&ifr->ifr_addr;
103        ifi->ifi_addr = Calloc(1, sizeof(struct sockaddr_in));
104        memcpy(ifi->ifi_addr, sinptr, sizeof(struct sockaddr_in));

105 #ifdef SIOCGIFBRDADDR
106     if (flags & IFF_BROADCAST) {
107         Ioctl(sockfd, SIOCGIFBRDADDR, &ifrcopy);
108         sinptr = (struct sockaddr_in*) &ifrcopy.ifr_broadaddr;
109         ifi->ifi_brdaddr = Calloc(1, sizeof(struct sockaddr_in));
110         memcpy(ifi->ifi_brdaddr, sinptr, sizeof(struct sockaddr_in));
111     }
112 #endif

113 #ifdef SIOCGIFDSTADDR
114     if (flags & IFF_POINTOPOINT) {
115         Ioctl(sockfd, SIOCGIFDSTADDR, &ifrcopy);
116         sinptr = (struct sockaddr_in*)&ifrcopy.ifr_dstaddr;
117         ifi->ifi_dstaddr = Calloc(1, sizeof(struct sockaddr_in));
118         memcpy(ifi->ifi_dstaddr, sinptr, sizeof(struct sockaddr_in));
119     }
120 #endif
121     break;

122 case AF_INET6:
123     sin6ptr = (struct sockaddr_in6*)&ifr->ifr_addr;
124     ifi->ifi_addr = Calloc(1, sizeof(struct sockaddr_in6));
125     memcpy(ifi->ifi_addr, sin6ptr, sizeof(struct sockaddr_in6));

126 #ifdef SIOCGIFDSTADDR
127     if (flags & IFF_POINTOPOINT) {
128         Ioctl(sockfd, SIOCGIFDSTADDR, &ifrcopy);
129         sin6ptr = (struct sockaddr_in6*)&ifrcopy.ifr_dstaddf;
130         ifi->ifi_dstaddr = Calloc(1, sizeof(struct sockaddr_in6));
131         memcpy(ifi->ifi_dstaddr, sin6ptr,
132             sizeof(struct sockaddr_in6));
133     }
134 #endif
135     break;

136 default:
137     break;
138 }
139 }
140 free(buf);
141 return(ifihead); /* указатель на первую структуру в связной списке */
142 }

```

102-104 Мы копируем IP-адрес, возвращенный из нашего начального вызова `SIOCGIFCONF` функции `ioctl`, в структуру, которую мы создаем.

106-119 Если интерфейс поддерживает широковещательную передачу, мы получаем широковещательный адрес с помощью вызова `SIOCGIFBRDADDR` функции `ioctl`. Мы выделяем память для структуры адреса сокета, содержащей этот адрес, и добавляем ее к структуре `ifi_info`, которую мы создаем. Аналогично, если интерфейс является интерфейсом типа «точка-точка», вызов `SIOCGIFBRDADDR` возвращает IP-адрес другого конца связи.

123-133 Обработка случая IPv6 — полная аналогия IPv4 за тем исключением, что вызов `SIOCGIFBRDADDR` не делается, потому что IPv6 не поддерживает широковещательную передачу.

В листинге 17.8 показана функция `free_ifi_info`, которой передается указатель, возвращенный функцией `get_ifi_info`. Эта функция освобождает всю динамически выделенную память.

Листинг 17.8. Функция `free_ifi_info`: освобождение памяти, которая была динамически выделена функцией `get_ifi_info`

```
//ioctl/get_ifi_info.c
143 void
144 free_ifi_info(struct ifi_info *ifihead)
145 {
146     struct ifi_info *ifi, *ifinext;

147     for (ifi = ifihead; ifi != NULL; ifi = ifinext) {
148         if (ifi->ifi_addr != NULL)
149             free(ifi->ifi_addr);
150         if (ifi->ifi_brdaddr != NULL)
151             free(ifi->ifi_brdaddr);
152         if (ifi->ifi_dstaddr != NULL)
153             free(ifi->ifi_dstaddr);
154         ifinext = ifi->ifi_next; /* невозможно получить ifi_next
                           после вызова freed */
155         free(ifi);
156     }
157 }
```

17.7. Операции с интерфейсами

Как мы показали в предыдущем разделе, запрос `SIOCGIFCONF` возвращает имя и структуру адреса сокета для каждого сконфигурированного интерфейса. Существует множество других вызовов, позволяющих установить или получить все остальные характеристики интерфейса. Версия `get` этих вызовов (`SIOCGXXX`) часто запускается программой `netstat`, а версия `set` (`SIOCSXXX`) — программой `ifconfig`. Любой пользователь может получить информацию об интерфейсе, в то время как установка этой информации требует прав привилегированного пользователя.

Эти вызовы получают или возвращают структуру `ifreq`, адрес которой задается в качестве третьего аргумента функции `ioctl`. Интерфейс всегда идентифицируется по имени: `le0`, `lo0`, `ppp0`, — то есть по имени, заданному в элементе `ifr_name` структуры `ifreq`.

Многие из этих запросов используют структуру адреса сокета, для того чтобы задать или возвратить IP-адрес или маску адреса. Для IPv4 адрес или маска содержится в элементе `sin_addr` из структуры адреса сокета Интернета. Для IPv6 они помещаются в элемент `sin6_addr` структуры адреса сокета IPv6.

- `SIOCGIFADDR`. Возвращает адрес направленной передачи в элементе `ifr_addr`.
- `SIOCSIFADDR`. Устанавливает адрес интерфейса из элемента `ifr_addr`. Также вызывается функция инициализации для интерфейса.
 - `SIOCGIFFLAGS`. Возвращает флаги интерфейса в элементе `ifr_flags`. Имена различных флагов определяются в виде `IFF_XXX` в заголовочном файле `<net/if.h>`. Флаги указывают, например, включен ли интерфейс (`IFF_UP`), является ли он интерфейсом типа «точка-точка» (`IFF_POINTOPOINT`), поддерживает ли широковещательную передачу (`IFF_BROADCAST`) и т.д.
 - `SIOCSIFFLAGS`. Устанавливает флаги из элемента `ifr_flags`.
 - `SIOCGIFDSTADDR`. Возвращает адрес типа «точка-точка» в элементе `ifr_dstaddr`.
 - `SIOCSIFDSTADDR`. Устанавливает адрес типа «точка-точка» из элемента `ifr_dstaddr`.
 - `SIOCGIFBRDADDR`. Возвращает широковещательный адрес в элементе `ifr_broadaddr`. Приложение сначала должно получить флаги интерфейса, а затем сделать корректный вызов: `SIOCGIFBRDADDR` для широковещательного интерфейса или `SIOCGIFDSTADDR` — для интерфейса типа «точка-точка».
 - `SIOCSIFBRDADDR`. Устанавливает широковещательный адрес из элемента `ifr_broadaddr`.
 - `SIOCGIFNETMASK`. Возвращает маску подсети в элементе `ifr_addr`.
 - `SIOCSIFNETMASK`. Устанавливает маску подсети из элемента `ifr_addr`.
 - `SIOCGIFMETRIC`. Возвращает метрику интерфейса в элементе `ifr_metric`. Метрика поддерживается ядром для каждого интерфейса, но используется демоном маршрутизации `routed`. Метрика интерфейса добавляется к счетчику количества переходов.

- **SIOCSIFMETRIC.** Устанавливает метрику интерфейса из элемента `ifr_metric`.

В этом разделе мы описали наиболее типичные операции интерфейсов. Во многих реализациях появились дополнительные операции.

17.8. Операции с кэшем ARP

Операции с кэшем ARP также осуществляются с помощью функции `ioctl`. В этих запросах используется структура `arpreq`, показанная в листинге 17.9 и определяемая в заголовочном файле `<net/if_arp.h>`.

Листинг 17.9. Структура `arpreq`, используемая с вызовами `ioctl` для кэша ARP

```
struct arpreq {  
    struct sockaddr arp_pa; /* адрес протокола */  
    struct sockaddr arp_ha; /* аппаратный адрес */  
    int             arp_flags; /* флаги */  
};  
  
#define ATF_INUSE 0x01 /* запись, которую нужно использовать */  
#define ATF_COM   0x02 /* завершенная запись */  
#define ATF_PERM  0x04 /* постоянная запись */  
#define ATF_PUBL  0x08 /* опубликованная запись (отсылается другим узлам) */
```

Третий аргумент функции `ioctl` должен указывать на одну из этих структур. Поддерживаются следующие три вызова:

- **SIOCSARP.** Добавляет новую запись в кэш ARP или изменяет существующую запись. `arp_pa` — это структура адреса сокета Интернета, содержащая IP-адрес, а `arp_ha` — это общая структура адреса сокета с элементом `ss_family`, равным `AF_UNSPEC`, и элементом `sa_data`, содержащим аппаратный адрес (например, 6-байтовый адрес Ethernet). Два флага `ATF_PERM` и `ATF_PUBL` могут быть заданы приложением. Два других флага, `ATF_INUSE` и `ATF_COM`, устанавливаются ядром.
- **SIOCDAWRP.** Удаляет запись из кэша ARP. Вызывающий процесс задает интернет-адрес удаляемой записи.
- **SIOCGARP.** Получает запись из кэша ARP. Вызывающий процесс задает интернет-адрес, и соответствующий адрес Ethernet возвращается вместе с флагами.

Добавлять или удалять записи может только привилегированный пользователь. Эти три вызова обычно делает программа `arp`.

ПРИМЕЧАНИЕ

Запросы функции `ioctl`, связанные с ARP, не поддерживаются в некоторых более новых системах, использующих для описанных операций ARP маршрутизирующие сокеты.

Обратите внимание, что невозможно с помощью функции `ioctl` перечислить все записи кэша ARP. Большинство версий команды `arp` при использовании флага `-a` (перечисление всех записей кэша ARP) считывают память ядра (`/dev/kmem`), чтобы получить текущее содержимое кэша ARP. Мы увидим более простой (и предпочтительный) способ, основанный на применении функции `sysctl`, описанной в разделе 18.4.

Пример: вывод аппаратного адреса узла

Теперь мы используем нашу функцию `my_addrs` для того, чтобы возвратить все IP-адреса узла. Затем для каждого IP-адреса мы делаем вызов `SIOCGARP` функции `ioctl`, чтобы получить и вывести аппаратные адреса. Наша программа показана в листинге 17.10.

Листинг 17.10. Вывод аппаратного адреса узла

```
//ioctl/prmac.c  
1 #include "unpifi.h"  
2 #include <net/if_arp.h>
```

```

3 int
4 main(int argc, char **argv)
5 {
6     int sockfd;
7     struct ifi_info *ifi;
8     unsigned char *ptr;
9     struct arpreq arpreq;
10    struct sockaddr_in *sin;

11    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
12    for (ifi = get_ifi_info(AF_INET, 0); ifi != NULL; ifi = ifi->ifi_next) {
13        printf("%s: ", Sock_ntop(ifi->ifi_addr, sizeof(struct sockaddr_in)));

14        sin = (struct sockaddr_in*)&arpreq.arp_pa;
15        memcpy(sin, ifi->ifi_addr, sizeof(struct sockaddr_in));

16        if (ioctl(sockfd, SIOCGARP, &arpreq) < 0) {
17            err_ret("ioctl SIOCGARP");
18            continue;
19        }

20        ptr = &arpreq.arp_ha.sa_data[0];
21        printf("%x:%x:%x:%x:%x:\n", *ptr, *(ptr+1),
22               *(ptr+2), *(ptr+3), *(ptr+4), *(ptr+5));
23    }
24    exit(0);
25 }

```

Получение списка адресов и проход в цикле по каждому из них

12 Мы вызываем функцию `get_ifi_info`, чтобы получить IP-адреса узла, а затем выполняем цикл по всем адресам.

Вывод IP-адреса

13 Мы выводим IP-адреса, используя функцию `inet_ntop`. Мы просим функцию `get_ifi_info` возвращать только адреса IPv4, так как ARP с IPv6 не используется.

Вызов функции `ioctl` и проверка ошибок

14-19 Мы заполняем структуру `arp_pa` как структуру адреса сокета IPv4, содержащую адрес IPv4. Вызывается функция `ioctl`, и если она возвращает ошибку (например, указанный адрес относится к интерфейсу, не поддерживающему ARP), мы выводим сообщение и переходим к следующему адресу.

Вывод аппаратного адреса

20-22 Выводится аппаратный адрес, возвращаемый `ioctl`.

При запуске этой программы на нашем узле `hpx` мы получаем:

```

hpx % prmac
192.6.38.100: 0:60:b0:c2:68:9b
192.168.1.1: 0:60:b0:b2:28:2b

```

127.0.0.1: ioctl SIOCGARP: Invalid argument

17.9. Операции с таблицей маршрутизации

Для работы с таблицей маршрутизации предназначены два вызова функции `ioctl`. Эти два вызова требуют, чтобы третий аргумент функции `ioctl` был указателем на структуру `rtentry`, которая определяется в заголовочном файле `<net/route.h>`. Обычно эти вызовы исходят от программы `route`. Их может делать только привилегированный пользователь. При наличии маршрутизирующих сокетов (глава 18) для выполнения этих запросов используются именно они, а не функция `ioctl`.

- `SIOCADDRT`. Добавить запись в таблицу маршрутизации.
- `SIOCDELRT`. Удалить запись из таблицы маршрутизации.

Нет способа с помощью функции `ioctl` перечислить все записи таблицы маршрутизации. Этую операцию обычно выполняет программа `netstat` с флагом `-r`. Программа получает таблицу маршрутизации, считывая память ядра (`/dev/kmem`). Как и в случае с просмотром кэша ARP, в разделе 18.4 мы увидим более простой (и предпочтительный) способ, предоставляемый функцией `sysctl`.

17.10. Резюме

Команды функции `ioctl`, используемые в сетевых приложениях, можно разделить на шесть категорий:

1. Операции с сокетами (находимся ли мы на отметке внеполосных данных?).
2. Операции с файлами (установить или сбросить флаг отсутствия блокировки).
3. Операции с интерфейсами (возвратить список интерфейсов, получить широковещательный адрес).
4. Операции с кэшем ARP (создать, изменить, получить, удалить).
5. Операции с таблицей маршрутизации (добавить или удалить).
6. Операции с потоками STREAMS (см. главу 31).

Мы будем использовать операции с сокетами и файлами, а получение списка интерфейсов — это настолько типичная операция, что для этой цели мы разработали собственную функцию. Мы будем применять ее много раз в оставшейся части книги. Вызовы функции `ioctl` с кэшем ARP и таблицей маршрутизации используются лишь несколькими специализированными программами.

Упражнения

1. В разделе 17.7 мы сказали, что широковещательный адрес, возвращаемый запросом `SIOCGIFBRDADDR`, возвращается в элементе `ifr_broadaddr`. Но на с. 173 [128] сказано, что он возвращается в элементе `ifr_dstaddr`. Имеет ли это значение?

2. Измените программу `get_ifi_info` так, чтобы она делала первый вызов `SIOCGIFCONF` для одной структуры `ifreq`, а затем каждый раз в цикле увеличивайте длину на размер одной из этих структур. Затем поместите в цикл операторы, которые выводили бы размер буфера при каждом вызове независимо от того, возвращает функция `ioctl` ошибку или нет, и при успешном выполнении выведите возвращаемую длину буфера. Запустите программу `prifinfo` и посмотрите, как ваша система обрабатывает вызов, когда размер буфера слишком мал. Выполните также семейство адресов для всех возвращаемых структур, семейство адресов которых не совпадает с указанным в первом аргументе функции `get_ifi_info`, чтобы увидеть, какие еще структуры возвращает ваша система.

3. Измените функцию `get_ifi_info` так, чтобы она возвращала информацию об адресе с альтернативным именем, если дополнительный адрес находится не в той подсети, в которой находится предыдущий адрес для данного интерфейса. Таким образом, наша версия из раздела 17.6 будет игнорировать альтернативные имена в диапазоне от 206.62.226.44 до 206.62.226.46, и это вполне нормально, поскольку они находятся в той же подсети, что и первичный адрес интерфейса 206.62.226.33. Но если альтернативное имя находится в другой подсети, допустим 192.3.4.5, возвратите структуру `ifi_info` с информацией о дополнительном адресе.

4. Если ваша система поддерживает вызов `SIOCGIGNUM` функции `ioctl`, измените листинг 17.4 так, чтобы запустить этот вызов, и используйте возвращаемое значение как начальный размер буфера.

Глава 18

Маршрутизирующие сокеты

18.1. Введение

Традиционно доступ к таблице маршрутизации Unix внутри ядра осуществлялся с помощью команд функции `ioctl`. В разделе 17.9 мы описали две операции: `SIOCADDR` и `SIOCDELR`, предназначенные для добавления и удаления маршрута. Мы также отметили, что не существует операции чтения всей таблицы маршрутизации — вместо этого программы, такие как `netstat`, считывают память ядра, для того чтобы получить содержимое таблицы маршрутизации. И еще одно добавление. Демонам маршрутизации, таким как `gated`, необходимо отслеживать сообщения ICMP (Internet Control Message Protocol — протокол управляющих сообщений Интернета) об изменении маршрутов, получаемых ядром, и для этого они часто создают символьный (неструктурированный) сокет ICMP (см. главу 28), а затем прослушивают на этом сокете все получаемые сообщения ICMP.

В 4.3BSD Reno интерфейс подсистемы маршрутизации ядра был упрощен за счет создания семейства адресов (домена) `AF_ROUTE`. Единственный тип сокетов, поддерживаемый для этого семейства, — это символьный сокет (raw socket). Маршрутизирующие сокеты поддерживают три типа операций.

1. Процесс может отправить ядру сообщение, записав его в маршрутизирующий сокет. Таким образом добавляются и удаляются маршруты.

2. Процесс может прочитать сообщение от ядра через маршрутизирующий сокет. Так ядро уведомляет процесс о том, что сообщение ICMP об изменении маршрутизации было получено и обработано.

Некоторые операции включают оба шага: например, процесс отправляет ядру сообщение через маршрутизирующий сокет, запрашивая всю информацию по данному маршруту, после чего через маршрутизирующий сокет считывает ответ ядра.

3. Процесс может использовать функцию `sysctl` (см. раздел 18.4) либо для просмотра таблицы маршрутизации, либо для перечисления всех сконфигурированных интерфейсов.

Первые две операции требуют прав привилегированного пользователя, а третью операцию может выполнить любой процесс.

ПРИМЕЧАНИЕ

Некоторые версии Unix из более новых ослабили требование к правам пользователя для операции открытия маршрутизирующего сокета и ограничивают только передачу сообщений, изменяющих таблицу маршрутизации ядра. Это позволяет любому процессу узнать маршрут при помощи команды `RTM_GET`, не являясь суперпользователем.

Технически третья операция выполняется при помощи общей функции `sysctl`, а не маршрутизирующего сокета. Но мы увидим, что среди ее входных параметров есть семейство адресов (для описываемых в этой главе операций используется семейство `AF_ROUTE`), а результат она возвращает в том же формате, который используется ядром для маршрутизирующего сокета. Действительно, в ядре 4.4BSD обработка функции `sysctl` для семейства `AF_ROUTE` является частью кода маршрутизирующего сокета [128, с. 632–643].

Функция `sysctl` появилась в 4.4BSD. К сожалению, не все реализации, поддерживающие маршрутизирующие сокеты, предоставляют ее. Например, AIX 4.2, Digital Unix 4.0 и Solaris 2.6 поддерживают маршрутизирующие сокеты, но ни одна из этих систем не поддерживает утилиту `sysctl`.

18.2. Структура адреса сокета канального уровня

Структуры адреса сокета канального уровня будут встречаться нам как значения, содержащиеся в некоторых сообщениях, возвращаемых на маршрутизирующем сокете. В листинге 18.1^[1] показано определение структуры, задаваемой в заголовочном файле `<net/if_dl.h>`.

Листинг 18.1. Структура адреса сокета канального уровня

```

struct sockaddr_dl {
    uint8_t      sdl_len;
    sa_family_t  sdl_family;   /* AF_LINK */
    uint16_t     sdl_index;    /* индекс интерфейса, присвоенный системой,
                                если > 0 */
    uint8_t      sdl_type;    /* тип интерфейса из <net/if_types.h>.
IFT_ETHER и т.д. */
    uint8_t      sdl_nlen;    /* длина имени, начинается с sdl_data[0] */
    uint8_t      sdl_alen;    /* длина адреса канального уровня */
    uint8_t      sdl_slen;    /* адрес селектора канального уровня */
    char        sdl_data[12]; /* минимальная рабочая область.
                                может быть больше; содержит имя
                                интерфейса и адрес канального уровня */
};

У каждого интерфейса имеется уникальный положительный индекс. Далее в этой главе мы увидим, каким образом он возвращается функциями if_nameindex и if_nametoindex. В главе 21 при обсуждении параметров многоадресных сокетов IPv6 и в главе 27 при обсуждении дополнительных параметров сокетов IPv6 и IPv4 мы вновь вернемся к этим функциям.

```

Элемент `sdl_data` содержит имя, и адрес канального уровня (например, 48-разрядный MAC-адрес интерфейса Ethernet). Имя начинается с `sdl_data[0]` и не заканчивается нулем. Начало адреса канального уровня смещено на `sdl_nlen` байтов относительно начала имени. В этом заголовочном файле для возвращения указателя на адрес канального уровня задается следующий макрос:

```
#define LLADDR(s) ((caddr_t)((s)->sdl_data + (s)->sdl_nlen))
```

Эти структуры адреса сокета имеют переменную длину [128, с. 89]. Если адрес канального уровня и имя превышают 12 байт, размер структуры будет больше 20 байт. В 32-разрядных системах размер обычно округляется в большую сторону, до следующего числа, кратного 4 байтам. Мы также увидим на рис. 22.1, что когда одна из этих структур возвращается параметром сокета `IP_RECVIF`, все три длины становятся нулевыми, а элемента `sdl_data` не существует.

18.3. Чтение и запись

Создав маршрутизирующий сокет, процесс может отправлять ядру команды путем записи в этот сокет и считывать из него информацию от ядра. Существует 12 различных команд маршрутизации, 5 из которых могут быть запущены процессом. Они определяются в заголовочном файле `<net/route.h>` и показаны в табл. 18.1.

Таблица 18.1. Типы сообщений, проходящих по маршрутизирующему сокету

Тип сообщения	К ядру?	От ядра?	Описание	Тип структуры
RTM_ADD	•	•	Добавить маршрут	rt_msghdr
RTM_CHANGE	•	•	Поменять шлюз, метрику или флаги	rt_msghdr
RTM_DELADDR		•	Адрес был удален из интерфейса	ifa_msghdr
RTM_DELETE	•	•	Удалить маршрут	rt_msghdr
RTM_GET	•	•	Сообщить о метрике и других характеристиках маршрута	rt_msghdr
RTM_IFINFO		•	Находится ли интерфейс в активном состоянии	if_msghdr
RTM_LOCK	•	•	Блокировка указанной метрики	rt_msghdr
RTM_LOSING		•	Возможно, неправильный маршрут	rt_msghdr
RTM_MISS	•		Поиск этого адреса завершился неудачно	rt_msghdr
RTM_NEWSDDR		•	Адрес добавлен к интерфейсу	ifa_msghdr
RTM_NEWMDDR		•	Групповой адрес добавлен к интерфейсу	ifma_msghdr
RTM_REDIRECT		•	Ядро получило указание использовать другой маршрут	rt_msghdr
RTM_RESOLVE		•	Запрос на определение адреса канального уровня по адресу получателя	rt_msghdr

На маршрутизирующем сокете происходит обмен пятью различными структурами, как показано в последнем столбце таблицы: `rt_msghdr`, `if_msghdr`, `if_announcemsgHdr`, `ifma_msghdr` и `ifa_msghdr`. Эти структуры представлены в листинге 18.2.

Листинг 18.2. Пять структур, возвращаемых с маршрутизирующими сообщениями

```
struct rt_msghdr { /* из <net/route.h> */
    u_short rtm_msflen; /* для пропуска некорректных сообщений */
    u_char rtm_version; /* для обеспечения двоичной совместимости в будущем */
    u_char rtm_type; /* тип сообщения */

    u_short rtm_index; /* индекс интерфейса, с которым связан адрес */
    int rtm_flags; /* флаги */
    int rtm_addrs; /* битовая маска, идентифицирующая sockaddr (строку адреса
                    сокета) в msg */
    pid_t rtm_pid; /* идентификация отправителя */
    int rtm_seq; /* для идентификации действия отправителем */
    int rtm_errno; /* причина неудачного выполнения */
    int rtm_use; /* из rtentry */
    u_long rtm_inits; /* какую метрику мы инициализируем */
    struct rt_metrics rtm_rmx; /* сами метрики */
};

struct if_msghdr { /* из <net/if.h> */
    u_short ifm_msflen; /* для пропуска некорректных сообщений */
    u_char ifm_version; /* для обеспечения двоичной совместимости в будущем */
    u_char ifm_type; /* тип сообщения */

    int ifm_addrs; /* как rtm_addrs */
    int ifm_flags; /* значение if_flags */
    u_short ifm_index; /* индекс интерфейса, с которым связан адрес */
    struct if_data ifm_data; /* статистические и другие сведения */
};

struct ifa_msghdr { /* из <net/if.h> */
    u_short ifam_msflen; /* для пропуска некорректных сообщений */
    u_char ifam_version; /* для обеспечения двоичной совместимости в будущем */
    u_char ifam_type; /* тип сообщения */

    int ifam_addrs; /* как rtm_addrs */
    int ifam_flags; /* значение ifa_flags */
    u_short ifam_index; /* индекс интерфейса, с которым связан адрес */
    int ifam_metric; /* значение ifa_metric */
};

struct ifma_msghdr { /* из <net/if.h> */
    u_short ifmam_msflen; /* для пропуска некорректных сообщений */
    u_char ifmam_version; /* для обеспечения двоичной совместимости в будущем */
    u_char ifmam_type; /* тип сообщения */
    int ifmam_addrs; /* аналог rtm_addrs */
    int ifmam_flags; /* значение ifa_flags */
    u_short ifmam_index; /* индекс связанного ifp */
};

struct if_announcemsgHdr { /* из <net/if.h> */
    u_short ifan_msflen; /* для пропуска некорректных сообщений */
    u_char ifan_version; /* для обеспечения двоичной совместимости в будущем */
```

```

    u_char ifan_type; /* тип сообщения */
    u_short ifan_index; /* индекс связанного ifp */
    char ifan_name[IFNAMSIZ]; /* название интерфейса, напр. "en0" */
    u_short ifan_what; /* тип объявления */
};


```

Первые три элемента каждой структуры одни и те же: длина, версия и тип сообщения. Тип — это одна из констант из первого столбца табл. 18.1. Элемент длины `xxx_msflen` позволяет приложению пропускать типы сообщений, которые оно не распознает.

Элементы `rtm_addrs`, `ifm_addrs` и `ifam_addrs` являются битовыми масками, указывающими, какая из возможных восьми структур адреса сокета следует за сообщением. В табл. 18.2 показаны константы и значения для битовой маски, определяемые в заголовочном файле `<net/route.h>`.

Таблица 18.2. Константы, используемые для ссылки на структуры адреса сокета в маршрутизирующих сообщениях

Битовая маска, константа	Битовая маска, значение	Индекс массива, константа	Индекс массива, значение	Структура адреса сокета содержит
RTA_DST	0x01	RTAX_DST	0	Адрес получателя
RTA_GATEWAY	0x02	RTAX_GATEWAY	1	Адрес шлюза
RTA_NETMASK	0x04	RTAX_NETMASK	2	Маска сети
RTA_GENMASK	0x08	RTAX_GENMASK	3	Маска клонирования
RTA_IFP	0x10	RTAX_IFP	4	Имя интерфейса
RTA_IFA	0x20	RTAX_IFA	5	Адрес интерфейса
RTA_AUTHOR	0x40	RTAX_AUTHOR	6	Отправитель запроса на перенаправление
RTA_BRD	0x80	RTAX_BRD	7	Адрес получателя типа «точка-точка» или широковещательный
		RTAX_MAX	8	Максимальное количество элементов

В том случае, когда имеется множество структур адреса сокета, они всегда располагаются в порядке, показанном в таблице.

Пример: получение и вывод записи из таблицы маршрутизации

Теперь мы покажем пример использования маршрутизирующих сокетов. Наша программа получает аргумент командной строки, состоящий из адреса IPv4 в точечно-десятичной записи, и отправляет ядру сообщение `RTM_GET` для получения этого адреса. Ядро ищет адрес в своей таблице маршрутизации IPv4 и возвращает сообщение `RTM_GET` с информацией о соответствующей записи из таблицы маршрутизации. Например, если мы выполним на нашем узле freebsd такой код

```

freebsd # getrt 206.168.112.219
dest: 0.0.0.0
gateway: 12.106.32.1
netmask: 0.0.0.0

```

мы увидим, что этот адрес получателя использует маршрут по умолчанию (который хранится в таблице маршрутизации с IP-адресом получателя 0.0.0.0 и маской 0.0.0.0). Маршрутизатор следующей ретрансляции — это интернет-шлюз нашей системы. Если мы выполним

```

freebsd # getrt 192.168.42.0
dest: 192.168.42.0
gateway: AF_LINK, index=2
netmask: 255.255.255.0

```

задав в качестве получателя главную сеть Ethernet, получателем будет сама сеть. Теперь шлюзом является исходящий интерфейс, возвращаемый в качестве структуры `sockaddr_dl` с индексом интерфейса 2.

Перед тем как представить исходный код, мы показываем на рис. 18.1, что именно мы пишем в маршрутизирующий сокет и что возвращает ядро.

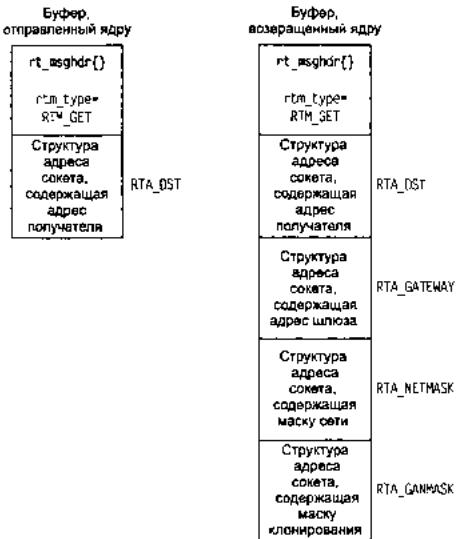


Рис. 18.1. Обмен данными с ядром на маршрутизирующем сокете для команды RTM_GET

Мы создаем буфер, содержащий структуру `rt_msghdr`, за которой следует структура адреса сокета, содержащая адрес получателя, информацию о котором должно найти ядро. Тип сообщения (`rtm_type`) — `RTM_GET`, а битовая маска (`rtm_addrs`) — `RTA_DST` (вспомните табл. 18.2). Эти значения указывают, что структура адреса сокета, следующая за структурой `rt_msghdr`, — это структура, содержащая адрес получателя. Эта команда может использоваться с любым семейством протоколов (представляющим таблицу маршрутизации), поскольку семейство адресов, в которое входит искомый адрес, указано в структуре адреса сокета.

После отправки сообщения ядру мы с помощью функции `read` читаем ответ, формат которого показан на рис. 18.1 справа: структура `rt_msghdr`, за которой следует до четырех структур адреса сокета. Какая из четырех структур адреса сокета возвращается, зависит от записи в таблице маршрутизации. Мы сможем идентифицировать возвращаемую структуру адреса сокета по значению элемента `rtm_addrs` возвращаемой структуры `rt_msghdr`. Семейство каждой структуры адреса сокета указано в элементе `ss_family`, и как мы видели в наших предыдущих примерах, первый раз сообщение `RST_GET` содержало информацию о том, что адрес шлюза является структурой адреса сокета IPv4, а второй раз это была структура адреса сокета канального уровня.

В листинге 18.3 показана первая часть нашей программы.

Листинг 18.3. Первая часть программы, запускающая команду RTM_GET на маршрутизирующем сокете

```

//route/getrt.c
1 #include "unproute.h"

2 #define BUflen (sizeof(struct rt_msghdr) + 512)
3 /* sizeof(struct sockaddr_in6) * 8 = 192 */
4 #define SEQ 9999

5 int
6 main(int argc, char **argv)
7 {
8     int sockfd;
9     char *buf;
10    pid_t pid;
11    ssize_t n;
12    struct rt_msghdr *rtm;
13    struct sockaddr *sa, *rti_info[RTAX_MAX];
14    struct sockaddr_in *sin;

15    if (argc != 2)

```

```

16    err_quit("usage: getrt <Ipaddress>");

17  sockfd = Socket(AF_ROUTE, SOCK_RAW, 0); /* необходимы права
                                         привилегированного пользователя */

18  buf = Calloc(1, BUFSIZE); /* инициализируется нулем */
19  rtm = (struct rt_msghdr*)buf;
20  rtm->rtm_msflen = sizeof(struct rt_msghdr) + sizeof(struct sockaddr_in);
21  rtm->rtm_version = RTM_VERSION;
22  rtm->rtm_type = RTM_GET;
23  rtm->rtm_addrs = RTA_DST;
24  rtm->rtm_pid = pid = getpid();
25  rtm->rtm_seq = SEQ;

26  sin = (struct sockaddr_in*)(rtm + 1);
27  sin->sin_len = sizeof(struct sockaddr_in);
28  sin->sin_family = AF_INET;
29  Inet_pton(AF_INET, argv[1], &sin->sin_addr);

30  Write(sockfd, rtm, rtm->rtm_msflen);
31  do {
32      n = Read(sockfd, rtm, BUFSIZE);
33  } while (rtm->rtm_type != RTM_GET || rtm->rtm_seq != SEQ ||
34      rtm->rtm_pid != pid);

```

1-3 Наш заголовочный файл `unproute.h` подключает некоторые необходимые файлы, а затем включает наш файл `unpr.h`. Константа `BUFSIZE` — это размер буфера, который мы размещаем в памяти для хранения нашего сообщения ядру вместе с ответом ядра. Нам необходимо место для одной структуры `rt_msghdr` и, возможно, восьми структур адреса сокета (максимальное число, которое может возвратиться через маршрутизирующий сокет). Поскольку структура адреса сокета IPv6 имеет размер 28 байт, то значения 512 нам более чем достаточно.

Создание маршрутизирующего сокета

17 Мы создаем символьный сокет в домене `AF_ROUTE`, что, как мы отмечали ранее, может потребовать прав привилегированного пользователя. Буфер размещается в памяти и инициализируется нулем.

Заполнение структуры `rt_msghdr`

18-25 Мы заполняем структуру `rt_msghdr` данными нашего запроса. В этой структуре хранится идентификатор процесса и порядковый номер, который мы выбираем. Мы сравним эти значения, когда будем искать правильный ответ.

Заполнение структуры адреса сокета адресом получателя

26-29 Следом за структурой `rt_msghdr` мы создаем структуру `sockaddr_in`, содержащую IPv4-адрес получателя, поиск которого будет проведен ядром в таблице маршрутизации. Все, что мы задаем — это длина адреса, семейство адреса и адрес.

Запись сообщения ядру (функция `write`) и чтение ответа (функция `read`)

30-34 Мы передаем сообщение ядру с помощью функции `write`, и с помощью функции `read` читаем ответ. Поскольку у других процессов могут быть открытые маршрутизирующие сокеты, а ядро передает

копию всех маршрутизирующих сообщений всем маршрутизирующем сокетам, мы должны проверить тип сообщения, порядковый номер и идентификатор процесса, чтобы узнать, что полученное сообщение — это ожидаемое нами сообщение.

Вторая часть этой программы показана в листинге 18.4. Она обрабатывает ответ.

Листинг 18.4. Вторая часть программы, запускающая команду RTM_GET на маршрутизирующем сокете

```
//route/getrt.c
35  rtm = (struct rt_msghdr*)buf;
36  sa = (struct sockaddr*)(rtm + 1);
37  get_rtaddrs(rtm->rtm_addrs, sa, rti_info);
38  if ((sa = rti_info[RTAX_DST]) != NULL)
39    printf("dest: %s\n", Sock_ntop_host(sa, sa->sa_len));

40  if ((sa = rti_info[RTAX_GATEWAY]) != NULL)
41    printf("gateway: %s\n", Sock_ntop_host(sa, sa->sa_len));
42  if ((sa = rti_info[RTAX_NETMASK]) != NULL)
43    printf("netmask: %s\n", Sock_masktop(sa, sa->sa_len));

44  if ((sa = rti_info[RTAX_GENMASK]) != NULL)
45    printf("genmask: %s\n", Sock_masktop(sa, sa->sa_len));

46  exit(0);
47 }
```

34-35 Указатель `rtm` указывает на структуру `rt_msghdr`, а указатель `sa` — на первую следующую за ней структуру адреса сокета.

36 `rtm_addrs` — это битовая маска той из возможных восьми структур адреса сокета, которая следует за структурой `rt_msghdr`. Наша функция `get_rtaddrs` (она показана в следующем листинге), получив эту маску и указатель на первую структуру адреса сокета (`sa`), заполняет массив `rti_info` указателями на соответствующие структуры адреса сокета. В предположении, что ядро возвращает все четыре структуры адреса сокета, показанные на рис. 18.1, полученный в результате массив `rti_info` будет таким, как показано на рис. 18.2.

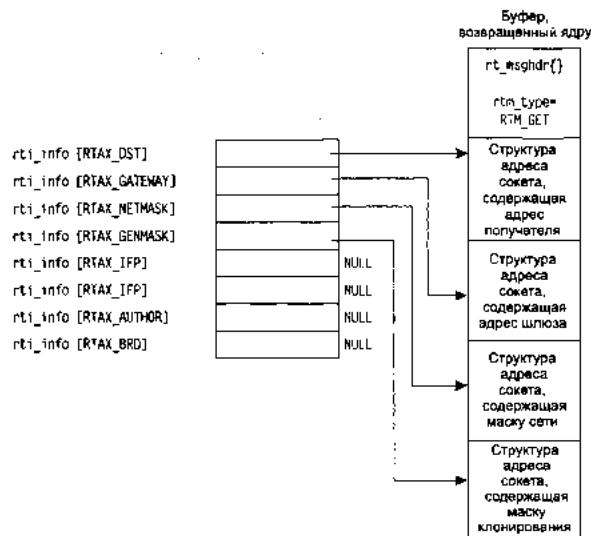


Рис. 18.2. Структура `rti_info`, заполненная с помощью нашей функции `get_rtaddrs`

Затем наша программа проходит массив `rti_info`, делая все, что ей нужно, с непустыми указателями массива.

37-44 Каждый из присутствующих четырех возможных адресов выводится. Мы вызываем нашу функцию `sock_ntop_host` для вывода адреса получателя и адреса шлюза, но для вывода двух масок подсети вызываем нашу функцию `sock_masktop`. Эту новую функцию мы покажем далее.

В листинге 18.5 показана наша функция `get_rtaddrs`, которую мы вызывали в листинге 18.4.

Листинг 18.5. Создание массива указателей на структуры адреса сокета в маршрутизирующем сообщении

```
//libroute/get_rtaddrs.c
1 #include "unproute.h"

2 /*
3  * Округляем 'a' до следующего значения, кратного 'size'
4  */
5 #define ROUNDUP(a, size) (((a) & ((size)-1)) ? (1 + ((a) | ((size)-1))) : (a))

6 /* Переходим к следующей структуре адреса сокета.
7  * Если sa_len равно 0, это значит, что
8  * размер выражен числом типа u_long).
9  */
10 #define NEXT_SA(ap) ap = (SA*) \
11 ((caddr_t)ap + (ap->sa_len ? ROUNDUP(ap->sa_len, sizeof(u_long)) : \
12 sizeof(u_long)))

13 void
14 get_rtaddrs(int addrs, SA *sa, SA **rti_info)
15 {
16     int i;

17     for (i = 0; i < RTAX_MAX; i++) {
18         if (addrs & (1 << i)) {
19             rti_info[i] = sa;
20             NEXT_SA(sa);
21         } else
22             rti_info[1] = NULL;
23     }
24 }
```

Цикл по восьми возможным указателям

Значение `RTAX_MAX` — максимальное число структур адреса сокета, возвращаемых от ядра в сообщении через маршрутизирующий сокет — равно 8. В цикле функции ведется поиск по каждой из восьми констант битовой маски `RTA_xxx` (см. табл. 18.2), которые могут быть присвоены элементам `rtm_addrs`, `ifm_addrs` и `ifam_addrs` структур, показанных в листинге 18.2. Если бит установлен, соответствующий элемент в массиве `rti_info` становится указателем на структуру адреса сокета; иначе элемент массива становится пустым указателем.

Переход к следующей структуре адреса сокета

2-12 Структуры адреса сокета имеют переменную длину, но в этом коде считается, что у каждой из них имеется поле `sa_len`, задающее длину структуры. Есть две сложности, с которыми придется столкнуться. Во-первых, маска подсети и маска клонирования могут возвращаться в структуре адреса сокета с нулевым значением поля `sa_len`, но на самом деле они занимают размер, представленный числом типа `unsigned long` (В главе 19 [128] обсуждается свойство клонирования таблицы маршрутизации 4.4BSD.) Это значение соответствует маске, состоящей только из нулевых битов, что мы видели в одном из приведенных выше примеров, когда для заданного по умолчанию маршрута маска подсети имела вид `0.0.0.0`. Во-вторых, каждая структура адреса сокета может быть заполнена в конце таким образом, что следующая начнется на определенной границе, которая в данном случае соответствует значению типа `unsigned long` (например, 4-байтовая граница для 32-разрядной архитектуры). Хотя структуры `sockaddr_in` занимают 16 байт и не требуют заполнения, маски часто имеют в конце заполнение.

Последняя функция, которую мы покажем в примере нашей программы, — это функция `sock_masktop`, представленная в листинге 18.6, возвращающая строку для одного из двух возможных значений масок. Маски хранятся в структурах адреса сокета. Элемент `sa_family` не задан, но имеется элемент `sa_len`, принимающий значения 0, 5, 6, 7 или 8 для 32-битовых масок IPv4. Когда длина больше нуля, действительная маска начинается с того же смещения от начала структуры, что и адрес IPv4 в структуре `sockaddr_in`: 4 байта от начала структуры (как показано на рис. 18.21 [128]), что соответствует элементу `sa_data[2]` общей структуры адреса сокета.

Листинг 18.6. Преобразование значения маски к формату представления

```
//libroute/sock_masktop.c
1 #include "unproute.h"

2 const char*
3 sock_masktop(SA *sa, socklen_t salen)
4 {
5     static char str[INET6_ADDRSTRLEN];
6     unsigned char *ptr = &sa->sa_data[2];

7     if (sa->sa_len == 0)
8         return ("0.0.0.0");
9     else if (sa->sa_len == 5)
10        snprintf(str, sizeof(str), "%d.0.0.0", *ptr);
11    else if (sa->sa_len == 6)
12        snprintf(str, sizeof(str), "%d.%d.0.0", *ptr, *(ptr + 1));
13    else if (sa->sa_len == 7)
14        snprintf(str, sizeof(str), "%d.%d.%d.0", *ptr, *(ptr + 1), *(ptr + 2));
15    else if (sa->sa_len == 8)
16        snprintf(str, sizeof(str), "%d.%d.%d.%d",
17                  *ptr, *(ptr + 1), *(ptr + 2), *(ptr + 3));
18    else
19        snprintf(str, sizeof(str), "(unknown mask, len = %d, family = %d)",
20                  sa->sa_len, sa->sa_family);
21    return (str);
22 }
```

7-21 Если длина равна нулю, то подразумевается маска 0.0.0.0. Если длина равна 5, хранится только первый байт 32-разрядной маски, а для оставшихся трех байтов подразумевается нулевое значение. Когда длина равна 8, хранятся все 4 байта маски.

В этом примере мы хотим прочитать ответ ядра, поскольку он содержит информацию, которую мы ищем. Но в общем случае возвращаемое значение нашей функции `write` на маршрутизирующем сокете сообщает нам, успешно ли была выполнена команда. Если это вся необходимая нам информация, мы вызываем функцию `shutdown` со вторым аргументом `SHUT_RD`, чтобы предотвратить отправку ответа. Например, если мы удаляем маршрут, то возвращение нуля функцией `write` означает успешное выполнение, а если удалить маршрут не удалось, возвращается ошибка `ESRCH` [128, с. 608]. Аналогично, когда добавляется маршрут, возвращение ошибки `EEXIST` при выполнении функции `write` означает, что запись уже существует. В нашем примере из листинга 18.3 функция `write` возвращает ошибку `ESRCH`, если записи в таблице маршрутизации не существует (допустим, у нашего узла нет заданного по умолчанию маршрута).

18.4. Операции функции `sysctl`

Маршрутизирующие сокеты нужны нам главным образом для проверки таблицы маршрутизации и списка интерфейсов при помощи функции `sysctl`. В то время как создание маршрутизирующего сокета (символьного сокета в домене `AF_ROUTE`) требует прав привилегированного пользователя, проверить таблицу маршрутизации и список интерфейсов с помощью функции `sysctl` может любой процесс.

```
#include <sys/param.h>
#include <sys/sysctl.h>
```

```

int sysctl(int *name, u_int namelen, void *oldp, size_t *oldlenp,
           void *newp, size_t newlen);
Возвращает: 0 в случае успешного выполнения

```

Эта функция использует имена, похожие на имена базы управляющей информации (Management Information Base, MIB) простого протокола управления сетью (Simple Network Management Protocol, SNMP). В главе 25 [111] подробно описываются SNMP и его MIB. Эти имена являются иерархическими.

Аргумент name — это массив целых чисел, задающий имя, а namelen задает число элементов массива. Первый элемент массива определяет, какой подсистеме ядра направлен запрос. Второй элемент определяет некоторую часть этой подсистемы, и т.д. На рис. 18.3 показана иерархическая организация с некоторыми константами, используемыми на первых трех уровнях.

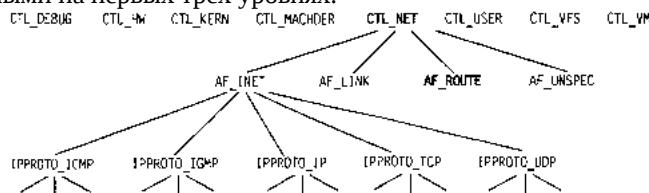


Рис. 18.3. Иерархическая организация имен функции sysctl

Для получения значений используется аргумент oldp. Он указывает на буфер, в котором ядро сохраняет значение. Аргумент oldlenp имеет тип «значение-результат»: когда функция вызывается, значение, на которое указывает oldlenp, задает размер этого буфера, а по завершении функции значением этого аргумента становится количество данных, сохраненных ядром в буфере. Если размера буфера недостаточно, возвращается ошибка ENOMEM. В специальном случае oldp может быть пустым указателем, а oldlenp — непустым указателем, и тогда ядро определяет, сколько данных возвратилось бы при вызове, сообщая это значение через oldlenp.

Чтобы установить новое значение, используется аргумент newp, указывающий на буфер размера newlen. Если новое значение не задается, newp должен быть пустым указателем, а newlen должен быть равен нулю.

В руководстве (man) по применению функции sysctl подробно описывается различная системная информация, которую можно получить с помощью этой функции: информация о файловых системах, виртуальной памяти, ограничениях ядра, аппаратных характеристиках и т.д. Нас интересует сетевая подсистема, на которую указывает первый элемент массива name, равный CTL_NET (константы CTL_xxx определяются в заголовочном файле <sys/sysctl.h>). Тогда второй элемент может быть одним из перечисленных ниже.

- **AF_INET.** Получение или установка переменных, влияющих на протоколы Интернета. Следующий уровень с помощью одной из констант IPPROTO_xxx задает протокол. BSD/OS 3.0 предоставляет на этом уровне около 30 переменных, управляющих такими свойствами, как генерация ядром переадресации ICMP, использование параметров TCP из RFC 1323, отправка контрольных сумм UDP и т.д. Пример подобного применения функции sysctl мы покажем в конце этого раздела.

- **AF_LINK.** Получение или установка информации канального уровня, такой как число интерфейсов PPP.

- **AF_ROUTE.** Возвращение информации либо о таблице маршрутизации, либо о списке интерфейсов. Мы вскоре опишем эту информацию.

- **AF_UNSPEC.** Получение или установка некоторых переменных уровня сокета, таких как максимальный размер буфера отправки или приема сокета.

Когда вторым элементом массива name является AF_ROUTE, третий элемент (номер протокола) всегда нулевой (поскольку протоколы внутри семейства AF_ROUTE отличаются от протоколов, например, в семействе AF_INET), четвертый элемент — это семейство адресов, а пятый и шестой элементы задают выполняемые действия. Вся эта информация обобщается в табл. 18.3.

Таблица 18.3. Информация функции sysctl, возвращаемая для маршрутизирующего домена
name[] Возвращает таблицу Возвращает кеш APR маршрутизации Возвращает список интерфейсов

0	CTL_NET	CTL_NET	CTL_NET
1	AF_ROUTE	AF_ROUTE	AF_ROUTE
2	0	0	0
3	AF_INET	AF_INET	AF_INET

4	NET_RT_DUMP	NET_RT_FLAGS	NET_RT_IFLIST
5	0	RTF_LLINFO	0

Поддерживаются три операции, задаваемые элементом name[4]. (Константы NET_RT_XXX определяются в заголовочном файле <sys/socket.h>.) Информация возвращается через указатель oldp при вызове функции sysctl. Этот буфер содержит переменное число сообщений RTM_XXX (см. табл. 18.1).

1. Операция NET_RT_DUMP возвращает таблицу маршрутизации для семейства адресов, заданного элементом name[3]. Если задано нулевое семейство адресов, возвращаются таблицы маршрутизации для всех семейств адресов.



Рис. 18.4. Информация возвращаемая функцией sysctl для команд CTL_NET и NET_RT_IFLIST

Таблица маршрутизации возвращается как переменное число сообщений RTM_GET, причем за каждым сообщением следует до четырех структур адреса сокета: получатель, шлюз, маска сети и маска клонирования записи в таблице маршрутизации. Пример такого сообщения мы показали в правой части рис. 18.1, а в нашем коде в листинге 18.4 проводится анализ одного из сообщений. В результате применения этой операции функции sysctl ядром возвращается одно или несколько таких сообщений.

2. Операция NET_RT_FLAGS возвращает таблицу маршрутизации для семейства адресов, заданного элементом name[3], но учитываются только те записи таблицы маршрутизации, для которых значение флага RTF_XXX равно указанному в элементе name[5]. У всех записей кэша ARP в таблице маршрутизации установлен бит флага RTF_LLINFO.

Информация возвращается в том же формате, что и в предыдущем пункте.

3. Операция NET_RT_IFLIST возвращает информацию обо всех сконфигурированных интерфейсах. Если элемент name[5] ненулевой, это номер индекса интерфейса и возвращается информация только об этом интерфейсе. (Более подробно об индексах интерфейсов мы поговорим в разделе 18.6.) Все адреса, присвоенные каждому интерфейсу, также возвращаются, и если элемент name[3] ненулевой, возвращаются только адреса для семейства адресов, указанного в этом элементе.

Для каждого интерфейса возвращается по одному сообщению RTM_IFINFO, за которым следует одно сообщение RTM_NEWADDR для каждого адреса, заданного для интерфейса. За сообщением RTM_IFINFO следует по одной структуре адреса сокета канального уровня, а за каждым сообщением RTM_NEWADDR — до трех структур адреса сокета: адрес интерфейса, маска сети и широковещательный адрес. Эти два сообщения представлены на рис. 18.4.

Пример: определяем, включены ли контрольные суммы UDP

Теперь мы приведем простой пример использования функции sysctl с протоколами Интернета для проверки, включены ли контрольные суммы UDP. Некоторые приложения UDP (например, BIND)

проверяют при запуске, включены ли контрольные суммы UDP, и если нет, пытаются включить их. Для того чтобы включить подобное свойство, требуются права привилегированного пользователя, но мы сейчас просто проверим, включено это свойство или нет. В листинге 18.7 представлена наша программа.

Листинг 18.7. Проверка включения контрольных сумм

```
//route/checkudpsum.c
1 #include "unproute.h"
2 #include <netinet/udp.h>
3 #include <netinet/ip_var.h>
4 #include <netinet/udp_var.h> /* для констант UDPCTL_xxx */

5 int
6 main(int argc, char **argv)
7 {
8     int mib[4], val;
9     size_t len;

10    mib[0] = CTL_NET;
11    mib[1] = AF_INET;
12    mib[2] = IPPROTO_UDP;
13    mib[3] = UDPCTL_CHECKSUM;

14    len = sizeof(val);
15    Sysctl(mib, 4, &val, &len, NULL, 0);
16    printf("udp checksum flag: %d\n", val);

17    exit(0);
18 }
```

Включение системных заголовков

2-4 Следует включить заголовочный файл `<netinet/udp_var.h>`, чтобы получить определение констант UDP функции `sysctl`. Для него требуются два других заголовка.

Вызов функции `sysctl`

10-16 Мы размещаем в памяти массив целых чисел с четырьмя элементами и храним константы, соответствующие иерархии, показанной на рис. 18.3. Поскольку мы только получаем переменную и не присваиваем ей значение, аргумент `newp` функции `sysctl` мы задаем как пустой указатель, и поэтому аргумент `newp` этой функции имеет нулевое значение, `oldp` указывает на нашу целочисленную переменную, в которую сохраняется результат, а `oldnp` указывает на переменную типа «значение- результат», хранящую размер этого целого числа. Мы выводим либо 0 (отключено), либо 1 (включено).

18.5. Функция `get_ifi_info` (повтор)

Вернемся к примеру из раздела 17.6 — возвращение всех активных интерфейсов в виде связного списка структур `ifi_info` (см. листинг 17.2). Программа `prifinfo` остается без изменений (см. листинг 17.3), но теперь мы покажем версию функции `get_ifi_info`, использующую функцию `sysctl` вместо вызова `SIOCGIFCONF` функции `ioctl` в листинге 17.4.

Сначала в листинге 18.8 мы представим функцию `net_rt_iflist`. Эта функция вызывает функцию `sysctl` с командой `NET_RT_IFLIST`, чтобы возвратить список интерфейсов для заданного семейства адресов.

Листинг 18.8. Вызов функции `sysctl` для возвращения списка интерфейсов

```
//libroute/net_rt_iflist.c
1 #include "unproute.h"
```

```

2 char*
3 net_rt_iflist(int family, int flags, size_t *lenp)
4 {
5     int mib[6];
6     char *buf;

7     mib[0] = CTL_NET;
8     mib[1] = AF_ROUTE;
9     mib[2] = 0;
10    mib[3] = family; /* только адреса этого семейства */
11    mib[4] = NET_RT_IFLIST;
12    mib[5] = flags; /* индекс интерфейса или 0.*/
13    if (sysctl(mib, 6, NULL, lenp, NULL, 0) < 0)
14        return (NULL);

15    if ((buf = malloc(*lenp)) == NULL)
16        return (NULL);
17    if (sysctl(mib, 6, buf, lenp, NULL, 0) < 0) {
18        free(buf);
19        return (NULL);
20    }
21    return (buf);
22 }
```

7-14 Инициализируется массив `mib`, как показано в табл. 18.3, для возвращения списка интерфейсов и всех сконфигурированных адресов заданного семейства. Затем функция `sysctl` вызывается дважды. В первом вызове функции третий аргумент нулевой, в результате чего в переменной, на которую указывает `lenp`, возвращается размер буфера, требуемый для хранения всей информации об интерфейсе.

15-21 Затем в памяти выделяется место для буфера, и функция `sysctl` вызывается снова, на этот раз с ненулевым третьим аргументом. При этом переменная, на которую указывает `lenp`, содержит при завершении функции число, равное количеству информации, хранимой в буфере, и эта переменная размещается в памятизывающим процессом. Указатель на буфер также возвращается вызывающему процессу.

ПРИМЕЧАНИЕ

Поскольку размер таблицы маршрутизации или число интерфейсов может изменяться между двумя вызовами функции `sysctl`, значение, возвращаемое при первом вызове, содержит поправочный множитель 10% [128, с. 639-640].

В листинге 18.9 показана первая половина функции `get_ifi_info`.

Листинг 18.9. Функция `get_ifi_info`, первая половина

```

//route/get_ifi_info.c
3 struct ifi_info *
4 get_ifi_info(int family, int doaliases)
5 {
6     int flags;
7     char *buf, *next, *lim;
8     size_t len;
9     struct if_msghdr *ifm;
10    struct ifa_msghdr *ifam;
11    struct sockaddr *sa, *rti_info[RTAX_MAX];
12    struct sockaddr_dl *sdl;
13    struct ifi_info *ifi, *ifisave, *ifihead, **ifipnext;
14    buf = Net_rt_iflist(family, 0, &len);
15    ifihead = NULL;
```

```

16 ifipnext = &ifihead;

17 lim = buf + len;
18 for (next = buf; next < lim; next += ifm->ifm_msflen) {
19     ifm = (struct if_msghdr*)next;
20     if (ifm->ifm_type == RTM_IFINFO) {
21         if (((flags = ifm->ifm_flags) & IFF_UP) == 0)
22             continue; /* игнорируем, если интерфейс не активен */

23     sa = (struct sockaddr*)(ifm + 1);
24     get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
25     if ((sa = rti_info[RTAX_IFP]) != NULL) {
26         ifi = Calloc(1, sizeof(struct ifi_info));
27         *ifipnext = ifi; /* предыдущий указатель указывал на эту
                           структуру */
28         ifipnext = &ifi->ifi_next; /* указатель на следующую структуру */

29         ifi->ifi_flags = flags;
30         if (sa->sa_family == AF_LINK) {
31             sdl = (struct sockaddr_dl*)sa;
32             ifi->ifi_index = sdl->sdl_index;
33             if (sdl->sdl_nlen > 0)
34                 snprintf(ifi->ifi_name, IFI_NAME, "%*s",
35                           sdl->sdl_nlen, &sdl->sdl_data[0]);
36             else
37                 snprintf(ifi->ifi_name, IFI_NAME, "index %d",
38                           sdl->sdl_index);

39             if ((ifi->ifi_hlen = sdl->sdl_alen) > 0)
40                 memcpy(ifi->ifi_haddr, LLADDR(sdl),
41                         min(IFI_HADDR, sdl->sdl_alen));
42         }
43     }

```

6-14 Мы объявляем локальные переменные и затем вызываем нашу функцию `net_rt_iflist`.

17-19 Цикл `for` — это цикл по всем сообщениям маршрутизации, попадающим в буфер в результате выполнения функции `sysctl`. Мы предполагаем, что сообщение — это структура `if_msghdr`, и рассматриваем поле `ifm_type` (вспомните, что первые три элемента трех структур идентичны, поэтому все равно, какую из трех структур мы используем для просмотра типа элемента).

Проверка, включен ли интерфейс

20-22 Для каждого интерфейса возвращается структура `RTM_IFINFO`. Если интерфейс не активен, он игнорируется.

Определение, какие структуры адреса сокета присутствуют

23-24 `sa` указывает на первую структуру адреса сокета, следующую за структурой `if_msghdr`. Наша функция `get_rtaddrs` инициализирует массив `rti_info` в зависимости от того, какие структуры адреса сокета присутствуют.

Обработка имени интерфейса

25-42 Если присутствует структура адреса сокета с именем интерфейса, в памяти размещается структура `ifi_info` и хранятся флаги интерфейса. Предполагаем семейством этой структуры адреса сокета является `AF_LINK`, что означает структуру адреса сокета канального уровня. Если элемент `sdl_nlen` ненулевой, имя интерфейса копируется в структуру `ifi_info`. В противном случае в качестве имени

хранится строка, содержащая индекс интерфейса. Если элемент `sdl_alen` ненулевой, аппаратный адрес (например, адрес Ethernet) копируется в структуру `ifi_info`, а его длина также возвращается как `ifi_hlen`.

В листинге 18.10 показана вторая часть нашей функции `get_ifi_info`, которая возвращает IP-адреса для интерфейса.

Листинг 18.10. Функция `get_ifi_info`, вторая часть

```
//route/get_ifi_info.c
44 } else if (ifm->ifm_type == RTM_NEWADDR) {
45     if (ifi->ifi_addr) { /* уже имеется IP-адрес для интерфейса */
46         if (doaliases == 0)
47             continue;
48
49     /* у нас имеется новый IP-адрес для существующего интерфейса */
50     ifisave = ifi;
51     ifi = Calloc(1, sizeof(struct ifi_info));
52     *ifipnext = ifi; /* предыдущий указатель указывал на эту
53                      структуру */
54     ifi->ifi_flags = ifi_save->ifi_flags;
55     ifi->ifi_index = ifisave->ifi_index;
56     ifi->ifi_hlen = ifisave->ifi_hlen;
57     memcpy(ifi->ifi_name, ifisave->ifi_name, IFI_NAME);
58     memcpy(ifi->ifi_haddr, ifisave->ifi_haddr, IFI_HADDR);
59 }
60
61 ifam = (struct ifa_msghdr*)next;
62 sa = (struct sockaddr*)(ifam + 1);
63 get_rtaddrs(ifam->ifam_addrs, sa, rti_info);
64
65 if ((sa = rti_info[RTAX_IFA]) != NULL) {
66     ifi->ifi_addr = Calloc(1, sa->sa_len);
67     memcpy(ifi->ifi_addr, sa, sa->sa_len);
68 }
69
70 if ((flags & IFF_BROADCAST) && (sa = rti_info[RTAX_BRD]) != NULL) {
71     ifi->ifi_brdaddr = Calloc(1, sa->sa_len);
72     memcpy(ifi->ifi_brdaddr, sa, sa->sa_len);
73 }
74
75 } else
76     err_quit("unexpected message type %d", ifm->ifm_type);
77 }
78 /* "ifihead" указывает на первую структуру в связном списке */
79 return (ifihead); /* указатель на первую структуру в связном списке */
80 }
```

Возвращение IP-адресов

44-65 Сообщение `RTM_NEWADDR` возвращается функцией `sysctl` для каждого адреса, связанного с интерфейсом: для первичного адреса и для всех альтернативных имен (псевдонимов). Если мы уже заполнили IP-адрес для этого интерфейса, то мы имеем дело с альтернативным именем. Поэтому если

вызывающему процессу нужен адрес псевдонима, мы должны выделить память для другой структуры `ifi_info`, скопировать заполненные поля и затем заполнить возвращенный адрес.

Возвращение широковещательного адреса и адреса получателя

66-75 Если интерфейс поддерживает широковещательную передачу, возвращается широковещательный адрес, а если интерфейс является интерфейсом типа «точка-точка», возвращается адрес получателя.

18.6. Функции имени и индекса интерфейса

Документ RFC 3493 [36] определяет четыре функции, обрабатывающие имена и индексы интерфейсов. Эти четыре функции используются во многих случаях, когда необходимо описать интерфейс. Они были предложены в процессе разработки API IPv6 (главы 21 и 27), однако индексы интерфейсов имеются и в API IPv4 (например, в вызове `IP_RECVIF` или `AF_LINK` для маршрутизирующего сокета). Основной принцип, объявляемый в этом документе, состоит в том, что каждый интерфейс имеет уникальное имя и уникальный положительный индекс (нуль в качестве индекса никогда не используется).

```
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname);
Возвращает: положительный индекс интерфейса в случае успешного выполнения, 0 в случае ошибки
```

```
char *if_indextoname(unsigned int ifindex, char *ifname);
Возвращает: указатель на имя интерфейса в случае успешного выполнения, NULL в случае ошибки
```

```
struct if_nameindex *if_nameindex(void);
Возвращает: непустой указатель в случае успешного выполнения, NULL в случае ошибки

void if_freenameindex(struct if_nameindex *Iptr);
Функция if_nametoindex возвращает индекс интерфейса, имеющего имя ifname. Функция if_indextoname возвращает указатель на имя интерфейса, если задан его индекс ifindex. Аргумент ifname указывает на буфер размера IFNAMSIZ (определенный в заголовочном файле <net/if.h> из листинга 17.1), который вызывающий процесс должен выделить для хранения результата, и этот указатель возвращается в случае успешного выполнения функции if_indextoname.
```

Функция `if_nameindex` возвращает указатель на массив структур `if_nameindex`:

```
struct if_nameindex {
    unsigned int if_index; /* 1, 2, ... */
    char *if_name; /* имя, завершающееся нулем: "le0", ... */
};
```

Последняя запись в этом массиве содержит структуру с нулевым индексом `if_index` и с пустым указателем `ifname`. Память для этого массива, а также для имен, на которые указывают элементы массива, выделяется динамически и освобождается при вызове функции `if_freenameindex`.

Теперь мы представим реализацию этих четырех функций с использованием маршрутизирующих сокетов.

Функция if_nametoindex

В листинге 18.11 показана функция `if_nametoindex`.

Листинг 18.11. Возвращение индекса интерфейса по его имени

```
//libroute/if_nametoindex.c
1 #include "unpifi.h"
2 #include "unproute.h"
```

```

3 unsigned int
4 if_nametoindex(const char *name)
5 {
6     unsigned int idx, namelen;
7     char *buf, *next, *lim;
8     size_t len;
9     struct if_msghdr *ifm;
10    struct sockadd *sa, *rti_info[RTAX_MAX];
11    struct sockaddr_dl *sdl;

12    if ((buf = net_rt_iflist(0, 0, &len)) == NULL)
13        return(0);

14    namelen = strlen(name);
15    lim = buf + len;
16    for (next = buf; next < lim; next += ifm->ifm_msglen) {
17        ifm = (struct if_msghdr*)next;
18        if (ifm->ifm_type == RTM_IFINFO) {
19            sa = (struct sockaddr*)(ifm + 1);
20            get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
21            if ((sa = rti_info[RTAX_IFP]) != NULL) {
22                if (sa->sa_family == AF_LINK) {
23                    sdl = (struct sockaddr_dl*)sa;
24                    if (sdl->sdl_nlen == namelen
25                        && strncmp(&sdl->sdl_data[0], name,
26                        sdl->sdl_nlen) == 0) {
27                        idx = sdl->sdl_index; /* сохранение перед
                                         вызовом free */
28                        free(buf);
29                        return(idx);
30                    }
31                }
32            }
33        }
34    }
35    free(buf);
36    return(0); /* индекс для имени не найден */
37 }

```

Получение списка интерфейсов

12-13 Наша функция `net_rt_iflist` возвращает список интерфейсов.

Обработка только сообщений RTM_IFINFO

17-30 Мы обрабатываем сообщения в буфере (см. рис. 18.4) в поисках сообщений типа `RTM_IFINFO`. Найдя такое сообщение, мы вызываем нашу функцию `get_rtaddrs`, чтобы установить указатели на структуры адреса сокета, а если присутствует структура имени интерфейса (элемент `RTAX_IFP` массива `rti_info`), то имя интерфейса сравнивается с аргументом.

Функция `if_indextoname`

Следующая функция, `if_indextoname`, показана в листинге 18.12.

Листинг 18.12. Возвращение имени интерфейса по его индексу

```
libroute/if_indextoname.c
1 #include "unpifi.h"
2 #include "unproute.h"

3 char*
4 if_indextoname(unsigned int index, char *name)
5 {
6     char *buf, *next, *lim;
7     size_t len;
8     struct if_msghdr *ifm;
9     struct sockaddr *sa, *rti_info[RTAX_MAX];
10    struct sockaddr_dl *sdl;

11    if ((buf = net_rt_iflist(0, index, &len)) == NULL)
12        return (NULL);

13    lim = buf + len;
14    for (next = buf; next < lim; next += ifm->ifm_msglen) {
15        ifm = (struct if_msghdr*)next;
16        if (ifm->ifm_type == RTM_IFINFO) {
17            sa = (struct sockaddr*)(ifm + 1);
18            get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
19            if ((sa = rti_info[RTAX_IFP]) != NULL) {
20                if (sa->sa_family == AF_LINK) {
21                    sdl = (struct sockaddr_dl*)sa;
22                    if (sdl->sdl_index == index) {
23                        int slen = min(IFNAMSIZ - 1, sdl->sdl_nlen);
24                        strncpy(name, sdl->sdl_data, slen);
25                        name[slen] = 0; /* завершающий нуль */
26                        free(buf);
27                        return (name);
28                    }
29                }
30            }
31        }
32    }
33    free(buf);
34    return (NULL); /* нет соответствия индексу */
35 }
```

Эта функция практически идентична предыдущей, но вместо поиска имени интерфейса мы сравниваем индекс интерфейса с аргументом вызывающего процесса. Кроме того, второй аргумент нашей функции `net_rt_iflist` — это заданный индекс, поэтому результат должен содержать информацию только для определенного интерфейса. Когда обнаруживается совпадение, возвращается имя интерфейса, к которому добавляется завершающий нуль.

Функция `if_nameindex`

Следующая функция, `if_nameindex`, возвращает массив структур `if_nameindex`, содержащих все имена интерфейсов и индексы. Она показана в листинге 18.13.

Листинг 18.13. Возвращение всех имен и индексов интерфейсов

```
//libroute/if_nameindex.c
1 #include "unpifi.h"
2 #include "unproute.h"
```

```

3 struct if_nameindex*
4 if_nameindex(void)
5 {
6     char *buf, *next, *lim;
7     size_t len;
8     struct if_msghdr *ifm;
9     struct sockaddr *sa, *rti_info[RTAX_MAX];
10    struct sockaddr_dl *sdl;
11    struct if_nameindex *result, *ifptr;
12    char *namptr;

13    if ((buf = net_it_iflist(0, 0, &len)) == NULL)
14        return (NULL);

15    if ((result = malloc(len)) == NULL) /* завышенная оценка */
16        return (NULL);
17    ifptr = result;
18    namptr = (char*)result + len; /* имена начинаются с конца буфера */

19    lim = buf + len;
20    for (next = buf; next < lim; next += ifm->ifm_msflen) {
21        ifm = (struct if_msghdr*)next;
22        if (ifm->ifm_type == RTM_IFINFO) {
23            sa = (struct sockaddr*)(ifm + 1);
24            get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
25            if ((sa = rti_info[RTAX_IFP]) != NULL) {
26                if (sa->sa_family == AF_LINK) {
27                    sdl = (struct sockaddr_in*)sa;
28                    namptr -= sdl->sdl_nlen + 1;
29                    strncpy(namptr, &sdl->sdl_data[0], sdl->sdl_nlen);
30                    namptr[sdl->sdl_nlen] = 0; /* завершающий нуль */
31                    ifptr->if_name = namptr;
32                    ifptr->if_index = sdl->sdl_index;
33                    ifptr++;
34                }
35            }
36        }
37    }
38    ifptr->if_name = NULL; /* отмечаем конец массива структур */
39    ifptr->if_index = 0;
40    free(buf);
41    return (result); /* вызывающий процесс должен освободить память
                       с помощью free(), когда все сделано */
43 }

```

Получение списка интерфейсов, выделение места для результата

13-18 Мы вызываем нашу функцию `net_rt_iflist` для возвращения списка интерфейсов. Мы также используем возвращаемый размер в качестве размера буфера, который мы размещаем в памяти для записи массива возвращаемых структур `if_nameindex`. Оценка необходимого размера буфера несколько завышена, но это проще, чем проходить список интерфейсов дважды: один раз для подсчета числа интерфейсов и общего размера имен, а второй — для записи этой информации. Мы создаем массив `if_nameindex` в начале этого буфера и записываем имена интерфейсов, начиная с конца буфера.

Обработка только сообщений RTM_IFINFO

22-36 Мы обрабатываем все сообщения, ища сообщения RTM_IFINFO и следующие за ними структуры адреса сокета. Имя и индекс интерфейса записываются в создаваемый нами массив.

Завершение массива

38-39 Последняя запись в массиве имеет пустой указатель if_name и нулевой индекс.

Функция if_freetenameindex

Последняя функция, показанная в листинге 18.13, освобождает память, которая была выделена для массива структур if_nameindex и хранящихся в нем имен.

Листинг 18.14. Освобождение памяти, выделенной функцией if_nameindex

```
43 void
44 if_freetenameindex(struct if_nameindex *ptr)
45 {
46     free(ptr);
47 }
```

Эта функция тривиальна, поскольку мы хранили и массив структур, и имена в одном и том же буфере. Если бы мы каждый раз вызывали функцию malloc, то для освобождения памяти нам бы пришлось проходить через весь массив, освобождать память, выделенную для каждого имени, а затем удалять сам массив (используя функцию free).

18.7. Резюме

Последняя из структур адреса сокета, с которой мы встретились в книге, это sockaddr_dl — структура адреса сокета канального уровня, имеющая переменную длину. Ядра Беркли-реализаций связывают их с интерфейсами, возвращая в одной из этих структур индекс интерфейса, его имя и аппаратный адрес.

В маршрутизирующий сокет процессом могут быть записаны 5 типов сообщений, и 12 различных сообщений могут быть асинхронно возвращены ядром через маршрутизирующий сокет. Мы привели пример, когда процесс запрашивает у ядра информацию о записи в таблице маршрутизации и ядро отвечает со всеми подробностями. Ответы ядра могут содержать до восьми структур адреса сокета, поэтому нам приходится анализировать сообщение, чтобы получить все фрагменты информации.

Функция sysctl предоставляет общий способ получения и хранения параметров операционной системы. При выполнении функции sysctl нас интересует получение следующей информации:

- список интерфейсов;
- таблица маршрутизации;
- кэш ARP.

Изменения API сокетов, требуемые IPv6, включают четыре функции для сопоставления имен интерфейсов и их индексов. Каждому интерфейсу присваивается уникальный положительный индекс. В Беркли-реализациях с каждым интерфейсом уже связан индекс, поэтому нам несложно реализовать эти функции с помощью функции sysctl.

Упражнения

1. Что, как вы считаете, будет хранить поле sdl_len в структуре адреса сокета канального уровня для устройства с именем eth10, адрес канального уровня которого является 64-разрядным адресом IEEE EUI-64?

2. В листинге 18.3 отключите параметр сокета SO_USELOOPBACK перед вызовом функции write. Что происходит?

Глава 19

Сокеты управления ключами

19.1. Введение

С появлением архитектуры безопасности IP (IPSec, см. RFC 2401 [64]) возникла потребность в стандартном механизме управления индивидуальными ключами шифрования и авторизации. Документ RFC 2367 [73] предлагает универсальный интерфейс управления ключами, который может использоваться с IPSec и иными криптографическими сетевыми службами. Подобно маршрутизирующему сокетам, этот интерфейс принадлежит к отдельному семейству протоколов, которое называется PF_KEY. В этом семействе поддерживаются только символьные сокеты.

ПРИМЕЧАНИЕ

Как отмечалось в разделе 4.2, на большинстве систем константа AF_KEY совпадает с PF_KEY. Однако в RFC 2367 совершенно четко утверждается, что с сокетами управления ключами должна использоваться константа PF_KEY.

Для открытия символьного сокета управления ключами требуются определенные привилегии. В системах с сегментированными привилегиями для этого действия должна иметься специальная привилегия. В обычных Unix-системах открывать такие сокеты может только привилегированный пользователь.

IPSec предоставляет криптографический сервис на базе *соглашений о безопасности* (*security association*, SA). Соглашение о безопасности представляет собой сочетание адресов отправителя и получателя (а при необходимости, транспортного протокола и портов), механизма (например, аутентификации) и ключей. К одному потоку трафика может относиться несколько соглашений (например, соглашения об аутентификации и о шифровании). Набор хранящихся в системе соглашений называется *базой данных соглашений о безопасности* (*security association database*, SADB).

База SADB может использоваться не только для работы с IPSec. В ней могут иметься записи для OSPFv2, RIPv2, RSVP и Mobile-IP. По этой причине нельзя считать, что сокеты PF_KEY предназначены только для работы с IPSec.

Для работы IPSec необходима также *база политик безопасности* (*security policy database*, SPDB). Политики описывают требования к трафику, например: «трафик между узлами А и В должен аутентифицироваться при помощи заголовков аутентификации IPSec (*authentication header*, AH); не удовлетворяющий требованию трафик должен сбрасываться». База соглашений описывает порядок выполнения требуемых для обеспечения безопасности действий, например, если трафик между узлами А и В использует заголовки аутентификации, то в SADB содержится соответствующий алгоритм и ключи. К сожалению, стандартного механизма управления SPDB не существует. Сокеты PF_KEY работают только с базой SADB, но не с SPDB. Реализация IPSec группы KAME использует для управления SPDB расширения PF_KEY, однако никаким стандартом эти расширения не описываются.

Сокеты управления ключами поддерживают три типа операций:

- Процесс может отправить сообщение ядру и всем остальным процессам с открытыми сокетами, записав это сообщение в свой сокет. Таким способом добавляются и удаляются записи в базе соглашений о безопасности. Таким же способом процессы, обеспечивающие собственную безопасность самостоятельно (типа OSPFv2), могут запрашивать ключи у демона-ключника (демона, управляющего ключами).
- Процесс может считать сообщение от ядра или иного процесса через сокет управления ключами. Это позволяет ядру запросить демона-ключника о добавлении соглашения о безопасности для нового сеанса TCP, который, согласно политике, подлежит определенной защите.
- Процесс может отправить ядру запрос дампа, и ядро в ответ передаст ему дамп текущей базы SADB. Это отладочная функция, которая может быть доступна не во всех системах.

19.2. Чтение и запись

Все сообщения в сокете управления ключами должны иметь одинаковые заголовки, соответствующие листингу 19.1^[1]. Сообщение может сопровождаться различными расширениями в зависимости от наличия дополнительной информации или необходимости ее предоставления. Все нужные структуры определяются в заголовочном файле `<net/pfkeyv2.h>`. Все сообщения и расширения подвергаются 64-разрядному выравниванию и дополняются до длин, кратных 64 разрядам. Все поля длины оперируют 64-разрядными единицами, то есть значение длины 1 означает реальную длину 8 байт. Расширение, не содержащее достаточного количества данных, дополняется произвольным образом до длины, кратной 64 разрядам.

Значение `sadb_msg_type` задает одну из десяти команд управления ключами. Типы сообщений перечислены в табл. 19.1. За заголовком `sadb_msg` может следовать произвольное количество расширений. Большинство сообщений имеют обязательные и необязательные расширения, которые будут описаны в соответствующих разделах. Шестнадцать типов расширений с названиями структур, их определяющих, перечислены в табл. 19.3.

Листинг 19.1. Заголовок сообщения управления ключами

```
struct sadb_msg {
    u_int8_t    sadb_msg_version; /* PF_KEY_V2 */
    u_int8_t    sadb_msg_type;    /* см. табл. 19.1 */
    u_int8_t    sadb_msg_errno;   /* код ошибки */
    u_int8_t    sadb_msg_satype; /* см. табл. 19.2 */
    u_int16_t   sadb_msg_len;    /* длина заголовка и расширений / 8 */
    u_int16_t   sadb_msg_reserved; /* нуль при передаче, игнорируется
                                    при получении */
    u_int32_t   sadb_msg_seq;    /* порядковый номер */
    u_int32_t   sadb_msg_pid;    /* идентификатор процесса отправителя
                                    или получателя */
};
```

Таблица 19.1. Типы сообщений

Тип сообщения	К ядру	От ядра	Описание
SADB_ACQUIRE	•	•	Запрос на создание записи в SADB
SADB_ADD	•	•	Добавление записи в полную базу безопасности
SADB_DELETE	•	•	Удаление записи
SADB_DUMP	•	•	Дамп SADB (используется для отладки)
SADB_EXPIRE	•		Уведомление об истечении срока действия записи
SADB_FLUSH	•	•	Очистка всей базы безопасности
SADB_GET	•	•	Получение записи
SADB_GETSPI	•	•	Выделение SPI для создания записи SADB
SADB_REGISTER	•		Регистрация для ответа на SADB_ACQUIRE
SADB_UPDATE	•	•	Обновление записи в частичной SADB

Таблица 19.2. Типы соглашений о безопасности

Тип соглашения	Описание
SADB_SATYPE_AH	Аутентифицирующий заголовок IPSec
SADB_SATYPE_ESP	ESP IPSec
SADB_SATYPE_MIP	Идентификация мобильных пользователей (Mobile IP)
SADB_SATYPE OSPFv2	Аутентификация OSPFv2
SADB_SATYPE_RIPV2	Аутентификация RIPv2
SADB_SATYPE_RSVP	Аутентификация RSVP
SADB_SATYPE_UNSPECIFIED	Не определен

Таблица 19.3. Типы расширений PF_KEY

Тип заголовка расширения	Описание	Структура
SADB_EXT_ADDRESS_DST	Адрес получателя SA	sadb_address

SADB_EXT_ADDRESS_PROXY	Адрес прокси-сервера SA	sadb_address
SADB_EXT_ADDRESS_SRC	Адрес отправителя SA	sadb_address
SADB_EXT_IDENTITY_DST	Личность получателя	sadb_ident
SADB_EXT_IDENTITY_SRC	Личность отправителя	sadb_ident
SADB_EXT_KEY_AUTH	Ключ аутентификации	sadb_key
SADB_EXT_KEY_ENCRYPT	Ключ шифрования	sadb_key
SADB_EXT_LIFETIME_CURRENT	Текущее время жизни SA	sadb_lifetime
SADB_EXT_LIFETIME_HARD	Жесткое ограничение на время жизни SA	sadb_lifetime
SADB_EXT_LIFETIME_SOFT	Гибкое ограничение на время жизни SA	sadb_lifetime
SADB_EXT_PROPOSAL	Предлагаемая ситуация	sadb_prop
SADB_EXT_SA	Соглашение о безопасности	sadb_sa
SADB_EXT_SENSITIVITY	Важность SA	sadb_sens
SADB_EXT_SPIRANGE	Диапазон допустимых значений SPI	sadb_spirange
SADB_EXT_SUPPORTED_AUTH	Поддерживаемые алгоритмы аутентификации	sadb_supported
SADB_EXT_SUPPORTED_ENCRYPT	Поддерживаемые алгоритмы шифрования	sadb_supported

Рассмотрим несколько примеров сообщений и расширений, используемых в типичных операциях с сокетами управления ключами.

19.3. Дамп базы соглашений о безопасности

Для дампа текущей базы соглашений о безопасности используется сообщение SADB_DUMP. Это самое простое из сообщений, поскольку оно не требует никаких расширений, а состоит только из 16-байтового заголовка sadb_msg. Когда процесс отправляет сообщение SADB_DUMP ядру через сокет управления ключами, ядро отвечает последовательностью сообщений SADB_DUMP по тому же сокету. В каждом сообщении содержится одна запись базы SADB. Конец последовательности обозначается сообщением со значением 0 в поле sadb_msg_seq.

Поле sadb_msg_satype позволяет запросить только записи определенного типа. Значения этого поля следует брать из табл. 19.2. При указании значения SADB_SATYPE_UNSPEC возвращаются все записи базы. Не все типы соглашений о безопасности поддерживаются всеми реализациями. Реализация KAME поддерживает только соглашения, относящиеся к IPSec (SADB_SATYPE_AH и SADB_SATYPE_ESP), поэтому при попытке получить дамп записей SADB_SATYPE_RIPV2 будет возвращена ошибка EINVAL. Если же записей, относящихся к запрошенному типу, в таблице нет (но они поддерживаются), функция возвращает ошибку ENOENT.

Программа, получающая записи из базы данных безопасности, приведена в листинге 19.2.

Листинг 19.2. Дамп базы соглашений о безопасности

```
//key/dump.c
1 void
2 sadb_dump(int type)
3 {
4     int s;
5     char buf[4096];
6     struct sadb_msg msg;
7     int goteof;

8     s = Socket(PF_KEY, SOCK_RAW, PF_KEY_V2);

9     /* формирование и отправка запроса SADB_DUMP */
10    bzero(&msg, sizeof(msg));
11    msg.sadb_msg_version = PF_KEY_V2;
12    msg.sadb_msg_type = SADB_DUMP;
13    msg.sadb_msg_satype = type;
14    msg.sadb_msg_len = sizeof(msg) / 8;
15    msg.sadb_msg_pid = getpid();
```

```

16 printf("Sending dump message:\n");
17 print_sadb_msg(&msg, sizeof(msg));
18 Write(s, &msg, sizeof(msg));

19 printf("\nMessages returned:\n");
20 /* считывание и вывод всех ответов SADB_DUMP */
21 goteof = 0;
22 while (goteof == 0) {
23     int msglen;
24     struct sadb_msg *msgp;

25     msglen = Read(s, &buf, sizeof(buf));
26     msgp = (struct sadb_msg*)&buf;
27     print_sadb_msg(msgp, msglen);
28     if (msgp->sadb_msg_seq == 0)
29         goteof = 1;
30 }
31 close(s);
32 }

33 int
34 main(int argc, char **argv)
35 {
36     int satype = SADB_SATYPE_UNSPEC;
37     int c;

38     opterr = 0; /* отключение записи в stderr для getopt() */
39     while ((c = getopt(argc, argv, "t:")) != -1) {
40         switch (c) {
41             case 't':
42                 if ((satype = getsatypename(optarg)) == -1)
43                     err_quit("invalid -t option %s", optarg);
44                 break;

45             default:
46                 err_quit("unrecognized option: %c", c);
47         }
48     }

49     sadb_dump(satype);
50 }

```

В этом листинге мы впервые встречаемся с функцией `getopt`, определяемой стандартом POSIX. Третий аргумент представляет собой строку символов, которые могут быть приняты в качестве аргументов командной строки: в нашем случае только `t`. За символом следует двоеточие, означающее, что за ключом должно быть указано численное значение. В программах, которые могут принимать несколько аргументов, эти аргументы должны объединяться. Например, в листинге 29.3 соответствующая строка имеет вид `0i:1:v`. Это означает, что ключи `i` и `l` сопровождаются дополнительными аргументами, а `0` и `v` — не сопровождаются.

Эта функция работает с четырьмя глобальными переменными, определенными в заголовочном файле `<unistd.h>`.

```

extern char *optarg;
extern int optind, opterr, optopt;

```

Перед вызовом `getopt` мы устанавливаем `opterr` в нуль, чтобы функция не направляла сообщений об ошибках в стандартный поток вывода этих сообщений, потому что мы хотим обрабатывать их самостоятельно. В стандарте POSIX говорится, что если первый символ третьего аргумента функции —

двоеточие, то это тоже должно отключать вывод сообщений в стандартный поток сообщений об ошибках, однако не все реализации в настоящий момент выполняют данное требование.

Открытие сокета PF_KEY

1-8 Сначала мы открываем сокет PF_KEY. Для этого требуются определенные привилегии, поскольку сокет дает доступ к управлению ключами.

Формирование запроса SADB_DUMP

9-15 Мы начинаем с обнуления структуры `sadb_msg`, что позволяет нам не инициализировать поля, которые должны остаться нулевыми. Затем мы заполняем все интересующие нас поля по отдельности.

Если при открытии сокета в качестве третьего аргумента использовалась константа `PF_KEY_V2`, все сообщения, направляемые в такой сокет, должны иметь версию `PF_KEY_V2`. Нужный нам тип сообщения — `SADB_DUMP`. Длина сообщения устанавливается равной длине заголовка без расширений, поскольку для запроса дампа расширения не нужны. Наконец, идентификатор процесса устанавливается равным идентификатору нашего процесса. Это обязательное условие для всех сообщений.

Отображение и отправка сообщения SADB_DUMP

16-18 Мы отображаем сообщение при помощи функции `print_sadb_msg`. Мы не приводим листинг этой функции, потому что он достаточно длинный и не представляет особого интереса, однако он включен в набор свободно распространяемых программ, доступный для скачивания с сайта этой книги. Функция принимает сообщение, подготовленное к отправке или полученное от ядра, и выводит всю содержащуюся в этом сообщении информацию в удобной для чтения форме.

После вызова функции подготовленное сообщение записывается в сокет.

Чтение ответов

19-30 Программа считывает сообщения и выводит их в цикле при помощи функции `print_sadb_msg`. Последнее сообщение последовательности имеет порядковый номер 0, что мы трактуем как «конец файла».

Закрытие сокета PF_KEY

31 Мы закрываем открытый в начале работы сокет управления ключами.

Обработка аргументов командной строки

38-48 На долю функции `main` остается не так уж много работы. Программа принимает единственный аргумент — тип соглашений о безопасности, которые должны быть запрошены из базы. По умолчанию тип равен `SADB_SATYPE_UNSPEC`. Указав другой тип в аргументе командной строки, пользователь может выбрать интересующие его записи. Наша программа вызывает нашу же функцию `getsatypebyname`, возвращающую значение типа (константу) по его названию.

Вызов функции `sadb_dump`

49 Наконец, мы вызываем функцию `sadb_dump`, которая уже была описана.

Пробный запуск

Ниже приведен пример выполнения программы дампа базы данных безопасности в системе с двумя статическими соглашениями о безопасности.

```
macosx % dump Sending dump message:  
SADB Message Dump, errno 0, satype Unspecified, seq 0, pid 20623  
Messages returned:  
SADB Message Dump, errno 0. satype IPsec AH, seq 1, pid 20623  
SA: SPI=258 Replay Window=0 State=Mature  
Authentication Algorithm: HMAC-MD5  
Encryption Algorithm: None  
[unknown extension 19]  
Current lifetime:  
0 allocations, 0 bytes  
added at Sun May 18 16:28:11 2003, never used  
Source address: 2.3.4.5/128 (IP proto 255)  
Dest address: 6.7.8.9/128 (IP proto 255)  
Authentication key, 128 bits: 0x20202020202020202020202020202023  
SADB Message Dump, errno 0, satype IPsec AH, seq 0, pid 20623  
SA: SPI=257 Replay Window=0 State=Mature  
Authentication Algorithm: HMAC-MD5  
Encryption Algorithm: None  
[unknown extension 19]  
Current lifetime:  
0 allocations, 0 bytes  
added at Sun May 18 16:26:24 2003, never used  
Source address: 1.2.3.4/128 (IP proto 255)  
Dest address: 5.6.7.8/128 (IP proto 255)  
Authentication key, 128 bits: 0x1010101010101001010101010101010101010101
```

19.4. Создание статического соглашения о безопасности

Наиболее прямолинейным методом добавления соглашения о безопасности в базу является отправка сообщения SADB_ADD с заполненными параметрами, которые могут задаваться вручную. Последнее затрудняет смену ключей, которая необходима для предотвращения криптоаналитических атак, но зато упрощает настройку. Элис и Боб договариваются о ключах и алгоритмах без использования линий связи. Мы приводим последовательность действий по созданию и отправке сообщения SADB_ADD.

Сообщение SADB_ADD обязано иметь три расширения: соглашение о безопасности, адрес и ключ. Оно может дополняться и другими расширениями: временем жизни, личными данными и параметром *важности* (*sensitivity*). Сначала мы опишем обязательные расширения.

Расширение SA описывается структурой sadb_sa, представленной в листинге 19.3.

Листинг 19.3. Расширение SA

```
struct sadb_sa {  
    u_int16_t sadb_sa_len;      /* длина расширения / 8 */  
    u_int16_t sadb_sa_exttype; /* SADB_EXT_SA */  
    u_int32_t sadb_sa_spi;     /* индекс параметров безопасности (SPI) */  
    u_int8_t  sadb_sa_replay;   /* размер окна защиты от повторов или нуль */  
    u_int8_t  sadb_sa_state;   /* состояние SA. см. табл. 19.4 */  
    u_int8_t  sadb_sa_auth;    /* алгоритм аутентификации, см. табл. 19.5 */  
    u_int8_t  sadb_sa_encrypt; /* алгоритм шифрования, см. табл. 19.5 */  
    u_int32_t sadb_sa_flags;   /* флаги */  
};
```

Таблица 19.4. Использование расширений

Состояние SA	Описание	Возможность
--------------	----------	-------------

		использования
SADB_SASTATE_LARVAL	В процессе создания	Нет
SADB_SASTATE_MATURE	Полностью сформированное	Да
SADB_SASTATE_DYING	Превышено гибкое ограничение на время жизни	Да
SADB_SASTATE_DEAD	Превышено жесткое ограничение на время жизни	Нет

Таблица 19.5. Алгоритмы аутентификации и шифрования

Алгоритм	Описание	Ссылка
SADB_AALG_NONE	Без аутентификации	
SADB_AALG_MD5HMAC	HMAC-MD5-96	RFC 2403
SADB_AALG_SHA1HMAC	HMAC-SHA-1-96	RFC 2404
SADB_EALG_NONE	Без шифрования	
SADB_EALG_DESCBC	DES-CBC	RFC 2405
SADB_EALG_3DESCBC	3DES-CBC	RFC 1851
SADB_EALG_NULL	NULL	RFC 2410

Поле `sadb_sa_spi` содержит индекс параметров безопасности (*security parameters index, SPI*). Это значение вместе с адресом получателя и используемым протоколом (например, IPSec AH) уникально идентифицирует соответствующее соглашение о безопасности. При получении пакета значение SPI используется для поиска соглашения, относящегося к пакету. При отправке пакета значение помещается в него для использования получателем. Никаких иных значений SPI не имеет, поэтому назначаться индекс может последовательно, в случайном порядке или с использованием метода, рекомендуемого собеседником. Поле `sadb_sa_replay` задает размер окна защиты от повторов. Поскольку статические соглашения о защите не дают возможности задействовать эту защиту, мы устанавливаем поле равным нулю. Значение поля `sadb_sa_state` меняется в зависимости от состояния динамически создаваемых соглашений о безопасности (см. табл. 19.4). Создаваемые вручную соглашения существуют исключительно в состоянии SADB_SASTATE_MATURE. С другими состояниями мы встретимся в разделе 19.5.

Поля `sadb_sa_auth` и `sadb_sa_encrypt` определяют алгоритмы аутентификации и шифрования для данного соглашения. Возможные значения этих полей перечислены в табл. 19.5. Единственное значение поля `sadb_sa_flags` определено в POSIX как константа `SADB_SAFLAGS_PFS`. Этот флаг требует *совершенной безопасности пересылки* (*perfect forward security*), которая состоит в том утверждении, что значение ключа не должно зависеть от предыдущих подключений или какого-либо главного ключа. Флаг используется при запросе ключей у приложения, заведующего ими, но не при создании статических соглашений.

Следующее обязательное расширение команды `SADB_ADD` должно содержать адреса отправителя и получателя, задаваемые константами `SADB_EXT_ADDRESS_SRC` и `SADB_EXT_ADDRESS_DST`. При необходимости может быть указан адрес прокси-сервера `SADB_EXT_ADDRESS_PROXY`. Подробнее об обработке адресов прокси-серверов вы можете прочесть в RFC 2367 [73]. Адреса задаются в структуре `sadb_address`, представленной в листинге 19.4. Поле `sadb_address_exttype` определяет тип адреса (отправителя, получателя или прокси-сервера). Поле `sadb_address_proto` позволяет выбрать протокол IP или произвольный протокол (значение 0). Поле `sadb_address_prefixlen` описывает значимый префикс адреса. Это позволяет использовать одно соглашение для множества адресов. За структурой `sadb_address` следует структура `sockaddr` соответствующего семейства (например, `sockaddr_in` или `sockaddr_in6`). Номер порта из структуры `sockaddr` используется только в том случае, если поле `sadb_address_proto` задает протокол, поддерживающий номера портов (например, `IPPROTO_TCP`).

Листинг 19.4. Структура `sadb_address`

```
struct sadb_address {
    u_int16_t sadb_address_len;      /* длина расширения с адресом / 8 */
    u_int16_t sadb_address_exttype;  /* SADB_EXT_ADDRESS_{SRC,DST,PROXY} */
    u_int8_t  sadb_address_proto;    /* протокол IP или 0 (любой) */
    u_int8_t  sadb_address_prefixlen; /* # значащих битов адреса */
    u_int16_t sadb_address_reserved; /* зарезервирован для послед. использования */
};

/* далее следует структура sockaddr соответствующего семейства */
```

Завершают список обязательных расширений сообщения SADB_ADD ключи аутентификации и шифрования — расширения SADB_EXT_KEY_AUTH и SADB_EXT_KEY_ENCRYPT, описываемые структурой `sadb_key` (листинг 19.5). Поле `sadb_key_exttype` определяет тип ключа (ключ аутентификации или шифрования), поле `sadb_key_bits` задает длину ключа в битах, а сам ключ следует за структурой `sadb_key`.

Листинг 19.5. Структура `sadb_key`

```
struct sadb_key {  
    u_int16_t sadb_key_len;      /* длина расширения с ключом / 8 */  
    u_int16_t sadb_key_exttype; /* SADB_EXT_KEY_{AUTH,ENCRYPT} */  
    u_int16_t sadb_key_bits;     /* # битов в ключе */  
    u_int16_t sadb_key_reserved; /* зарезервировано для расширения */  
};
```

```
/* далее следуют данные о самом ключе */
```

Программа, добавляющая статическую запись в базу данных безопасности, представлена в листинге 19.6.

Листинг 19.6. Программа, использующая команду SADB_ADD

```
//key/add.c  
33 void  
34 sadb_add(struct sockaddr *src, struct sockaddr *dst, int type, int alg,  
35 int spi, int keybits, unsigned char *keydata)  
36 {  
37     int s;  
38     char buf[4096], *p; /* XXX */  
39     struct sadb_msg *msg;  
40     struct sadb_sa *saext;  
41     struct sadb_address *addrext;  
42     struct sadb_key *keyext;  
43     int len;  
44     int mypid;  
  
45     s = Socket(PF_KEY, SOCK_RAW, PF_KEY_V2);  
  
46     mypid = getpid();  
  
47     /* Формирование и запись запроса SADB_ADD */  
48     bzero(&buf, sizeof(buf));  
49     p = buf;  
50     msg = (struct sadb_msg*)p;  
51     msg->sadb_msg_version = PF_KEY_V2;  
52     msg->sadb_msg_type = SADB_ADD;  
53     msg->sadb_msg_satype = type;  
54     msg->sadb_msg_pid = getpid();  
55     len = sizeof(*msg);  
56     p += sizeof(*msg);  
  
57     saext = (struct sadb_sa*)p;  
58     saext->sadb_sa_len = sizeof(*saext) / 8;  
59     saext->sadb_sa_exttype = SADB_EXT_SA;  
60     saext->sadb_sa_spi = htonl(spi);  
61     saext->sadb_sa_replay = 0; /* статические ключи не защищают от повтора */  
62     saext->sadb_sa_state = SADB_SASTATE_MATURE;  
63     saext->sadb_sa_auth = alg;  
64     saext->sadb_sa_encrypt = SADB_EALG_NONE;  
65     saext->sadb_sa_flags = 0;  
66     len += saext->sadb_sa_len * 8;  
67     p += saext->sadb_sa_len * 8;
```

```

68 addrext = (struct sadb_address*)p;
69 addrext->sadb_address_len = (sizeof(*addrext) + salen(src) + 7) / 8;
70 addrext->sadb_address_exttype = SADB_EXT_ADDRESS_SRC;
71 addrext->sadb_address_proto = 0; /* any protocol */
72 addrext->sadb_address_prefixlen = prefix_all(src);
73 addrext->sadb_address_reserved = 0;
74 memcpy(addrext + 1, src, salen(src));
75 len += addrext->sadb_address_len * 8,
76 p += addrext->sadb_address_len * 8;

77 addrext = (struct sadb_address*)p;
78 addrext->sadb_address_len = (sizeof(*addrext) + salen(dst) + 7) / 8;
79 addrext->sadb_address_exttype = SADB_EXT_ADDRESS_DST;
80 addrext->sadb_address_proto = 0; /* any protocol */
81 addrext->sadb_address_prefixlen = prefix_all(dst);
82 addrext->sadb_address_reserved = 0;
83 memcpy(addrext + 1, dst, salen(dst));
84 len += addrext->sadb_address_len * 8;
85 p += addrext->sadb_address_len * 8;

86 keyext = (struct sadb_key*)p;
87 /* обеспечивает выравнивание */
88 keyext->sadb_key_len = (sizeof(*keyext) + (keybits / 8) + 7) / 8;
89 keyext->sadb_key_exttype = SADB_EXT_KEY_AUTH;
90 keyext->sadb_key_bits = keybits;
91 keyext->sadb_key_reserved = 0;
92 memcpy(keyext + 1, keydata, keybits / 8);
93 len += keyext->sadb_key_len * 8;
94 p += keyext->sadb_key_len * 8;

95 msg->sadb_msg_len = len / 8;
96 printf("Sending add message:\n");
97 print_sadb_msg(buf, len);
98 Write(s, buf, len);

99 printf("\nReply returned:\n");
100 /* считывание и вывод ответа SADB_ADD, игнорируя любые другие */
101 for (;;) {
102     int msglen;
103     struct sadb_msg *msgp;

104     msglen = Read(s, &buf, sizeof(buf));
105     msgp = (struct sadb_msg*)&buf;
106     if (msgp->sadb_msg_pid == mypid &&
107         msgp->sadb_msg_type == SADB_ADD) {
108         print_sadb_msg(msgp, msglen);
109         break;
110     }
111 }
112 close(s);
113 }

```

Открытие сокета PF_KEY и сохранение PID

55-56 Как и в предыдущей программе, мы открываем сокет PF_KEY и сохраняем идентификатор нашего процесса для последующего его использования.

Формирование общего заголовка сообщений

47-56 Мы формируем заголовок сообщения SADB_ADD. Поле sadb_msg_len устанавливается непосредственно перед отправкой сообщения, поскольку оно должно соответствовать истинной его длине. В переменной len хранится текущая длина сообщения, а указатель p всегда указывает на первый неиспользуемый байт буфера.

Добавление расширения SA

57-67 Мы добавляем обязательное расширение SA (см. листинг 19.3). Поле sadb_sa_spi должно иметь сетевой порядок байтов, поэтому нам приходится применять функцию htonl к значению в порядке байтов узла. Мы отключаем защиту от повторов и устанавливаем состояние SA равным SADB_SASTATE_MATURE (см. табл. 19.4). Алгоритм аутентификации выбирается в соответствии с аргументом командной строки, а шифрование отключается при помощи константы SADB_EALG_NONE.

Добавление адреса отправителя

68-76 К сообщению добавляется расширение SADB_EXT_ADDRESS_SRC, содержащее адрес отправителя для соглашения о безопасности.

Значение протокола устанавливается равным нулю, что подразумевает действительность соглашения для всех протоколов. Длина префикса устанавливается равной соответствующей длине версии IP (то есть 32 разряда для IPv4 и 128 разрядов для IPv6). При расчете значения поля длины мы добавляем к реальному значению число 7 перед делением на 8, что гарантирует выравнивание по 64-разрядной границе, обязательное для всех расширений, передаваемых через сокеты PF_KEY. Структура sockaddr копируется в буфер после заголовка расширения.

Добавление адреса получателя

77-85 Адрес получателя добавляется в сообщение SADB_EXT_ADDRESS_DST. Процедура в точности совпадает с описанной выше.

Добавление ключа

86-94 К сообщению добавляется расширение SADB_EXT_KEY_AUTH, содержащее ключ авторизации. Расчет поля длины производится точно так же, как и для обоих адресов. Ключ переменной длины требует соответствующего количества дополняющих нулей. Мы устанавливаем значение количества битов и копируем ключ вслед за заголовком расширения.

Запись сообщения в сокет

95-98 Мы выводим сообщение на экран вызовом функции print_sadb_msg, после чего записываем его в сокет.

Считывание ответа

99-111 Мы считываем все сообщения из сокета до тех пор, пока не будет получено сообщение, адресованное нашему процессу (проверяется по PID) и имеющее тип SADB_ADD. Это сообщение выводится на экран функцией `print_sadb_msg`, после чего программа завершает работу.

Пример

Мы запускаем программу, требуя от нее установки соглашения о безопасности, касающегося трафика между узлами 127.0.0.1 и 127.0.0.1 (то есть локального трафика):

```
macosx % add 127.0.0.1 127.0.0.1 HMAC-SHA-1-96 160 \
0123456789abcdef0123456789abcdef01234567
Sending add message:
SADB Message Add, errno 0, satype IPsec AH, seq 0, pid 6246
SA: SPI=39030 Replay Window=0 State=Mature
Authentication Algorithm: HMAC-SHA-1
Encryption Algorithm: None
Source address: 127.0.0.1/32
Dest address: 127.0.0.1/32
Authentication key. 160 bits: 0x0123456789abcdef0123456789abcdef01234567
Reply returned:
SADB Message Add, errno 0, satype IPsec AH, seq 0, pid 6246
SA: SPI=39030 Replay Window=0 State=Mature
Authentication Algorithm: HMAC-SHA-1
Encryption Algorithm: None
Source address: 127.0.0.1/32
Dest address: 127.0.0.1/32
```

Обратите внимание, что в ответе системы отсутствует ключ. Дело в том, что ответ направляется на все сокеты PF_KEY, которые, однако, могут принадлежать к разным доменам, а данные о ключах не должны передаваться между доменами. После добавления записи в базу данных мы даем команду `ping 127.0.0.1`, чтобы проверить, задействуется ли соглашение о безопасности, после чего запрашиваем дамп базы данных и смотрим, что в ней изменилось.

```
macosx % dump
Sending dump message:
SADB Message Dump, errno 0, satype Unspecified, seq 0, pid 6283
Messages returned:
SADB Message Dump, errno 0, satype IPsec AH, seq 0, pid 6283
SA: SPI=39030 Replay Window=0 State=Mature
Authentication Algorithm: HMAC-SHA-1
Encryption Algorithm: None
[unknown extension 19]
Current lifetime:
36 allocations. 0 bytes
added at Thu Jun 5 21:01:31 2003, first used at Thu Jun 5 21:15:07 2003
Source address: 127.0.0.1/128 (IP proto 255)
Dest address: 127.0.0.1/128 (IP proto 255)
Authentication key. 160 bits: 0x0123456789abcdef0123456789abcdef01234567
```

Из этого дампа видно, что ядро изменило значение протокола с 0 на 255. Это артефакт реализации, а не общее свойство сокетов PF_KEY. Кроме того, ядро изменило длину префикса с 32 на 128. Это какая-то проблема, связанная с протоколами IPv4 и IPv6. Ядро возвращает расширение (с номером 19), которое не обрабатывается нашей программой выведения дампа. Неизвестные расширения пропускаются (их длина имеется в соответствующем поле). Наконец, возвращается расширение времени жизни (листинг 19.7), содержащее информацию о текущем времени жизни соглашения о безопасности.

Листинг 19.7. Структура расширения времени жизни

```
struct sadb_lifetime {
    u_int16_t sadb_lifetime_len;      /* длина расширения / 8 */
    u_int16_t sadb_lifetime_exttype; /* SADB_EXT_LIFETIME_{SOFT,HARD,CURRENT} */
```

```

u_int32_t sadb_lifetime_allocations; /* количество соединений, конечных
                                       точек или потоков */
u_int64_t sadb_lifetime_bytes; /* количество байтов */
u_int64_t sadb_lifetime_addtime; /* время создания либо время от создания
                                   до устаревания */
u_int64_t sadb_lifetime_usetime; /* время первого использования или время от
                                   первого использования до устаревания */

};

Расширения времени жизни бывают трех типов. Расширения SADB_LIFETIME_SOFT и
SADB_LIFETIME_HARD задают гибкое и жесткое ограничения на время жизни соглашения. Сообщение
SADB_EXPIRE отправляется ядром в случае превышения гибкого ограничения на время жизни. После
достижения жесткого ограничения использование соглашения прекращается. Расширение
SADB_LIFETIME_CURRENT возвращается в ответ на SADB_DUMP, SADB_EXPIRE и SADB_GET и описывает
соответствующие параметры текущего соглашения.

```

19.5. Динамическое управление SA

Для повышения безопасности требуется периодическая смена ключей. Обычно для этого используется протокол типа IKE (RFC 2409 [43]).

ПРИМЕЧАНИЕ

В момент написания этой книги рабочая группа IETF по IPSec разрабатывала замену для протокола IKE.

Демон, обеспечивающий безопасность, регистрируется в ядре при помощи сообщения SADB_REGISTER, указывая в поле sadb_msg_satype (см. табл. 19.2) тип соглашения о безопасности, которое он умеет обрабатывать. Если демон может работать с несколькими типами соглашений, он должен отправить несколько сообщений SADB_REGISTER, зарегистрировав в каждом из них ровно один тип SA. В ответном сообщении SADB_REGISTER ядро указывает поддерживаемые алгоритмы шифрования или аутентификации (в отдельном расширении), а также длины ключей для этих алгоритмов. Расширение поддерживаемых алгоритмов описывается структурой sadb_supported, представленной в листинге 19.8. Структура содержит заголовок, за которым следуют описания алгоритма шифрования или аутентификации в полях sadb_alg.

Листинг 19.8. Структура, описывающая поддерживаемые алгоритмы

```

struct sadb_supported {
    u_int16_t sadb_supported_len;      /* длина расширения и списка алгоритмов / 8 */
    u_int16_t sadb_supported_exttype; /* SADB_EXT_SUPPORTED_{AUTH,ENCRYPT} */
    u_int32_t sadb_supported_reserved; /* зарезервировано для расширения в будущем */

};

/* далее следует список алгоритмов */
struct sadb_alg {
    u_int8_t  sadb_alg_id;           /* идентификатор алгоритма из табл. 19.5 */
    u_int8_t  sadb_alg_ivlen;        /* длина IV или нуль */
    u_int16_t sadb_alg_minbits;     /* минимальная длина ключа */
    u_int16_t sadb_alg_maxbits;     /* максимальная длина ключа */
    u_int16_t sadb_alg_reserved;    /* зарезервировано для расширения в будущем */
};

```

После заголовка sadb_supported следует по одной структуре sadb_alg для каждого алгоритма, поддерживаемого системой. На рис. 19.1 представлен возможный ответ на сообщение, регистрирующее обработчик SA типа SADB_SATYPE_ESP.

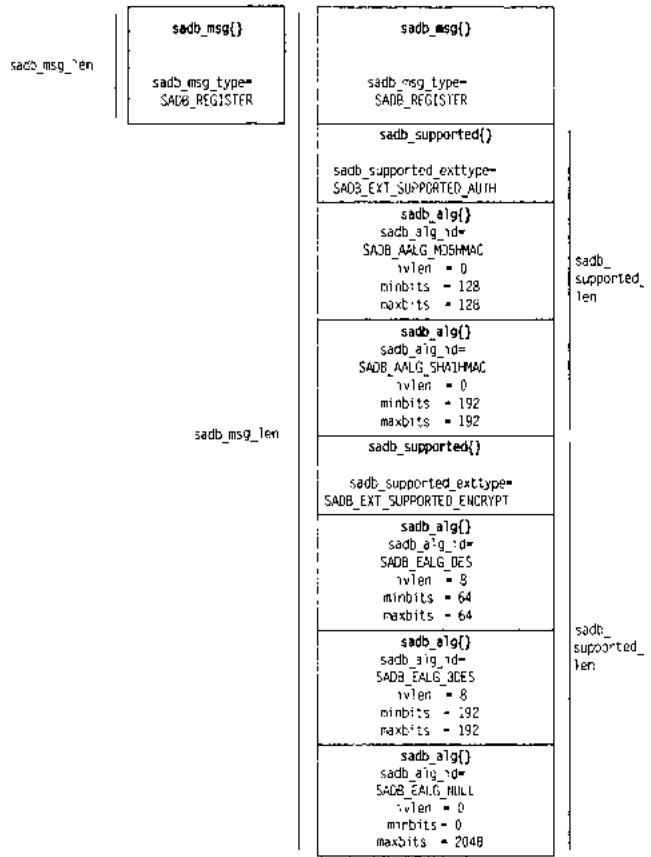


Рис. 19.1. Данные, возвращаемые ядром в ответ на команду SADB_REGISTER

Программа, представленная в листинге 19.9, просто регистрируется в ядре в качестве обработчика заданного механизма безопасности и выводит ответ ядра, содержащий список поддерживаемых алгоритмов.

Листинг 19.9. Регистрация демона-обработчика

```
//key/register.c
1 void
2 sadb_register(int type)
3 {
4     int s;
5     char buf[4096]; /* XXX */
6     struct sadb_msg msg;
7     int goteof;
8     int mypid;

9     s = Socket(PF_KEY, SOCK_RAW, PF_KEY_V2);

10    mypid = getpid();

11    /* формирование и отправка запроса SADB_REGISTER */
12    bzero(&msg, sizeof(msg));
13    msg.sadb_msg_version = PF_KEY_V2;
14    msg.sadb_msg_type = SADB_REGISTER;
15    msg.sadb_msg_satype = type;
16    msg.sadb_msg_len = sizeof(msg) / 8;
17    msg.sadb_msg_pid = mypid;
18    printf("Sending register message:\n");
19    print_sadb_msg(&msg, sizeof(msg));
```

```
20 Write(s, &msg, sizeof(msg));  
21 printf("\nReply returned:\n");  
22 /* Чтение и вывод ответа SADB_REGISTER, игнорирование всех прочих  
сообщений */  
23 for (;;) {  
24     int msglen;  
25     struct sadb_msg *msgp;  
26     msglen = Read(s, &buf, sizeof(buf));  
27     msgp = (struct sadb_msg*)&buf;  
28     if (msgp->sadb_msg_pid == mypid &&  
29         msgp->sadb_msg_type == SADB_REGISTER) {  
30         print_sadb_msg(msgp, msglen);  
31         break;  
32     }  
33 }  
34 close(s);  
35 }
```

Открытие сокета PF_KEY

1-9 Мы открываем сокет PF_KEY.

Сохранение PID

10 Поскольку ядро будет адресовать нам свои сообщения по идентификатору процесса, нам необходимо сохранить его, чтобы иметь возможность впоследствии отбирать интересующие нас сообщения.

Создание сообщения SADB_REGISTER

11-17 Подобно SADB_DUMP, сообщение SADB_REGISTER не требует никаких расширений. Мы обнуляем сообщение, после чего заполняем интересующие нас поля структуры.

Вывод и отправка сообщения

18-20 Мы отображаем подготовленное сообщение на экране при помощи функции `print_sadb_msg`, после чего записываем сообщение в сокет.

Ожидание ответа

23-30 Мы считываем сообщения из сокета, ожидая ответа на наше сообщение о регистрации. Ответ адресован по идентификатору процесса и представляет собой сообщение SADB_REGISTER. Он содержит список поддерживаемых алгоритмов, который выводится нашей функцией `print_sadb_msg`.

Пример

Мы запускаем программу `register` в системе, поддерживающей на несколько протоколов больше, чем описано в RFC 2367.

```
macosx % register -t ah
Sending register message:
SADB Message Register, errno 0, satype IPsec AH, seq 0, pid 20746
Reply returned:
SADB Message Register, errno 0, satype IPsec AH, seq 0, pid 20746
Supported authentication algorithms:
HMAC-MD5 ivlen 0 bits 128-128
HMAC-SHA-1 ivlen 0 bits 160-160
Keyed MD5 ivlen 0 bits 128-128
Keyed SHA-1 ivlen 0 bits 160-160
Null ivlen 0 bits 0-2048
SHA2-256 ivlen 0 bits 256-256
SHA2-384 ivlen 0 bits 384-384
SHA2-512 ivlen 0 bits 512-512
Supported encryption algorithms:
DES-CBC ivlen 8 bits 64-64
3DES-CBC ivlen 8 bits 192-192
Null ivlen 0 bits 0-2048
Blowfish-CBC ivlen 8 bits 40-448
CAST128-CBC ivlen 8 bits 40-128
AES ivlen 16 bits 128-256
```

Если ядру требуется связаться с собеседником, а соответствующая политика требует наличия соглашения о безопасности, но соглашение таковое отсутствует, ядро отправляет на зарегистрировавшиеся для данного типа соглашения сокеты управления ключами сообщение SADB_ACQUIRE, в расширениях которого содержатся предлагаемые ядром алгоритмы и длины ключей. Предложение может представлять собой комбинацию поддерживаемых системой средств безопасности и политики, ограничивающей набор средств для конкретного собеседника. Алгоритмы, длины ключей и времена жизни объединяются в список в порядке предпочтительности использования. Когда демон-ключник получает сообщение SADB_ACQUIRE, он выполняет действия, необходимые для выбора ключа, удовлетворяющего одной из предложенных ядром комбинаций, и устанавливает этот ключ в ядро. Для выбора SPI из нужного диапазона демон отправляет ядру сообщение SADB_GETSPI. В ответ на это сообщение ядро создает соглашение о безопасности в состоянии SADB_SASTATE_LARVAL. Затем демон согласовывает параметры безопасности с удаленным собеседником, используя предоставленный ядром SPI, после чего отправляет ядру сообщение SADB_UPDATE для завершения создания соглашения и перевода его в рабочее состояние (SADB_SASTATE_MATURE). Динамически создаваемые соглашения обычно снабжаются гибким и жестким ограничениями на время жизни. Когда истекает один из этих сроков, ядро отправляет сообщение SADB_EXPIRE, в котором указывается, какое именно достигнуто ограничение. По достижении гибкого ограничения соглашение переходит в состояние SADB_SASTATE_DYING, в котором оно еще может использоваться, однако процессу следует получить новое соглашение. Если же достигнуто жесткое ограничение, соглашение переходит в состояние SADB_SASTATE_DEAD, в котором оно больше не может использоваться для обеспечения безопасности и должно быть удалено из базы данных.

19.6. Резюме

Сокеты управления ключами используются для взаимодействия с ядром, демонами-ключниками и другими обеспечивающими безопасность сущностями (такими как маршрутизирующие демоны). Соглашения о безопасности могут создаваться статически или динамически посредством протокола согласования ключей. Динамические ключи обычно характеризуются определенным временем жизни, по истечении которого (гибкое ограничение) демон-ключник получает соответствующее уведомление. Если соглашение не обновляется до достижения жесткого ограничения, оно становится недействительным.

Между процессами и ядром через сокет управления ключами могут передаваться сообщения десяти типов. Каждому типу сообщений сопоставляются обязательные и необязательные расширения. Все сообщения, отправляемые процессом, передаются на все открытые сокеты управления ключами (однако при этом из сообщений удаляются расширения, содержащие «уязвимые» данные).

Упражнения

1. Напишите программу, открывающую сокет PF_KEY и выводящую все получаемые через этот сокет сообщения.

2. Изучите сведения о новом протоколе, предложенном рабочей группой IETF по IPSec взамен IKE. Эти сведения находятся на странице <http://www.ietf.org/html.charters/ipsec-charter.html>.

Глава 20

Широковещательная передача

20.1. Введение

В этой главе мы расскажем о широковещательной передаче (*broadcasting*), а в следующей главе — о многоадресной передаче (*multicasting*). Во всех предыдущих примерах рассматривалась направленная (одноадресная) передача (*unicasting*), когда процесс общается только с одним определенным процессом. Действительно, TCP работает только с адресами направленной передачи, хотя UDP и символьные сокеты поддерживают и другие парадигмы передачи. В табл. 20.1 представлено сравнение различных видов адресации.

Таблица 20.1. Различные формы адресации

Тип	IPv4	IPv6	TCP	UDP	Количество идентифицируемых интерфейсов	Количество интерфейсов, куда доставляется сообщение
Направленная передача	•	•	•	•	Один	Один
Передача наиболее подходящему узлу		•	Пока нет	•	Набор	Один из набора
Многоадресная передача	Не обязательно	•		•	Набор	Все в наборе
Широковещательная передача	•			•	Все	Все

С введением IPv6 к парадигмам адресации добавилась *передача наиболее подходящему узлу (anycasting)*. Ее вариант для IPv4 не получил широкого распространения. Он описан в RFC 1546 [88]. Передача наиболее подходящему узлу для IPv6 определяется в документе RFC 3513 [44]. Этот режим позволяет обращаться к одной (обычно «ближайшей» в некоторой метрике) из множества систем, предоставляющих одинаковые сервисы. Правильная конфигурация системы маршрутизации позволяет узлам пользоваться сервисами передачи наиболее подходящему узлу по IPv4 и IPv6 путем добавления одного и того же адреса в протокол маршрутизации в нескольких местах. Однако RFC 3513 разрешает иметь адреса такого типа только маршрутизаторам; узлы не имеют права предоставлять сервисы передачи наиболее подходящему узлу. На момент написания этой книги интерфейс API для использования адресов передачи наиболее подходящему узлу еще не определен. Архитектура IPv6 в настоящий момент находится на стадии совершенствования, и в будущем узлы, вероятно, получат возможность динамически предоставлять сервисы передачи наиболее подходящему узлу.

Вот наиболее важные положения из табл. 20.1:

- Поддержка многоадресной передачи не обязательна для IPv4, но обязательна для IPv6.
- Поддержка широковещательной передачи не обеспечивается в IPv6: любое приложение IPv4, использующее широковещательную передачу, для совместимости с IPv6 должно быть преобразовано так, чтобы использовать вместо широковещательной передачи многоадресную.
- Широковещательная и многоадресная передачи требуют наличия протокола UDP или символьного IP и не работают с TCP.

Одним из применений широковещательной передачи является поиск сервера в локальной подсети, когда известно, что сервер находится в этой локальной подсети, но его IP-адрес для направленной передачи неизвестен. Иногда эту процедуру называют *обнаружением ресурса (resource discovery)*. Другое применение — минимизация сетевого трафика в локальной сети, когда несколько клиентов взаимодействуют с одним сервером. Можно привести множество примеров интернет-приложений, использующих для этой цели широковещательную передачу. Некоторые из них используют и многоадресную передачу.

- Протокол разрешения адресов (Address Resolution Protocol, ARP). Это фундаментальная часть IPv4, а не пользовательское приложение. ARP отправляет широковещательный запрос в локальную подсеть, суть

которого такова: «Система с IP-адресом a.b.c.d, идентифицируйте себя и сообщите свой аппаратный адрес».

■ Протокол начальной загрузки (Bootstrap Protocol, BOOTP). Клиент предполагает, что сервер находится в локальной подсети, и посыпает запрос на широковещательный адрес (часто 255.255.255.255, поскольку клиент еще не знает IP-адреса, маски подсети и адреса ограниченной широковещательной передачи в этой подсети).

■ Протокол синхронизации времени (Network Time Protocol, NTP). В обычном сценарии клиент NTP конфигурируется с IP-адресом одного или более серверов, которые будут использоваться для определения времени, и опрашивает серверы с определенной частотой (с периодом 64 с или больше). Клиент обновляет свои часы, используя сложные алгоритмы, основанные на значении истинного времени (time-of-day), возвращаемом серверами, и величине периода RTT обращения к серверам. Но в широковещательной локальной сети вместо того, чтобы каждый клиент обращался к одному серверу, сервер может отправлять текущее значение времени с помощью широковещательных сообщений каждые 64 с для всех клиентов в локальной подсети, ограничивая тем самым сетевой трафик.

■ Демоны маршрутизации. Наиболее часто используемый демон маршрутизации *routed* распространяет по локальной сети широковещательные сообщения, содержащие таблицу маршрутизации. Это позволяет всем другим маршрутизаторам, соединенным с локальной сетью, получать объявления маршрутизации. При этом в конфигурацию каждого маршрутизатора не обязательно должны входить IP-адреса соседних маршрутизаторов. Это свойство также используется (многие могут отметить, что «используется неправильно») узлами локальной сети, прослушивающими объявления о маршрутизации и изменяющими в соответствии с этим свои таблицы маршрутизации.

Следует отметить, что многоадресная передача может заменить оба варианта применения широковещательной передачи (обнаружение ресурса и ограничение сетевого трафика). Проблемы широковещательной передачи мы обсудим далее в этой главе, а также в следующей главе.

20.2. Широковещательные адреса

Если мы обозначим адрес IPv4 в виде {*subnetid*, *hostid*}, где *subnetid* означает биты, относящиеся к маске сети (или префиксу CIDR), а *hostid* — все остальные биты, мы получим два типа широковещательных адресов. Поле, целиком состоящее из единичных битов, обозначим -1.

1. Широковещательный адрес подсети: {*subnetid*, -1}. Сообщение адресуется на все интерфейсы в заданной подсети. Например, в подсети 192.168.42/24 широковещательным адресом будет 192.168.42.255.

Обычно маршрутизаторы не передают широковещательные сообщения дальше из подсети [128, с. 226-227]. На рис. 20.1 изображен маршрутизатор, соединенный с двумя подсетями 192.168.42/24 и 192.168.123/24.

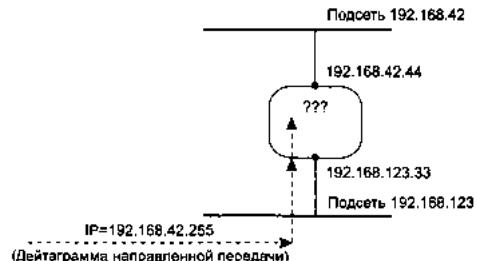


Рис. 20.1. Передает ли маршрутизатор дальше широковещательное сообщение, направленное в подсеть?

Маршрутизатор получает дейтаграмму IP направленной передачи в подсети 192.168.123/24 с адресом получателя 192.168.42.255 (адрес широковещательной передачи для подсети другого интерфейса). Обычно маршрутизатор не передает дейтаграмму дальше в подсеть 192.168.42/24. У некоторых систем имеется параметр конфигурации, позволяющий передавать широковещательные сообщения, направленные в подсеть (см. приложение E [111]).

ПРИМЕЧАНИЕ

Пересылка широковещательных сообщений, направленных в подсеть, делает возможным атаки типа «отказ в обслуживании» особого класса, получившего название «усиление» (amplification). Например, отправка эхо-запроса ICMP на широковещательный адрес подсети может привести к получению нескольких ответов. При подмене IP-адреса отправителя это дает возможность загрузить канал жертвы, снизив ее доступность. По этой причине рекомендуется отключать соответствующий параметр настройки маршрутизаторов.

Учитывая вышеизложенное, не рекомендуется писать приложения, рассчитанные на пересылку широковещательных сообщений из одной подсети в другую, за исключением тех случаев, когда приложение разрабатывается для использования в полностью контролируемой сети, где включение пересылки не приведет к нарушению безопасности.

2. Локальный широковещательный адрес: {-1,-1} или 255.255.255.255. Дейтаграммы, предназначенные для этого ограниченного адреса, никогда не передаваться маршрутизатором.

Из четырех типов широковещательных адресов адрес широкого вещания для подсети является на сегодняшний день наиболее общим. Но более старые системы продолжают отправлять дейтаграммы, предназначенные для адреса 255.255. 255.255. Кроме того, некоторые еще более старые системы не воспринимают широковещательный адрес подсети и только отправляемые на адрес 255.255.255.255 дейтаграммы интерпретируют как широковещательные.

ПРИМЕЧАНИЕ

Адрес 255.255.255.255 предназначен для использования в качестве адреса получателя во время процесса начальной загрузки такими приложениями, как DHCP и ВООТР, которым еще не известен IP-адрес узла.

Возникает вопрос: что делает узел, когда приложение посыпает дейтаграмму UDP на адрес 255.255.255.255? Большинство узлов допускают это (если процесс установил параметр сокета SO_BROADCAST) и преобразуют адрес получателя в широковещательный адрес исходящего интерфейса, направленный в подсеть. Для отправки пакета на конкретный адрес 255.255.255.255 часто приходится работать непосредственно с канальным уровнем.

Может появиться другой вопрос: что делает узел с несколькими сетевыми интерфейсами, когда приложение посыпает дейтаграмму UDP на адрес 255.255.255.255? Некоторые системы посыпают одно широковещательное сообщение с основного интерфейса (с интерфейса, который был сконфигурирован первым) с IP-адресом получателя, равным широковещательному адресу подсети этого интерфейса [128, с. 736]. Другие системы посыпают по одной копии дейтаграммы с каждого интерфейса, поддерживающего широковещательную передачу. В разделе 3.3.6 RFC 1122 [10] по этому вопросу не сказано ничего. Однако если приложению нужно отправить широковещательное сообщение со всех интерфейсов, поддерживающих широковещательную передачу, то в целях переносимости оно должно получить конфигурацию интерфейсов (см. раздел 16.6) и выполнить по одному вызову sendto для каждого из них, указав в качестве адреса получателя широковещательный адрес подсети этого интерфейса.

20.3. Направленная и широковещательная передачи

Прежде чем рассматривать широковещательную передачу, необходимо уяснить, что происходит, когда дейтаграмма UDP отправляется на адрес направленной передачи. На рис. 20.2 представлены три узла Ethernet.

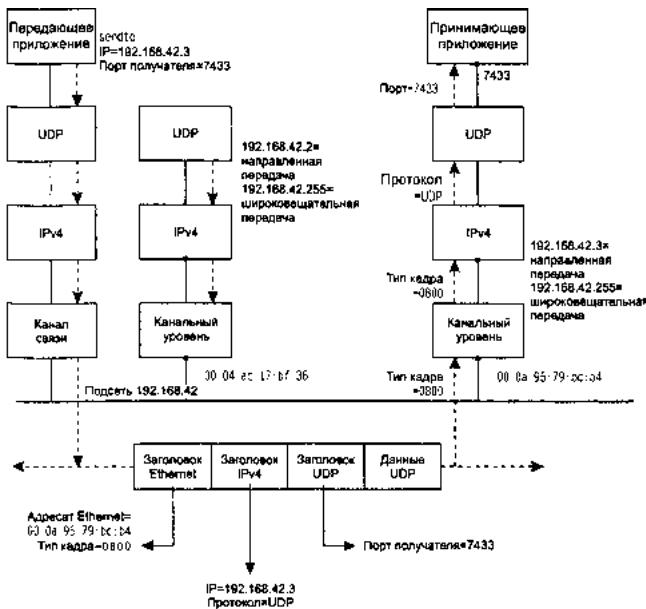


Рис. 20.2. Пример направленной передачи дейтаграммы UDP

Адрес подсети Ethernet — 192.168.42/24. 24 разряда адреса относятся к маске сети, а 8 разрядов — к идентификатору узла. Приложение на узле, изображенном слева, вызывает функцию `sendto` для сокета UDP, отправляя дейтаграмму на адрес 192.168.42.3, порт 7433. Уровень UDP добавляет в начало дейтаграммы заголовок UDP и передает дейтаграмму UDP уровню IP. IP добавляет заголовок IPv4 и определяет исходящий интерфейс. В случае использования сети Ethernet активизируется протокол ARP для определения адреса Ethernet, соответствующего IP-адресу получателя: 08:00:20:03:f6:42. Затем пакет посыпается как кадр Ethernet с 48-разрядным адресом получателя Ethernet. Поле типа кадра Ethernet будет равно 0x0800, что соответствует пакету IPv4. Тип кадра для пакета IPv6 — 0x86dd.

Интерфейс Ethernet на узле, изображенном в центре, видит проходящий кадр и сравнивает адрес получателя Ethernet со своим собственным адресом Ethernet (02:00:8c:2f:4e:00). Поскольку они не равны, интерфейс игнорирует кадр. Поскольку кадр является кадром направленной передачи, этот узел не тратит на его обработку никаких ресурсов. Интерфейс игнорирует кадр.

Интерфейс Ethernet на узле, изображенном справа, также видит проходящий кадр, и когда он сравнивает адрес получателя Ethernet со своим собственным адресом Ethernet, они оказываются одинаковыми. Этот интерфейс считывает весь кадр, возможно, генерирует аппаратное прерывание при завершении считывания кадра и драйвер устройства читает кадр из памяти интерфейса. Поскольку тип кадра — 0x0800, пакет помещается в очередь ввода IP.

Когда уровень IP обрабатывает пакет, он сначала сравнивает IP-адрес получателя (192.168.42.3) со всеми собственными IP-адресами. (Вспомним, что узел может иметь несколько сетевых интерфейсов. Также вспомним наше обсуждение модели системы с жесткой привязкой (strong end system model) и системы с гибкой привязкой (weak end system model) в разделе 8.8.) Поскольку адрес получателя — это один из собственных IP-адресов узла, пакет принимается.

Затем уровень IP проверяет поле протокола в заголовке IPv4. Его значение для UDP равно 17, поэтому далее дейтаграмма IP передается UDP.

Уровень UDP проверяет порт получателя (и, возможно, также порт отправителя, если сокет UDP является присоединенным) и в нашем примере помещает дейтаграмму в соответствующий приемный буфер сокета. При необходимости процесс возобновляется для чтения вновь полученной дейтаграммы.

Ключевым моментом на этом рисунке является то, что дейтаграмма IP при направленной передаче принимается только одним узлом, заданным с помощью IP-адреса получателя. Другие узлы подсети не задействуются в этом процессе.

Теперь мы рассмотрим похожий пример в той же подсети, но при этом приложение будет отправлять дейтаграмму UDP на широковещательный адрес для подсети 192.168.42.255. Этот пример представлен на рис. 20.3.

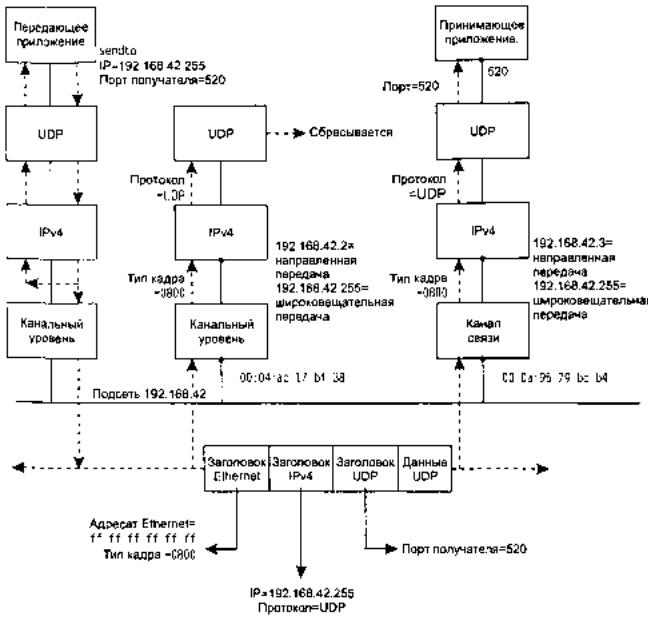


Рис. 20.3. Пример широковещательной дейтаграммы UDP

Когда узел, изображенный слева, отправляет дейтаграмму, он замечает, что IP-адрес получателя — это широковещательный адрес подсети, и сопоставляет ему адрес Ethernet, состоящий из 48 единичных битов: ff:ff:ff:ff:ff:ff. Это заставляет *каждый* интерфейс Ethernet в подсети получить кадр. Оба узла, изображенные на правой части рисунка, работающие с IPv4, получат кадр. Поскольку тип кадра Ethernet — 0800, оба узла передают пакет уровню IP. Так как IP-адрес получателя совпадает с широковещательным адресом для каждого из двух узлов, и поскольку поле протокола — 17 (UDP), оба узла передают пакет UDP.

Узел, изображенный справа, передает дейтаграмму UDP приложению, связанному с портом UDP 520. Приложению не нужно выполнять никаких специальных действий, чтобы получить широковещательную дейтаграмму UDP — оно лишь создает сокет UDP и связывает номер приложения порта с сокетом. (Предполагается, как обычно, что связанный IP-адрес — INADDR_ANY.)

Но на узле, изображенном в центре, с портом UDP 520 не связано никакое приложение. UDP этого узла игнорирует полученную дейтаграмму. Узел не должен отправлять сообщение ICMP о недоступности порта, поскольку это может вызвать *лавину широковещательных сообщений* (*broadcast storm*): ситуацию, в которой множество узлов сети генерируют ответы приблизительно в одно и то же время, в результате чего сеть просто невозможно использовать в течение некоторого времени. Кроме того, не совсем понятно, что должен предпринять получатель сообщения об ошибке: что, если некоторые получатели будут сообщать об ошибках, а другие — нет?

В этом примере мы также показываем дейтаграмму, которую изображенный слева узел доставляет сам себе. Это свойство широковещательных сообщений: по определению широковещательное сообщение идет к каждому узлу подсети, включая отправляющий узел [128, с. 109–110]. Мы также предполагаем, что отправляющее приложение связано с портом, на который оно отправляет дейтаграммы (порт 520), поэтому оно получит копию каждой отправленной им широковещательной дейтаграммы. (Однако в общем случае не требуется, чтобы процесс связывался с портом UDP, на который он отправляет дейтаграммы.)

ПРИМЕЧАНИЕ

В этом примере мы демонстрируем закольцовку, которая осуществляется либо на уровне IP, либо на канальном уровне, создающем копию [128, с. 109–110] и отправляющем ее вверх по стеку протоколов. Сеть могла бы использовать физическую закольцовку, но это может вызвать проблемы в случае сбоев сети (например, линия Ethernet без терминатора).

Этот пример отражает фундаментальную проблему, связанную с широковещательной передачей: каждый узел IPv4 в подсети, даже не выполняющий соответствующего приложения, должен полностью обрабатывать широковещательную дейтаграмму UDP при ее прохождении вверх по стеку протоколов,

включая уровень UDP, прежде чем сможет ее проигнорировать. (Вспомните наше обсуждение следом за листингом 8.11). Более того, каждый не-IP-узел в подсети (скажем, узел, на котором работает IPX Novell) должен также получать целый кадр на канальном уровне, перед тем как он сможет проигнорировать этот кадр (в данном случае мы предполагаем, что узел не поддерживает кадры определенного типа — для дейтаграммы IPv4 тип равен `0x0800`). Если приложение генерирует дейтаграммы IP с большой скоростью (например, аудио- или видеоданные), то такая ненужная обработка может серьезно повлиять на остальные узлы подсети. В следующей главе мы увидим, как эта проблема решается с помощью многоадресной передачи.

ПРИМЕЧАНИЕ

Для рис. 20.3 мы специально выбрали порт UDP 520. Это порт, используемый демоном `routed` для обмена пакетами по протоколу информации о маршрутизации (Routing Information Protocol, RIP). Все маршрутизаторы в подсети, использующие RIP, будут отправлять широковещательную дейтаграмму UDP каждые 30 секунд. Если в подсети имеется 200 узлов, в том числе два маршрутизатора, использующих RIP, то 198 узлов должны будут обрабатывать (и игнорировать) эти широковещательные дейтаграммы каждые 30 с, если ни на одном из них не запущен демон `routed`. Протокол RIP версии 2 использует многоадресную передачу именно для того, чтобы избавиться от этой проблемы.

20.4. Функция `dg_cli` при использовании широковещательной передачи

Мы еще раз изменим нашу функцию `dg_cli`, на этот раз дав ей возможность отправлять широковещательные сообщения стандартному серверу времени и даты UDP (см. табл. 2.1) и выводить все ответы. Единственное изменение, внесенное нами в функцию `main` (см. листинг 8.3), состоит в изменении номера порта получателя на 13:

```
servaddr.sin_port = htons(13);
```

Сначала мы откомпилируем измененную функцию `main` с прежней функцией `dg_cli` из листинга 8.4 и запустим ее на узле freebsd:

```
freebsd % udpccli01 192.168.42.255  
hi
```

```
sendto error: Permission denied
```

Аргумент командной строки — это широковещательный адрес подсети для присоединенной сети Ethernet. Мы вводим строку, программа вызывает функцию `sendto`, и возвращается ошибка `EACCESS`. Мы получаем ошибку, потому что нам не разрешается посыпал дейтаграмму на широковещательный адрес получателя, если мы не указали ядру явно, что будем передавать широковещательное сообщение. Мы выполняем это условие, установив параметр сокета `SO_BROADCAST` (см. табл. 7.1).

ПРИМЕЧАНИЕ

Беркли-реализации реализуют эту «защиту от дурака» (sanity check). Однако Solaris 2.5 принимает дейтаграмму, предназначенную для широковещательного адреса, даже если мы не задаем параметр сокета `SO_BROADCAST`. Стандарт POSIX требует установки параметра сокета `SO_BROADCAST` для отправки широковещательной дейтаграммы.

В 4.2BSD широковещательная передача была привилегированной операцией, и параметра сокета `SO_BROADCAST` не существовало. В 4.3BSD этот параметр был добавлен и каждому процессу стало разрешено его устанавливать.

Теперь мы изменим нашу функцию `dg_cli`, как показано в листинге 20.1^[1]. Эта версия устанавливает параметр сокета `SO_BROADCAST` и выводит все ответы, полученные в течение 5 с.

Листинг 20.1. Функция `dg_cli`, осуществляющая широковещательную передачу

```
//bcast/dgclibcast1.c  
1 #include "unp.h"
```

```

2 static void recvfrom_alarm(int);

3 void
4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
5 {
6     int n;
7     const int on = 1;
8     char sendline[MAXLINE], recvline[MAXLINE + 1];
9     socklen_t len;
10    struct sockaddr *preply_addr;

11   preply_addr = Malloc(servlen);

12   Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

13   Signal(SIGALRM, recvfrom_alarm);

14   while (Fgets(sendline, MAXLINE, fp) != NULL) {

15       Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

16       alarm(5);

17       for (;;) {
18           len = servlen;
19           n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
20           if (n < 0) {
21               if (errno == EINTR)
22                   break; /* окончание ожидания ответов */
23               else
24                   err_sys("recvfrom error");
25           } else {
26               recvline[n] = 0; /* завершающий нуль */
27               printf("from %s: %s",
28                     Sock_ntop_host(preply_addr, len), recvline);
29           }
30       }
31   }
32   free(preply_addr);
33 }

34 static void
35 recvfrom_alarm(int signo)
36 {
37     return; /* прерывание recvfrom() */
38 }

```

Выделение памяти для адреса сервера, установка параметра сокета

11-13 Функция malloc выделяет в памяти пространство для адреса сервера, возвращаемого функцией recvfrom. Устанавливается параметр сокета SO_BROADCAST, устанавливается обработчик сигнала SIGALRM.

Чтение строки, отправка сокету, чтение всех ответов

14-24 Следующие два вызова, `fgets` и `sendto`, выполняются так же, как и в предыдущих версиях этой функции. Но поскольку мы посылаем широковещательную дейтаграмму, мы можем получить множество ответов. Мы вызываем в цикле функцию `recvfrom` и выводим все ответы, полученные в течение 5 с. По истечении 5 с генерируется сигнал `SIGALRM`, вызывается наш обработчик сигнала и функция `recvfrom` возвращает ошибку `EINTR`.

Выход каждого полученного ответа

25-29 Для каждого полученного ответа мы вызываем функцию `sock_ntop_host`, которая в случае IPv4 возвращает строку, содержащую IP-адрес сервера в точечно-десятичной записи. Эта строка выводится вместе с ответом сервера.

Если мы запустим программу, задав широковещательный адрес подсети 192. 168.42.255, мы увидим следующее:

```
bsdi % udpcli01 192.168.42.255 hi
from 192.168.42.2: Sat Aug 2 16.42.45 2003
from 192.168.42.1: Sat Aug 2 14.42.45 2003
from 192.168.42.3: Sat Aug 2 14.42.45 2003
hello
from 192.168.42.3: Sat Aug 2 14.42.57 2003
from 192.168.42.2: Sat Aug 2 16.42.57 2003
from 192.168.42.1: Sat Aug 2 14.42.57 2003
```

Каждый раз мы набираем строку ввода, чтобы сгенерировать выходную дейтаграмму UDP, и каждый раз получаем три ответа, причем отвечает и отправляющий узел. Как мы отмечали ранее, получателями широковещательной дейтаграммы являются все узлы в сети, включая отправляющий. Каждый ответ является направленным, поскольку адрес отправителя запроса, используемый каждым сервером в качестве адреса получателя ответа, — это адрес направленной передачи.

Все системы сообщают одно и то же время, поскольку на них используется NTP (Network Time Protocol — протокол синхронизации времени).

Фрагментация IP-пакетов и широковещательная передача

В Беркли-ядрах фрагментация широковещательных дейтаграмм запрещена. Если размер IP-дейтаграммы, посылаемой на широковещательный адрес, превышает размер MTU исходящего интерфейса, возвращается ошибка `EMSGSIZE` [128, с. 233–234]. Эта стратегия впервые появилась в 4.2BSD. На самом деле нет никаких технических препятствий для фрагментирования широковещательных дейтаграмм, но широковещательная передача сама по себе связана со значительной нагрузкой на сеть, поэтому не стоит дополнительно увеличивать эту нагрузку, используя фрагментацию.

Можно наблюдать этот сценарий с нашей программой из листинга 20.1. Мы перенаправляем стандартный поток ввода для чтения из файла, содержащего 2000-байтовую строку, которая потребует фрагментации в Ethernet:

```
bsdi % udpcli01 192.168.42.255 < 2000line
sendto error: Message too long
```

ПРИМЕЧАНИЕ

Это ограничение реализовано в AIX, FreeBSD и MacOS. Linux, Solaris и HP-UX фрагментируют дейтаграммы, отправленные на широковещательный адрес. Однако в целях переносимости приложение, которому нужно сделать широковещательный запрос, должно определять MTU для интерфейса, через который будет отправлено сообщение, при помощи параметра `SIOCGIPMTU` функции `iocctl`, после чего вычесть размер заголовков IP и транспортного протокола. Альтернативный подход: выбрать типичное значение MTU (например, 1500 для Ethernet) и использовать его в качестве константы.

20.5. Ситуация гонок

Ситуация гонок (*race condition*) обычно возникает, когда множество процессов получают доступ к общим для них данным, но корректность результата зависит от порядка выполнения процессов. Поскольку порядок выполнения процессов в типичных системах Unix зависит от множества факторов, которые могут меняться от запуска к запуску, иногда результат корректен, а иногда — нет. Наиболее сложным для отладки типом гонок является такой, когда результат получается некорректным только изредка. Более подробно о ситуациях гонок мы поговорим в главе 26, когда будем обсуждать взаимные исключения (mutex) и условные переменные (condition variables). При программировании потоков всегда возникают проблемы с ситуациями гонок, поскольку значительное количество данных является общим для всех потоков (например, все глобальные переменные).

Ситуации гонок другого типа часто возникают при работе с сигналами. Проблемы возникают, потому что сигнал, как правило, может быть доставлен в любой момент во время выполнения нашей программы. POSIX позволяет нам блокировать доставку сигнала, но при выполнении операций ввода-вывода это часто не дает эффекта.

Чтобы понять эту проблему, рассмотрим пример. Ситуация гонок возникает при выполнении программы из листинга 20.1. Потратьте несколько минут и посмотрите, сможете ли вы ее обнаружить. (*Подсказка:* в каком месте программы мы можем находиться, когда доставляется сигнал?) Вы можете также инициировать ситуацию гонок следующим образом: изменить аргумент функции `alarm` с 5 на 1 и добавить вызов `sleep(1)` сразу же после `printf`.

Когда мы после внесения этих изменений наберем первую строку ввода, эта строка будет отправлена как широковещательное сообщение, а мы установим аргумент функции `alarm` равным 1 с. Мы блокируемся в вызове функции `recvfrom`, а затем для нашего сокета приходит первый ответ, вероятно, в течение нескольких миллисекунд. Ответ возвращается функцией `recvfrom`, но затем мы входим в спящее состояние на одну секунду. Принимаются остальные ответы и помещаются в приемный буфер сокета. Но пока мы находимся в спящем состоянии, время таймера `alarm` истекает и генерируется сигнал `SIGALRM`. При этом вызывается наш обработчик сигнала, затем он возвращает управление и прерывает функцию `sleep`, в которой мы блокированы. Далее мы повторяем цикл и читаем установленные в очередь ответы с паузой в одну секунду каждый раз, когда выводится ответ. Прочитав все ответы, мы снова блокируемся в вызове функции `recvfrom`, однако таймер уже не работает. Мы окажемся навсегда заблокированы в вызове функции `recvfrom`. Фундаментальная проблема здесь в том, что наша цель — обеспечить прерывание блокирования в функции `recvfrom` обработчиком сигнала, однако сигнал может быть доставлен в любое время, и наша программа в момент доставки сигнала может находиться в любом месте бесконечного цикла `for`.

Теперь мы проанализируем четыре различных варианта решения этой проблемы: одно некорректное и три различных корректных решения.

Блокирование и разблокирование сигнала

Наше первое (некорректное) решение снижает вероятность появления ошибки, блокируя сигнал и предотвращая его доставку, пока наша программа выполняет оставшуюся часть цикла `for`. Эта версия представлена в листинге 20.2.

Листинг 20.2. Блокирование сигналов при выполнении в цикле `for` (некорректное решение)

```
//bcast/dglibcast3.c
1 #include "unp.h"

2 static void recvfrom_alarm(int);

3 void
4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
5 {
6     int n;
7     const int on = 1;
8     char sendline[MAXLINE], recvline[MAXLINE + 1];
9     sigset(SIGALRM, sigset_alarm);
```

```

10  socklen_t len;
11  struct sockaddr *preply_addr;
12
13  preply_addr = Malloc(servlen);
14
15  Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
16
17  Sigemptyset(&sigset_alarm);
18  Sigaddset(&sigset_alarm, SIGALRM);
19
20  Signal(SIGALRM, recvfrom_alarm);
21
22  while (Fgets(sendline, MAXLINE, fp) != NULL) {
23      Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
24      alarm(5);
25      for (;;) {
26          len = servlen;
27          Sigprocmask(SIG_UNBLOCK, &sigset_alarm, NULL);
28          n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
29          Sigprocmask(SIG_BLOCK, &sigset_alarm, NULL);
30          if (n < 0) {
31              if (errno == EINTR)
32                  break; /* окончание ожидания ответа */
33              else
34                  err_sys("recvfrom error");
35          } else {
36              recvline[n] = 0; /* завершающий нуль */
37              printf("from %s: %s",
38                  Sock_ntop_host(preply_addr, len), recvline);
39      }
40  }
41  free(preply_addr);
42
43 static void
44 recvfrom_alarm(int signo)
45 {
46     return; /* выход из recvfrom() */
47 }

```

Объявление набора сигналов и инициализация

14-15 Мы объявляем набор сигналов, инициализируем его как пустой набор (`sigemptyset`) и включаем бит, соответствующий сигналу SIGALRM (`sigaddset`).

Разблокирование и блокирование сигнала

21-24 Перед вызовом функции `recvfrom` мы разблокируем сигнал (с тем, чтобы он мог быть доставлен, пока наша программа блокирована), а затем блокируем его, как только завершается функция `recvfrom`. Если сигнал генерируется (истекает время таймера), когда сигнал блокирован, то ядро запоминает этот факт, но доставить сигнал (то есть вызвать наш обработчик) не может, пока сигнал не будет разблокирован. В этом состоит принципиальная разница между *генерацией* сигнала и его *доставкой*. В главе 10 [110] предоставлена более подробная информация обо всех аспектах обработки сигналов POSIX.

Если мы откомпилируем и запустим эту программу, нам будет казаться, что она работает нормально, но все программы, порождающие ситуацию гонок, большую часть времени работают без каких-либо проблем! Проблема остается: разблокирование сигнала, вызов функции `recvfrom` и блокирование сигнала — все эти действия являются независимыми системными вызовами. Будем считать, что функция `recvfrom` возвращает последний ответ на нашу дейтаграмму, а сигнал доставляется между вызовом функции `recvfrom` и блокированием сигнала. Следующий вызов функции `recvfrom` заблокируется навсегда. Мы ограничили размер окна, но проблема осталась.

Вариант решения может быть установка глобального флага при доставке сигнала его обработчиком:

```
recvfrom_alarm(int signo) {
    had_alarm = 1;
    return;
}
```

Флаг сбрасывается в 0 каждый раз, когда вызывается функция `alarm`. Наша функция `dg_cli` проверяет этот флаг перед вызовом функции `recvfrom` и не вызывает ее, если флаг ненулевой.

```
for (;;) {
    len = servlen;
    Sigprocmask(SIG_UNBLOCK, &sigset_alrm, NULL);
    if (had_alarm == 1)
        break;
    n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
```

Если сигнал был сгенерирован во время его блокирования (после предыдущего возвращения из функции `recvfrom`), то после разблокирования в этой части кода он будет доставлен перед завершением функции `sigprocmask`, устанавливающей наш флаг. Однако между проверкой флага и вызовом функции `recvfrom` существует промежуток времени, в течение которого сигнал может быть сгенерирован и доставлен, и если это произойдет, вызов функции `recvfrom` заблокируется навсегда (разумеется, мы считаем при этом, что не приходит никаких дополнительных ответов).

Блокирование и разблокирование сигнала с помощью функции `pselect`

Одним из корректных решений будет использование функции `pselect` (см. раздел 6.9), как показано в листинге 20.3.

Листинг 20.3. Блокирование и разблокирование сигналов с помощью функции `pselect`

```
//bcast/dgclibcast4.c
1 #include "unp.h"

2 static void recvfrom_alarm(int);

3 void
4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
5 {
6     int n;
7     const int on = 1;
8     char sendline[MAXLINE], recvline[MAXLINE + 1];
9     fd_set rset;
10    sigset_t sigset_alrm, sigset_empty;
11    socklen_t len;
12    struct sockaddr *preply_addr;

13    preply_addr = Malloc(servlen);

14    Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

15    FD_ZERO(&rset);

16    Sigemptyset(&sigset_empty);
```

```

17 Sigemptyset(&sigset_alrm);
18 Sigaddset(&sigset_alrm, SIGALRM);

19 Signal(SIGALRM, recvfrom_alarm);

20 while (Fgets(sendline, MAXLINE, fp) != NULL) {
21   Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

22   Sigprocmask(SIG_BLOCK, &sigset_alrm, NULL);
23   alarm(5);
24   for (;;) {
25     FD_SET(sockfd, &rset);
26     n = pselect(sockfd + 1, &rset, NULL, NULL, NULL, &sigset_empty);
27     if (n < 0) {
28       if (errno == EINTR)
29         break;
30     else
31       err_sys("pselect error");
32   } else if (n != 1)
33     err_sys("pselect error; returned %d", n);

34   len = servlen;
35   n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
36   recvline[n] = 0; /* завершающий нуль */
37   printf("from %s: %s",
38   Sock_ntop_host(preply_addr, len), recvline);
39 }
40 }
41 free(preply_addr);
42 }

43 static void
44 recvfrom_alarm(int signo)
45 {
46   return; /* просто прерываем recvfrom() */
47 }

```

22-23 Мы блокируем сигнал SIGALRM и вызываем функцию pselect. Последний аргумент этой функции — указатель на нашу переменную sigset_empty, являющуюся набором сигналов, в котором нет блокированных сигналов (все сигналы разблокированы). Функция pselect сохранит текущую маску сигналов (которая блокирует SIGALRM), проверит заданные дескрипторы, заблокируется при необходимости с маской сигналов, установленной в пустой набор, но перед завершением функции маска сигналов процесса будет переустановлена в исходное значение, которое она имела при вызове функции pselect. Ключ к пониманию функции pselect в том, что установка маски сигналов, проверка дескрипторов и переустановка маски сигнала — это атомарные операции по отношению к вызывающему процессу.

34-38 Если наш сокет готов для чтения, мы вызываем функцию recvfrom, зная, что она не заблокируется.

Как мы упоминали в разделе 6.9, функция pselect — относительно новая среди других, описываемых спецификацией POSIX. Из всех систем, показанных на рис. 1.7, эту функцию поддерживают только FreeBSD и Linux. Тем не менее в листинге 20.4 представлена простая, хотя и некорректная ее реализация. Мы приводим ее здесь, несмотря на некорректность, чтобы продемонстрировать три стадии решения: установку маски сигнала в значение, заданное вызывающей функцией, с сохранением текущей маски, проверку дескрипторов и переустановку маски сигнала.

Листинг 20.4. Простая некорректная реализация функции pselect

```
//lib/pselect.c
9 #include "unp.h"
```

```

10 int
11 pselect(int nfds, fd_set *rset, fd_set *wset, fd_set *xset,
12 const struct timespec *ts, const sigset_t *sigmask)
13 {
14     int n;
15     struct timeval tv;
16     sigset_t savemask;

17     if (ts != NULL) {
18         tv.tv_sec = ts->tv_sec;
19         tv.tv_usec = ts->tv_nsec / 1000; /* наносекунды -> микросекунды */
20     }
21     sigprocmask(SIG_SETMASK, sigmask, &savemask); /* маска вызывающего
процесса */
22     n = select(nfds, rset, wset, xset, (ts == NULL) ? NULL : &tv);
23     sigprocmask(SIG_SETMASK, &savemask, NULL); /* восстанавливаем
исходную маску */

24     return (n);
25 }
```

Использование функций sigsetjmp и siglongjmp

Нашу проблему можно решить корректно, если отказаться от прерывания блокированного системного вызова обработчиком сигнала, вместо этого вызвав из обработчика сигнала функцию `siglongjmp`. Этот метод называется *нелокальным оператором goto (nonlocal goto)*, поскольку мы можем использовать его для перехода из одной функции в другую. В листинге 20.5 проиллюстрирована эта технология.

Листинг 20.5. Вызов функций `sigsetjmp` и `siglongjmp` из обработчика сигнала

```

//bcast/dgclibcast5.c
1 #include "unp.h"
2 #include <setjmp.h>

3 static void recvfrom_alarm(int);
4 static sigjmp_buf jmpbuf;

5 void
6 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
7 {
8     int n;
9     const int on = 1;
10    char sendline[MAXLINE], recvline[MAXLINE + 1];
11    socklen_t len;
12    struct sockaddr *preply_addr;

13    preply_addr = Malloc(servlen);

14    Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

15    Signal(SIGALRM, recvfrom_alarm);

16    while (Fgets(sendline, MAXLINE, fp) != NULL) {

17        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

18        alarm(5);
```

```

19  for (;;) {
20      if (sigsetjmp(jmpbuf, 1) != 0)
21          break;
22      len = servlen;
23      n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
24      recvline[n] = 0; /* null terminate */
25      printf("from %s: %s",
26             Sock_ntop_host(preply_addr, len), recvline);
27  }
28 }
29 free(preply_addr);
30 }

31 static void
32 recvfrom_alarm(int signo)
33 {
34     siglongjmp(jmpbuf, 1);
35 }

```

Размещение буфера перехода в памяти

4 Мы выделяем буфер перехода, который будет использовать наша функция и ее обработчик сигнала.

Вызов функции *sigsetjmp*

20-23 Когда мы вызываем функцию *sigsetjmp* непосредственно из нашей функции *dg_cli*, она устанавливает буфер перехода и возвращает нуль. Мы продолжаем работать дальше и вызываем функцию *recvfrom*.

Обработка сигнала *SIGALRM* и вызов функции *siglongjmp*

31-35 Когда сигнал доставлен, мы вызываем функцию *siglongjmp*. Это заставляет *sigsetjmp* в функции *dg_cli* возвратить значение, равное второму аргументу (1), который должен быть ненулевым. Это приведет к завершению цикла *for* в функции *dg_cli*.

Использование функций *sigsetjmp* и *siglongjmp* подобным образом гарантирует, что мы не останемся навсегда блокированы в вызове функции *recvfrom* из-за доставки сигнала в неподходящее время. Однако такое решение создает иную потенциальную проблему. Если сигнал доставляется в тот момент, когда функция *printf* осуществляет вывод данных, управление будет передано из *printf* обратно на *sigsetjmp*. При этом в структурах данных *printf* могут возникнуть противоречия. Чтобы предотвратить эту проблему, следует объединить блокирование и разблокирование сигналов, показанное в листинге 20.2, с помощью нелокального оператора *goto*.

Применение IPC в обработчике сигнала функции

Существует еще один корректный путь решения нашей проблемы. Вместо того чтобы просто возвращать управление и, как мы надеемся, прерывать блокированную функцию *recvfrom*, наш обработчик сигнала при помощи средств IPC (Interprocess Communications — взаимодействие процессов) может сообщить функции *dg_cli* о том, что время таймера истекло. Это аналогично предложению, сделанному нами раньше, когда обработчик сигнала устанавливал глобальную переменную *had_alarm* по истечении времени таймера. Глобальная переменная использовалась как некая разновидность IPC (поскольку она была доступна и нашей функции, и обработчику сигнала). Однако при таком решении наша функция

должна была проверять эту переменную, что могло привести к проблемам синхронизации в том случае, когда сигнал доставлялся приблизительно в это же время.

Листинг 20.6 демонстрирует использование канала внутри процесса. Обработчик сигналов записывает в канал 1 байт, когда истекает время таймера, а наша функция dg_cli считывает этот байт, чтобы определить, когда завершить свой цикл for. Что замечательно в этом решении — проверка готовности канала осуществляется функцией select. С ее помощью мы проверяем, готов ли к считыванию сокет или канал.

Листинг 20.6. Использование канала в качестве IPC между обработчиком сигнала и нашей функцией

```
//bcast/dgclibcast6.c
1 #include "unp.h"

2 static void recvfrom_alarm(int);
3 static int pipefd[2];
4 void
5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
6 {
7     int n, maxfdp1;
8     const int on = 1;
9     char sendline[MAXLINE], recvline[MAXLINE + 1];
10    fd_set rset;
11    socklen_t len;
12    struct sockaddr *preply_addr;

13    preply_addr = Malloc(servlen);

14    Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

15    Pipe(pipefd);
16    maxfdp1 = max(sockfd, pipefd[0]) + 1;

17    FD_ZERO(&rset);

18    Signal(SIGALRM, recvfrom_alarm);

19    while (Fgets(sendline, MAXLINE, fp) != NULL) {
20        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

21        alarm(5);
22        for (;;) {
23            FD_SET(sockfd, &rset);
24            FD_SET(pipefd[0], &rset);
25            if ((n = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
26                if (errno == EINTR)
27                    continue;
28                else
29                    err_sys("select error");
30            }

31            if (FD_ISSET(sockfd, &rset)) {
32                len = servlen;
33                n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr,
34                             &len);
35                recvline[n] = 0; /* null terminate */
36                printf("from %s: %s",
37                      Sock_ntop_host(preply_addr, len), recvline);
38            }
        }
    }
}
```

```

39     if (FD_ISSET(pipefd[0], &rset)) {
40         Read(pipefd[0], &n, 1); /* истекшее время */
41         break;
42     }
43 }
44 }
45 free(preply_addr);
46 }

47 static void
48 recvfrom_alarm(int signo)
49 {
50     Write(pipefd[1], "", 1); /* в канал пишется один нулевой байт */
51     return;
52 }

```

Создание канала

15 Мы создаем обычный канал Unix. Возвращаются два дескриптора: pipefd[0] доступен для чтения, а pipefd[0] — для записи.

ПРИМЕЧАНИЕ

Мы могли бы использовать функцию socketpair и получить двусторонний канал. В некоторых системах, особенно SVR4, обычный канал Unix всегда является двусторонним, и мы можем и читать, и записывать на любом конце этого канала.

Функция select на сокете и считывающем конце канала

23-30 Мы вызываем функцию select и на сокете, и на считывающем конце канала.

47-52 Когда доставляется сигнал SIGALRM, наш обработчик сигналов записывает в канал 1 байт, в результате чего считывающий конец канала становится готовым для чтения. Наш обработчик сигнала также возвращает управление, возможно, прерывая функцию select. Следовательно, если функция select возвращает ошибку EINTR, мы игнорируем эту ошибку, зная, что считывающий конец канала также готов для чтения, что завершит цикл for.

Чтение из канала

38-41 Когда считывающий конец канала готов для чтения, мы с помощью функции read считываем нулевой байт, записанный обработчиком сигнала, и игнорируем его. Но прибытие этого нулевого байта указывает нам на то, что истекло время таймера, и мы с помощью функции break выходим из бесконечного цикла for.

20.6. Резюме

При широковещательной передаче посыпаетсядейтаграмма, которую получают все узлы. Недостатком широковещательной передачи является то, что каждый узел в подсети должен обрабатыватьдейтаграмму, вплоть до уровня UDP в случаедейтаграммы UDP, даже если на узле не выполняетсяприложение-адресат. Для приложений с большими потоками данных, таких как аудио- и видео-приложения, это может привести к повышенной нагрузке на все узлы. В следующей главе мы увидим, что многоадресная передача решает эту проблему, поскольку позволяет не получатьдейтаграмму узлам, не заинтересованным в этом.

Использование версии нашего эхо-клиента UDP, который отправляет серверу времени и даты широковещательные дейтаграммы и затем выводит все его ответы, полученные в течение 5 с, позволяет нам рассмотреть ситуацию гонок, возникающую при применении сигнала SIGALRM. Общим способом помещения тайм-аута в операцию чтения является использование функции `alarm` и сигнала `SIGALRM`, но он несет в себе неявную ошибку, типичную для сетевых приложений. Мы показали один некорректный и три корректных способа решения этой проблемы:

- использование функции `pselect`,
- использование функций `sigsetjmp` и `siglongjmp`,
- использование средств IPC (обычно канала) между обработчиком сигнала и главным циклом.

Упражнения

1. Запустите клиент UDP, используя функцию `dg_cli`, выполняющую широковещательную передачу (см. листинг 20.1). Сколько ответов вы получаете? Всегда ли ответы приходят в одном и том же порядке? Синхронизированы ли часы у узлов в вашей подсети?

2. Поместите несколько функций `printf` в листинг 20.6 после завершения функции `select`, чтобы увидеть, возвращает ли она ошибку или указание на готовность к чтению одного из двух дескрипторов. Возвращает ли ваша система ошибку `EINTR` или сообщение о готовности канала к чтению, когда истекает время таймера `alarm`?

3. Запустите такую программу, как `tcpdump`, если это возможно, и просмотрите широковещательные пакеты в вашей локальной сети (команда `tcpdump ether broadcast`). К каким наборам протоколов относятся эти широковещательные пакеты?

Глава 21

Многоадресная передача

21.1. Введение

Как показано в табл. 20.1, адрес направленной передачи идентифицирует *одиночный* интерфейс, широковещательный адрес идентифицирует все интерфейсы в подсети, а адрес многоадресной передачи — *набор (множество)* интерфейсов. Направленная и широковещательная передача — это конечные точки спектра адресации (один интерфейс или все), а цель многоадресной передачи — обеспечить возможность адресации на участок спектра между этими конечными точками. Дейтаграмму многоадресной передачи должны получать только заинтересованные в ней интерфейсы, то есть интерфейсы на тех узлах, на которых запущены приложения, желающие принять участие в сеансе многоадресной передачи. Кроме того, широковещательная передача обычно ограничена локальными сетями, в то время как многоадресная передача может использоваться как в локальной, так и в глобальной сети. Существуют приложения, которые ежедневно участвуют в многоадресной передаче через всю сеть Интернет.

Дополнения к API сокетов, необходимые для поддержки многоадресной передачи, — это девять параметров сокетов. Три из них влияют на отправку дейтаграмм UDP на адрес, а шесть — на получение узлом дейтаграмм многоадресной передачи.

21.2. Адрес многоадресной передачи

При описании адресов многоадресной передачи необходимо провести различия между IPv4 и IPv6.

Адреса IPv4 класса D

Адреса класса D, лежащие в диапазоне от 224.0.0.0 до 239.255.255.255, в IPv4 являются адресами многоадресной передачи (см. табл. А.1). Младшие 28 бит адреса класса D образуют *идентификатор группы многоадресной передачи (multicast group ID)*, а 32-разрядный адрес называется *адресом группы (group address)*.

На рис. 21.1 показано, как адреса многоадресной передачи сопоставляются адресам Ethernet. Сопоставление адресов групп IPv4 для сетей Ethernet описывается в RFC 1112 [26], для сетей FDDI — в RFC 1390 [59], а для сетей типа Token Ring — в RFC 1469 [97]. Чтобы обеспечить возможность сравнения полученных в результате адресов Ethernet, мы также показываем сопоставление для адресов групп IPv6.

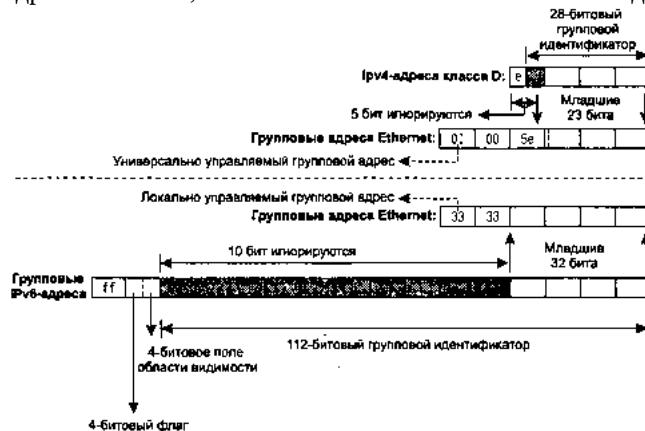


Рис. 21.1. Сопоставление адресам Ethernet адресов многоадресной передачи IPv4 и IPv6

Если рассматривать лишь сопоставление адресов IPv4, то в 24 старших битах адреса Ethernet всегда будет **01:00:5e**. Следующий бит всегда нулевой, а 23 младших бита копируются из 23 младших битов группового адреса. Старшие 5 бит группового адреса при сопоставлении игнорируются. Это значит, что 32 групповых адреса сопоставляются одиночному адресу Ethernet, то есть соответствие не является взаимнооднозначным.

Младшие 2 бита первого байта адреса Ethernet идентифицируют адрес как универсально управляемый групповой адрес. «Универсально управляемый» означает то, что 24 старших бита были присвоены IEEE (Institute of Electrical and Electronics Engineers — Институт инженеров по электротехнике и электронике), а групповые адреса многоадресной передачи распознаются и обрабатываются получающими интерфейсами специальным образом.

Существует несколько специальных адресов многоадресной передачи IPv4:

- 224.0.0.1 — это группа *всех узлов* (*all-hosts group*). Все узлы в подсети, имеющие возможность многоадресной передачи, должны присоединиться к этой группе интерфейсами, поддерживающими многоадресную передачу. (Мы поговорим о том, что значит присоединиться к группе, несколько позже.)

- 224.0.0.2 — это группа *всех маршрутизаторов* (*all-routers group*). Все маршрутизаторы многоадресной передачи в подсети должны присоединиться к этой группе интерфейсами, поддерживающими многоадресную передачу.

Диапазон адресов от 224.0.0.0 до 224.0.0.255 (который мы можем также записать в виде 224.0.0.0/24), называется *локальным на канальном уровне* (*link local*). Эти адреса предназначены для низкоуровневого определения топологии и служебных протоколов, и дейтаграммы, предназначенные для любого из этих адресов, никогда не передаются маршрутизатором многоадресной передачи дальше. Более подробно об области действия различных групповых адресов IPv4 мы поговорим после того, как рассмотрим адреса многоадресной передачи IPv6.

Адреса многоадресной передачи IPv6

Старший байт адреса многоадресной передачи IPv6 имеет значение ff. На рис. 21.1 показано сопоставление 16-байтового адреса многоадресной передачи IPv6 6-байтовому адресу Ethernet. Младшие 32 бита группового адреса копируются в младшие 32 бита адреса Ethernet. Старшие 2 байта адреса Ethernet имеют значение 33:33. Это сопоставление для сетей Ethernet описано в RFC 2464 [23], то же сопоставление для FDDI — в RFC 2467 [24], а сопоставление для сетей типа Token Ring — в RFC 2470 [25].

Младшие два бита первого байта адреса Ethernet определяют адрес как локально администрируемый групповой адрес. «Локально администрируемый» — это значит, что нет гарантий, что адрес уникален по отношению к IPv6. В этой сети кроме IPv6 могут быть и другие наборы протоколов, использующие те же два старших байта адреса Ethernet. Как мы отмечали ранее, групповые адреса распознаются и обрабатываются получающими интерфейсами специальным образом.

Имеется два формата адресов многоадресной передачи IPv6 (рис. 21.2). Когда флаг P имеет значение 0, флаг T интерпретируется как обозначение принадлежности адреса к группе *заранее известных* (*well-known* — значение 0) или к группе *временных* (*transient* — значение 1). Если флаг P равен 1, адрес считается назначенным на основе одноадресного префикса (см. RFC 3306 [40]). При этом флаг T также должен иметь значение 1 (многоадресные адреса на основе одноадресных всегда являются временными), а поля plen и prefix устанавливаются равными длине и значению префикса соответственно. Верхние два бита этого поля зарезервированы. Адреса многоадресной передачи IPv6 имеют также 4-разрядное поле *области действия* (*scope*), которое будет описано ниже. Документ RFC 3307 [39] описывает механизм выделения младших 32 разрядов группового адреса IPv6 (идентификатора группы) в зависимости от значения флага P.

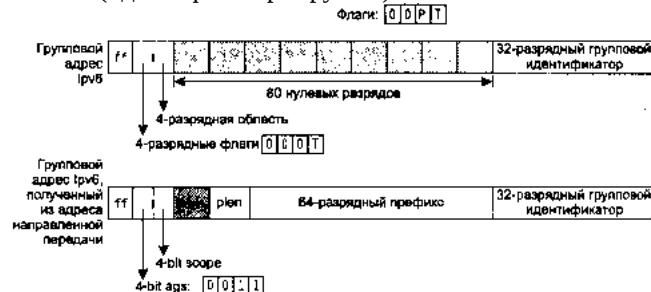


Рис. 21.2. Формат адресов многоадресной передачи IPv6

Существует несколько специальных адресов многоадресной передачи IPv6:

- ff02:1 — это группа *всех узлов* (*all-nodes group*). Все узлы подсети (компьютеры, маршрутизаторы, принтеры и т.д.), имеющие возможность многоадресной передачи, должны присоединиться к этой группе всеми своими интерфейсами, поддерживающими многоадресную передачу. Этот адрес аналогичен адресу

многоадресной передачи IPv4 224.0.0.1. Однако поскольку многоадресная передача является неотъемлемой частью IPv6, присоединение к группе является обязательным (в отличие от IPv4).

ПРИМЕЧАНИЕ

Хотя группа IPv4 называется all-hosts, а группа IPv6 — all-nodes, назначение у них одно и то же. Группа IPv6 была переименована, чтобы подчеркнуть, что в нее должны входить маршрутизаторы, принтеры и любые другие IP-устройства подсети, а не только компьютеры (hosts).

- ff02::2 — группа *всех маршрутизаторов* (*all-routers group*). Все маршрутизаторы многоадресной передачи в подсети должны присоединиться к этой группе интерфейсами, поддерживающими многоадресную передачу. Он аналогичен адресу многоадресной передачи IPv4 224.0.0.2.

Область действия адресов многоадресной передачи

Адреса многоадресной передачи IPv6 имеют собственное 4-разрядное поле *области действия* (*scope*), определяющее, насколько «далеко» будет передаваться пакет многоадресной передачи. Пакеты IPv6 вообще имеют поле предела количества транзитных узлов, которое ограничивает количество передач через маршрутизаторы (*hop limit field*). Поле области действия может принимать следующие значения:

- 1: локальная в пределах узла (*node-local*);
- 2: локальная в пределах физической сети (подсети) (*link-local*);
- 4: локальная в пределах области администрирования (*admin-local*);
- 5: локальная в пределах сайта (*site-local*);
- 8: локальная в пределах организации (*organization-local*);
- 14: глобальная (*global*).

Оставшиеся значения — это еще не присвоенные либо зарезервированные значения. Дейтаграмма, локальная в пределах узла, не должна выводиться интерфейсом, а дейтаграмма, локальная в пределах сети, никогда не должна передаваться в другую сеть маршрутизатором. Что понимается под областью администрирования, сайтом или организацией, зависит от администраторов маршрутизаторов многоадресной передачи. Адреса многоадресной передачи IPv6, различающиеся только областью действия, считаются относящимися к разным группам.

В IPv4 нет отдельного поля области действия для многоадресных пакетов. Исторически поле TTL IPv4 в заголовке IP выполняло также роль поля области действия многоадресной передачи: TTL, равное нулю, означает адрес, локальный в пределах узла, 1 — локальный в пределах сети, значения до 32 — локальный в пределах сайта, до 64 — локальный в пределах региона, до 128 — локальный в пределах континента (это означает, что пакеты не передаются по низкоскоростным и загруженным каналам, даже если они проложены в пределах одного континента) и до 255 — неограниченная область действия (глобальный). Двойное использование поля TTL привело к ряду сложностей, подробно описанных в документе RFC 2365 [75].

Хотя использование поля TTL IPv4 для области действия является принятой и рекомендуемой практикой, предпочтительнее административное управление областями действия, если оно возможно. При этом диапазон адресов от 239.0.0.0 до 239.255.255.255 определяется как *пространство многоадресной передачи IPv4 с административным ограничением области действия* (*administratively scoped IPv4 multicast space*) [75]. Это верхняя граница пространства адресов многоадресной передачи. Адреса в этом диапазоне задаются организацией локально, но их уникальность за пределами организации не гарантируется. Организация должна настроить свои пограничные маршрутизаторы многоадресной передачи таким образом, чтобы пакеты многоадресной передачи, предназначенные для любого из этих адресов, не передавались вовне.

Административно управляемые адреса многоадресной передачи IPv4 затем делятся на локальную область действия и локальную в пределах организации область действия, первая из которых аналогична (но не является семантическим эквивалентом) области действия IPv6, локальной в пределах сайта. Различные правила определения области действия мы приводим в табл. 21.1.

Таблица 21.1. Область действия адресов многоадресной передачи IPv4 и IPv6

Область действия	Значение поля области действия в IPv6	Значение поля TTL в IPv4	Административное управление областью действия в IPv4
Локальная в пределах узла	1	0	
Локальная в пределах сети	2	1	от 224.0.0.0 до 224.0.0.255
Локальная в пределах сайта	5	<32	от 239.255.0.0 до 239.255.255.255
Локальная в пределах организации	8		от 239.192.0.0 до 239.195.255.255
Глобальная	14	<255	от 224.0.1.0 до 238.255.255.255

Сеансы многоадресной передачи

Сочетание адреса многоадресной передачи IPv4 или IPv6 и порта транспортного уровня часто называется *сеансом (session)*, особенно если речь идет о передаче потокового мультимедиа. Например, телеконференция может объединять два сеанса: один аудио- и один видео-. Практически во всех случаях сеансы используют разные порты, а иногда и разные группы, что обеспечивает определенную гибкость для получателей. Например, один клиент может получать только аудиопоток, тогда как другой — аудио- и видео-. Если бы сеансы использовали один и тот же групповой адрес, это было бы невозможно.

21.3. Сравнение многоадресной и широковещательной передачи в локальной сети

Вернемся к примерам, представленным на рис. 20.2 и 20.3, чтобы показать, что происходит в случае многоадресной передачи. В примере, показанном на рис. 21.3, мы будем использовать IPv4, хотя для IPv6 последовательность операций будет такой же.

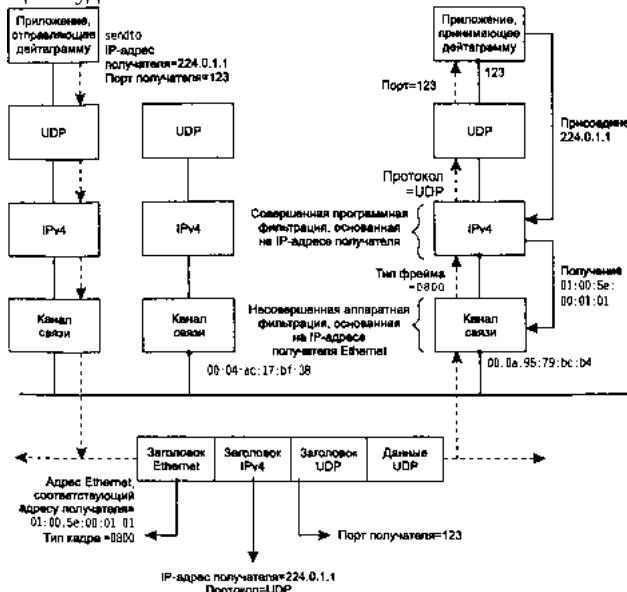


Рис. 21.3. Пример многоадресной передачи дейтаграммы UDP

Принимающее приложение на узле, изображенном справа, запускается и создает сокет UDP, связывает порт 123 с сокетом и затем присоединяется к группе 224.0.1.1. Мы вскоре увидим, что операция «присоединения» выполняется при помощи вызова функции `setsockopt`. Когда это происходит, уровень IPv4 сохраняет внутри себя информацию и затем сообщает соответствующему канальному уровню, что нужно получить кадры Ethernet, предназначенные адресу 01:00:5e:00:01:01 (см. раздел 12.11 [128]). Это соответствующий IP-адресу многоадресной передачи адрес Ethernet, к которому приложение только что присоединилось (с учетом сопоставления адресов, показанного на рис. 21.1).

Следующий шаг для отправляющего приложения на узле, изображенном слева, — создание сокета UDP и отправка дейтаграммы на адрес 224.0.1.1, порт 123. Для отправки дейтаграммы многоадресной передачи не требуется никаких специальных действий — приложению не нужно присоединяться к группе. Отправляющий узел преобразует IP-адрес в соответствующий адрес получателя Ethernet, и кадр отправляется. Обратите внимание, что кадр содержит и адрес получателя Ethernet (проверяемый интерфейсами), и IP-адрес получателя (проверяемый уровнями IP).

Мы предполагаем, что узел, изображенный в центре рисунка, не поддерживает многоадресную передачу IPv4 (поскольку поддержка многоадресной передачи IPv4 не обязательна). Узел полностью игнорирует кадр, поскольку, во-первых, адрес получателя Ethernet не совпадает с адресом интерфейса; во-вторых, адрес получателя Ethernet не является широковещательным адресом Ethernet, и в-третьих, интерфейс не получал указания принимать сообщения с адресами многоадресной передачи (то есть адресами, у которых младший бит старшего байта равен 1, как на рис. 21.1).

ПРИМЕЧАНИЕ

Когда интерфейс получает указание принимать кадры, предназначенные для определенного группового адреса Ethernet, многие современные сетевые адаптеры Ethernet применяют к адресу хэш-функцию, вычисляя значение от 0 до 511. Затем один из 512 бит массива устанавливается равным 1. Когда кадр проходит по кабелю, предназначенному для группового адреса, та же хэш-функция применяется интерфейсом к адресу получателя (первое поле в кадре), и снова вычисляется значение от 0 до 511. Если соответствующий бит в массиве установлен, кадр будет получен интерфейсом; иначе он игнорируется. Старые сетевые адаптеры использовали массив размером 64 бита, поэтому вероятность получения ненужных кадров была выше. С течением времени, поскольку все больше и больше приложений используют многоадресную передачу, этот размер, возможно, еще возрастет. Некоторые сетевые карты уже сейчас осуществляют совершенную фильтрацию (perfect filtering). У других карт возможность фильтрации многоадресной передачи отсутствует вовсе, и получая указание принять определенный групповой адрес, они должны принимать все кадры многоадресных передач (иногда это называется режимом смешанной многоадресной передачи). Одна популярная сетевая карта выполняет совершенную фильтрацию для 16 групповых адресов, а также имеет 512-битовую хэш-таблицу. Другая выполняет совершенную фильтрацию для 80 адресов, а остальные обрабатывают в смешанном режиме. Даже если интерфейс выполняет совершенную фильтрацию, все равно требуется совершенная программная фильтрация в пределах IP, поскольку сопоставление групповых адресов IP с аппаратными адресами не является взаимооднозначным.

Канальный уровень, изображенный справа, получает кадр на основе так называемой *несовершенной фильтрации* (*imperfect filtering*), которая выполняется интерфейсом с использованием адреса получателя Ethernet. Мы говорим, что эта фильтрация несовершенна, потому что если интерфейс получает указание принимать кадры, предназначенные для одного определенного группового адреса Ethernet, может случиться так, что он будет получать кадры, предназначенные также для других групповых адресов Ethernet.

Если предположить, что канальный уровень, изображенный справа, получает кадр, то поскольку тип кадра Ethernet — IPv4, пакет передается уровню IP. Поскольку полученный пакет был предназначен IP-адресу многоадресной передачи, уровень IP сравнивает этот адрес со всеми адресами многоадресной передачи, к которым присоединились приложения на узле. Мы называем это *совершенной фильтрацией*, так как она основана на полном 32-разрядном адресе класса D в заголовке IPv4. В этом примере пакет принимается уровнем IP и передается уровню UDP, который, в свою очередь, передает дейтаграмму сокету, связанному с портом 123.

Существует еще три сценария, не показанных нами на рис. 21.3.

1. На узле запущено приложение, присоединившееся к адресу многоадресной передачи 225.0.1.1. Поскольку 5 верхних битов группового адреса игнорируются при сопоставлении с адресом Ethernet, этот интерфейс узла будет также получать кадры с адресом получателя Ethernet 01:00:5e:00:01:01. В этом случае пакет будет проигнорирован при осуществлении совершенной фильтрации на уровне IP.

2. На узле запущено приложение, присоединившееся к некоторой группе. Соответствующий адрес Ethernet этой группы является одним из тех, которые интерфейс может получить случайно, поскольку он

запрограммирован на получение сообщений на адрес 01:00:5e:00:01:01 (то есть сетевая карта выполняет несовершенную фильтрацию). Этот кадр будет проигнорирован либо канальным уровнем, либо уровнем IP.

3. Пакет предназначен для той же группы 224.0.1.1, но для другого порта, скажем 4000. Узел, изображенный справа на рис. 21.3, получает пакет, далее этот пакет принимается уровнем IP, но если не существует сокета, связанного с портом 4000, пакет будет проигнорирован уровнем UDP.

ВНИМАНИЕ

Эти сценарии показывают нам, что для того чтобы процесс мог получать дейтаграммы многоадресной передачи, он должен присоединиться к группе и связаться с портом.

21.4. Многоадресная передача в глобальной сети

Многоадресная передача внутри локальной сети, описанная нами в предыдущем разделе, проста. Один узел посыпает пакет многоадресной передачи, и любой заинтересованный узел получает этот пакет. Преимущество многоадресной передачи перед широковещательной состоит в сокращении нагрузки на все узлы, не заинтересованные в получении пакетов многоадресной передачи.

Многоадресная передача имеет преимущества и при работе в глобальных сетях. Рассмотрим глобальную сеть, изображенную на рис. 21.4.

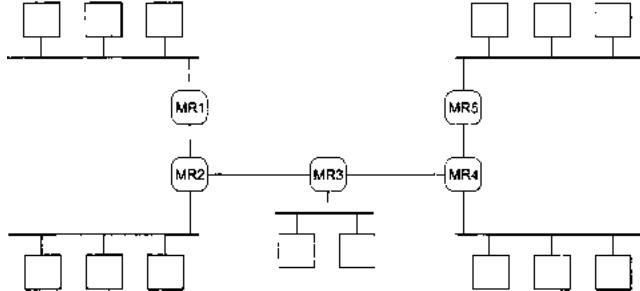


Рис. 21.4. Пять локальных сетей с пятью маршрутизаторами многоадресной передачи

Здесь изображены пять локальных сетей, соединенных пятью маршрутизаторами многоадресной передачи.

Будем считать, что некая программа запущена на пяти из показанных узлов (скажем, программа прослушивания группового аудиосеанса), и эти пять программ присоединяются к данной группе. Тогда каждый из пяти узлов присоединяется к группе. Мы также считаем, что каждый маршрутизатор многоадресной передачи общается с соседними маршрутизаторами многоадресной передачи при помощи протокола маршрутизации многоадресной передачи (*multicast routing protocol*), который мы обозначим просто MRP. Это показано на рис. 21.5.

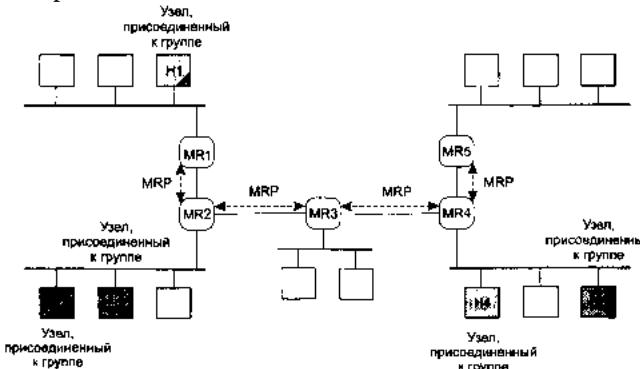


Рис. 21.5. Присоединение пяти узлов к группе многоадресной передачи в глобальной сети

Когда процесс на узле присоединяется к группе, этот узел отправляет всем присоединенным к той же сети маршрутизаторам многоадресной передачи сообщение IGMP, информирующее их о том, что узел только что присоединился к группе. Затем маршрутизаторы обмениваются этой информацией по MRP, так

что каждый маршрутизатор знает, что делать, если он получит пакет, предназначенный для конкретного адреса многоадресной передачи.

ПРИМЕЧАНИЕ

Адресация многоадресной передачи — не до конца исследованная тема, и ее описание может легко составить отдельную книгу.

Теперь будем считать, что процесс на узле, изображенном слева вверху, начинает отправлять пакеты на адрес многоадресной передачи. Допустим, этот процесс отправляет аудиопакеты, ожидаемые получателями многоадресной передачи. Эти пакеты показаны на рис. 21.6.

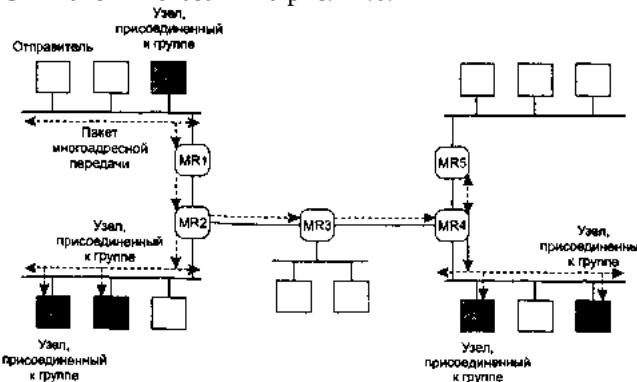


Рис. 21.6. Отправка пакетов на адрес многоадресной передачи в глобальной сети

Проследим шаги, которые проходит пакет от отправителя до получателей.

■ Пакеты многоадресной передачи рассылаются отправителем в левой верхней локальной сети. Получатель H1 получает их (так как он присоединился к группе), как и MR1 (поскольку маршрутизатор многоадресной передачи должен получать все пакеты многоадресного вещания).

■ MR1 передает пакет многоадресной передачи дальше маршрутизатору MR2, поскольку протокол маршрутизации многоадресной передачи сообщил MR1, что MR2 должен получить пакеты, предназначенные для этой группы.

■ MR2 передает этот пакет присоединенной локальной сети, поскольку узлы H2 и H3 входят в группу. Он также создает копию пакета и отправляет ее MR3.

Создание копии пакета маршрутизатором свойственно только многоадресной передаче. Пакет направленной передачи никогда не дублируется при передаче маршрутизаторами.

■ MR3 с отправляет пакет многоадресной передачи маршрутизатору MR4, но не передает копию в свою локальную сеть, потому что ни один из узлов в этой сети не присоединился к группе.

■ MR4 передает пакет на присоединенную локальную сеть, поскольку узлы H4 и H5 входят в группу. Он не создает копии пакета и не отправляет пакет маршрутизатору MR, поскольку ни один из узлов присоединенной к MR локальной сети не входит в группу, и MR4 знает об этом из информации о маршрутизации многоадресной передачи, которой он обменялся с MR.

Две менее желательные альтернативы многоадресной передаче в глобальной сети — *лавинная адресация* (*broadcast flooding*) и отправка индивидуальных копий каждому получателю. В первом случае отправитель будет передавать широковещательные пакеты, а каждый маршрутизатор будет передавать пакет с каждого из своих интерфейсов, кроме принимающего. Ясно, что это увеличит число незаинтересованных узлов и маршрутизаторов, которым придется получать этот пакет.

Во втором случае отправитель должен знать IP-адреса всех получателей и отослать каждому по копии пакета. В случае с пятью пакетами, который представлен на рис. 21.6, это потребует пяти пакетов в локальной сети отправителя, четырех пакетов, идущих от MR1 к MR2, и двух пакетов, идущих от MR2 к MR3 и к MR4. А если получателей будет миллион?!

21.5. Многоадресная передача от отправителя

Внедрение многоадресной передачи в глобальные сети было затруднено несколькими обстоятельствами. Главная проблема заключается в том, что протокол маршрутизации MRP, описанный в разделе 21.4, должен обеспечивать доставку данных от всех отправителей (которые могут располагаться в сети совершенно произвольным образом) всем получателям (которые также могут быть размещены произвольно). Еще одна проблема связана с выделением адресов: адресов многоадресной передачи IPv4 недостаточно для того, чтобы можно было статически назначать их всем, кому они нужны, как это делается с адресами направленной передачи. Чтобы передавать многоадресные сообщения в глобальной сети, не конфликтую с другими отправителями, нужно иметь уникальный адрес, однако механизма глобального выделения адресов еще не существует.

Многоадресная передача от отправителя (*source-specific multicast, SSM*) [47] представляет собой эффективное решение этих проблем. Она состоит в соединении адреса группы с адресом отправителя.

- При подключении к группе получатели предоставляют маршрутизаторам не только адрес группы, но и адрес отправителя. Это устраняет проблему поиска, потому что теперь маршрутизатор точно знает, где находится отправитель. Однако при этом сохраняется удобство масштабирования приложений, потому что отправителю все так же не нужно знать адреса всех своих получателей. Такое решение очень сильно упрощает протоколы маршрутизации многоадресной передачи.

- Идентификатор группы перестает быть групповым адресом и становится комбинацией адреса отправителя (адреса направленной передачи) и адреса группы (адреса многоадресной передачи). Такая комбинация называется в SSM *каналом* (*channel*). Благодаря этому отправитель может выбрать любой адрес многоадресной передачи, так как уникальность канала обеспечивается уже уникальностью адреса отправителя. Сеанс SSM представляет собой комбинацию адреса отправителя, адреса группы и порта.

SSM обеспечивает некоторую защиту от подмены адреса, потому что отправителю 2 становится значительно труднее передавать сообщения по каналу отправителя 1, так как идентификатор этого канала включает в себя адрес отправителя 1. Подмена все еще остается возможной, однако серьезно усложняется.

21.6. Параметры сокетов многоадресной передачи

Для поддержки многоадресной передачи программным интерфейсом приложений (API) требуется только пять новых параметров сокетов. Поддержка фильтрации отправителей, необходимая для SSM, требует еще четырех параметров. В табл. 21.2 показаны три параметра, не имеющих отношения к членству в группах, а также тип данных аргумента, который предполагается использовать в вызове функций `getsockopt` или `setsockopt` для IPv4 и IPv6. В табл. 21.3 представлены оставшиеся шесть параметров сокетов для IPv4, IPv6 и не зависящего от IP-версии API. Во втором столбце показан тип данных переменной, указатель на которую является четвертым аргументом функций `getsockopt` и `setsockopt`. Все девять параметров действительны с функцией `setsockopt`, но шесть предназначенных для входа и выхода из группы не могут быть использованы в вызове функции `getsockopt`.

Таблица 21.2. Параметры сокетов многоадресной передачи

Параметр	Тип данных	Описание
IP_MULTICAST_IF	struct in_addr	Интерфейс по умолчанию для исходящих многоадресных пакетов
IP_MULTICAST_TTL	u_char	TTL для исходящих многоадресных пакетов
IP_MULTICAST_LOOP	u_char	Включение и отключение закольцовки для исходящих многоадресных пакетов
IPV6_MULTICAST_IF	u_int	Интерфейс по умолчанию для исходящих многоадресных пакетов
IPV6_MULTICAST_HOPS	int	Предел количества прыжков для и сходящих многоадресных пакетов
IPV6_MULTICAST_LOOP	u_int	Включение и отключение закольцовки для исходящих многоадресных пакетов

Таблица 21.3. Параметры сокета, определяющие членство в группах многоадресной передачи

Параметр	Тип данных	Описание
IP_ADD_MEMBERSHIP	struct ip_mreq	Присоединение к группе многоадресной передачи

IP_DROP_MEMBERSHIP	struct ip_mreq	Отсоединение от группы многоадресной передачи
IP_BLOCK_SOURCE	struct ip_mreq_source	Блокирование источника из группы, к которой выполнено присоединение
IP_UNBLOCK_SOURCE	struct ip_mreq_source	Разблокирование ранее заблокированного источника
IP_ADD_SOURCE_MEMBERSHIP	struct ip_mreq_source	Присоединение к группе источника
IP_DROP_SOURCE_MEMBERSHIP	struct ip_mreq_source	Отсоединение от группы источника
IPV6_JOIN_GROUP	struct ipv6_mreq	Присоединение к группе многоадресной передачи
IPV6_LEAVE_GROUP	struct ipv6_mreq	Отсоединение от группы многоадресной передачи
MCAST_JOIN_GROUP	struct group_req	Присоединение к группе многоадресной передачи
MCAST_LEAVE_GROUP	struct group_req	Отсоединение от группы многоадресной передачи
MCAST_BLOCK_SOURCE	struct group_source_req	Блокирование источника из группы, к которой выполнено присоединение
MCAST_UNBLOCK_SOURCE	struct group_source_req	Разблокирование ранее заблокированного источника
MCAST_JOIN_SOURCE_GROUP	struct group_source_req	Присоединение к группе источника
MCAST_LEAVE_SOURCE_GROUP	struct group_source_req	Отсоединение от группы источника

ПРИМЕЧАНИЕ

Параметры IPv4 TTL и закольцовки получают аргумент типа `u_char`, в то время как IPv6 — параметры предела транзитных узлов и закольцовки получают аргументы соответственно типа `int` и `u_int`. Распространенная ошибка программирования с параметрами многоадресной передачи IPv4 — вызов функции `setsockopt` с аргументом типа `int` для задания TTL или закольцовки (что не разрешается [128, с. 354–355]), поскольку большинство других параметров сокетов, представленных в табл. 7.1, имеют целочисленные аргументы. Изменения, внесенные в IPv6, должны уменьшить вероятность ошибок.

Теперь мы опишем каждый из девяти параметров сокетов более подробно. Обратите внимание, что эти девять параметров концептуально идентичны в IPv4 и IPv6 — различаются только их названия и типы аргументов.

■ `IP_ADD_MEMBERSHIP`, `IPV6_JOIN_GROUP`, `MCAST_JOIN_GROUP`. Назначение этих параметров — присоединение к группе на заданном локальном интерфейсе. Мы задаем локальный интерфейс одним из его направленных адресов для IPv4 или индексом интерфейса для IPv6. Следующие три структуры используются при присоединении к группе или при отсоединении от нее:

```

struct ip_mreq {
    struct in_addr imr_multiaddr; /* IPv4-адрес многоадресной
                                   передачи класса D */
    struct in_addr imr_interface; /* IPv4-адрес локального
                                   интерфейса */
};

struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr; /* IPv6-адрес многоадресной
                                   передачи */
    unsigned int ipv6mr_interface; /* индекс интерфейса или 0 */
};

struct group_req {

```

```

unsigned int gr_interface;           /* индекс интерфейса или 0 */
struct sockaddr_storage gr_group;  /* адрес многоадресной передачи
IPv4 или IPv6 */
};

};


```

Если локальный интерфейс задается как универсальный адрес (`INADDR_ANY` для IPv4) или как нулевой индекс IPv6, то конкретный локальный интерфейс выбирается ядром.

Мы говорим, что узел принадлежит к данной группе на данном интерфейсе, если один или более процессов в настоящий момент принадлежат к этой группе на этом интерфейсе.

Сокет может быть присоединен к нескольким группам, но к каждой группе должен быть присоединен уникальный адрес или уникальный интерфейс. Это свойство можно использовать на узле с несколькими сетевыми интерфейсами: создается один сокет, которому присваивается один адрес многоадресной передачи, но благодаря наличию разных интерфейсов этот сокет может быть присоединен к разным группам.

Вспомните из табл. 21.1, что частью адреса многоадресной передачи IPv6 является поле области действия. Как мы отмечали, адреса многоадресной передачи IPv6, отличающиеся только областью действия, являются различными. Следовательно, если реализация протокола синхронизации времени (network time protocol, NTP) хочет получать все пакеты NTP независимо от их области действия, она должна будет присоединиться к адресу `ff01:101` (локальный в пределах узла), `ff02:101` (локальный в пределах физической сети), `ff05:101` (локальный в пределах сайта), `ff08:101` (локальный в пределах организации) и `ff0e:101` (глобальный). Все присоединения могут выполняться на одном сокете. Можно установить параметр сокета `IPV6_PKTINFO` (см. раздел 22.8), чтобы функция `recvmsg` возвращала адрес получателя каждой дейтаграммы.

Независимый от версии IP параметр сокета (`MCAST_JOIN_GROUP`) аналогичен соответствующему параметру IPv6 за тем исключением, что он использует структуру `sockaddr_storage` вместо `in6_addr` для передачи адреса ядру. Структура `sockaddr_storage` (см. листинг 3.4) достаточно велика для хранения адреса любой версии, поддерживаемой системой.

ПРИМЕЧАНИЕ

В большинстве реализаций число присоединений, допустимых для одного сокета, ограничено. Предел определяется константой `IP_MAX_MEMBERSHIPS` (для Беркли-реализаций ее значение равно 20). В некоторых реализациях это ограничение снято, в других оно значительно превышает значение для Беркли-реализаций.

Когда интерфейс, на котором будет происходить присоединение, не задан, Беркли-ядра ищут адрес многоадресной передачи в обычной таблице маршрутизации IP и используют полученный в результате интерфейс [128, с. 357]. Некоторые системы для обработки этой ситуации устанавливают маршрут для всех адресов многоадресной передачи (то есть маршрут с адресом получателя 224.0.0.0/8 для IPv4) в процессе инициализации.

Для IPv6 сделано изменение — при задании интерфейса используется индекс, а не локальный адрес направленной передачи, как было в IPv4. Это позволяет выполнять присоединение на ненумерованных интерфейсах и конечных точках туннелей.

Изначально в API многоадресной передачи IPv6 использовалась константа `IPV6_ADD_MEMBERSHIP`, а не `IPV6_JOIN_GROUP`. Во всех остальных отношениях интерфейс программирования не изменился. Описанная далее функция `mcast_join` скрывает это отличие.

■ `IP_DROP_MEMBERSHIP`, `IPV6_LEAVE_GROUP` и `MCAST_LEAVE_GROUP`. Назначение этих параметров — выход из группы на заданном локальном интерфейсе. С этими параметрами сокета применяются те же структуры, которые мы только что показали для присоединения к группе. Если локальный интерфейс не задан (то есть его значение равно `INADDR_ANY` для IPv4 или индекс интерфейса равен нулю для IPv6), удаляется первое совпадающее с искомым вхождение в группу.

Если процесс присоединился к группе, но не выходил из группы явно, то при закрытии сокета (либо явном, либо по завершении процесса) вхождение в группу прекращается автоматически. Возможна ситуация, когда несколько процессов на узле присоединились к одной и той же группе, и в этом случае узел остается членом группы, пока последний процесс не выйдет из группы.

ПРИМЕЧАНИЕ

Изначально в API многоадресной передачи IPv6 использовалась константа IPV6_DROP_MEMBERSHIP, а не IPV6_LEAVE_GROUP. Во всех остальных отношениях интерфейс программирования не изменился. Описанная далее функция mcast_leave скрывает это отличие.

- IP_BLOCK_SOURCE, MCAST_BLOCK_SOURCE. Блокируют получение трафика через данный сокет от конкретного источника для конкретной группы и интерфейса. Если все сокеты, присоединенные к группе, заблокировали один и тот же источник, система может проинформировать маршрутизаторы о нежелательности трафика, что может повлиять на маршрутизацию многоадресного трафика в сети. Локальный интерфейс задается одним из его направленных адресов для IPv4 или индексом для независимого от версии API. Для блокирования и разблокирования источника используются две приведенные ниже структуры:

```
struct ip_mreq_source {  
    struct in_addr imr_multiaddr; /* IPv4-адрес многоадресной  
                                   передачи класса D */  
    struct in_addr imr_sourceaddr; /* IPv4-адрес источника */  
    struct in_addr imr_interface; /* IPv4-адрес локального  
                                   интерфейса */  
};  
  
struct group_source_req {  
    unsigned int gsr_interface;      /* индекс интерфейса или 0 */  
    struct sockaddr_storage gsr_group; /* адрес многоадресной  
                                       передачи IPv4 или IPv6 */  
    struct sockaddr_storage gsr_source; /* адрес источника IPv4  
                                       или IPv6 */  
};
```

Если локальный интерфейс задается как универсальный адрес (INADDR_ANY для IPv4) или как нулевой индекс IPv6, то конкретный локальный интерфейс выбирается ядром.

Запрос на блокирование источника действует только для присоединенных групп, то есть таких, которые уже были присоединены к указанному интерфейсу параметром IP_ADD_MEMBERSHIP, IPV6_JOIN_GROUP или MCAST_JOIN_GROUP.

- IP_UNBLOCK_SOURCE, MCAST_UNBLOCK_SOURCE. Разблокирование заблокированного ранее источника. Аргументы должны быть в точности те же, что и у предшествовавшего запроса IP_BLOCK_SOURCE или MCAST_BLOCK_SOURCE.

Если локальный интерфейс задается как универсальный адрес (INADDR_ANY для IPv4) или как нулевой индекс IPv6, то конкретный локальный интерфейс выбирается ядром.

- IP_ADD_SOURCE_MEMBERSHIP, MCAST_JOIN_SOURCE_GROUP. Присоединение к группе конкретного источника на заданном локальном интерфейсе. С этим параметром используются те же структуры, что и с параметрами блокирования и разблокирования источника. Сокет не должен быть присоединен к той же группе без указания источника (параметры IP_ADD_MEMBERSHIP, IPV6_JOIN_GROUP, MCAST_JOIN_GROUP).

Если локальный интерфейс задается как универсальный адрес (INADDR_ANY для IPv4) или как нулевой индекс IPv6, то конкретный локальный интерфейс выбирается ядром.

- IP_DROP_SOURCE_MEMBERSHIP, MCAST_LEAVE_SOURCE_GROUP. Отключение от группы источника конкретного локального интерфейса. Используются те же структуры, что и с предыдущими параметрами сокетов. Если локальный интерфейс не указан (значение INADDR_ANY для IPv4 или 0 для независимого от версии API), отключается первая группа, удовлетворяющая заданным значениям.

Если процесс присоединяется к группе источника, но не отключается от нее явно, отсоединение производится автоматически при закрытии сокета (явном или также автоматическом, при завершении процесса). Несколько процессов узла могут присоединиться к одной и той же группе источника, в случае чего узел остается в группе до тех пор, пока из нее не выйдет последний процесс.

- IP_MULTICAST_IF и IPV6_MULTICAST_IF. Назначение этих параметров - задание интерфейса для исходящих дейтаграмм многоадресной передачи, отправленных на этом сокете. Этот интерфейс задается

либо структурой `in_addr` для IPv4, либо индексом интерфейса для IPv6. Если задано значение `INADDR_ANY` для IPv4 или нулевой индекс интерфейса для IPv6, то удаляется любой интерфейс, ранее заданный этим параметром сокета, и система будет выбирать интерфейс каждый раз при отправке дейтаграммы.

Будьте внимательны, четко различая локальный интерфейс, заданный (или выбранный), когда процесс присоединяется к группе (интерфейс для получения приходящих дейтаграмм многоадресной передачи), и локальный интерфейс, заданный (или выбранный) для исходящих дейтаграмм.

ПРИМЕЧАНИЕ

Беркли-ядра выбирают интерфейс для исходящих дейтаграмм многоадресной передачи по умолчанию при помощи обычной таблицы маршрутизации IP. В ней выполняется поиск маршрута к групповому адресу получателя, после чего используется соответствующий интерфейс. Это та же технология, что используется для выбора принимающего интерфейса, если процесс не задает его в процессе присоединения к группе. При этом считается, что если для данного адреса многоадресной передачи существует маршрут (возможно, маршрут, заданный по умолчанию в таблице маршрутизации), то соответствующий интерфейс должен использоваться для ввода и вывода.

■ `IP_MULTICAST_TTL` и `IPV6_MULTICAST_HOPS`. Назначение этих параметров - установка значения поля TTL в случае IPv4 или предельного количества транзитных узлов в случае IPv6 для исходящих дейтаграмм многоадресной передачи. По умолчанию значение обоих параметров равно 1, что ограничивает дейтаграмму локальной подсетью.

■ `IP_MULTICAST_LOOP` и `IPV6_MULTICAST_LOOP`. Назначение этих параметров - включение или отключение локальной закольцовки для дейтаграмм многоадресной передачи. По умолчанию закольцовка включена: копия каждой дейтаграммы многоадресной передачи, посланной процессом на узле, будет отправлена обратно на этот узел и обработана им, как любая другая полученная дейтаграмма, если узел принадлежит данной группе на исходящем интерфейсе.

Это аналогично широковещательной передаче, где мы видели, что широковещательные сообщения, посланные на узле, также обрабатываются на нем, как полученные дейтаграммы (см. рис. 20.3). (Но в случае широковещательной передачи нет возможности отключить закольцовку.) Это значит, что если процесс входит в ту группу, которой он отправляет дейтаграммы, он будет получать свои собственные передачи.

ПРИМЕЧАНИЕ

Описываемая здесь закольцовка является внутренней и выполняется на уровне IP или выше. Если интерфейс получает копии своих передач, RFC 1112 [26] требует, чтобы драйвер игнорировал эти копии. В этом документе также утверждается, что параметр закольцовки по умолчанию включен «в целях оптимизации производительности для протоколов верхнего уровня, которые ограничивают членство в группе до одного процесса на узел (например, маршрутизирующих протоколов)».

Первые шесть пар параметров сокетов (`ADD_MEMBERSHIP/JION_GROUP`, `DROP_MEMBERSHIP/LEAVE_GROUP`, `BLOCK_SOURCE, UNBLOCK_SOURCE, ADD_SOURCE_MEMBERSHIP/JION_SOURCE_GROUP`, `DROP_SOURCE_MEMBERSHIP/LEAVE_SOURCE_GROUP`) влияют на получение дейтаграмм многоадресной передачи, в то время как последние три пары параметров влияют на отправку дейтаграмм многоадресной передачи (интерфейс для исходящих сообщений, TTL или предел количества транзитных узлов, закольцовка). Ранее мы отмечали, что для отправки дейтаграммы многоадресной передачи ничего особенного не требуется. Если ни один параметр сокетов многоадресной передачи не задан перед отправкой дейтаграммы, интерфейс для исходящей дейтаграммы будет выбран ядром, TTL или предел количества транзитных узлов будут равны 1, а копия отправленной дейтаграммы будет посыпаться обратно (то есть будет включена закольцовка).

Чтобы получить дейтаграмму многоадресной передачи, процесс должен присоединиться к группе, а также связать при помощи функции `bind` сокет UDP с номером порта, который будет использоваться как

номер порта получателя для дейтаграмм, отсылаемых данной группе. Это две отдельные операции, и обе они являются обязательными. Присоединение к группе указывает уровню IP узла и канальному уровню, что необходимо получать дейтаграммы многоадресной передачи, отправленные этой группе. Связывая порт, приложение указывает UDP, что требуется получать отправляемые на этот порт дейтаграммы. Некоторые приложения в дополнение к связыванию порта также связывают при помощи функции bind адрес многоадресной передачи с сокетом. Это предотвращает доставку сокету любых других дейтаграмм, которые могли быть получены для этого порта.

ПРИМЕЧАНИЕ

Исторически Беркли-реализации требуют только, чтобы некоторый сокет на узле присоединился к группе — это не обязательно тот сокет, который связывается с портом и затем получает дейтаграммы многоадресной передачи. Однако есть вероятность, что эти реализации могут доставлять дейтаграммы многоадресной передачи приложениям, не знающим о многоадресной передаче. Более новые ядра требуют, чтобы процесс связывался с портом и устанавливал какой-нибудь параметр сокета многоадресной передачи для сокета как указатель того, что приложение знает о многоадресной передаче. Самый обычный параметр сокета многоадресной передачи — признак присоединения к группе. Для Solaris 2.5 характерны некоторые отличия: дейтаграммы многоадресной передачи доставляются только на те сокеты, которые присоединились к группе и связались с портом. В целях переносимости все приложения многоадресной передачи должны присоединяться к группе и связываться с портом.

Более новый интерфейс многоадресного сервиса требует, чтобы уровень IP доставлял многоадресные пакеты сокету только в том случае, если этот сокет присоединился к группе или источнику. Такое требование было введено с IGMPv3 (RFC 3376 [16]), чтобы разрешить фильтрацию источников и многоадресную передачу от источника. Таким образом ужесточается требование на присоединение к группе, но зато ослабляется требование на связывание группового адреса. Однако для наибольшей переносимости со старыми и новыми интерфейсами приложения должны присоединяться к группам и связывать сокеты с групповыми адресами.

Некоторые более старые узлы, имеющие возможность многоадресной передачи, не позволяют связывать адрес многоадресной передачи с сокетом при помощи функции bind. В целях переносимости приложение может игнорировать ошибку функции bind при связывании адреса многоадресной передачи с сокетом и делать повторную попытку с адресом INADDR_ANY или in6addr_any.

21.7. Функция mcast_join и родственные функции

Несмотря на то что параметры сокетов многоадресной передачи для IPv4 аналогичны параметрам сокетов многоадресной передачи для IPv6, есть достаточно много различий, из-за которых не зависящий от протокола код, использующий многоадресную передачу, усложняется и содержит множество директив #ifdef. Наилучшим решением будет использование приведенных ниже восьми функций, позволяющих скрыть различия реализаций:

```
#include "unp.h"

int mcast_join(int sockfd, const struct sockaddr *grp,
    socklen_t grplen, const char *ifname, u_int ifindex);
int mcast_leave(int sockfd, const struct sockaddr *grp,
    socklen_t grplen);
int mcast_block_source(int sockfd,
    const struct sockaddr *src, socklen_t srclen,
    const struct sockaddr *grp, socklen_t grplen);
int mcast_unblock_source(int sockfd,
    const struct sockaddr *src, socklen_t srclen,
    const struct sockaddr *grp, socklen_t grplen);
int mcast_join_source_group(int sockfd,
    const struct sockaddr *src, socklen_t srclen,
```

```
const struct sockaddr *grp, socklen_t grplen,
const char *ifname, u_int ifindex);
int mcast_leave_source_group(int sockfd,
    const struct sockaddr *src, socklen_t srclen,
    const struct sockaddr *grp, socklen_t grplen);
int mcast_set_if(int sockfd, const char *ifname, u_int ifindex);
int mcast_set_loop(int sockfd, int flag);
int mcast_set_ttl(int sockfd, int ttl);
Все перечисленные выше функции возвращают: 0 в случае успешного выполнения, -1 в случае ошибки
```

```
int mcast_get_if(int sockfd);
Возвращает: неотрицательный индекс интерфейса в случае успешного выполнения, -1 в случае ошибки
```

```
int mcast_get_loop(int sockfd);
Возвращает: текущий флаг закольцовки в случае успешного выполнения, -1 в случае ошибки

int mcast_get_ttl(int sockfd);
Возвращает: текущее значение TTL или предельное количество транзитных узлов в случае успешного выполнения, -1 в случае ошибки
```

Функция `mcast_join` присоединяет узел к группе. IP-адрес этой группы содержится в структуре адреса сокета, на которую указывает аргумент `grp`, а длина этой структуры задается аргументом `grplen`. Мы можем задать интерфейс, на котором должно происходить присоединение к группе, либо через имя интерфейса (непустой аргумент `ifname`), либо через ненулевой индекс интерфейса (непустой аргумент `ifindex`). Если ни одно из этих значений не задано, ядро самостоятельно выбирает интерфейс, на котором происходит присоединение к группе. Вспомните, что в случае IPv6 для работы с параметрами сокета интерфейс задается по его индексу. Если для сокета IPv6 известно имя интерфейса, нужно вызвать функцию `if_nametoindex`, чтобы получить индекс интерфейса. В случае параметра сокета IPv4 мы задаем интерфейс по его IP-адресу направленной передачи. Если для сокета IPv4 интерфейс задан по имени, нужно вызвать функцию `ioctl` с запросом `SIOCGIFADDR` для получения IP-адреса направленной передачи для этого интерфейса. Если для сокета IPv4 задан индекс интерфейса, мы сначала вызываем функцию `if_indextoname`, чтобы получить имя интерфейса, а затем обрабатываем имя так, как только что было сказано.

ПРИМЕЧАНИЕ

Пользователи обычно задают имя интерфейса `le0` или `ether0`, а IP-адрес и индекс интерфейса не используются. Например, `tcpdump` является одной из немногих программ, позволяющих пользователю задавать интерфейс, а ее параметр `-i` принимает имя интерфейса в качестве аргумента.

Функция `mcast_leave` выводит узел из группы с IP-адресом, содержащимся в структуре адреса сокета, на которую указывает аргумент `grp`.

Функция `mcast_block_source` блокирует получение через конкретный сокет пакетов, относящихся к определенной группе и исходящих от определенного источника. IP-адреса группы и источника хранятся в структурах адреса сокета, на которые указывают аргументы `grp` и `src` соответственно. Длины структур задаются параметрами `srclen` и `grplen`. Для успешного завершения функции необходимо, чтобы до ее вызова уже была вызвана функция `mcast_join` для того же сокета и той же группы.

Функция `mcast_unblock_source` разблокирует получение трафика от источника из заданной группы. Аргументы `src`, `srclen`, `grp` и `grplen` имеют тот же смысл, что и аргументы предыдущей функции, и должны совпадать с ними по значениям.

Функция `mcast_join_source_group` выполняет присоединение к группе источника. Адрес источника и адрес группы содержатся в структурах адреса сокета, на которые указывают аргументы `src` и `grp`. Длины структур задаются параметрами `srclen` и `grplen`. Интерфейс, присоединяемый к группе, может быть задан

именем (ненулевой аргумент `ifname`) или индексом (`ifindex`). Если интерфейс не задан явно, ядро выбирает его самостоятельно.

Функция `mcast_leave_source_group` выполняет отсоединение от группы источника. Адреса источника и группы содержатся в структурах адреса сокета, на которые указывают аргументы `src` и `grp`. Длины структур задаются параметрами `srcalen` и `grplen`. Подобно `mcast_leave`, `mcast_leave_source_group` не требует указания интерфейса: она всегда отсоединяет от группы первый интерфейс, удовлетворяющий условиям.

Функция `mcast_set_if` устанавливает индекс интерфейса по умолчанию для исходящих дейтаграмм многоадресной передачи. Если аргумент `ifname` непустой, он задает имя интерфейса. Иначе положительное значение аргумента `ifindex` будет задавать индекс интерфейса. В случае IPv6 имя сопоставляется индексу с использованием функции `if_nametoindex`. В случае IPv4 сопоставление имени или индекса IP-адресу направленной передачи интерфейса происходит так же, как для функции `mcast_join`.

Функция `mcast_set_loop` устанавливает параметр закольцовки либо в 0, либо в 1, а функция `mcast_set_ttl` TTL в случае IPv4 или предел количества транзитных узлов в случае IPv6. Функции `mcast_get_XXX` возвращают соответствующие значения.

Пример: функция `mcast_join`

В листинге 21.1^[1] показана первая часть функции `mcast_join`. Эта часть демонстрирует простоту интерфейса программирования, не зависящего от протокола.

Листинг 21.1. Присоединение к группе: сокет IPv4

```
//lib/mcast_join.c
1 #include "unp.h"
2 #include <net/if.h>

3 int
4 mcast_join(int sockfd, const SA *grp, socklen_t grplen,
5 const char *ifname, u_int ifindex)
6 {
7 #ifdef MCAST_JOIN_GROUP
8     struct group_req req;
9     if (ifindex > 0) {
10         req.gr_interface = ifindex;
11     } else if (ifname != NULL) {
12         if ((req.gr_interface = if_nametoindex(ifname)) == 0) {
13             errno = ENXIO; /* интерфейс не найден */
14             return(-1);
15         }
16     } else
17         req.gr_interface = 0;
18     if (grplen > sizeof(req.gr_group)) {
19         errno = EINVAL;
20         return -1;
21     }
22     memcpy(&req.gr_group, grp, grplen);
23     return (setsockopt(sockfd, family_to_level(grp->sa_family),
24 MCAST_JOIN_GROUP, &req, sizeof(req)));
25 }
```

Обработка индекса

9-17 Если при вызове был указан индекс интерфейса, функция использует его непосредственно. В противном случае (при указании имени интерфейса), имя преобразуется в индекс вызовом `if_nametoindex`. Если ни имя, ни индекс не заданы, интерфейс выбирается ядром.

Копирование адреса и вызов setsockopt

18-22 Адрес сокета копируется непосредственно в поле группы. Вспомните, что поле это имеет тип `sockaddr_storage`, а потому достаточно велико для хранения адреса любого типа, поддерживаемого системой. Для предотвращения переполнения буфера (при ошибках в программе) мы проверяем размер `sockaddr` и возвращаем `EINVAL`, если он слишком велик.

23-24 Присоединение к группе выполняется вызовом `setsockopt`. Аргумент `level` определяется на основании семейства группового адреса вызовом нашей собственной функции `family_to_level`. Некоторые системы допускают несоответствие аргумента `level` семейству адреса сокета, например использование `IPPROTO_IP` с `MCast_JOIN_GROUP`, даже если сокет относится к семейству `AF_INET6`, но это верно не для всех систем, поэтому мы и должны выполнить преобразование семейства к нужному значению `level`. Листинг этой тривиальной функции в книге мы не приводим, но исходный код этой функции вы можете скачать вместе со всеми остальными программами.

В листинге 21.2 представлена вторая часть функции `mcast_join`, обрабатывающая сокеты IPv4.

Листинг 21.2. Присоединение к группе: обработка сокета IPv4

```
26 switch (grp->sa_family) {
27 case AF_INET: {
28     struct ip_mreq mreq;
29     struct ifreq ifreq;

30     memcpy(&mreq.imr_multiaddr,
31         &((const struct sockaddr_in*)grp)->sin_addr,
32         sizeof(struct in_addr));

33     if (ifindex > 0) {
34         if (if_indextoname(ifindex, ifreq.ifr_name) == NULL) {
35             errno = ENXIO; /* i/f index not found */
36             return(-1);
37         }
38         goto doioctl;
39     } else if (ifname != NULL) {
40         strncpy(ifreq.ifr_name, ifname, IFNAMSIZ);
41     doioctl:
42         if (ioctl(sockfd, SIOCGIFADDR, &ifreq) < 0)
43             return(-1);
44         memcpy(&mreq.imr_interface,
45             &((struct sockaddr_in*)&ifreq.ifr_addr)->sin_addr,
46             sizeof(struct in_addr));
47     } else
48         mreq.imr_interface.s_addr = htonl(INADDR_ANY);

49     return(setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
50         &mreq, sizeof(mreq)));
51 }
```

Обработка индекса

33-38 Адрес многоадресной передачи IPv4 в структуре адреса сокета копируется в структуру `ip_mreq`. Если индекс был задан, вызывается функция `if_indextoname`, сохраняющая имя в нашей структуре `ip_mreq`. Если это выполняется успешно, мы переходим на точку вызова `ioctl`.

Обработка имени

39-46 Имя вызывающего процесса копируется в структуру `ip_mreq`, а вызов `SIOCGIFADDR` функции `ioctl` возвращает адрес многоадресной передачи, связанный с этим именем. При успешном выполнении адрес IPv4 копируется в элемент `imr_interface` структуры `ip_mreq`.

Значения по умолчанию

47-48 Если ни индекс, ни имя не заданы, используется универсальный адрес, что указывает ядре на необходимость выбрать интерфейс.

49-50 Функция `setsockopt` выполняет присоединение к группе.

Третья, и последняя, часть функции, обрабатывающая сокеты IPv6, приведена в листинге 21.3.

Листинг 21.3. Присоединение к группе: обработка сокета IPv6

```
52 #ifdef IPV6
53 case AF_INET6: {
54     struct ipv6_mreq mreq6;
55
56     memcpy(&mreq6.ipv6mr_multiaddr,
57           &((const struct sockaddr_in6*) grp)->sin6_addr,
58           sizeof(struct in6_addr));
59
60     if (ifindex > 0) {
61         mreq6.ipv6mr_interface = ifindex;
62     } else if (ifname != NULL) {
63         if ((mreq6.ipv6mr_interface = if_nametoindex(ifname)) == 0) {
64             errno = ENXIO; /* интерфейс не найден */
65             return(-1);
66         }
67     } else
68         mreq6.ipv6mr_interface = 0;
69
70     return(setsockopt(sockfd, IPPROTO_IP, IPV6_JOIN_GROUP,
71                      &mreq6, sizeof(mreq6)));
72 }
73#endif
74
75 default:
76     errno = EAFNOSUPPORT;
77     return(-1);
78 }
79#endif
80 }
```

Копирование адреса

55-57 Сначала адрес IPv6 копируется из структуры адреса сокета в структуру `ipv6_mreq`.

Обработка индекса или имени интерфейса или выбор интерфейса по умолчанию

58-66 Если был задан индекс, он записывается в элемент `ipv6mr_interface`. Если индекс не задан, но задано имя, то для получения индекса вызывается функция `if_nametoindex`. В противном случае для функции `setsockopt` индекс устанавливается в 0, что указывает ядре на необходимость выбрать интерфейс.

67-68 Выполняется присоединение к группе.

Пример: функция `mcast_set_loop`

В листинге 21.4 показана наша функция `mcast_set_loop`.

Поскольку аргументом является дескриптор сокета, а не структура адреса сокета, мы вызываем нашу функцию `sockfd_to_family`, чтобы получить семейство адресов сокета. Устанавливается соответствующий параметр сокета.

Мы не показываем исходный код для всех остальных функций `mcast_XXX`, так как он свободно доступен в Интернете (см. предисловие).

Листинг 21.4. Установка параметра закольцовки для многоадресной передачи

```
//lib/mcast_set_loop.c
1 #include "unp.h"

2 int
3 mcast_set_loop(int sockfd, int onoff)
4 {
5     switch (sockfd_to_family(sockfd)) {
6     case AF_INET:{
7         u_char flag;

8         flag = onoff;
9         return (setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_LOOP,
10             &flag, sizeof(flag)));
11     }

12 #ifdef IPV6
13     case AF_INET6:{
14         u_int flag;

15         flag = onoff;
16         return (setsockopt(sockfd, IPPROTO_IPV6, IPV6_MULTICAST_LOOP,
17             &flag, sizeof(flag)));
18     }
19 #endif

20     default:
21         errno = EPROTONOSUPPORT;
22         return (-1);
23     }
24 }
```

21.8 Функция `dg_cli`, использующая многоадресную передачу

Мы изменяем нашу функцию `dg_cli`, показанную в листинге 20.1, просто удаляя вызов функции `setsockopt`. Как мы сказали ранее, для отправки дейтаграмм многоадресной передачи не нужно устанавливать ни одного параметра сокета многоадресной передачи, если нас устраивают заданные по умолчанию настройки интерфейса исходящих пакетов, значения TTL и параметра закольцовки. Мы запускаем нашу программу, задавая в качестве адреса получателя группу всех узлов (`all-hosts group`):

```
macosx % udpccli01 224.0.1.1
hi there
from 172.24.37.78: hi there MacOS X
from 172.24.37.94: hi there FreeBSD
```

Отвечают оба узла, находящиеся в подсети. На обоих работают многоадресные эхо-серверы. Каждый ответ является направленным, поскольку адрес отправителя запроса, используемый сервером в качестве адреса получателя ответа, является адресом направленной передачи.

Фрагментация IP и многоадресная передача

В конце раздела 20.4 мы отмечали, что в большинстве систем фрагментация широковещательнойдейтаграммы не допускается по стратегическим соображениям. Фрагментация допускается при многоадресной передаче, что мы можем легко проверить, используя тот же файл с 2000-байтовой строкой:

```
macosx % udpcli01 224.0.1.1 < 2000line
from 172.24.37.78: xxxxxxxx[...]
from 172.24.37.94: xxxxxxxx[...]
```

21.9. Получение анонсов сеансов многоадресной передачи

Многоадресная инфраструктура представляет собой часть Интернета, в которой разрешена многоадресная передача между доменами. Во всем Интернете многоадресная передача не разрешена. Многоадресная инфраструктура Интернета начала свое существование в 1992 году. Тогда она называлась MBone и была оверлейной сетью. В 1998 году MBone была признана частью инфраструктуры Интернета. Внутри предприятий многоадресная передача используется достаточно широко, но междоменная передача поддерживается гораздо меньшим числом серверов.

Для участия в мультимедиа-конференции по сети многоадресной передачи достаточно того, чтобы сайту был известен групповой адрес конференции и порты UDP для потоков данных (например, аудио и видео). Протокол анонсирования сеансов (*Session Announcement Protocol, SAP*) определяет эту процедуру, описывая заголовки пакетов и частоту, с которой эти анонсы при помощи многоадресной передачи передаются по инфраструктуре многоадресной передачи. Этот протокол описан в RFC 2974 [42]. Протокол описания сеанса (*Session Description Protocol, SDP*) [41] описывает технические параметры сеанса связи (в частности, он определяет, как задаются адреса многоадресной передачи и номера портов UDP). Сайт, желающий анонсировать сеанс, периодически посылает пакет многоадресной передачи, содержащий описание сеанса, для известной группы на известный порт UDP. Для получения этих анонсов сайты запускают программу под названием *sdr*. Эта программа не только получает объявления сеансов, но и предоставляет интерактивный интерфейс пользователя, позволяющий пользователю отправлять свои собственные анонсы.

В этом разделе мы продемонстрируем прием пакетов многоадресной передачи, создав пример простой программы, лишь получающей анонсы сеансов. В данном случае мы стремимся показать простоту устройства получателя пакетов при многоадресной передаче, а не исследовать подробности конкретного приложения.

В листинге 21.5 показана наша программа *main*, получающая периодические анонсы SAP/SDP.

Листинг 21.5. Программа *main*, получающая периодические анонсы SAP/SDP

```
//mysdr/main.c
1 #include "unp.h"

2 #define SAP_NAME "sap.mcast.net" /* имя группы и порт по умолчанию */
3 #define SAP_PORT "9875"

4 void loop(int, socklen_t);

5 int
6 main(int argc, char **argv)
7 {
8     int sockfd;
9     const int on = 1;
10    socklen_t salen;
11    struct sockaddr *sa;

12    if (argc == 1)
13        sockfd = Udp_client(SAP_NAME, SAP_PORT, (void**)&sa, &salen);
14    else if (argc == 4)
15        sockfd = Udp_client(argv[1], argv[2], (void**)&sa, &salen);
16    else
17        err_quit("usage: mysdr <mcast-addr> <port#> <interface-name>");
```

```

18 Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
19 Bind(sockfd, sa, salen);

20 Mcast_join(sockfd, sa, salen, (argc == 4) ? argv[3], NULL, 0);

21 loop(sockfd, salen); /* получение и вывод */

22 exit(0);
23 }

```

Заранее известные имя и порт

2-3 Адрес многоадресной передачи, заданный для анонсов SAP — 224.2.127.254, а его имя — sap.mcast.net. Все заранее известные адреса многоадресной передачи (см. <http://www.iana.org/assignments/multicast-addresses>) появляются в DNS в иерархии mcast.net. Заранее известный порт UDP — это порт 9875.

Создание сокета UDP

12-17 Мы вызываем нашу функцию `udp_client`, чтобы просмотреть имя и порт, и она заполняет соответствующую структуру адреса сокета. Если не заданы аргументы командной строки, мы используем значения по умолчанию. В противном случае мы получаем адрес многоадресной передачи, порт и имя интерфейса из аргументов командной строки.

Связывание порта с помощью функции bind

18-19 Мы устанавливаем параметр сокета `SO_REUSEADDR`, чтобы позволить множеству экземпляров этой программы запуститься на узле, и с помощью функции `bind` связываем порт с сокетом. Связывая адрес многоадресной передачи с сокетом, мы запрещаем сокету получать какие-либо другие дейтаграммы UDP, которые могут быть получены для этого порта. Связывание этого адреса многоадресной передачи не является обязательным, но оно обеспечивает возможность фильтрации, благодаря чему ядро может не принимать пакеты, которые его не интересуют.

Присоединение к группе

20 Мы вызываем нашу функцию `mcast_join`, чтобы присоединиться к группе. Если имя интерфейса было задано в качестве аргумента командной строки, оно передается нашей функции, иначе мы позволяем ядру выбрать интерфейс, на котором будет происходить присоединение к группе.

21 Мы вызываем нашу функцию `loop`, показанную в листинге 21.6, чтобы прочитать и вывести все анонсы.

Листинг 21.6. Цикл, получающий и выводящий анонсы SAP/SDP

```

//mysdr/loop.c
1 #include "mysdr.h"

2 void
3 loop(int sockfd, socklen_t salen)
4 {
5     socklen_t len;
6     ssize_t n;
7     char *p;
8     struct sockaddr *sa;
9     struct sap_packet {

```

```

10     uint32_t sap_header;
11     uint32_t sap_src;
12     char      sap_data[BUFFSIZE];
13 } buf;

14 sa = Malloc(salen);

15 for (;;) {
15   len = salen;
17   n = Recvfrom(sockfd, &buf, sizeof(buf) - 1, 0, sa, &len);
18   ((char *)&buf)[n] = 0; /* завершающий нуль */
19   buf.sap_header = ntohs(buf.sap_header);
20   printf("From %s hash 0x%0x\n" Sock_ntop(sa, len),
21         buf.sap_header & SAP_HASH_MASK);
22   if (((buf.sap_header & SAP_VERSION_MASK) >> SAP_VERSION_SHIFT) > 1) {
23     err_msg("... version field not 1 (0x%08x)", buf.sap_header);
24     continue;
25   }
26   if (buf.sap_header & SAP_IPV6) {
27     err_msg("... IPv6");
28     continue;
29   }
30   if (buf.sap_header & (SAP_DELETE|SAP_ENCRYPTED|SAP_COMPRESSED)) {
31     err_msg("... can't parse this packet type (0x%08x)",
32             buf.sap_header);
33     continue;
34   }
35   p = buf.sap_data + ((buf.sap_header & SAP_AUTHLEN_MASK)
36   >> SAP_AUTHLEN_SHIFT);
37   if (strcmp(p, "application/sdp") == 0)
38     p += 16;
39   printf("%s\n", p);
40 }
41 }
```

Формат пакета

9-13 Структура `sap_packet` описывает пакет SDP: 32-разрядный заголовок SAP, за которым следует 32-разрядный адрес отправителя и сам анонс. Анонс представляет собой строки текста в стандарте ISO 8859-1 и не может превышать 1024 байта. В каждой дейтаграмме UDP допускается только один анонс сеанса.

Чтение дейтаграммы UDP, вывод параметров отправителя и содержимого

15-21 Функция `recvfrom` ждет следующую дейтаграмму UDP, предназначенную нашему сокету. Когда она приходит, мы помещаем в конец буфера пустой байт, исправляем порядок байтов заголовка и выводим адрес отправителя пакета и хэш SAP.

Проверка заголовка SAP

22-34 Мы проверяем заголовок SAP, чтобы убедиться, что он относится к одному из тех типов, с которыми мы умеем работать. Пакеты SAP с адресами IPv6 в заголовках, а также сжатые и зашифрованные пакеты мы не обрабатываем.

Поиск начала и вывод анонса

35-39 Мы пропускаем аутентифицирующие данные и тип пакета, после чего выводим содержимое оставшейся части.

В листинге 21.7 показано несколько типичных примеров результата выполнения нашей программы.

Листинг 21.7. Типичный анонс SAP/SDP

```
freebsd % mysdr
From 128.223.83.33:1028 hash 0x0000 v=0
o=- 60345 0 IN IP4 128.223.214.198
s=U0 Broadcast - NASA Videos - 25 Years of Progress
i=25 Years of Progress, parts 1-13. Broadcast with Cisco System's
IP/TV using MPEG1 codec (6 hours 5 Minutes; repeats) More information
about IP/TV and the client needed to view this program is available
from http://videolab.uoregon.edu/download.html
u=http://videolab.uoregon.edu/
e=Hans Kuhn <multicast@lists.uoregon.edu>
p=Hans Kuhn <541/346-1758>
b=AS:1000
t=0 0
a=type:broadcast
a=tool:IP/TV Content Manager 3.2.24
a=x-iptv-file:1 name y:25yop1234567890123.mpg
m=video 63096 RTP/AVP 32 31 96
c=IN IP4 224.2.245.25/127
a=framerate:30
a=rtpmap:96 WBIH/90000
a=x-iptv-srvr:video blaster2.uoregon.edu file 1 loop
m=audio 31954 RTP/AVP 14 96 0 3 5 97 98 99 100 101 102 10 11 103 104 105 106
c=IN IP4 224.2.216.85/127
a=rtpmap:96 X-WAVE/8000
a=rtpmap:97 L8/8000/2
a=rtpmap:98 L8/8000
a=rtpmap:99 L8/22050/2
a=rtpmap:100 L8/22050
a=rtpmap:101 L8/11025/2
a=rtpmap:102 L8/11025
a=rtpmap:103 L16/22050/2
a=rtpmap:104 L16/22050
a=rtpmap:105 L16/11025/2
a=rtpmap:106 L16/11025
a=x-iptv-srvr:audio blaster2.uoregon.edu file 1 loop
```

Этот анонс описывает рассылки, посвященные истории NASA (National Aeronautics and Space Administration — НАСА, государственная организация США, занимающаяся исследованием космоса). Описание сеанса SDP состоит из множества строк следующего формата:

type=*value*

где *type* всегда является одним символом, значение которого зависит от регистра, а *value* — это структурированная текстовая строка, зависящая от значения *type*. Пробелы справа и слева от знака равенства недопустимы. *v=0* (в нашем случае) обозначает версию (version).

■ *o*= обозначает источник (origin). В данном случае имя пользователя не указано, 60345 — идентификатор сеанса, 0 — номер версии этого сеанса, IN — тип сети, IP4 — тип адреса, 128.223.214.198 — адрес. В результате объединения этих пяти элементов — имя пользователя, идентификатор сеанса, тип сети, тип адреса и адрес — образуется глобально уникальный идентификатор сеанса.

■ *s*= задает имя сеанса (session name), а *i*= — это информация о сеансе (information). *u*= указывает URI (Uniform Resource Identifier — уникальный идентификатор ресурса), по которому можно найти более

подробную информацию по тематике данного сеанса, а `p=` и `e=` задают номер телефона (phone number) и адрес электронной почты (e-mail) ответственного за данную конференцию.

- `b=` позволяет оценить пропускную способность, необходимую для приема данного сеанса.
- `t=` задает время начала и время окончания сеанса в единицах NTP (Network Time Protocol — синхронизирующий сетевой протокол), то есть число секунд, прошедшее с 1 января 1900 года, измеренное в соответствии с UTC (Universal Time Coordinated — универсальное скоординированное время). Данный сеанс является постоянным и не имеет конкретных моментов начала и окончания, поэтому соответствующие времена полагаются нулевыми.
- Строки `a=` представляют собой атрибуты, либо сеанса, если они помещены до первой строки `m=`, либо мультимедиа, если они помещены после первой строки `m=`.
- Строки `m=` — это анонсы мультимедиа. Первая строка говорит нам о том, что видео передается на порт 63 096 в формате RTP с использованием профиля аудио и видео (Audio/Video Profile, AVP) с возможными типами данных 32, 31 и 96 (то есть MPEG, H.261 и WBH соответственно). Стока `c=` сообщает о соединении. В данном случае используется протокол IPv4 с групповым адресом 224.2.245.25 и TTL = 127. Хотя между этими числами стоит символ «косая черта», как в формате CIDR, они ни в коем случае не должны трактоваться как префикс и маска.

Следующая строка `m=` говорит, что аудиопоток передается на порт 31 954 и может иметь один из типов RTP/AVP, некоторые из которых являются стандартными, в то время как другие указаны ниже в виде атрибутов `a=rtpmap::`. Стока `c=` сообщает нам сведения об аудиосоединении: IPv4 с групповым адресом 224.2.216.85 и TTL = 127.

21.10. Отправка и получение

Программа для получения анонсов сеанса многоадресной передачи, показанная в предыдущем разделе, могла только получать дейтаграммы многоадресной передачи. Теперь мы создадим простую программу, способную и отправлять, и получать дейтаграммы многоадресной передачи. Наша программа состоит из двух частей. Первая часть отправляет дейтаграмму многоадресной передачи определённой группе каждые 5 с. Эта дейтаграмма содержит имя узла отправителя и идентификатор процесса. Вторая часть программы — это бесконечный цикл, присоединяющийся к той группе, которой первая часть программы отправляет данные. В этом цикле выводится каждая полученная дейтаграмма (содержащая имя узла и идентификатор процесса отправителя). Это позволяет нам запустить программу на множестве узлов в локальной сети и посмотреть, какой узел получает дейтаграммы от каких отправителей.

В листинге 21.8 показана функция `main` нашей программы.

Листинг 21.8. Создание сокетов, вызов функции `fork` и запуск отправителя и получателя

```
//mcast/main.c
1 #include "unp.h"

2 void recv_all(int, socklen_t);
3 void send_all(int, SA *, socklen_t);

4 int
5 main(int argc, char **argv)
6 {
7     int sendfd, recvfd;
8     const int on = 1;
9     socklen_t salen;
10    struct sockaddr *sasend, *sarecv;

11    if (argc != 3)
12        err_quit("usage: sendrecv <IP-multicast-address> <port#>");

13    sendfd = Udp_client(argv[1], argv[2], (void**)&sasend, &salen);

14    recvfd = Socket(sasend->sa_family, SOCK_DGRAM, 0);

15    Setsockopt(recvfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
```

```

16  sarecv = Malloc(salen);
17  memcpy(sarecv, sasend, salen);
18  Bind(recvfd, sarecv, salen);

19  Mcast_join(recvfd, sasend, salen, NULL, 0);
20  Mcast_set_loop(sendfd, 0);

21  if (Fork() == 0)
22      recv_all(recvfd, salen); /* дочерний процесс -> получение */

23  send_all(sendfd, sasend, salen); /* родитель -> отправка */
24 }

```

Мы создаем два сокета, один для отправки и один для получения. Нам нужно, чтобы принимающий сокет связался при помощи функции `bind` с группой и портом, допустим 239.255.1.2, порт 8888. (Вспомните, что мы могли просто связать универсальный IP-адрес и порт 8888, но связывание с определенным адресом многоадресной передачи предотвращает получение сокетом других дейтаграмм, которые могут прийти на порт получателя 8888.) Далее, нам нужно, чтобы принимающий сокет присоединился к группе. Отправляющий сокет будет отправлять дейтаграммы на этот же адрес многоадресной передачи и этот же порт, то есть на 239.255.1.2, порт 8888. Но если мы попытаемся использовать один сокет и для отправки, и для получения, то адресом отправителя для функции `bind` будет 239.255.1.2.8888 (здесь используется нотация `netstat`), а адресом получателя для функции `sendto` — также 239.255.1.2.8888. Но адрес отправителя, связанный с сокетом, становится IP-адресом отправителя дейтаграммы UDP, а RFC 1122 [10] запрещает дейтаграмме IP иметь IP-адрес отправителя, являющийся адресом многоадресной или широковещательной передачи. (См. также упражнение 21.2.) Следовательно, мы создаем два сокета: один для отправки, другой для получения.

Создание отправляющего сокета

13 Наша функция `udp_client` создает отправляющий сокет, обрабатывая два аргумента командной строки, которые задают адрес многоадресной передачи и номер порта. Эта функция также возвращает структуру адреса сокета, готовую к вызовам функции `sendto`, и длину этой структуры.

Создание принимающего сокета и связывание (при помощи функции `bind`) с адресом многоадресной передачи и портом

14-18 Мы создаем принимающий сокет, используя то же семейство адресов, что и при создании отправляющего сокета, и устанавливаем параметр сокета `SO_REUSEADDR`, чтобы разрешить множеству экземпляров этой программы одновременно запускаться на узле. Затем мы выделяем в памяти пространство для структуры адреса этого сокета, копируем ее содержимое из структуры адреса отправляющего сокета (адрес и порт которого взяты из аргументов командной строки) и при помощи функции `bind` связываем адрес многоадресной передачи и порт с принимающим сокетом.

Присоединение к группе и выключение закольцовки

19-20 Мы вызываем нашу функцию `mcast_join`, чтобы присоединиться к группе на получающем сокете, а также нашу функцию `mcast_set_loop`, чтобы отключить закольцовку на отправляющем сокете. Для присоединения задаем имя интерфейса в виде пустого указателя и нулевой индекс интерфейса, что указывает ядру на необходимость выбрать интерфейс самостоятельно.

Функция `fork` и вызов соответствующих функций

21-23 Мы вызываем функцию `fork`, после чего дочерним процессом становится получающий цикл, а родительским — отправляющий.

Наша функция `sendmail`, отправляющая по одной дейтаграмме многоадресной передачи каждые 5 с, показана в листинге 21.9. Функция `main` передает в качестве аргументов дескриптор сокета, указатель на структуру адреса сокета, содержащую адрес получателя многоадресной передачи и порт, и длину структуры.

Листинг 21.9. Отправка дейтаграммы многоадресной передачи каждые 5 с

```
//mcast/send.c
1 #include "unp.h"
2 #include <sys/utsname.h>

3 #define SENDRATE 5 /* отправка дейтаграмм каждые 5 с */

4 void
5 send_all(int sendfd, SA *sadest, socklen_t salen)
6 {
7     static char line[MAXLINE]; /* имя узла и идентификатор процесса */
8     struct utsname myname;

9     if (uname(&myname) < 0)
10    err_sys("uname error");
11    sprintf(line, sizeof(line), "%s, %d\n", myname, nodename, getpid());

12    for (;;) {
13        Sendto(sendfd, line, strlen(line), 0, sadest, salen);
14        sleep(SENDRATE);
15    }
16 }
```

Получение имени узла и формирование содержимого дейтаграммы

9-11 Мы получаем имя узла из функции `uname` и создаем строку вывода, содержащую это имя и идентификатор процесса.

Отправка дейтаграммы, переход в режим ожидания

12-15 Мы отправляем дейтаграмму и с помощью функции `sleep` переходим в состояние ожидания на 5 с.

Функция `recv_all`, содержащая бесконечный цикл получения, показана в листинге 21.10.

Листинг 21.10. Получение всех дейтаграмм многоадресной передачи для группы, к которой мы присоединились

```
//mcast/recv.c
1 #include "unp.h"

2 void
3 recv_all(int recvfd, socklen_t salen)
4 {
5     int n;
6     char line[MAXLINE + 1];
7     socklen_t len;
8     struct sockaddr *safrom;

9     safrom = Malloc(salen);
```

```
10  for (;;) {
11      len = salen;
12      n = Recvfrom(recvfd, line, MAXLINE, 0, safrom, &len);
13
14      line[n] = 0; /* завершающий нуль */
15      printf("from %s: %s", Sock_ntop(safrom, len), line);
16  }
```

Размещение в памяти структуры адреса сокета

9 При каждом вызове функции `recvfrom` в памяти выделяется пространство для структуры адреса сокета, в которую записывается адрес отправителя.

Чтение и вывод дейтаграмм

10-15 Каждая дейтаграмма считывается функцией `recvfrom`, дополняется символом конца строки (то есть нулем) и выводится.

Пример

Мы запускаем программу в двух системах: freebsd4 и macosx. Каждая система видит пакеты, отправляемые другой.

```
freebsd4 % sendrecv 239.255.1.2 8888
from 172.24.37.78:51297: macosx, 21891
from 172.24.37.78:51297: macosx, 21891
from 172.24.37.78:51297: macosx, 21891
from 172.24.37.78:51297: macosx, 21891

macosx % sendrecv 239.255.1.2 8888
from 172.24.37.94.1215: freebsd4, 55372
from 172.24.37.94.1215: freebsd4, 55372
from 172.24.37.94.1215: freebsd4, 55372
from 172.24.37.94.1215: freebsd4, 55372
```

21.11. SNTP: простой синхронизирующий сетевой протокол

Синхронизирующий сетевой протокол (Network Time Protocol, NTP) — это сложный протокол синхронизации часов в глобальной или локальной сети. Его точность часто может достигать миллисекунд. В RFC 1305 [76] этот протокол подробно описан, а в RFC 2030 [77] рассматривается протокол SNTP — упрощенная версия NTP, предназначенная для узлов, которым не требуется функциональность полной реализации NTP. Типичной является ситуация, когда несколько узлов в локальной сети синхронизируют свои часы через Интернет с другими узлами NTP, а затем распространяют полученное значение времени в локальной сети с использованием либо широковещательной, либо многоадресной передачи.

В этом разделе мы создадим клиент SNTP, который прослушивает широковещательные или групповые сообщения NTP на всех присоединенных сетях, а затем выводит разницу во времени между пакетом NTP и текущим истинным временем узла. Мы не пытаемся изменить это время, поскольку для этого необходимы права привилегированного пользователя.

Файл `ntp.h`, показанный в листинге 21.11, содержит некоторые из основных определений формата пакета NTP.

Листинг 21.11. Заголовок `ntp.h`: формат пакета NTP и определения

```
//ssntp/ntp.h
1 #define JAN_1970 2208988800UL /* 1970 - 1900 в секундах */
```

```

2 struct l_fixedpt { /* 64-разрядное число с фиксированной точкой */
3     uint32_t int_part;
4     uint32_t fraction;
5 };

6 struct s_fixedpt { /* 32-разрядное число с фиксированной точкой */
7     u_short int_part;
8     u_short fraction;
9 };

10 struct ntpdata { /* заголовок NTP */
11     u_char status;
12     u_char stratum;
13     u_char ppoll;
14     int    precision:8;
15     struct s_fixedpt distance;
16     struct s_fixedpt dispersion;
17     uint32_t refid;
18     struct l_fixedpt reftime;
19     struct l_fixedpt org;
20     struct l_fixedpt rec;
21     struct l_fixedpt xmt;
22 };

23 #define VERSION_MASK 0x38
24 #define MODE_MASK 0x07

25 #define MODE_CLIENT 3
26 #define MODE_SERVER 4
27 #define MODE_BROADCAST 5

```

2-22 `l_fixedpt` задает 64-разрядные числа с фиксированной точкой, используемые NTP для отметок времени, а `s_fixedpt` — 32-разрядные значения с фиксированной точкой, также используемые NTP. Структура `ntpdata` представляет 48-байтовый формат пакета NTP.

В листинге 21.12 представлена функция `main`.

Листинг 21.12. Функция main

```

//ssntp/main.c
1 #include "sntp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;
6     char buf[MAXLINE];
7     ssize_t n;
8     socklen_t salen, len;
9     struct ifi_info *ifi;
10    struct sockaddr *mcastsa, *wild, *from;
11    struct timeval now;

12    if (argc != 2)
13        err_quit("usage: ssntp <Ipadress>");

14    sockfd = Udp_client(argv[1], "ntp", (void**)&mcastsa, &salen);

15    wild = Malloc(salen);

```

```

16 memcpy(wild, mcastsa. salen); /* копируем семейство и порт */
17 sock_set_wild(wild, salen);
18 Bind(sockfd, wild, salen); /* связываем сокет с универсальным[3]
адресом */
19 #ifdef MCAST
20 /* получаем список интерфейсов и обрабатываем каждый интерфейс */
21 for (ifi = Get_ifi_info(mcastsa->sa_family, 1); ifi != NULL;
22 ifi = ifi->ifi_next) {
23 if (ifi->ifi_flags & IFF_MULTICAST) {
24 Mcast_join(sockfd, mcastsa, salen, ifi->ififname, 0);
25 printf("joined %s on %s\n",
26 Sock_ntop(mcastsa, salen), ifi->ifif_name);
27 }
28 }
29#endif
30 from = Malloc(salen);
31 for (;;) {
32 len = salen;
33 n = Recvfrom(sockfd, buf, sizeof(buf), 0, from, &len);
34 Gettimeofday(&now, NULL);
35 sntp_proc(buf, n, &now);
36 }
37 }

```

Получение IP-адреса многоадресной передачи

12-14 При выполнении программы пользователь должен задать в качестве аргумента командной строки адрес многоадресной передачи, к которому он будет присоединяться. В случае IPv4 это будет 224.0.1.1 или имя ntp.mcast.net. В случае IPv6 это будет ff05::101 для области действия NTP, локальной в пределах сайта. Наша функция `udp_client` выделяет в памяти пространство для структуры адреса сокета корректного типа (либо IPv4, либо IPv6) и записывает адрес многоадресной передачи и порт в эту структуру. Если эта программа выполняется на узле, не поддерживающем многоадресную передачу, может быть задан любой IP-адрес, так как в этой структуре задействуются только семейство адресов и порт. Обратите внимание, что наша функция `udp_client` не связывает адрес с сокетом (то есть не вызывает функцию `bind`) — она лишь создает сокет и заполняет структуру адреса сокета.

Связывание универсального адреса с сокетом

15-18 Мы выделяем в памяти пространство для другой структуры адреса сокета и заполняем ее, копируя структуру, заполненную функцией `udp_client`. При этом задаются семейство адреса и порт. Мы вызываем нашу функцию `sock_set_wild`, чтобы присвоить IP-адресу универсальный адрес, а затем вызываем функцию `bind`.

Получение списка интерфейсов

20-22 Наша функция `get_ifi_info` возвращает информацию обо всех интерфейсах и адресах. Запрашиваемое нами семейство адреса берется из структуры адреса сокета, заполненной функцией `udp_client` на основе аргумента командной строки.

Присоединение к группе

23-27 Мы вызываем нашу функцию `mcast_join`, чтобы присоединиться к группе, заданной аргументом командной строки для каждого интерфейса, поддерживающего многоадресную передачу. Все эти присоединения происходят на одном сокете, который использует эта программа. Как отмечалось ранее, количество присоединений на одном сокете ограничено константой `IP_MAX_MEMBERSHIPS` (которая обычно имеет значение 20), но лишь немногие многоинтерфейсные узлы используют столько интерфейсов.

Чтение и обработка всех пакетов NTP

30-36 В памяти размещается другая структура адреса сокета для хранения адреса, возвращаемого функцией `recvfrom`, и программа входит в бесконечный цикл, считывая все пакеты NTP, которые получает узел, и вызывая нашу функцию `sntp_proc` (описывается далее) для обработки пакета. Поскольку сокет был связан с универсальным адресом и присоединение к группе произошло на всех интерфейсах, поддерживающих многоадресную передачу, сокет должен получить любой пакет NTP направленной, широковещательной или многоадресной передачи, получаемый узлом. Перед вызовом функции `sntp_proc` мы вызываем функцию `gettimeofday`, чтобы получить текущее время, потому что функция `sntp_proc` вычисляет разницу между временем пакета и текущим временем.

Наша функция `sntp_proc`, показанная в листинге 21.13, обрабатывает пакет NTP.

Листинг 21.13. Функция `sntp_proc`: обработка пакета NTP

```
//ssntp/sntp_proc.c
1 #include "sntp.h"

2 void
3 sntp proc(char *buf, ssize_t n, struct timeval *nowptr)
4 {
5     int version, mode;
6     uint32_t nsec, useci;
7     double usecf;
8     struct timeval diff;
9     struct ntpdata *ntp;

10    if (n < (ssize_t)sizeof(struct ntpdata)) {
11        printf("\npacket too small: %d bytes\n", n);
12        return;
13    }

14    ntp = (struct ntpdata*)buf;
15    version = (ntp->status & VERSION_MASK) >> 3;
16    mode = ntp->status & MODE_MASK;
17    printf("\nv%d, mode %d, strat %d, ", version, mode, ntp->stratum);
18    if (mode == MODE_CLIENT) {
19        printf("client\n");
20        return;
21    }

22    nsec = ntohl(ntp->xmt.int_part) - JAN_1970;
23    useci = ntohl(ntp->xmt.fraction); /* 32-разрядная дробь */
24    usecf = useci; /* дробь в double */
25    usecf /= 4294967296.0; /* деление на 2**32 -> [0, 1.0) */
26    useci = usecf * 1000000.0; /* дробь в миллионную часть */

27    diff.tv_sec = nowptr->tv_sec - nsec;
28    if ((diff.tv_usec = nowptr->tv_usec - useci) < 0) {
29        diff.tv_usec += 1000000;
30        diff.tv_sec--;
31    }
```

```
32 useci = (diff.tv_sec * 1000000) + diff.tv_usec; /* diff в мс */
33 printf("clock difference = %d usec\n", useci);
34 }
```

Ратификация пакета

10-21 Сначала мы проверяем размер пакета, затем выводим его версию, режим и слой (stratum) сервера. Если режимом является MODE_CLIENT, пакет является запросом клиента, а не ответом сервера, и мы игнорируем его.

Получение времени передачи из пакета NTP

22-34 В пакете NTP нас интересует поле xmt — отметка времени. Это 64-разрядное значение с фиксированной точкой, определяющее момент отправки пакета сервером. Поскольку отметки времени NTP отсчитывают секунды начиная с 1 января 1900 года, а отметки времени Unix — с 1 января 1970 года, сначала мы вычитаем JAN_1970 (число секунд в 70 годах) из целой части.

Дробная часть — это 32-разрядное целое без знака, которое может принимать значение от 0 до 4 294 967 295 включительно. Оно копируется из 32-разрядного целого (usecf) в переменную с плавающей точкой двойной точности (usecf) и делится на 4 294 967 296 (2^{32}). Результат больше либо равен 0.0 и меньше 1.0. Мы умножаем это число на 1 000 000 — число микросекунд в секунде, записывая результат в переменную useci как 32-разрядное целое без знака.

Число микросекунд лежит в интервале от 0 до 999 999 (см. упражнение 21.5). Мы преобразуем значение в микросекунды, поскольку отметка времени Unix, возвращаемая функцией gettimeofday, возвращается как два целых числа: число секунд и число микросекунд, прошедшее с 1 января 1970 года (UTC). Затем мы вычисляем и выводим разницу между истинным временем узла и истинным временем сервера NTP в микросекундах.

Один из факторов, не учитываемых нашей программой, — это задержка в сети между клиентом и сервером. Но мы считаем, что пакеты NTP обычно приходят как широковещательные или многоадресные пакеты в локальной сети, а в этом случае задержка в сети составит всего несколько миллисекунд.

Если мы запустим эту программу на узле macosx с сервером NTP на узле freebsd4, который с помощью многоадресной передачи отправляет пакеты NTP в сеть Ethernet каждые 64 с, то получим следующий результат:

```
macosx # ssntp 224.0.1.1
joined 224.0.1.1.123 on lo0
joined 224.0.1.1.123 on en1
v4, mode 5, strat 3, clock difference = 661 usec
v4, mode 5, strat 3, clock difference = -1789 usec
v4, mode 5, strat 3, clock difference = -2945 usec
v4, mode 5, strat 3, clock difference = -3689 usec
v4, mode 5, strat 3, clock difference = -5425 usec
v4, mode 5, strat 3, clock difference = -6700 usec
v4, mode 5, strat 3, clock difference = -8520 usec
```

Перед запуском нашей программы мы завершили на узле работу NTP-сервера, поэтому когда наша программа запускается, время очень близко к времени сервера. Мы видим, что этот узел отстал на 9181 мс за 384 с работы программы, то есть за 24 ч он отстанет на 2 с.

21.12. Резюме

Для запуска приложения многоадресной передачи в первую очередь требуется присоединиться к группе, заданной для этого приложения. Тем самым уровень IP получает указание присоединиться к группе, что, в свою очередь, указывает канальному уровню на необходимость получать кадры многоадресной передачи, отправляемые на соответствующий адрес многоадресной передачи аппаратного уровня. Многоадресная передача использует преимущество аппаратной фильтрации, имеющееся у большинства интерфейсных карт, и чем качественнее фильтрация, тем меньше число нежелательных

получаемых пакетов. Использование аппаратной фильтрации сокращает нагрузку на все узлы, не задействованные в приложении.

Многоадресная передача в глобальной сети требует наличия маршрутизаторов, поддерживающих многоадресную передачу, и протокола маршрутизации многоадресной передачи. Поскольку не все маршрутизаторы в Интернете имеют возможность многоадресной передачи, для этой цели используется IP-инфраструктура многоадресной передачи.

API для многоадресной передачи обеспечивают девять параметров сокетов:

- присоединение к группе на интерфейсе;
- выход из группы;
- блокирование передачи от источника;
- разблокирование заблокированного источника;
- присоединение интерфейса к группе многоадресной передачи от источника;
- выход из группы многоадресной передачи от источника;
- установка интерфейса по умолчанию для исходящих пакетов многоадресной передачи;
- установка значения TTL или предельного количества транзитных узлов для исходящих пакетов многоадресной передачи;
- включение или отключение закольцовки для пакетов многоадресной передачи.

Первые шесть параметров предназначены для получения пакетов многоадресной передачи, последние три — для отправки. Существует достаточно большая разница между указанными параметрами сокетов IPv4 и IPv6. Вследствие этого код многоадресной передачи, зависящий от протокола, очень быстро становится «замусорен» директивами `#ifdef`. Мы разработали 12 наших собственных функций с именами, начинающимися с `mcast_`, для упрощения написания приложений многоадресной передачи, работающих как с IPv4, так и с IPv6.

Упражнения

1. Скомпилируйте программу, показанную в листинге 20.5, и запустите ее, задав в командной строке IP-адрес 224.0.0.1. Что произойдет?

2. Измените программу из предыдущего примера, чтобы связать IP-адрес 224.0.0.1 и порт 0 с сокетом. Запустите ее. Разрешается ли вам связывать адрес многоадресной передачи с сокетом при помощи функции `bind`? Если у вас есть такая программа, как `tcpdump`, понаблюдайте за пакетами в сети. Каков IP-адрес отправителя посылаемой вами дейтаграммы?

3. Один из способов определить, какие узлы в вашей подсети имеют возможность многоадресной передачи, заключается в запуске утилиты `ping` для группы всех узлов, то есть для адреса 224.0.0.1. Попробуйте это сделать.

4. Одним из способов обнаружения маршрутизаторов многоадресной передачи в вашей подсети является запуск утилиты `ping` для группы всех маршрутизаторов — 224.0.0.2. Попробуйте это сделать.

5. Один из способов узнать, соединен ли ваш узел с многоадресной IP-инфраструктурой — запустить нашу программу из раздела 21.9, подождать несколько минут и посмотреть, появляются ли анонсы сеанса. Попробуйте сделать это и посмотрите, получите ли вы какие-нибудь анонсы.

6. Выполните вычисления в листинге 21.12 при условии, что дробная часть отметки времени NTP равна 1 073 741 824 (одна четвертая от 2^{32}).

Выполните еще раз эти же вычисления для максимально возможной дробной части ($2^{32} - 1$).

Измените реализацию функции `mcast_set_if` для IPv4 так, чтобы запоминать имя каждого интерфейса, для которого она получает IP-адрес. Это позволит избежать нового вызова функции `ioctl` для данного интерфейса.

Глава 22

Дополнительные сведения о сокетах udp

22.1. Введение

Эта глава объединяет различные темы, касающиеся приложений, использующих сокеты UDP. Для начала нас интересует, как определяется адрес получателя дейтаграммы UDP и интерфейс, на котором дейтаграмма была получена, поскольку сокет, связанный с портом UDP и универсальным адресом, может получать дейтаграммы направленной, широковещательной и многоадресной передачи на любом интерфейсе.

TCP — это потоковый протокол, использующий окно *переменной величины* (*sliding window*), поэтому в TCP отсутствует такое понятие, как граница записи, и невозможно переполнение буфера получателя отправителем в результате передачи слишком большого количества данных. Однако в случае UDP каждой операции ввода соответствует одна дейтаграмма UDP (запись), поэтому возникает вопрос: что произойдет, когда полученная дейтаграмма окажется больше приемного буфера приложения?

UDP — это ненадежный протокол, однако существуют приложения, в которых UDP использовать целесообразнее, чем TCP. Мы рассмотрим факторы, под влиянием которых UDP оказывается предпочтительнее TCP. В UDP-приложения необходимо включать ряд функций, в некоторой степени компенсирующих ненадежность UDP: тайм-аут и повторную передачу, обработку потерянных дейтаграмм и порядковые номера для сопоставления ответов запросам. Мы разработаем набор функций, которые сможем вызывать из наших приложений UDP.

Если реализация не поддерживает параметр сокета `IP_RECVSTADDR`, один из способов определить IP-адрес получателя UDP-дейтаграммы заключается в связывании всех интерфейсных адресов и использовании функции `select`.

Большинство серверов UDP являются последовательными, но существуют приложения, обменивающиеся множеством дейтаграмм UDP между клиентом и сервером, что требует параллельной обработки. Примером может служить TFTP (Trivial File Transfer Protocol — упрощенный протокол передачи файлов). Мы рассмотрим два варианта подобного согласования — с использованием суперсервера `inetd` и без него.

В завершение этой главы мы рассмотрим информацию о пакете, которая может быть передана во вспомогательных данных дейтаграммы IPv6: IP-адрес отправителя, отправляющий интерфейс, предельное количество транзитных узлов исходящих дейтаграмм и адрес следующего транзитного узла. Аналогичная информация — IP-адрес получателя, принимающий интерфейс и предельное количество транзитных узлов — может быть получена вместе с дейтаграммой IPv6.

22.2. Получение флагов, IP-адреса получателя и индекса интерфейса

Исторически функции `sendmsg` и `recvmsg` использовались только для передачи дескрипторов через доменные сокеты Unix (см. раздел 15.7), но даже это происходило сравнительно редко. Однако в настоящее время популярность этих двух функций растет по двум причинам:

1. Элемент `msg_flags`, добавленный в структуру `msghdr` в реализации 4.3BSD Reno, возвращает приложению флаги сообщения. Эти флаги мы перечислили в табл. 14.2.

2. Вспомогательные данные используются для передачи все большего количества информации между приложением и ядром. В главе 27 мы увидим, что IPv6 продолжает эту тенденцию.

В качестве примера использования функции `recvmsg` мы напишем функцию `recvfrom_flags`, аналогичную функции `recvfrom`, но дополнительно позволяющую получить:

- возвращаемое значение `msg_flags`;
- адрес получателя полученной дейтаграммы (из параметра сокета `IP_RECVSTADDR`);
- индекс интерфейса, на котором была получена дейтаграмма (параметр сокета `IP_RECVIF`).

Чтобы можно было получить два последних элемента, мы определяем в нашем заголовке `inpr.h` следующую структуру:

```
struct in_pktinfo {  
    struct in_addr ipi_addr; /* IPv4-адрес получателя */
```

```

    int          ipi_ifindex; /* индекс интерфейса, на котором была
                               получена дейтаграмма */
};


```

Мы выбрали имена структуры и ее элементов так, чтобы получить определенное сходство со структурой IPv6 `in6_pktnfo`, возвращающей те же два элемента для сокета IPv6 (см. раздел 22.8). Наша функция `recvfrom_flags` будет получать в качестве аргумента указатель на структуру `in_pktnfo`, и если этот указатель не нулевой, возвращать структуру через указатель.

Проблема построения этой структуры состоит в том, что неясно, что возвращать, если недоступна информация, которая должна быть получена из параметра сокета `IP_RECVSTADDR` (то есть реализация не поддерживает данный параметр сокета). Обработать индекс интерфейса легко, поскольку нулевое значение может использоваться как указание на то, что индекс неизвестен. Но для IP-адреса все 32-разрядные значения являются действительными. Мы выбрали такое решение: адрес получателя 0.0.0.0 возвращается в том случае, когда действительное значение недоступно. Хотя это реальный IP-адрес, использовать его в качестве IP-адреса получателя не разрешается (RFC 1122 [10]). Он будет действителен только в качестве IP-адреса отправителя во время начальной загрузки узла, когда узел еще не знает своего IP-адреса.

ПРИМЕЧАНИЕ

К сожалению, Беркли-ядра принимают дейтаграммы, предназначенные для адреса 0.0.0.0 [128, с. 218-219]. Это устаревшие адреса широковещательной передачи, генерируемые ядрами 4.2BSD.

Первая часть нашей функции `recvfrom_flags` представлена в листинге 22.1^[1]. Эта функция предназначена для использования с сокетом UDP.

Листинг 22.1. Функция `recvfrom_flags`: вызов функции `recvmsg`

```

//advio/recvfromflags.c
1 #include "unp.h"
2 #include <sys/param.h> /* макрос ALIGN для макрояда CMSG_NXTHDR() */

3 ssize_t
4 recvfrom_flags(int fd, void *ptr, size_t nbytes, int *flagsp,
5     SA *sa, socklen_t *salenptr, struct unp_in_pktnfo *pktp)
6 {
7     struct msghdr msg;
8     struct iovec iov[1];
9     ssize_t n;

10 #ifdef HAVE_MSGHDR_MSG_CONTROL
11     struct cmsghdr *cmphdr;
12     union {
13         struct cmsghdr cm;
14         char control[CMSG_SPACE(sizeof(struct in_addr)) +
15             CMSG_SPACE(sizeof(struct unp_in_pktnfo))];
16     } control_un;

17     msg.msg_control = control_un.control;
18     msg.msg_controllen = sizeof(control_un.control);
19     msg.msg_flags = 0;
20 #else
21     bzero(&msg, sizeof(msg)); /* обнуление msg_accrightslen = 0 */
22 #endif

23     msg.msg_name = sa;
24     msg.msg_namelen = *salenptr;
25     iov[0].iov_base = ptr;

```

```

26 iov[0].iov_len = nbytes;
27 msg.msg_iov = iov;
28 msg.msg iovlen = 1;

29 if ((n = recvmsg(fd, &msg, *flagsp)) < 0)
30 return(n);
31 *salenptr = msg.msg_namelen; /* возвращение результатов */
32 if (pktp)
33 bzero(pktp, sizeof(struct unp_in_pktinfo)); /* 0.0.0.0. интерфейс = 0 */

```

Подключаемые файлы

1-2 Использование макроопределения `CMSG_NXTHDR` требует подключения заголовочного файла `<sys/param.h>`.

Аргументы функции

3-5 Аргументы функции аналогичны аргументам функции `recvfrom` за исключением того, что четвертый аргумент является указателем на целочисленный флаг (так что мы можем возвратить флаги, возвращаемые функцией `recvmsg`), а седьмой аргумент новый: это указатель на структуру `unp_in_pktinfo`, содержащую IPv4-адрес получателя пришедшей дейтаграммы и индекс интерфейса, на котором дейтаграмма была получена.

Различия реализаций

10-22 При работе со структурой `msghdr` и различными константами `MSG_XXX` мы встречаемся со множеством различных реализаций. Одним из вариантов обработки таких различий может быть использование имеющейся в языке С возможности условного подключения (директива `#ifdef`). Если реализация поддерживает элемент `msg_control`, то выделяется пространство для хранения значений, возвращаемых параметрами сокета `IP_RECVSTADDR` и `IP_RECVIF`, и соответствующие элементы инициализируются.

Заполнение структуры `msghdr` и вызов функции `recvmsg`

23-33 Заполняется структура `msghdr` и вызывается функция `recvmsg`. Значения элементов `msg_namelen` и `msg_flags` должны быть переданы обратно вызывающему процессу. Они являются аргументами типа «значение-результат». Мы также инициализируем структуру вызывающего процесса `unp_in_pktinfo`, устанавливая IP-адрес 0.0.0.0 и индекс интерфейса 0.

В листинге 22.2 показана вторая часть нашей функции.

Листинг 22.2. Функция `recvfrom_flags`: возвращаемые флаги и адрес получателя

```

//advio/recvfrom_flags.c
34 #ifndef HAVE_MSGHDR_MSG_CONTROL
35 *flagsp = 0; /* возвращение результатов */
36 return(n);
37 #else

38 *flagsp = msg.msg_flags; /* возвращение результатов */
39 if (msg.msg_controllen < sizeof(struct cmsghdr) ||
40 (msg.msg_flags & MSG_CTRUNC) || pktp == NULL)
41 return(n);

42 for (cmptr = CMSG_FIRSTHDR(&msg); cmptr != NULL;

```

```

43     cmptr = CMSG_NXTHDR(&msg, cmptr)) {

44 #ifdef IP_RECVSTADDR
45     if (cmptr->cmsg_level == IPPROTO_IP &&
46         cmptr->cmsg_type == IP_RECVSTADDR) {

47         memcpy(&pktp->ipi_addr, CMSG_DATA(cmptr),
48             sizeof(struct in_addr));
49         continue;
50     }
51 #endif

52 #ifdef IP_RECVIF
53     if (cmptr->cmsg_level == IPPROTO_IP && cmptr->cmsg_type == IP_RECVIF) {
54         struct sockaddr_dl *sdl;

55         sdl = (struct sockaddr_dl*)CMSG_DATA(cmptr);
56         pktp->ipi_ifindex = sdl->sdl_index;
57         continue;
58     }
59 #endif
60     err_quit("unknown ancillary data, len = %d, level = %d, type = %d",
61     cmptr->cmsg_len, cmptr->cmsg_level, cmptr->cmsg_type);
62 }
63 return(n);
64 #endif /* HAVE_MSGHDR_MSG_CONTROL */
65 }

```

34-37 Если реализация не поддерживает элемента `msg_control`, мы просто обнуляем возвращаемые флаги и завершаем функцию. Оставшаяся часть функции обрабатывает информацию, содержащуюся в структуре `msg_control`.

Возвращение при отсутствии управляющей информации

38-41 Мы возвращаем значение `msg_flags` и передаем управление вызывающей функции в том случае, если нет никакой управляющей информации, управляющая информация была обрезана или вызывающий процесс не требует возвращения структуры `unp_in_pktinfo`.

Обработка вспомогательных данных

42-43 Мы обрабатываем произвольное количество объектов вспомогательных данных с помощью макросов `CMSG_FIRSTHDR` и `CMSG_NEXTHDR`.

Обработка параметра сокета IP_RECVSTADDR

47-54 Если в составе управляющей информации был возвращен IP-адрес получателя (см. рис. 14.2), он возвращается вызывающему процессу.

Обработка параметра сокета IP_RECVIF

55-63 Если в составе управляющей информации был возвращен индекс интерфейса, он возвращается вызывающему процессу. На рис. 22.1 показано содержимое возвращенного объекта вспомогательных данных.

cmsghdr()			
cmmsg_len	20	cmmsg_level	IPPROTO_IP
cmmsg_type	IP_RECVIF	len	fam index
		len	BAF_LLINK
		len	IPT_NONE, 0, 0, 0

Рис. 22.1. Объект вспомогательных данных, возвращаемый для параметра IP_RECVIF

Вспомните структуру адреса сокета канального уровня (см. листинг 18.1). Данные, возвращаемые в объекте вспомогательных данных, представлены в одной из этих структур, но длины трех элементов являются нулевыми (длина имени, адреса и селектора). Следовательно, нет никакой необходимости указывать эти значения, и таким образом структура имеет размер 8 байт, а не 20, как было в листинге 18.1. Возвращаемая нами информация — это индекс интерфейса.

Пример: вывод IP-адреса получателя и флага обрезки дейтаграммы

Для проверки нашей функции мы изменим функцию dg_echo (см. листинг 8.2) так, чтобы она вызывала функцию recvfrom_flags вместо функции recvfrom. Новая версия функции dg_echo показана в листинге 22.3.

Листинг 22.3. Функция dg_echo,зывающая нашу функцию recvfrom_flags

```
//advio/dgechoaddr.c
1 #include "unpifi.h"

2 #undef MAXLINE
3 #define MAXLINE 20 /* устанавливаем новое значение, чтобы
   пронаблюдать обрезку дейтаграмм */

4 void
5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
6 {
7     int flags;
8     const int on = 1;
9     socklen_t len;
10    ssize_t n;
11    char mesg[MAXLINE], str[INET6_ADDRSTRLEN], ifname[IFNAMSIZ];
12    struct in_addr in_zero;
13    struct in_pktinfo pktinfo;

14 #ifdef IP_RECVSTADDR
15     if (setsockopt(sockfd, IPPROTO_IP, IP_RECVSTADDR, &on, sizeof(on)) < 0)
16         err_ret("setsockopt of IP_RECVSTADDR");
17 #endif
18 #ifdef IP_RECVIF
19     if (setsockopt(sockfd, IPPROTO_IP, IP_RECVIF, &on, sizeof(on)) < 0)
20         err_ret("setsockopt of IP_RECVIF");
21 #endif
22     bzero(&in_zero, sizeof(struct in_addr)); /* IPv4-адрес, состоящий
   из одних нулей */

23    for (;;) {
24        len = clilen;
25        flags = 0;
26        n = Recvfrom_flags(sockfd, mesg, MAXLINE, &flags,
27                           pcliaddr, &len, &pktinfo);
28        printf("%d-byte datagram from %s", n, Sock_ntop(pcliaddr, len));
29        if (memcmp(&pktinfo.ipi_addr, &in_zero, sizeof(in_zero)) != 0)
30            printf(", to %s", Inet_ntop(AF_INET, &pktinfo.ipi_addr,
31                                         str, sizeof(str)));
32        if (pktinfo.ipi_ifindex > 0)
```

```

33     printf(", recv i/f = %s",
34     If_indextoname(pktinfo.ipi_ifindex, ifname));
35 #ifdef MSG_TRUNC
36     if (flags & MSG_TRUNC)
37         printf(" (datagram truncated)");
38 #endif
39 #ifdef MSG_CTRUNC
40     if (flags & MSG_CTRUNC)
41         printf(" (control info truncated)");
42 #endif
43 #ifdef MSG_BCAST
44     if (flags & MSG_BCAST)
45         printf(" (broadcast)");
46 #endif
47 #ifdef MSG_MCAST
48     if (flags & MSG_MCAST)
49         printf(" (multicast)");
50 #endif
51     printf("\n");

52     Sendto(sockfd, mesg, n, 0, pcliaddr, len);
53 }
54 }
```

Изменение MAXLINE

2-3 Мы удаляем существующее определение MAXLINE, имеющееся в нашем заголовочном файле `upr.h`, и задаем новое значение — 20. Это позволит нам увидеть, что произойдет, когда мы получим дейтаграмму UDP, превосходящую размер буфера, переданного функции (в данном случае функции `recvmsg`).

Установка параметров сокета IP_RECVSTADDR и IP_RECVIF

14-21 Если параметр сокета `IP_RECVSTADDR` определен, мы включаем его. Аналогично включается параметр сокета `IP_RECVIF`.

Чтение дейтаграммы, вывод IP-адреса отправителя и порта

24-28 Дейтаграмма читается с помощью вызова функции `recvfrom_flags`. IP-адрес отправителя и порт ответа сервера преобразуются в формат представления функцией `sock_ntop`.

Вывод IP-адреса получателя

29-31 Если возвращаемый IP-адрес ненулевой, он преобразуется в формат представления функцией `inet_ntop` и выводится.

Вывод имени интерфейса, на котором была получена дейтаграмма

32-34 Если индекс интерфейса ненулевой, его имя будет возвращено функцией `if_indextoname`. Это имя наша функция печатает на экране.

Проверка различных флагов

35-51 Мы проверяем четыре дополнительных флага и выводим сообщение, если какие-либо из них установлены.

22.3. Обрезанные дейтаграммы

В системах, происходящих от BSD, при получении UDP-дейтаграммы, размер которой больше буфера приложения, функция `recvmsg` устанавливает флаг `MSG_TRUNC` в элементе `msg_flags` структуры `msghdr` (см. табл. 14.2). Все Беркли-реализации, поддерживающие структуру `msghdr` с элементом `msg_flags`, обеспечивают это уведомление.

ПРИМЕЧАНИЕ

Это пример флага, который должен быть возвращен процессу ядром. В разделе 14.3 мы упомянули о проблеме разработки функций `recv` и `recvfrom`: их аргумент `flags` является целым числом, что позволяет передавать флаги от процесса к ядру, но не наоборот.

К сожалению, не все реализации подобным образом обрабатывают ситуацию, когда размер дейтаграммы UDP оказывается больше, чем предполагалось. Возможны три сценария:

1. Лишние байты игнорируются, и приложение получает флаг `MSG_TRUNC`, что требует вызова функции `recvmsg`.
2. Игнорирование лишних байтов без уведомления приложения.
3. Сохранение лишних байтов и возвращение их в последующих операциях чтения на сокете.

ПРИМЕЧАНИЕ

POSIX задает первый тип поведения: игнорирование лишних байтов и установку флага `MSG_TRUNC`. Ранние реализации SVR4 действуют по третьему сценарию.

Поскольку способ обработки дейтаграмм, превышающих размер приемного буфера приложения, зависит от реализации, одним из решений, позволяющий обнаружить ошибку, будет всегда использовать буфер приложения на 1 байт больше самой большой дейтаграммы, которую приложение предположительно может получить. Если все же будет получена дейтаграмма, длина которой равна размеру буфера, это явно будет свидетельствовать об ошибке.

22.4. Когда UDP оказывается предпочтительнее TCP

В разделах 2.3 и 2.4 мы описали основные различия между UDP и TCP. Поскольку мы знаем, что TCP надежен, а UDP — нет, возникает вопрос: когда следует использовать UDP вместо TCP и почему? Сначала перечислим преимущества UDP:

- Как видно из табл. 20.1, UDP поддерживает широковещательную и направленную передачу. Действительно, использование UDP обязательно, если приложению требуется широковещательная или многоадресная передача. Эти два режима адресации мы рассматривали в главах 20 и 21.
- UDP не требует установки и разрыва соединения. В соответствии с рис. 2.5 UDP позволяет осуществить обмен запросом и ответом в двух пакетах (если предположить, что размеры запроса и ответа меньше минимального размера MTU между двумя оконечными системами). В случае TCP требуется около 10 пакетов, если считать, что для каждого обмена «запрос-ответ» устанавливается новое соединение TCP.

Для анализа количества передаваемых пакетов важным фактором является также число циклов обращения пакетов, необходимых для получения ответа. Это становится важно, если время ожидания превышает пропускную способность, как показано в приложении A [112]. В этом тексте сказано, что минимальное время транзакции для запроса-ответа UDP равно $RTT + SPT$, где RTT — это время обращения между клиентом и сервером, а SPT — время обработки запроса сервером. Однако в случае TCP,

если для осуществления каждой последовательности «запрос-ответ» используется новое соединение TCP, минимальное время транзакции будет равно $2 \times RTT + SPT$, то есть на один период RTT больше, чем для UDP.

В отношении второго пункта очевидно, что если соединение TCP используется для множества обменов «запрос-ответ», то стоимость установления и разрыва соединения амортизируется во всех запросах и ответах. Обычно это решение предпочтительнее, чем использование нового соединения для каждого обмена «запрос-ответ». Тем не менее существуют приложения, использующие новое соединение для каждого цикла «запрос-ответ» (например, старые версии HTTP). Кроме того, существуют приложения, в которых клиент и сервер обмениваются в одном цикле «запрос-ответ» (например, DNS), а затем могут не обращаться друг к другу в течение часов или дней.

Теперь мы перечислим функции TCP, отсутствующие в UDP. Это означает, что приложение должно само реализовывать эти функции, если они ему необходимы. Мы говорим «необходимы», потому что не все свойства требуются всем приложениям. Например, может не возникнуть необходимости повторно передавать потерянные сегменты для аудиоприложений реального времени, если приемник способен интерполировать недостающие данные. Также для простых транзакций «запрос-ответ» может не потребоваться управление потоком, если два конца соединения заранее договорятся о размерах наибольшего запроса и ответа.

■ *Положительные подтверждения, повторная передача потерянных пакетов, обнаружение дубликатов и упорядочивание пакетов, порядок следования которых был изменен сетью.* TCP подтверждает получение всех данных, позволяя обнаруживать потерянные пакеты. Реализация этих двух свойств требует, чтобы каждый сегмент данных TCP содержал порядковый номер, по которому можно впоследствии проверить получение данного сегмента. Требуется также, чтобы TCP прогнозировал значение тайм-аута повторной передачи для соединения и чтобы это значение последовательно обновлялось по мере изменения сетевого трафика между конечными точками.

■ *Оконное управление потоком.* Принимающий TCP сообщает отправляющему, какое буферное пространство он выделил для приема данных, и отправляющий не может превышать этого ограничения. То есть количество неподтвержденных данных отправителя никогда не может стать больше объявленного размера окна принимающего.

■ *Медленный старт и предотвращение перегрузки.* Это форма управления потоком, осуществляемого отправителем, служащая для определения текущей пропускной способности сети и позволяющая контролировать ситуацию во время переполнения сети. Все современные TCP-приложения должны поддерживать эти два свойства, и опыт (накопленный еще до того, как эти алгоритмы были реализованы в конце 80-х) показывает, что протоколы, не снижающие скорость передачи при перегрузке сети, лишь усугубляют эту перегрузку (см., например, [52]).

Суммируя вышеизложенное, мы можем сформулировать следующие рекомендации:

■ UDP должен использоваться для приложений широковещательной и многоадресной передачи. Если требуется какая-либо форма защиты от ошибок, то соответствующая функциональность должна быть добавлена клиентам и серверам. Однако приложения часто используют широковещательную и многоадресную передачу, когда некоторое (предположительно небольшое) количество ошибок вполне допустимо (например, потеря аудио- или видеопакетов). Имеются приложения многоадресной передачи, требующие надежной доставки (например, пересылка файлов при помощи многоадресной передачи), но в каждом конкретном случае мы должны решить, компенсируется ли выигрышем в производительности, получаемым за счет использования многоадресной передачи (отправка одного пакета N получателям вместо отправки N копий пакета через N соединений TCP), дополнительное усложнение приложения для обеспечения надежности соединений.

■ UDP может использоваться для простых приложений «запрос-ответ», но тогда обнаружение ошибок должно быть встроено в приложение. Минимально это означает включение подтверждений, тайм-аутов и повторных передач. Управление потоком часто не является существенным для обеспечения надежности, если запросы и ответы имеют достаточно разумный размер. Мы приводим пример реализации этой функциональности в приложении UDP, представленном в разделе 22.5. Факторы, которые нужно учитывать, — это частота соединения клиента и сервера (нужно решить, можно ли не разрывать установленное соединение TCP между транзакциями) и количество данных, которыми обмениваются клиент и сервер (если в большинстве случаев при работе данного приложения требуется много пакетов, стоимость установления и разрыва соединения TCP становится менее значимым фактором).

■ UDP не следует использовать для передачи большого количества данных (например, при передаче файлов). Причина в том, что оконное управление потоком, предотвращение переполнения и медленный

старт должны быть встроены в приложение вместе с функциями, перечисленными в предыдущем пункте. Это означает, что мы фактически заново изобретаем TCP для одного конкретного приложения. Нам следует оставить производителям заботу об улучшении производительности TCP и сконцентрировать свои усилия на самом приложении.

Из этих правил есть исключения, в особенности для существующих приложений. Например, TFTP использует UDP для передачи большого количества данных. Для TFTP был выбран UDP, поскольку, во-первых, его реализация проще в отношении кода начальной загрузки (800 строк кода C для UDP в сравнении с 4500 строками для TCP, например в [128]), а во-вторых, TFTP используется только для начальной загрузки систем в локальной сети, а не для передачи большого количества данных через глобальные сети. Однако при этом требуется, чтобы в TFTP были предусмотрены такие свойства, как собственное поле порядкового номера (для подтверждений), тайм-аут и возможность повторной передачи.

NFS (Network File System — сетевая файловая система) является другим исключением из правила: она также использует UDP для передачи большого количества данных (хотя некоторые могут возразить, что в действительности это приложение типа «запрос-ответ», использующее запросы и ответы больших размеров). Отчасти это можно объяснить исторически сложившимися обстоятельствами: в середине 80-х, когда была разработана эта система, реализации UDP были быстрее, чем TCP, и система NFS использовалась только в локальных сетях, где потеря пакетов, как правило, происходит на несколько порядков реже, чем в глобальных сетях. Но как только в начале 90-х NFS начала использоваться в глобальных сетях, а реализации TCP стали обгонять UDP в отношении производительности при передаче большого количества данных, была разработана версия 3 системы NFS для поддержки TCP. Теперь большинство производителей предоставляют NFS как для TCP, так и для UDP. Аналогичные причины (большая скорость по сравнению с TCP в начале 80-х плюс преобладание локальных сетей над глобальными) привели к тому, что в Apollo NCS (предшественник DCE RPC) сначала использовали UDP, а не TCP, хотя современные реализации поддерживают UDP и TCP.

Мы могли бы сказать, что применение UDP сокращается, поскольку сегодня хорошие реализации TCP не уступают в скорости сетям и все меньше разработчиков готовы встраивать в приложения UDP функциональность, свойственную TCP. Но предсказываемое увеличение количества мультимедиа-приложений в будущем десятилетии должно привести к возрастанию популярности UDP, поскольку их работа обычно подразумевает использование многоадресной передачи, требующей наличия UDP.

22.5. Добавление надежности приложению UDP

Если мы хотим использовать UDP для приложения типа «запрос-ответ», как было отмечено в предыдущем разделе, мы должны добавить нашему клиенту две функции:

- тайм-аут и повторную передачу, которые позволяют решать проблемы, возникающие в случае потери дейтаграмм;
- порядковые номера, позволяющие клиенту проверить, что ответ приходит на определенный запрос.

Эти два свойства предусмотрены в большинстве существующих приложений UDP, использующих простую модель «запрос-ответ»: например, распознаватели DNS, агенты SNMP, TFTP и RPC. Мы не пытаемся использовать UDP для передачи большого количества данных: наша цель — приложение, посылающее запрос и ожидающее ответа на этот запрос.

ПРИМЕЧАНИЕ

Использование дейтаграмм по определению не может быть надежным, следовательно, мы специально не называем данный сервис «надежным сервисом дейтаграмм». Действительно, термин «надежная дейтаграмма» — это оксюморон. Речь идет лишь о том, что приложение до некоторой степени обеспечивает надежность, добавляя соответствующие функциональные возможности «поверх» ненадежного сервиса дейтаграмм (UDP).

Добавление порядковых номеров осуществляется легко. Клиент подготавливает порядковый номер для каждого запроса, а сервер должен отразить этот номер обратно в своем ответе клиенту. Это позволяет клиенту проверить, что данный ответ пришел на соответствующий запрос.

Более старый метод реализации тайм-аутов и повторной передачи заключался в отправке запроса и ожидании в течение N секунд. Если ответ не приходил, осуществлялась повторная передача и снова на

ожидание ответа отводилось N секунд. Если это повторялось несколько раз, отправка запроса прекращалась. Это так называемый линейный таймер повторной передачи (на рис. 6.8 [111] показан пример клиента TFTP, использующего эту технологию. Многие клиенты TFTP до сих пор пользуются этим методом).

Проблема при использовании этой технологии состоит в том, что количество времени, в течение которого дейтаграмма совершает цикл в объединенной сети, может варьироваться от долей секунд в локальной сети до нескольких секунд в глобальной. Факторами, влияющими на время обращения (RTT), являются расстояние, скорость сети и переполнение. Кроме того, RTT между клиентом и сервером может быстро меняться со временем при изменении условий в сети. Нам придется использовать тайм-ауты и алгоритм повторной передачи, который учитывает действительное (измеряемое) значение периода RTT и изменения RTT с течением времени. В этой области ведется большая исследовательская работа, в основном направленная на TCP, но некоторые идеи применимы к любым сетевым приложениям.

Мы хотим вычислить тайм-аут повторной передачи (RTO), чтобы использовать его при отправке каждого пакета. Для того чтобы выполнить это вычисление, мы измеряем RTT — действительное время обращения для пакета. Каждый раз, измеряя RTT, мы обновляем два статистических показателя: $srtt$ — слаженную оценку RTT, и $rttvar$ — слаженную оценку среднего отклонения. Последняя является хорошей приближенной оценкой стандартного отклонения, но ее легче вычислять, поскольку для этого не требуется извлечения квадратного корня. Имея эти два показателя, мы вычисляем RTO как сумму $srtt$ и $rttvar$, умноженного на четыре. В [52] даются все необходимые подробности этих вычислений, которые мы можем свести к четырем следующим уравнениям:

```
delta = measuredRTT - srtt
srtt ← srtt + g × delta
rttvar ← rttvar + h (|delta| - rttvar)
RTO = srtt + 4 × rttvar
```

δ — это разность между измеренным RTT и текущим слаженным показателем RTT ($srtt$). g — это приращение, применяемое к показателю RTT, равное $1/8$. h — это приращение, применяемое к слаженному показателю среднего отклонения, равное $1/4$.

ПРИМЕЧАНИЕ

Два приращения и множитель 4 в вычислении RTO специально выражены степенями числа 2 и могут быть вычислены с использованием операций сдвига вместо деления и умножения. На самом деле реализация TCP в ядре (см. раздел 25.7 [128]) для ускорения вычислений обычно использует арифметику с фиксированной точкой, но мы для простоты используем в нашем коде вычисления с плавающей точкой.

Другой важный момент, отмеченный в [52], заключается в том, что по истечении времени таймера повторной передачи для следующего RTO должно использоваться экспоненциальное смещение (*exponential backoff*). Например, если первое значение RTO равно 2 с и за это время ответа не получено, следующее значение RTO будет равно 4 с. Если ответ все еще не последовал, следующее значение RTO будет 8 с, затем 16 и т.д.

Алгоритмы Джекобсона (Jacobson) реализуют вычисление RTO при измерении RTT и увеличение RTO при повторной передаче. Однако, когда клиент выполняет повторную передачу и получает ответ, возникает проблема неопределенности повторной передачи (*retransmission ambiguity problem*). На рис. 22.2 показаны три возможных сценария, при которых истекает время ожидания повторной передачи:

- запрос потерян;
- ответ потерян;
- значение RTO слишком мало.

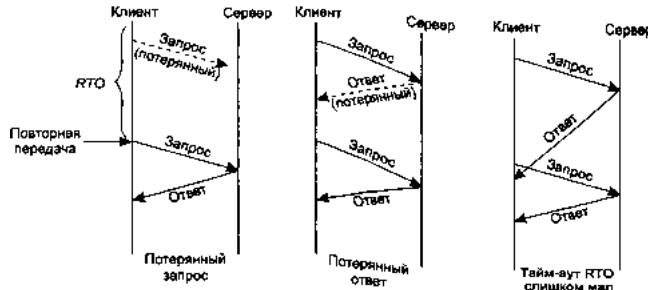


Рис. 22.2. Три сценария, возможные при истечении времени таймера повторной передачи

Когда клиент получает ответ на запрос, отправленный повторно, он не может сказать, какому из запросов соответствует ответ. На рисунке, изображенном справа, ответ соответствует начальному запросу, в то время как на двух других рисунках ответ соответствует второму запросу.

Алгоритм Карна (Karn) [58] обрабатывает этот сценарий в соответствии со следующими правилами, применяемыми в любом случае, когда ответ получен на запрос, отправленный более одного раза:

- Если для запроса и ответа было измерено значение RTT, не следует использовать его для обновления оценочных значений, так как мы не знаем, какому запросу соответствует ответ.
- Поскольку ответ пришел до того, как истекло время нашего таймера повторной передачи, используйте для следующего пакета текущее значение RTO. Только когда мы получим ответ на запрос, который не был передан повторно, мы изменяем значение RTT и снова вычисляем RTO.

При написании наших функций RTT применить алгоритм Карна несложно, но оказывается, что существует и более изящное решение. Оно используется в расширениях TCP для сетей с высокой пропускной способностью, то есть сетей, обладающих либо широкой полосой пропускания, либо большим значением RTT, либо обоими этими свойствами (RFC 1323 [53]). Кроме добавления порядкового номера к началу каждого запроса, который сервер должен отразить, мы добавляем *отметку времени*, которую сервер также должен отразить. Каждый раз, отправляя запрос, мы сохраняем в этой отметке значение текущего времени. Когда приходит ответ, мы вычисляем величину RTT для этого пакета как текущее время минус значение отметки времени, отраженной сервером в своем ответе. Поскольку каждый запрос несет отметку времени, отражаемую сервером, мы можем вычислить RTT для *каждого* ответа, который мы получаем. Теперь нет никакой неопределенности. Более того, поскольку сервер только отражает отметку времени клиента, клиент может использовать для отметок времени любые удобные единицы, и при этом не требуется, чтобы клиент и сервер синхронизировали часы.

Пример

Свяжем теперь всю эту информацию воедино в примере. Мы начнем с функции `main` нашего клиента UDP, представленного в листинге 8.3, и изменим в ней только номер порта с `SERV_PORT` на 7 (стандартный эхо-сервер, см. табл. 2.1).

В листинге 22.4 показана функция `dg_cli`. Единственное изменение по сравнению с листингом 8.4 состоит в замене вызовов функций `sendto` и `recvfrom` вызовом нашей новой функции `dg_send_recv`.

Перед тем как представить функцию `dg_send_recv` и наши функции RTT, которые она вызывает, мы показываем в листинге 22.5 нашу схему реализации функциональных свойств, повышающих надежность клиента UDP. Все функции, имена которых начинаются с `rtt_`, описаны далее.

Листинг 22.4. Функция `dg_cli`, вызывающая нашу функцию `dg_send_recv`

```
//rtt/dg_cli.c
1 #include "unp.h"

2 ssize_t Dg_send_recv(int, const void*, size_t, void*, size_t,
3 const SA*, socklen_t);

4 void
5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
6 {
7    ssize_t n;
8    char sendline[MAXLINE], recvline[MAXLINE + 1];
```

```
9 while (Fgets(sendline, MAXLINE, fp) != NULL) {  
10    n = Dg_send_recv(sockfd, sendline, strlen(sendline),  
11                      recvline, MAXLINE, pservaddr, servlen);
```

```
12    recvline[n] = 0; /* завершающий нуль */  
13    Fputs(recvline, stdout);  
14 }  
15 }
```

Листинг 22.5. Схема функций RTT и последовательность их вызова

```
static sigjmp_buf jmpbuf;
```

```
{
```

формирование запроса

```
signal(SIGALRM, sig_alrm); /* устанавливаем обработчик сигнала */  
rtt_newpack(); /* инициализируем значение счетчика rexmt нулем */  
sendagain:  
sendto();  
  
alarm(rtt_start()); /* задаем аргумент функции alarm равным RTO */  
if (sigsetjmp(jmpbuf, 1) != 0) {  
    if (rtt_timeout()) /* удваиваем RTO, обновляем оценочные значения */  
        отказываемся от дальнейших попыток  
    goto sendagain; /* повторная передача */  
}  
do {  
    recvfrom();  
} while (неправильный порядковый номер);  
alarm(0); /* отключаем сигнал alarm */  
rtt_stop(); /* вычисляем RTT и обновляем оценочные значения */
```

обрабатываем ответ

```
}
```

```
void sig_alrm(int signo) {  
    siglongjmp(jmpbuf, 1);  
}
```

Если приходит ответ, но его порядковый номер отличается от предполагаемого, мы снова вызываем функцию `recvfrom`, но не отправляем снова тот же запрос и не перезапускаем работающий таймер повторной передачи. Обратите внимание, что в крайнем правом случае на рис. 22.2 последний ответ, полученный на отправленный повторно запрос, будет находиться в приемном буфере сокета до тех пор, пока клиент не решит отправить следующий запрос (и получить на него ответ). Это нормально, поскольку клиент прочитает этот ответ, отметит, что порядковый номер отличается от предполагаемого, проигнорирует ответ и снова вызовет функцию `recvfrom`.

Мы вызываем функции `sigsetjmp` и `siglongjmp`, чтобы предотвратить возникновение ситуации гонок с сигналом `SIGALRM`, который мы описали в разделе 20.5. В листинге 22.6 показана первая часть нашей функции `dg_send_recv`.

Листинг 22.6. Функция `dg_send_recv`: первая половина

```
//rtt/dg_send_recv.c  
1 #include "unprtt.h"  
2 #include <setjmp.h>
```

```

3 #define RTT_DEBUG

4 static struct rtt_info rttinfo;
5 static int rttinit = 0;
6 static struct msghdr msgsend, msgrecv;
/* предполагается, что обе структуры инициализированы нулем */
7 static struct hdr {
8     uint32_t seq; /* порядковый номер */
9     uint32_t ts; /* отметка времени при отправке */
10 } sendhdr, recvhdr;

11 static void signalrm(int signo);
12 static sigjmp_buf jmpbuf;

13 ssize_t
14 dg_send_recv(int fd, const void *outbuff, size_t outbytes,
15   void *inbuff, size_t inbytes,
16   const SA *destaddr, socklen_t destlen)
17 {
18     ssize_t n;
19     struct iovec iovsend[2], iovrecv[2];
20     if (rttinit == 0) {
21         rtt_init(&rttinfo); /* первый вызов */
22         rttinit = 1;
23         rtt_d_flag = 1;
24     }
25     sendhdr.seq++;
26     msgsend.msg_name = destaddr;
27     msgsend.msg_namelen = destlen;
28     msgsend.msg iov = iovsend;
29     msgsend.msg iovlen = 2;
30     iovsend[0].iov_base = &sendhdr;
31     iovsend[0].iov_len = sizeof(struct hdr);
32     iovsend[1].iov_base = outbuff;
33     iovsend[1].iov_len = outbytes;
34     msgrecv.msg_name = NULL;
35     msgrecv.msg_namelen = 0;
36     msgrecv.msg iov = iovrecv;
37     msgrecv.msg iovlen = 2;
38     iovrecv[0].iov_base = &recvhdr;
39     iovrecv[0].iov_len = sizeof(struct hdr);
40     iovrecv[1].iov_base = inbuff;
41     iovrecv[1].iov_len = inbytes;

```

1-5 Мы включаем новый заголовочный файл unprtt.h, показанный в листинге 22.8, который определяет структуру `rtt_info`, содержащую информацию RTT для клиента. Мы определяем одну из этих структур и ряд других переменных.

Определение структур `msghdr` и структуры `hdr`

6-10 Мы хотим скрыть от вызывающего процесса добавление порядкового номера и отметки времени в начало каждого пакета. Проще всего использовать для этого функцию `writev`, записав свой заголовок (структур `hdr`), за которым следуют данные вызывающего процесса, в виде однойдейтаграммы UDP. Вспомните, что результатом выполнения функции `writev` на дейтаграммном сокете является отправка однойдейтаграммы. Это проще, чем заставлять вызывающий процесс выделять для нас место в начале буфера, а также быстрее, чем копировать наш заголовок и данные вызывающего процесса в один буфер

(под который мы должны выделить память) для каждой функции sendto. Но поскольку мы работаем с UDP и нам необходимо задать адрес получателя, следует использовать возможности, предоставляемые структурой iovec функций sendmsg и recvmsg и отсутствующие в функциях sendto и recvfrom. Вспомните из раздела 14.5, что в некоторых системах доступна более новая структура msghdr, включающая вспомогательные данные (`msg_control`), тогда как в более старых системах вместо них применяются элементы `msg_accright` (так называемые права доступа — access rights), расположенные в конце структуры. Чтобы избежать усложнения кода директивами `#ifdef` для обработки этих различий, мы объявляем две структуры `msghdr` как `static`. При этом они инициализируются только нулевыми битами, а затем неиспользованные элементы в конце структур просто игнорируются.

Инициализация при первом вызове

20-24 При первом вызове нашей функции мы вызываем функцию `rtt_init`.

Заполнение структур `msghdr`

25-41 Мы заполняем две структуры `msghdr`, используемые для ввода и вывода. Для данного пакета мы увеличиваем на единицу порядковый номер отправки, но не устанавливаем отметку времени отправки, пока пакет не будет отправлен (поскольку он может отправляться повторно, а для каждой повторной передачи требуется текущая отметка времени).

Вторая часть функции вместе с обработчиком сигнала `sig_alarm` показана в листинге 22.7.

Листинг 22.7. Функция `dg_send_recv`: вторая половина

```
//rtt/dg_send_recv.c
42 Signal(SIGALRM, sig_alarm);
43 rtt_newpack(&rttinfo); /* инициализируем для этого пакета */

44 sendagain:
45 sendhdr.ts = rtt_ts(&rttinfo);
46 Sendmsg(fd, &msgsend, 0);

47 alarm(rtt_start(&rttinfo)); /* вычисляем тайм-аут. запускаем таймер */
48 if (sigsetjmp(jmpbuf, 1) != 0) {
49     if (rtt_timeout(&rttinfo) < 0) {
50         err_msg("dg_send_recv: no response from server, giving up");
51         rttinit = 0; /* повторная инициализация для следующего вызова */
52         errno = ETIMEDOUT;
53         return (-1);
54     }
55     goto sendagain;
56 }
57 do {
58     n = Recvmsg(fd, &msgrecv, 0);
59 } while (n < sizeof(struct hdr) || recvhdr.seq != sendhdr.seq);

60 alarm(0); /* останавливаем таймер SIGALRM */
61 /* вычисляем и записываем новое значение оценки RTT */
62 rtt_stop(&rttinfo, rtt_ts(&rttinfo) - recvhdr.ts);

63 return (n - sizeof(struct hdr)); /* возвращаем размер полученной
64                               дейтаграммы */

65 static void
66 sig_alarm(int signo)
```

```
67 {  
68     siglongjmp(jmpbuf, 1);  
69 }
```

Установка обработчика сигналов

42-43 Для сигнала SIGALRM устанавливается обработчик сигналов, а функция rtt_newpack устанавливает счетчик повторных передач в нуль.

Отправка дейтаграммы

45-47 Функция rtt_ts получает текущую отметку времени. Отметка времени хранится в структуре hdr, которая добавляется к данным пользователя. Одиночная дейтаграмма UDP отправляется функцией sendmsg. Функция rtt_start возвращает количество секунд для этого тайм-аута, а сигнал SIGALRM контролируется функцией alarm.

Установка буфера перехода

48 Мы устанавливаем буфер перехода для нашего обработчика сигналов с помощью функции sigsetjmp. Мы ждем прихода следующей дейтаграммы, вызывая функцию recvmsg. (Совместное использование функций sigsetjmp и siglongjmp вместе с сигналом SIGALRM мы обсуждали применительно к листингу 20.5.) Если время таймера истекает, функция sigsetjmp возвращает 1.

Обработка тайм-аута

49-55 Когда возникает тайм-аут, функция rtt_timeout вычисляет следующее значение RTO (используя экспоненциальное смещение) и возвращает -1, если нужно прекратить попытки передачи дейтаграммы, или 0, если нужно выполнить очередную повторную передачу. Когда мы прекращаем попытки, мы присваиваем переменной errno значение ETIMEDOUT и возвращаемся в вызывающую функцию.

Вызов функции recvmsg, сравнение порядковых номеров

57-59 Мы ждем прихода дейтаграммы, вызывая функцию recvmsg. Длина полученной дейтаграммы не должна быть меньше размера структуры hdr, а ее порядковый номер должен совпадать с порядковым номером запроса, ответом на который предположительно является эта дейтаграмма. Если при сравнении хотя бы одно из этих условий не выполняется, функция recvmsg вызывается снова.

Выключение таймера и обновление показателей RTT

60-62 Когда приходит ожидаемый ответ, функция alarm отключается, а функция rtt_stop обновляет оценочное значение RTT. Функция rtt_ts возвращает текущую отметку времени, и отметка времени из полученной дейтаграммы вычитается из текущей отметки, что дает в результате RTT.

Обработчик сигнала SIGALRM

65-69 Вызывается функция siglongjmp, результатом выполнения которой является то, что функция sigsetjmp в dg_send_recv возвращает 1.

Теперь мы рассмотрим различные функции RTT, которые вызывались нашей функцией dg_send_recv. В листинге 22.8 показан заголовочный файл unprtt.h.

Листинг 22.8. Заголовочный файл unprtt.h

```
//lib/unprtt.h
1 #ifndef __unp_rtt_h
2 #define __unp_rtt_h

3 #include "unp.h"

4 struct rtt_info {
5     float    rtt_rtt;      /* последнее измеренное значение RTT в секундах */
6     float    rtt_srtt;     /* сглаженная оценка RTT в секундах */
7     float    rtt_rttvar;   /* сглаженные средние значения отклонений
                           в секундах */
8     float    rtt_rto;      /* текущее используемое значение RTO, в секундах */
9     int      rtt_nrexmt;  /* количество повторных передач: 0, 1, 2, ... */
10    uint32_t rtt_base;    /* число секунд, прошедшее после 1.1.1970 в начале */
11};

12 #define RTT_RXTMIN    2 /* минимальное значение тайм-аута для
                           повторной передачи, в секундах */
13 #define RTT_RXTMAX    60 /* максимальное значение тайм-аута для
                           повторной передачи, в секундах */
14 #define RTT_MAXNREXMT 3 /* максимально допустимое количество
                           повторных передач однойдейтаграммы */

15 /* прототипы функций */
16 void    rtt_debug(struct rtt_info*);
17 void    rtt_init(struct rtt_info*);
18 void    rtt_newpack(struct rtt_info*);
19 int     rtt_start(struct rtt_info*);
20 void    rtt_stop(struct rtt_info*, uint32_t);
21 int     rtt_timeout(struct rtt_info*);
22 uint32_t rtt_ts(struct rtt_info*);
23 extern int rtt_d_flag; /* может быть ненулевым при наличии
                           дополнительной информации */
24#endif /* __unp_rtt_h */
```

Структура rtt_info

4-11 Эта структура содержит переменные, необходимые для того, чтобы определить время передачи пакетов между клиентом и сервером. Первые четыре переменных взяты из уравнений, приведенных в начале этого раздела.

12-14 Эти константы определяют минимальный и максимальный тайм-ауты повторной передачи и максимальное число возможных повторных передач.

В листинге 22.9 показан макрос RTT_RTOCALC и первые две из четырех функций RTT.

Листинг 22.9. Макрос RTT_RTOCALC, функции rtt_minmax и rtt_init

```
//lib/rtt.c
1 #include "unprtt.h"

2 int rtt_d_flag = 0; /* отладочный флаг; может быть установлен в
                      ненулевое значение вызывающим процессом */
3 /* Вычисление значения RTO на основе текущих значений:
4 * сглаженное оценочное значение RTT + четырежды сглаженная
5 * величина отклонения.
6 */
7 #define RTI_RTOCALC(ptr) ((ptr)->rtt_srtt + (4.0 * (ptr)->rtt_rttvar))
```

```

8 static float
9 rtt_minmax(float rto)
10 {
11   if (rto < RTT_RXTMIN)
12     rto = RTT_RXTMIN;
13   else if (rto > RTT_RXTMAX)
14     rto = RTT_RXTMAX;
15   return (rto);
16 }

17 void
18 rtt_init(struct rtt_info *ptr)
19 {
20   struct timeval tv;

21   gettimeofday(&tv, NULL);
22   ptr->rtt_base = tv.tv_sec; /* количество секунд, прошедших с 1.1.1970 */

23   ptr->rtt_rtt = 0;
24   ptr->rtt_srtt = 0;
25   ptr->rtt_rttvar = 0.75;
26   ptr->rtt_rto = rtt_minmax(RTT_RTOCALC(ptr));
27   /* первое RTO (srtt + (4 * rttvar)) = 3 с */
28 }

3-7 Макрос вычисляет RTO как сумму оценочной величины RTT и оценочной величины среднего отклонения, умноженной на четыре.

8-16 Функция rtt_minmax проверяет, что RTO находится между верхним и нижним пределами, заданными в заголовочном файле unprtt.h.

17-28 Функция rtt_init вызывается функцией dg_send_recv при первой отправке пакета. Функция gettimeofday возвращает текущее время и дату в той же структуре timeval, которую мы видели в функции select (см. раздел 6.3). Мы сохраняем только текущее количество секунд с момента начала эпохи Unix, то есть с 00:00:00 1 января 1970 года (UTC). Измеряемое значение RTT обнуляется, а сглаженная оценка RTT и среднее отклонение принимают соответственно значение 0 и 0,75, в результате чего начальное RTO равно 3 с ( $4 \times 0,75$ ).

В листинге 22.10 показаны следующие три функции RTT.

Листинг 22.10. Функции rtt_ts, rtt_newpack и rtt_start

//lib/rtt.c

34 uint32_t
35 rtt_ts(struct rtt_info *ptr)
36 {
37   uint32_t ts;
38   struct timeval tv;

39   gettimeofday(&tv, NULL);
40   ts = ((tv.tv_sec - ptr->rtt_base) * 1000) + (tv.tv_usec / 1000);
41   return (ts);
42 }

43 void
44 rtt_newpack(struct rtt_info *ptr)
45 {
46   ptr->rtt_nrexmt = 0;
47 }

48 int

```

```

49 rtt_start(struct rtt_info *ptr)
50 {
51     return ((int)(ptr->rtt_rto + 0.5)); /* округляем float до int */
52     /* возвращенное значение может быть использовано как аргумент
53         alarm(rtt_start(&foo)) */

```

34-42 Функция `rtt_ts` возвращает текущую отметку времени для вызывающего процесса, которая должна содержаться в отправляемой дейтаграмме в виде 32-разрядного целого числа без знака. Мы получаем текущее время и дату из функции `gettimeofday` и затем вычитаем число секунд в момент вызова функции `rtt_init` (значение, хранящееся в элементе `rtt_base` структуры `rtt_info`). Мы преобразуем это значение в миллисекунды, а также преобразуем в миллисекунды значение, возвращаемое функцией `gettimeofday` в микросекундах. Тогда отметка времени является суммой этих двух значений в миллисекундах.

Разница во времени между двумя вызовами функции `rtt_ts` представляется количеством миллисекунд между этими двумя вызовами. Но мы храним отметки времени в 32-разрядном целом числе без знака, а не в структуре `timeval`.

43-47 Функция `rtt_newpack` просто обнуляет счетчик повторных передач. Эта функция должна вызываться всегда, когда новый пакет отправляется в первый раз.

48-53 Функция `rtt_start` возвращает текущее значение RTO в миллисекундах. Возвращаемое значение затем может использоваться в качестве аргумента функции `alarm`.

Функция `rtt_stop`, показанная в листинге 22.11, вызывается после получения ответа для обновления оценочного значения RTT и вычисления нового значения RTO.

Листинг 22.11. Функция `rtt_stop`: обновление показателей RTT и вычисление нового

```

//lib/rtt.c
62 void
63 rtt_stop(struct rtt_info *ptr, uint32_t ms)
64 {
65     double delta;

66     ptr->rtt_rtt = ms / 1000.0; /* измеренное значение RTT в секундах */

67     /*
68      * Обновляем оценочные значения RTT среднего отклонения RTT.
69      * (См. статью Джекобсона (Jacobson). SIGCOMM'88. Приложение A.)
70      * Здесь мы для простоты используем числа с плавающей точкой.
71     */

72     delta = ptr->rtt_rtt - ptr->rtt_srtt;
73     ptr->rtt_srtt += delta / 8; /* g - 1/8 */

74     if (delta < 0.0)
75         delta = -delta; /* |delta| */

76     ptr->rtt_rttvar += (delta - ptr->rtt_rttvar) / 4; /* h - 1/4 */

77     ptr->rtt_rto = rtt_minmax(RTT_RTOCALC(ptr));
78 }

```

62-78 Вторым аргументом является измеренное RTT, полученное вызывающим процессом при вычитании полученной в ответе отметки времени из текущей (функция `rtt_ts`). Затем применяются уравнения, приведенные в начале этого раздела, и записываются новые значения переменных `rtt_srtt`, `rtt_rttvar` и `rtt_rto`.

Последняя функция, `rtt_timeout` показана в листинге 22.12. Эта функция вызывается, когда истекает время таймера повторных передач.

Листинг 22.12. Функция `rtt_timeout`: применение экспоненциального смещения

```

//lib/rtt.c
83 int

```

```

84 rtt_timeout(struct rtt_info *ptr)
85 {
86     ptr->rtt_rto *= 2; /* следующее значение RTO */

87     if (++ptr->rtt_nrexmt > RTT_MAXNREXMT)
88         return (-1); /* закончилось время, отпущенное на попытки отправить
эти пакет */
89     return (0);
90 }

```

86 Текущее значение RTO удваивается — в этом и заключается экспоненциальное смещение.

87-89 Если мы достигли максимально возможного количества повторных передач, возвращается значение -1, указывающее вызывающему процессу, что дальнейшие попытки передачи должны прекратиться. В противном случае возвращается 0.

В нашем примере клиент соединялся дважды с двумя различными эхо-серверами в Интернете утром рабочего дня. Каждому серверу было отправлено по 500 строк. По пути к первому серверу было потеряно 8 пакетов, по пути ко второму — 16. Один из потерянных шестнадцати пакетов, предназначенных второму серверу, был потерян дважды, то есть пакет пришлось дважды передавать повторно, прежде чем был получен ответ. Все остальные потерянные пакеты пришлось передать повторно только один раз. Мы могли убедиться, что эти пакеты были действительно потеряны, посмотрев на выведенные порядковые номера каждого из полученных пакетов. Если пакет лишь опоздал, но не был потерян, после повторной передачи клиент получает два ответа: соответствующий запоздавшему первому пакету и повторно переданному. Обратите внимание, что у нас нет возможности определить, что именно было потеряно (и привело к необходимости повторной передачи клиентского запроса) — сам клиентский запрос или же ответ сервера, посланный после получения такого запроса.

ПРИМЕЧАНИЕ

Для первого издания этой книги автор написал для проверки этого клиента сервер UDP, который случайным образом игнорировал пакеты. Теперь он не используется. Нужно только соединить клиент с сервером через Интернет, и тогда нам почти гарантирована потеря некоторых пакетов!

22.6. Связывание с адресами интерфейсов

Одно из типичных применений функции `get_ifi_info` связано с приложениями UDP, которым нужно выполнять мониторинг всех интерфейсов на узле, чтобы знать, когда и на какой интерфейс приходит дейтаграмма. Это позволяет получающей программе узнавать адрес получателя дейтаграммы UDP, так как именно по этому адресу определяется сокет, на который доставляется дейтаграмма, даже если узел не поддерживает параметр сокета `IP_RECVDSTADDR`.

ПРИМЕЧАНИЕ

Вспомните наше обсуждение в конце раздела 22.2. Если узел использует более распространенную модель системы с гибкой привязкой (см. раздел 8.8), IP-адрес получателя может отличаться от IP-адреса принимающего интерфейса. В этом случае мы можем определить только адрес получателя дейтаграммы, который не обязательно должен быть адресом, присвоенным принимающему интерфейсу. Чтобы определить принимающий интерфейс, требуется параметр сокета `IP_RECVIF` или `IPV6_PKTINFO`.

В листинге 22.13 показана первая часть примера применения этой технологии к эхо-серверу UDP, который связывается со всеми адресами направленной передачи, широковещательной передачи и, наконец, с универсальными адресами.

Листинг 22.13. Первая часть сервера UDP, который с помощью функции `bind` связывается со всеми адресами

```

//advio/udpserv03.c
1 #include "unpifi.h"

2 void mydg_echo(int, SA*, socklen_t, SA*);

3 int
4 main(int argc, char **argv)
5 {
6     int sockfd;
7     const int on = 1;
8     pid_t pid;
9     struct ifi_info *ifi, *ifihead;
10    struct sockaddr_in *sa, cliaddr, wildaddr;

11    for (ifihead = ifi = Get_ifi_info(AF_INET, 1);
12         ifi != NULL; ifi = ifi->ifi_next) {

13        /* связываем направленный адрес */
14        sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

15        Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

16        sa = (struct sockaddr_in*)ifi->ifi_addr;
17        sa->sin_family = AF_INET;
18        sa->sin_port = htons(SERV_PORT);
19        Bind(sockfd, (SA*)sa, sizeof(*sa));
20        printf("bound %s\n", Sock_ntop((SA*)sa, sizeof(*sa)));

21        if ((pid = Fork()) == 0) { /* дочерний процесс */
22            mydg_echo(sockfd, (SA*)&cliaddr, sizeof(cliaddr), (SA*)sa);
23            exit(0); /* не выполняется */
24        }

```

Вызов функции *get_ifi_info* для получения информации об интерфейсе

11-12 Функция *get_ifi_info* получает все адреса IPv4, включая дополнительные (псевдонимы), для всех интерфейсов. Затем программа перебирает все структуры *ifi_info*.

Создание сокета UDP и связывание адреса направленной передачи

13-20 Создается сокет UDP, и с ним связывается адрес направленной передачи. Мы также устанавливаем параметр сокета *S0_REUSEADDR*, поскольку мы связываем один и тот же порт (*параметр SERV_PORT*) для всех IP-адресов.

ПРИМЕЧАНИЕ

Не все реализации требуют, чтобы был установлен этот параметр сокета. Например, Беркли-реализации не требуют этого параметра и позволяют с помощью функции *bind* связать уже связанный порт, если новый связываемый IP-адрес не является универсальным адресом и отличается от всех IP-адресов, уже связанных с портом. Однако Solaris 2.5 для успешного связывания с одним и тем же портом второго адреса направленной передачи требует установки этого параметра.

Порождение дочернего процесса для данного адреса

21-24 Вызывается функция `fork`, порождающая дочерний процесс. В этом дочернем процессе вызывается функция `mydg_echo`, которая ждет прибытия любой дейтаграммы на сокет и отсылает ее обратно отправителю.

В листинге 22.14 показана следующая часть функции `main`, которая обрабатывает широковещательные адреса.

Листинг 22.14. Вторая часть сервера UDP, который с помощью функции `bind` связывается со всеми адресами

```
//advio/udpserv03.c
25  if (ifi->ifi_flags & IFF_BROADCAST) {
26      /* пытаемся связать широковещательный адрес */
27      sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
28      Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

29      sa = (struct sockaddr_in*)ifi->ifi_brdaddr;
30      sa->sin_family = AF_INET;
31      sa->sin_port = htons(SERV_PORT);
32      if (bind(sockfd, (SA*)sa, sizeof(*sa)) < 0) {
33          if (errno == EADDRINUSE) {
34              printf("EADDRINUSE: %s\n",
35                  Sock_ntop((SA*)sa, sizeof(*sa)));
36              Close(sockfd);
37              continue;
38          } else
39              err_sys("bind error for %s",
40                  Sock_ntop((SA*)sa, sizeof(*sa)));
41      }
42      printf("bound %s\n", Sock_ntop((SA*)sa, sizeof(*sa)));

43      if ((pid = Fork()) == 0) { /* дочерний процесс */
44          mydg_echo(sockfd, (SA*)&cliaddr, sizeof(cliaddr),
45          (SA*)sa);
46          exit(0); /* не выполняется */
47      }
48  }
```

Связывание с широковещательными адресами

25-42 Если интерфейс поддерживает широковещательную передачу, создается сокет UDP и с ним связывается широковещательный адрес. На этот раз мы позволим функции `bind` завершиться с ошибкой `EADDRINUSE`, поскольку если у интерфейса имеется несколько дополнительных адресов (псевдонимов) в одной подсети, то каждый из различных адресов направленной передачи будет иметь один и тот же широковещательный адрес. Подобный пример приведен после листинга 17.3. В этом сценарии мы предполагаем, что успешно выполнится только первая функция `bind`.

Порождение дочернего процесса

43-47 Порождается дочерний процесс, и он вызывает функцию `mydg_echo`.

Заключительная часть функции `main` показана в листинге 22.15. В этом коде при помощи функции `bind` происходит связывание с универсальным адресом для обработки любого адреса получателя, отличного от адресов направленной и широковещательной передачи, которые уже связаны. На этот сокет

будут приходить только дейтаграммы, предназначенные для ограниченного широковещательного адреса (255.255.255.255).

Листинг 22.15. Заключительная часть сервера UDP, связывающегося со всеми адресами

```
//advio/udpserv03.c
50 /* связываем универсальный адрес */
51 sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
52 Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

53 bzero(&wildaddr, sizeof(wildaddr));
54 wildaddr.sin_family = AF_INET;
55 wildaddr.sin_addr.s_addr = htonl(INADDR_ANY);
56 wildaddr.sin_port = htons(SERV_PORT);
57 Bind(sockfd, (SA*)&wildaddr, sizeof(wildaddr));
58 printf("bound %s\n", Sock_ntop((SA*)&wildaddr, sizeof(wildaddr)));

59 if ((pid = Fork()) == 0) { /* дочерний процесс */
60   mydg_echo(sockfd, (SA*)&cliaddr, sizeof(cliaddr), (SA*)sa);
61   exit(0); /* не выполняется */
62 }
63 exit(0);
64 }
```

Создание сокета и связывание с универсальным адресом

50-62 Создается сокет UDP, устанавливается параметр сокета SO_REUSEADDR и происходит связывание с универсальным IP-адресом. Порождается дочерний процесс, вызывающий функцию mydg_echo.

Завершение работы функции main

63 Функция main завершается, и сервер продолжает выполнять работу, как и все порожденные дочерние процессы.

Функция mydg_echo, которая выполняется всеми дочерними процессами, показана в листинге 22.16.

Листинг 22.16. Функция mydg_echo

```
//advio/udpserv03.c
65 void
66 mydg_echo(int sockfd, SA *pcliaddr, socklen_t clilen, SA *myaddr)
67 {
68   int n;
69   char mesg[MAXLINE];
70   socklen_t len;

71   for (;;) {
72     len = clilen;
73     n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
74     printf("child %d, datagram from %s", getpid(),
75     Sock_ntop(pcliaddr, len));
76     printf(", to %s\n", Sock_ntop(myaddr, clilen));

77     Sendto(sockfd, mesg, n, 0, pcliaddr, len);
78   }
79 }
```

Новый аргумент

65-66 Четвертым аргументом этой функции является IP-адрес, связанный с сокетом. Этот сокет должен получать только дейтаграммы, предназначенные для данного IP-адреса. Если IP-адрес является универсальным, сокет должен получать только те дейтаграммы, которые не подходят ни для какого другого сокета, связанного с тем же портом.

Чтение дейтаграммы и отражение ответа

71-78 Дейтаграмма читается с помощью функции `recvfrom` и отправляется клиенту обратно с помощью функции `sendto`. Эта функция также выводит IP-адрес клиента и IP-адрес, который был связан с сокетом.

Запустим эту программу на нашем узле `solaris` после установки псевдонима для интерфейса `hme0` Ethernet. Адрес псевдонима: узел 200 в сети 10.0.0/24.

```
solaris % udpserv03
bound 127.0.0.1:9877      интерфейс закольцовки
bound 10.0.0.200:9877    направленный адрес интерфейса hme0:1
bound 10.0.0.255:9877    широковещательный адрес интерфейса hme0:1
bound 192.168.1.20:9877  направленный адрес интерфейса hme0
bound 192.168.1.255:9877 широковещательный адрес интерфейса hme0
bound 0.0.0.0.9877       универсальный адрес
```

При помощи утилиты `netstat` мы можем проверить, что все сокеты связаны с указанными IP-адресами и портом:

```
solaris % netstat -na | grep 9877
127.0.0.1.9877          Idle
10.0.0.200.9877         Idle
*.9877                  Idle
192.129.100.100.9877   Idle
*.9877                  Idle
*.9877                  Idle
```

Следует отметить, что для простоты мы создаем по одному дочернему процессу на сокет, хотя возможны другие варианты. Например, чтобы ограничить число процессов, программа может управлять всеми дескрипторами сама, используя функцию `select` и не вызывая функцию `fork`. Проблема в данном случае будет заключаться в усложнении кода. Хотя использовать функцию `select` для всех дескрипторов несложно, нам придется осуществить некоторое сопоставление каждого дескриптора связанному с ним IP-адресу (вероятно, с помощью массива структур), чтобы иметь возможность вывести IP-адрес получателя после того, как на определенном сокете получена дейтаграмма. Часто бывает проще использовать отдельный процесс или поток для каждой операции или дескриптора вместо мультиплексирования множества различных операций или дескрипторов одним процессом.

22.7. Параллельные серверы UDP

Большинство серверов UDP являются последовательными (iterative): сервер ждет запрос клиента, считывает запрос, обрабатывает его, отправляет обратно ответ и затем ждет следующий клиентский запрос. Но когда обработка запроса клиента занимает длительное время, желательно так или иначе совместить во времени обработку различных запросов.

Определение «длительное время» означает, что другой клиент вынужден ждать в течение некоторого заметного для него промежутка времени, пока обслуживается текущий клиент. Например, если два клиентских запроса приходят в течение 10 мс и предоставление сервиса каждому клиенту занимает в среднем 5 с, то второй клиент будет вынужден ждать ответа около 10 с вместо 5 с (если бы запрос был принят в обработку сразу же по прибытии).

В случае TCP проблема решается просто — требуется лишь породить дочерний процесс с помощью функции `fork` (или создать новый поток, что мы увидим в главе 23) и дать возможность дочернему процессу выполнять обработку нового клиента. При использовании TCP ситуация существенно упрощается за счет того, что каждое клиентское соединение уникально: пара сокетов TCP уникальна для каждого соединения. Но в случае с UDP мы вынуждены рассматривать два различных типа серверов.

1. Первый тип — простой сервер UDP, который читает клиентский запрос, посыпает ответ и затем завершает работу с клиентом. В этом сценарии сервер, читающий запрос клиента, может с помощью функции `fork` породить дочерний процесс и дать ему возможность обработать запрос. «Запрос», то есть содержимое дейтаграммы и структура адреса сокета, содержащая адрес протокола клиента, передаются дочернему процессу в виде копии содержащей области памяти из функции `fork`. Затем дочерний процесс посыпает свой ответ непосредственно клиенту.

2. Второй тип — сервер UDP, обменивающийся множеством дейтаграмм с клиентом. Проблема здесь в том, что единственный номер порта сервера, известный клиенту, — это номер заранее известного порта. Клиент посыпает первую дейтаграмму своего запроса на этот порт, но как сервер сможет отличить последующие дейтаграммы этого клиента от запросов новых клиентов? Типичным решением этой проблемы для сервера будет создание нового сокета для каждого клиента, связывание при помощи функции `bind` динамически назначаемого порта с этим сокетом и использование этого сокета для всех своих ответов. При этом требуется, чтобы клиент запомнил номер порта, с которого был отправлен первый ответ сервера, и отправлял последующие дейтаграммы уже на этот порт.

Примером второго типа сервера UDP является сервер TFTP (Trivial File Transfer Protocol — упрощенный протокол передачи файлов). Передача файла с помощью TFTP обычно требует большого числа дейтаграмм (сотен или тысяч, в зависимости от размера файла), поскольку этот протокол отправляет в одной дейтаграмме только 512 байт. Клиент отправляет дейтаграмму на известный порт сервера (69), указывая, какой файл нужно отправить или получить. Сервер читает запрос, но отправляет ответ с другого сокета, который он создает и связывает с динамически назначаемым портом. Все последующие дейтаграммы между клиентом и сервером используют для передачи этого файла новый сокет. Это позволяет главному серверу TFTP продолжать обработку других клиентских запросов, приходящих на порт 69, в то время как происходит передача файла (возможно, в течение нескольких секунд или даже минут).

Если мы рассмотрим автономный сервер TFTP (то есть случай, когда не используется демон `inetd`), то получим сценарий, показанный на рис. 22.3. Мы считаем, что динамически назначаемый порт, связанный дочерним процессом с его новым сокетом, — это порт 2134.

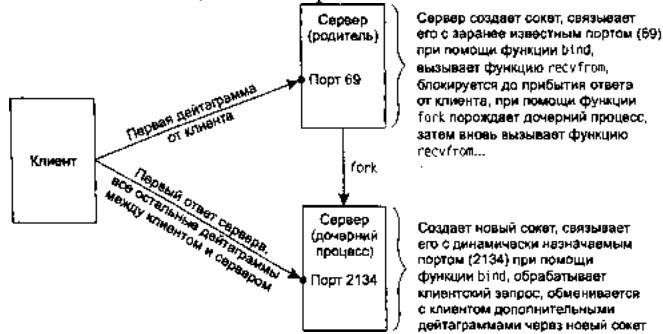


Рис. 22.3. Процессы, происходящие на автономном параллельном UDP-сервере

Если используется демон `inetd`, сценарий включает еще один шаг. Вспомните из табл. 13.4, что большинство серверов UDP задают аргумент `wait-flag` как `wait`. В описании, которое следовало за рис. 13.4, мы сказали, что при указанном значении этого флага демон `inetd` приостанавливает выполнение функции `select` на сокете до завершения дочернего процесса, давая возможность этому дочернему процессу считать дейтаграмму, доставленную на сокет. На рис. 22.4 показаны все шаги.

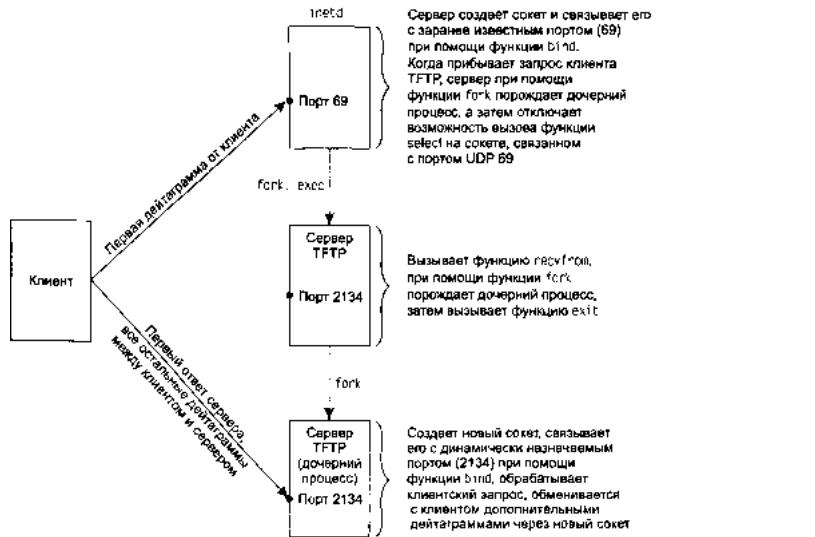


Рис. 22.4. Параллельный сервер UDP, запущенный демоном inetd

Сервер TFTP, являясь дочерним процессом функции `inetd`, вызывает функцию `recvfrom` и считывает клиентский запрос. Затем он с помощью функции `fork` порождает собственный дочерний процесс, и этот дочерний процесс будет обрабатывать клиентский запрос. Затем сервер TFTP вызывает функцию `exit`, отправляя демону `inetd` сигнал `SIGCHLD`, который, как мы сказали, указывает демону `inetd` снова вызвать функцию `select` на сокете, связанном с портом UDP 69.

22.8. Информация о пакетах IPv6

IPv6 позволяет приложению определять до пяти характеристик исходящей дейтаграммы:

- IPv6-адрес отправителя;
- индекс интерфейса для исходящих дейтаграмм;
- предельное количество транзитных узлов для исходящих дейтаграмм;
- адрес следующего транзитного узла;
- класс исходящего трафика.

Эта информация отправляется в виде вспомогательных данных с функцией `sendmsg`. Для сокета можно задать постоянные параметры, которые будут действовать на все отправляемые пакеты (раздел 27.7).

Для полученного пакета могут быть возвращены четыре аналогичных характеристики. Они возвращаются в виде вспомогательных данных с функцией `recvmsg`:

- IPv6-адрес получателя;
- индекс интерфейса для входящих дейтаграмм;
- предельное количество транзитных узлов для входящих дейтаграмм.
- класс входящего трафика.

На рис. 22.5 показано содержимое вспомогательных данных, о которых рассказывается далее.

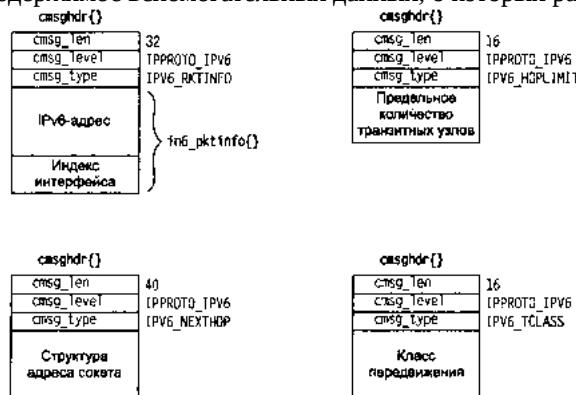


Рис. 22.5. Вспомогательные данные для информации о пакете IPv6

Структура `in6_pktinfo` содержит либо IPv6-адрес отправителя и индекс интерфейса для исходящей дейтаграммы, либо IPv6-адрес получателя и индекс интерфейса для получаемой дейтаграммы:

```
struct in6_pktinfo {  
    struct in6_addr ipi6_addr; /* IPv6-адрес отправителя/получателя */  
    int ipi6_ifindex; /* индекс интерфейса для исходящей/получаемой дейтаграммы */  
};
```

Эта структура определяется в заголовочном файле `<netinet/in.h>`, подключение которого позволяет ее использовать. В структуре `cmsghdr`, содержащей вспомогательные данные, элемент `cmsg_level` будет иметь значение `IPPROTO_IPV6`, элемент `cmsg_type` будет равен `IPV6_PKTINFO` и первый байт данных будет первым байтом структуры `in6_pktinfo`. В примере, приведенном на рис. 22.5, мы считаем, что между структурой `cmsghdr` и данными нет заполнения и целое число занимает 4 байта.

Чтобы отправить эту информацию, никаких специальных действий не требуется — нужно только задать управляющую информацию во вспомогательных данных функции `sendmsg`. Чтобы информация добавлялась ко всем отправляемым через сокет пакетам, необходимо установить параметр сокета `IPV6_PKTINFO` со значением `in6_pktinfo`. Возвращать эту информацию функция `recvmsg` будет, только если приложение включит параметр сокета `IPV6_RECVPKTINFO`.

Исходящий и входящий интерфейсы

Интерфейсы на узле IPv6 идентифицируются небольшими целыми положительными числами, как мы сказали в разделе 18.6. Вспомните, что ни одному интерфейсу не может быть присвоен нулевой индекс. При задании исходящего интерфейса ядро само выберет исходящий интерфейс, если значение `ip6_ifindex` нулевое. Если приложение задает исходящий интерфейс для пакета многоадресной передачи, то любой интерфейс, заданный параметром сокета `IPV6_MULTICAST_IF`, заменяется на интерфейс, заданный вспомогательными данными (но только для данной дейтаграммы).

Адрес отправителя и адрес получателя IPv6

IPv6-адрес отправителя обычно определяется при помощи функции `bind`. Но если адрес отправителя поставляется вместе с данными, это может снизить непроизводительные затраты. Этот параметр также позволяет серверу гарантировать, что адрес отправителя ответа совпадает с адресом получателя клиентского запроса — некоторым клиентам требуется такое условие, которое сложно выполнить в случае IPv4 (см. упражнение 22.4).

Когда IPv6-адрес отправителя задан в качестве вспомогательных данных и элемент `ip6_addr` структуры `in6_pktinfo` имеет значение `IN6ADDR_ANY_INIT`, возможны следующие сценарии: если адрес в настоящий момент связан с сокетом, он используется в качестве адреса отправителя; если в настоящий момент никакой адрес не связан с сокетом, ядро выбирает адрес отправителя. Если же элемент `ip6_addr` не является неопределенным адресом, но сокет уже связался с адресом отправителя, то значением элемента `ip6_addr` перекрывается уже связанный адрес, но только для данной операции вывода. Затем ядро проверяет, действительно ли запрашиваемый адрес отправителя является адресом направленной передачи, присвоенным узлу.

Когда структура `in6_pktinfo` возвращается в качестве вспомогательных данных функцией `recvmsg`, элемент `ip6_addr` содержит IPv6-адрес получателя из полученного пакета. По сути, это аналог параметра сокета `IP_RECVDSTADDR` для IPv4.

Задание и получение предельного количества транзитных узлов

Предельное количество транзитных узлов обычно задается параметром сокета `IPV6_UNICAST_HOPS` для дейтаграмм направленной передачи (см. раздел 7.8) или параметром сокета `IPV6_MULTICAST_HOPS` для дейтаграмм многоадресной передачи (см. раздел 21.6). Задавая предельное количество транзитных узлов в составе вспомогательных данных, мы можем заменить как значение этого предела, задаваемое ядром по умолчанию, так и ранее заданное значение — и для направленной, и для многоадресной передачи, но только для одной операции вывода. Предел количества транзитных узлов полученного пакета используется в таких программах, как `traceroute`, и в некоторых приложениях IPv6, которым нужно проверять, что полученное значение равно 255 (то есть что пакет не пересыпался маршрутизаторами).

Полученное предельное количество транзитных узлов возвращается в виде вспомогательных данных функцией `recvmsg`, только если приложение включает параметр сокета `IPV6_RECVHOPLIMIT`. В структуре `cmsghdr`, содержащей эти вспомогательные данные, элемент `cmsg_level` будет иметь значение `IPPROTO_IPV6`, элемент `cmsg_type` — значение `IPV6_HOPLIMIT`, а первый байт данных будет первым байтом целочисленного предела повторных передач. Мы показали это на рис. 22.5. Нужно понимать, что значение, возвращаемое в качестве вспомогательных данных, — это действительное значение из полученной дейтаграммы, в то время как значение, возвращаемое функцией `getsockopt` с параметром `IPV6_UNICAST_HOPS`, является значением по умолчанию, которое ядро будет использовать для исходящих дейтаграмм на сокете.

Чтобы задать предельное количество транзитных узлов для исходящих пакетов, никаких специальных действий не требуется — нам нужно только указать управляющую информацию в виде вспомогательных данных для функции `sendmsg`. Обычные значения для предельного количества транзитных узлов лежат в диапазоне от 0 до 255 включительно, но если целочисленное значение равно -1, это указывает ядру, что следует использовать значение по умолчанию.

ПРИМЕЧАНИЕ

Предельное количество транзитных узлов не содержится в структуре `in6_pktnfo` — некоторые серверы UDP хотят отвечать на запросы клиентов, посылая ответы на том же интерфейсе, на котором был получен запрос, с совпадением IPv6-адреса отправителя ответа и IPv6-адреса получателя запроса. Для этого приложение может включить параметр сокета `IPV6_RECVPKTINFO`, а затем использовать полученную управляющую информацию из функции `recvmsg` в качестве управляющей информации для функции `sendmsg` при отправке ответа. Приложению вообще никак не нужно проверять или изменять структуру `in6_pktnfo`. Но если в этой структуре содержался бы предел количества транзитных узлов, приложение должно было бы проанализировать полученную управляющую информацию и изменить значение этого предела, поскольку полученный предел не является желательным значением для исходящего пакета.

Задание адреса следующего транзитного узла

Объект вспомогательных данных `IPV6_NEXTHOP` задает адрес следующего транзитного узла дейтаграммы в виде структуры адреса сокета. В структуре `cmsghdr`, содержащей эти вспомогательные данные, элемент `cmsg_level` будет иметь значение `IPPROTO_IPV6`, элемент `cmsg_type` — значение `IPV6_NEXTHOP`, а первый байт данных будет первым байтом структуры адреса сокета.

На рис. 22.5 мы показали пример такого объекта вспомогательных данных, считая, что структура адреса сокета — это 24-байтовая структура `sockaddr_in6`. В этом случае узел, идентифицируемый данным адресом, должен быть соседним для отправляющего узла. Если этот адрес совпадает с адресом получателя IPv6-дейтаграммы, мы получаем эквивалент параметра сокета `SO_DONTROUTE`. Установка этого параметра требует прав привилегированного пользователя. Адрес следующего транзитного узла можно устанавливать для всех пакетов на сокете, если включить параметр сокета `IPV6_NEXTHOP` со значением `sockaddr_in6` (раздел 27.7). Для этого необходимо обладать правами привилегированного пользователя.

Задание и получение класса трафика

Объект вспомогательных данных `IPV6_TCLASS` задает класс трафика для дейтаграммы. Элемент `cmsg_level` структуры `cmsghdr`, содержащей эти данные, будет равен `IPPROTO_IPV6`, элемент `cmsg_type` будет равен `IPV6_TCLASS`, а первый байт данных будет первым байтом целочисленного (4-байтового) значения класса трафика (см. рис. 22.5). Согласно разделу А.3, класс трафика состоит из полей DSCP и ECN. Эти поля должны устанавливаться одновременно. Ядро может маскировать или игнорировать указанное пользователем значение, если ему это нужно (например, если ядро реализует ECN, оно может установить биты ECN равными какому-либо значению, игнорируя два бита, указанных с параметром `IPV6_TCLASS`). Класс трафика обычно лежит в диапазоне 0–255. Значение -1 говорит ядру о необходимости использовать значение по умолчанию.

Чтобы задать класс трафика для пакета, нужно отправить вспомогательные данные вместе с этим пакетом. Чтобы задать класс трафика для всех пакетов, отправляемых через сокет, необходимо использовать параметр сокета `IPV6_TCLASS` (раздел 27.7). Класс трафика для принятого пакета возвращается функцией `recvmsg` во вспомогательных данных, только если приложение включило параметр сокета `IPV6_RECVTCLASS`.

22.9. Управление транспортной MTU IPv6

IPv6 предоставляет приложениям средства для управления механизмом обнаружения транспортной MTU (раздел 2.11). Значения по умолчанию пригодны для подавляющего большинства приложений, однако специальные программы могут настраивать процедуру обнаружения транспортной MTU так, как им нужно. Для этого имеется четыре параметра сокета.

Отправка с минимальной MTU

При работе в режиме детектирования транспортной MTU пакеты фрагментируются по MTU исходящего интерфейса или по транспортной MTU в зависимости от того, какое значение оказывается меньше. IPv6 требует минимального значения MTU 1280 байт. Это значение должно поддерживаться любой линией передачи. Фрагментация сообщений по этому минимальному значению позволяет не тратить ресурсы на обнаружение транспортной MTU (потерянные пакеты и задержки в процессе обнаружения), но зато не дает возможности отправлять большие пакеты (что более эффективно).

Минимальная MTU может использоваться приложениями двух типов. Во-первых, это приложения многоадресной передачи, которым нужно избегать порождения множества ICMP-сообщений «Message too big». Во-вторых, это приложения, выполняющие небольшие по объему транзакции с большим количеством адресатов (например, DNS). Обнаружение MTU для многоадресного сеанса может быть недостаточно выгодным, чтобы компенсировать затраты на получение и обработку миллионов ICMP-сообщений, а приложения типа DNS обычно связываются с серверами недостаточно часто, чтобы можно было рисковать утратой пакетов.

Использование минимальной MTU обеспечивается параметром сокета `IPV6_USE_MIN_MTU`. Для него определено три значения: -1 (по умолчанию) соответствует использованию минимальной MTU для многоадресных передач и обнаруженной транспортной MTU для направленных передач; 0 соответствует обнаружению транспортной MTU для всех передач; 1 означает использование минимальной MTU для всех адресатов.

Параметр `IPV6_USE_MIN_MTU` может быть передан и во вспомогательных данных. В этом случае элемент `cmsg_level` структуры `cmsg_hdr` должен иметь значение `IPPROTO_IPV6`, элемент `cmsg_type` должен иметь значение `IPV6_USE_MIN_MTU`, а первый байт данных должен быть первым байтом четырехбайтового целочисленного значения параметра.

Получение сообщений об изменении транспортной MTU

Для получения уведомлений об изменении транспортной MTU приложение может включить параметр сокета `IPV6_RECVPATHMTU`. Этот флаг разрешает доставку транспортной MTU во вспомогательных данных каждый раз, когда эта величина меняется. Функция `recvmsg` в этом случае возвратит дейтаграмму нулевой длины, но со вспомогательными данными, в которых будет помещена транспортная MTU. Элемент `cmsg_level` структуры `cmsg_hdr` будет иметь значение `IPPROTO_IPV6`, элемент `cmsg_type` будет `IPV6_PATHMTU`, а первый байт данных будет первым байтом структуры `ip6_mtuinfo`. Эта структура содержит адрес узла, для которого изменилась транспортная MTU, и новое значение этой величины в байтах.

```
struct ip6_mtuinfo {  
    struct sockaddr_in6 ip6m_addr; /* адрес узла */  
    uint32_t          ip6m_mtu;   /* транспортная MTU  
                                 в порядке байтов узла */  
};
```

Эта структура определяется включением заголовочного файла `<netinet/in.h>`.

Определение текущей транспортной MTU

Если приложение не отслеживало изменения MTU при помощи параметра `IPV6_RECVPATHMTU`, оно может определить текущее значение транспортной MTU *присоединенного* сокета при помощи параметра `IPV6_PATHMTU`. Этот параметр доступен только для чтения и возвращает он структуру `ip6_mtuinfo` (см. выше), в которой хранится текущее значение MTU. Если значение еще не было определено, возвращается значение MTU по умолчанию для исходящего интерфейса. Значение адреса из структуры `ip6_mtuinfo` в данном случае не определено.

Отключение фрагментации

По умолчанию стек IPv6 фрагментирует исходящие пакеты по транспортной MTU. Приложениям типа `traceroute` автоматическая фрагментация не нужна, потому что им нужно иметь возможность самостоятельно определять транспортную MTU. Параметр сокета `IPV6_DONTFRAG` используется для отключения автоматической фрагментации: значение 0 (по умолчанию) разрешает фрагментацию, тогда как значение 1 отключает ее.

Когда автоматическая фрагментация отключена, вызов `send` со слишком большим пакетом может возвратить ошибку `EMSGSIZE`, но это не является обязательным. Единственным способом определить необходимость фрагментации пакета является использование параметра сокета `IPV6_RECVPATHMTU`, который мы описали выше.

Параметр `IPV6_DONTFRAG` может передаваться и во вспомогательных данных. При этом элемент `cmsg_level` структуры `cmsg_hdr` должен иметь значение `IPPROTO_IPV6`, а элемент `cmsg_type` должен иметь значение `IPV6_DONTFRAG`. Первый байт данных должен быть первым байтом четырехбайтового целого.

22.10. Резюме

Существуют приложения, которым требуется знать IP-адрес получателя дейтаграммы UDP и интерфейс, на котором была получена эта дейтаграмма. Чтобы получать эту информацию в виде вспомогательных данных для каждой дейтаграммы, можно установить параметры сокета `IP_RECVDSTADDR` и `IP_RFCVIF`. Аналогичная информация вместе с предельным значением количества транзитных узлов полученной дейтаграммы для сокетов IPv6 становится доступна при включении параметра сокета `IPV6_PKTINFO`.

Несмотря на множество полезных свойств, предоставляемых протоколом TCP и отсутствующих в UDP, существуют ситуации, когда следует отдать предпочтение UDP. UDP *должен* использоваться для широковещательной или многоадресной передачи. UDP *может* использоваться в простых сценариях «запрос-ответ», но тогда приложение должно само обеспечить некоторую функциональность, повышающую надежность протокола UDP. UDP *не следует* использовать для передачи большого количества данных.

В разделе 22.5 мы добавили нашему клиенту UDP определенные функциональные возможности, повышающие его надежность за счет обнаружения факта потери пакетов, для чего используются тайм-аут и повторная передача. Мы изменили тайм-аут повторной передачи динамически, снабжая каждый пакет отметкой времени и отслеживая два параметра: период обращения RTT и его среднее отклонение. Мы также добавили порядковые номера, чтобы проверять, что данный ответ — это ожидаемый нами ответ на определенный запрос. Наш клиент продолжал использовать простой протокол остановки и ожидания (*stop-and-wait*), а приложения такого типа допускают применение UDP.

Упражнения

1. Почему в листинге 22.16 функция `printf` вызывается дважды?
2. Может ли когда-нибудь функция `dg_send_recv` (см. листинги 22.6 и 22.7) возвратить нуль?
3. Перепишите функцию `dg_send_recv` с использованием функции `select` и ее таймера вместо `alarm`, `SIGALRM`, `sigsetjmp` и `siglongjmp`.
4. Как может сервер IPv4 гарантировать, что адрес отправителя в его ответе совпадает с адресом получателя клиентского запроса? (Аналогичную функциональность предоставляет параметр сокета `IPV6_PKTINFO`.)
5. Функция `main` в разделе 22.6 является зависящей от протокола (IPv4). Перепишите ее, чтобы она стала не зависящей от протокола. Потребуйте, чтобы пользователь задал один или два аргумента

командной строки, первый из которых — необязательный IP-адрес (например, 0.0.0.0 или 0::0), а второй — обязательный номер порта. Затем вызовите функцию `udp_client`, чтобы получить семейство адресов, номер порта и длину структуры адреса сокета.

Что произойдет, если вы вызовете функцию `udp_client`, как было предложено, не задавая аргумент `hostname`, поскольку функция `udp_client` не задает значение `AI_PASSIVE` функции `getaddrinfo`?

6. Соедините клиент, показанный в листинге 22.4, с эхо-сервером через Интернет, изменив функции `rtt_` так, чтобы выводилось каждое значение RTT. Также измените функцию `dg_send_recv`, чтобы она выводила каждый полученный порядковый номер. Изобразите на графике полученные в результате значения RTT вместе с оценочными значениями RTT и среднего отклонения.

Глава 23

Дополнительные сведения о сокетах SCTP

23.1. Введение

В этой главе мы займемся углубленным рассмотрением SCTP, изучим особенности этого протокола и параметры сокетов, при помощи которых он управляет. Мы обсудим некоторые специальные вопросы, в частности, управление обнаружением отказов, доставку неупорядоченных данных, а также уведомления. Мы будем щедро иллюстрировать наши утверждения примерами программ, которые помогут читателю получить представление об использовании расширенных функций SCTP.

SCTP — протокол, ориентированный на передачу сообщений. Он способен доставлять сообщения конечному пользователю как целиком, так и по частям. Доставка по частям включается только в том случае, если приложение отправляет собеседнику большие сообщения (то есть такие, размер которых превышает половину размера буфера). Части разных сообщений никогда не смешиваются друг с другом. Приложение получает сообщение либо одним вызовом функции чтения, либо несколькими последовательными вызовами. Метод работы с механизмом частичной доставки мы продемонстрируем на примере вспомогательной функции.

Серверы SCTP могут быть как последовательными, так и параллельными в зависимости от того, какой тип интерфейса выберет разработчик приложения. SCTP предоставляет средства извлечения ассоциации из сокета типа «один-ко-многим» в отдельный сокет типа «один-к-одному». Благодаря этому появляется возможность создания последовательно-параллельных серверов.

23.2. Сервер типа «один-ко-многим» с автоматическим закрытием

Вспомните программу-сервер, которую мы написали в главе 10. Эта программа не отслеживала ассоциации. Сервер рассчитывал, что клиент самостоятельно закроет ассоциацию, удалив тем самым данные о ее состоянии. Однако такой подход делает сервер уязвимым: что если клиент откроет ассоциацию, но никаких данных не пришлет? Для такого клиента будут выделены ресурсы, которые он не использует. Неудачное стечание обстоятельств может привести к DoS-атаке на нашу реализацию SCTP со стороны неактивных клиентов. Для предотвращения подобных ситуаций в SCTP была добавлена функция автоматического закрытия ассоциаций (autoclose).

Автоматическое закрытие позволяет конечной точке SCTP задавать максимальную длительность бездействия ассоциации. Ассоциация считается бездействующей, если по ней не передаются никакие данные (ни в одном направлении). Если длительность бездействия превышает установленное ограничение, ассоциация автоматически закрывается реализацией SCTP.

Особое внимание следует уделить выбору ограничения на время бездействия. Значение не должно быть слишком маленьким, иначе сервер может в какой-то момент обнаружить, что ему требуется передать данные по уже закрытой ассоциации. На повторное открытие ассоциации будут затрачены ресурсы, да и вообще маловероятно, что клиент будет готов принять входящую ассоциацию. В листинге 23.1^[1] приведена новая версия кода нашего сервера, в которую добавлены вызовы, защищающие этот сервер от неактивных клиентов. Как отмечалось в разделе 7.10, функция автоматического закрытия по умолчанию отключена и должна быть включена явным образом при помощи параметра сокета SCTP_AUTOCLOSE.

Листинг 23.1. Включение автоматического закрытия сокета на сервере

```
//sctp/sctpserv04.c
14 if (argc == 2)
15     stream_increment = atoi(argv[1]);
16 sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
17 close_time = 120;
18 Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_AUTOCLOSE,
19             &close_time, sizeof(close_time));
20 bzero(&servaddr, sizeof(servaddr));
21 servaddr.sin_family = AF_INET;
```

```
22 servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
23 servaddr.sin_report = htons(SERV_PORT);
```

Установка автоматического закрытия

17-19 Сервер устанавливает ограничение на простой ассоциаций равным 120 с и помещает это значение в переменную `close_time`. Затем сервер вызывает функцию `setsockopt` с параметром `SCTP_AUTOCLOSE`, устанавливающим выбранное ограничение. В остальном код сервера остается прежним.

Теперь SCTP будет автоматически закрывать ассоциации, простоявшие более двух минут. Автоматическое закрытие ассоциаций уменьшает расходы ресурсов сервера на неактивных клиентов.

23.3. Частичная доставка

Механизм частичной доставки (partial delivery) используется стеком SCTP каждый раз, когда требуется доставить приложению большое сообщение. Сообщение считается «большим», если SCTP решает, что у него недостаточно ресурсов на его обработку. Частичная доставка накладывает на работу SCTP определенные ограничения:

- объем памяти, занимаемой сообщением в буфере, должен превосходить некоторое пороговое значение;
- доставка может выполняться только последовательно от начала сообщения до первого отсутствующего блока;
- после включения механизма частичной доставки приложение не может получить никакие другие сообщения до тех пор, пока «большое» сообщение не будет им полностью считано из буфера. Таким образом, большое сообщение блокирует все остальные, которые в противном случае могли бы быть доставлены (в том числе и по другим потокам).

В реализации SCTP, выполненной группой КАМЕ, используется пороговое значение, равное половине объема приемного буфера сокета. На момент написания этой книги объем приемного буфера по умолчанию составляет 131 072 байта. Если параметр сокета `SO_RCVBUF` не меняется, механизм частичной доставки будет включаться только для сообщений, превышающих 65 536 байт. Мы продолжим расширение новой версии сервера из раздела 10.2, написав функцию-обертку для вызова `sctp_recvmsg`. Затем мы создадим новый сервер, который будет использовать эту функцию. В листинге 23.2 представлена функция-обертка, способная работать с механизмом частичной доставки.

Листинг 23.2. Работа с API частичной доставки

```
//sctp/sctp_pdapirev.c
1 #include "unp.h"

2 static uint8_t *sctp_pdapi_readbuf=NULL;
3 static int sctp_pdapi_rdbuf_sz=0;

4 uint8_t*
5 pdapi_recvmsg(int sock_fd,
6   int *rdlen,
7   SA *from,
8   int *from_len, struct sctp_sndrcvinfo *sri, int *msg_flags)
9 {
10  int rdsz, left, at_in_buf;
11  int frmlen=0;

12  if (sctp_pdapi_readbuf == NULL) {
13    sctp_pdapi_readbuf = (uint8_t*)Malloc(SCTP_PDAPI_INCR_SZ);
14    sctp_pdapi_rdbuf_sz = SCTP_PDAPI_INCR_SZ;
15  }
16  at_in_buf = Sctp_recvmsg(sock_fd, sctp_pdapi_readbuf, sctp_pdapi_rdbuf_sz,
17    from, from_len,
18    sri.msg_flags);
```

```

19 if (at_in_buf < 1) {
20     *rdlen = at_in_buf;
21     return(NULL);
22 }
23 while ((*msg_flags & MSG_EOR) == 0) {
24     left = sctp_pdapi_rdbuf_sz = at_in_buf;
25     if (left < SCTP_PDAPI_NEED_MORE_THRESHOLD) {
26         sctp_pdapi_readbuf =
27             realloc(sctp_pdapi_readbuf,
28                     setp_pdapi_rdbuf_sz + SCTP_PDAPI_INCR_SZ);
29         if (sctp_pdapi_readbuf == NULL) {
30             err_quit("sctp_pdapi ran out of memory");
31         }
32         sctp_pdapi_rdbuf_sz += SCTP_PDAPI_INCR_SZ;
33         left = sctp_pdapi_rdbuf_sz - at_in_buf;
34     }
35     rdsz = Sctp_recvmsg(sock_fd, &sctp_pdapi_readbuf[at_in_buf],
36     left, NULL, &frmlen, NULL, msg_flags);
37     at_in_buf += rdsz;
38 }
39 *rdlen = at_in_buf;
40 return(sctp_pdapi_readbuf);
41 }

```

Подготовка статического буфера

12-15 Если статический буфер не выделен, функция выделяет его и инициализирует переменную, хранящую информацию о состоянии этого буфера.

Чтение сообщения

16-18 Первое сообщение считывается из сокета вызовом `sctp_recvmsg`.

Обработка ошибки чтения

19-22 Если `sctp_recvmsg` возвращает ошибку или признак конца файла EOF, соответствующий код возвращается вызвавшему нашу функцию процессу без всяких изменений.

Если сообщение считано не полностью

23-24 Если флаги сообщения показывают, что оно было считано не полностью, мы вызываем функцию `sctp_recvmsg` снова. Предварительно мы вычисляем объем свободного места в буфере.

Проверка необходимости увеличения статического буфера

25-34 Если остаток приемного буфера оказался меньше некоторого минимального значения, этот буфер необходимо увеличить. С этой целью мы вызываем функцию `realloc`, выделяющую буфер большего размера, после чего копируем в новый буфер данные из старого буфера. Если по какой-то причине размер буфера не может быть увеличен, функция завершает свою работу с выводом сообщения об ошибке.

Получение данных

35-36 Новые данныечитываются из буфера вызовом `sctp_recvmsg`.

Шаг вперед

37-38 Функция увеличивает индекс буфера, после чего возвращается на проверку полного считывания сообщения.

После завершения цикла

39-40 После завершения цикла функция копирует количество считанных байтов в буфер, указатель на который передается ей вызвавшим процессом, и возвращает этому процессу указатель на собственный буфер.

Теперь мы можем изменить сервер SCTP таким образом, чтобы он использовал нашу новую функцию. Новый код представлен в листинге 23.3.

Листинг 23.3. Сервер SCTP, использующий API частичной доставки

```
//sctp/sctpserv05.c
26 for (;;) {
27     len = sizeof(struct sockaddr_in);
28     bzero(&sri, sizeof(sri));
29     readbuf = pdapi_recvmsg(sock_fd, &rd_sz,
30     (SA*)&cliaddr, &len, &sri, &msg_flags);
31     if (readbuf == NULL)
32         continue;
```

Чтение сообщения

29-30 Сервер вызывает новую функцию-обертку интерфейса частичной доставки. Предварительно обнуляется переменная `sri`.

Проверка наличия считанных данных

31-32 Обратите внимание, что теперь серверу приходится проверять объем буфера, чтобы убедиться, что чтение было успешным. Если буфер оказывается нулевым, программа переходит на начало цикла.

23.4. Уведомления

В разделе 9.14 уже отмечалось, что приложение может подписаться на уведомления, общее количество которых составляет 7 штук. Пока что наше приложение игнорировало все события, за исключением прихода новых данных. В этом разделе приводятся примеры приема и интерпретации уведомлений SCTP о других событиях транспортного уровня. В листинге 23.4 представлена функция, отображающая все получаемые уведомления. Нам придется изменить и код сервера, чтобы разрешить доставку уведомлений обо всех происходящих событиях. Однако сервер не будет использовать получаемые уведомления для чего-либо конкретного.

Листинг 23.4. Функция вывода уведомлений

```
1 #include "unp.h"
2 void
3 print_notification(char *notify_buf)
4 {
```

```
5 union sctp_notification *snp;
6 struct sctp_assoc_change *sac;
7 struct sctp_paddr_change *spc;
8 struct sctp_remote_error *sre;
9 struct sctp_send_failed *ssf;
10 struct sctp_shutdown_event *sse;
11 struct sctp_adaption_event *ae;
12 struct sctp_pdapi_event *pdapi,
13 const char *str;

14.snp = (union sctp_notification*)notify_buf;
15 switch (snp->sn_header.sn_type) {
16 case SCTP_ASSOC_CHANGE:
17     sac = &snp->sn_assoc_change;
18     switch (sac->sac_state) {
19         case SCTP_COMM_UP:
20             str = "COMMUNICATION UP";
21             break;
22         case SCTP_COMM_LOST:
23             str = "COMMUNICATION LOST";
24             break;
25         case SCTP_RESTART:
26             str = "RESTART";
27             break;
28         case SCTP_SHUTDOWN_COMP:
29             str = "SHUTDOWN COMPLETE";
30             break;
31         case SCTP_CANT_STR_ASSOC:
32             str = "CAN'T START ASSOC";
33             break;
34     default:
35         str = "UNKNOWN";
36         break;
37 } /* конец ветвления switch (sac->sac_state) */
38 printf("SCTP_ASSOC_CHANGE %s, assoc=0x%x\n", str,
39 (uint32_t)sac->sac_assoc_id);
40 break;
41 case SCTP_PEER_ADDR_CHANGE:
42     spc = &snp->sn_paddr_change;
43     switch (spc->spc_state) {
44         case SCTP_ADDR_AVAILABLE:
45             str = "ADDRESS AVAILABLE";
46             break;
47         case SCTP_ADDR_UNREACHABLE:
48             str = "ADDRESS UNREACHABLE";
49             break;
50         case SCTP_ADDR_REMOVED:
51             str = "ADDRESS REMOVED";
52             break;
53         case SCTP_ADDR_ADDED:
54             str = "ADDRESS ADDED";
55             break;
56         case SCTP_ADDR MADE_PRIM:
57             str = "ADDRESS MADE PRIMARY";
58             break;
59     default:
```

```

60     str = "UNKNOWN";
61     break;
62 } /* конец ветвления switch (spc->spc_state) */
63 printf("SCTP_PEER_ADDR_CHANGE %s, addr=%s, assoc=0x%llx\n", str,
64     Sock_ntop((SA*)&spc->spc_aaddr, sizeof(spc->spc_aaddr)),
65     (uint32_t)spc->spc_assoc_id);
66     break;
67 case SCTP_REMOTE_ERROR:
68     sre = &snp->sn_remote_error;
69     printf("SCTP_REMOTE_ERROR: assoc=0x%lx error=%d\n",
70     (uint32_t)sre->sre_assoc_id, sre->sre_error);
71     break;
72 case SCTP_SEND_FAILED:
73     ssf = &snp->sn_send_failed;
74     printf("SCTP_SEND_FAILED: assoc=0x%lx error=%d\n",
75     (uint32_t)ssf->ssf_assoc_id, ssf->ssf_error);
76     break;
77 case SCTP_ADAPTION_INDICATION:
78     ae = &snp->sn_adaption_event;
79     printf("SCTP_ADAPTION_INDICATION: 0x%lx\n",
80     (u_int)ae->sai_adaption_ind);
81     break;
82 case SCTP_PARTIAL_DELIVERY_EVENT:
83     pdapi = &snp->sn_pdapi_event;
84     if (pdapi->pdapi_indication == SCTP_PARTIAL_DELIVERY_ABORTED)
85         printf("SCTP_PARTIAL_DELIVERY_ABORTED\n");
86     else
87         printf("Unknown SCTP_PARTIAL_DELIVERY_EVENT 0x%lx\n",
88         pdapi->pdapi_indication);
89     break;
90 case SCTP_SHUTDOWN_EVENT:
91     sse = &snp->sn_shutdown_event;
92     printf("SCTP_SHUTDOWN_EVENT: assoc=0x%lx\n",
93     (uint32_t)sse->sse_assoc_id);
94     break;
95 default:
96     printf("Unknown notification event type=0x%lx\n",
97     snp->sn_header.sn_type);
98 }
99 }

```

Преобразование буфера и начало ветвления

14-15 Функция преобразует буфер вызова к типу union, после чего разыменовывает структуру sn_header и тип sn_type и выполняет ветвление по значению соответствующего поля.

Обработка изменения состояния ассоциации

16-40 Если функция обнаруживает в буфере уведомление об изменении ассоциации, она выводит тип прошедшего изменения.

Изменение адреса собеседника

16-40 Если получено уведомление об изменении адреса собеседника, функция распечатывает событие и новый адрес.

Ошибка на удаленном узле

67-71 Если получено уведомление об ошибке на удаленном узле, функция отображает сообщение об этом вместе с идентификатором ассоциации, для которой получено уведомление. Мы не пытаемся декодировать и отобразить сообщение об ошибке, присланное собеседником. При необходимости эти сведения можно получить из поля `sre_data` структуры `sctp_remote_error`.

Ошибка отправки сообщения

72-76 Если получено уведомление об ошибке отправки сообщения, мы можем сделать вывод, что сообщение не было отправлено собеседнику. Это означает, что либо ассоциация завершает работу и вскоре будет получено уведомление об изменении ее состояния (если оно еще не было получено) или же сервер использует расширение частичной надежности и отправка сообщения оказалась неудачной из-за наложенных ограничений. Данные, которые все-таки были переданы, помещаются в поле `ssf_data`, которая наша функция не использует.

Индикация уровня адаптера

77-81 Если получено уведомление об уровне адаптера, функция отображает соответствующее 32-разрядное значение, полученное в сообщении INIT или INIT-ACK.

Уведомление механизма частичной доставки

82-89 Если получено уведомление механизма частичной доставки, функция выводит на экран соответствующее сообщение. Единственное определенное на момент написания этой книги событие, связанное с частичной доставкой, состоит в ее аварийном завершении.

Уведомление о завершении ассоциации

90-94 Если получено уведомление о завершении ассоциации, мы можем сделать вывод, что собеседник выполняет корректное закрытие. За этим уведомлением обычно следует уведомление об изменении состояния ассоциации, которое приходит б момент окончания последовательности пакетов, завершающих ассоциацию. Код сервера, использующего нашу новую функцию, приведен в листинге 23.5.

Листинг 23.5. Сервер, обрабатывающий уведомления о событиях

```
//sctp/sctpserv06.c
21 bzero(&evnts, sizeof(evnts));
22 evnts.sctp_data_io_event = 1;
23 evnts.sctp_association_event = 1;
24 evnts.sctp_address_event = 1;
25 evnts.sctp_send_failure_event = 1;
26 evnts.sctp_peer_error_event = 1;
27 evnts.sctp_shutdown_event = 1;
28 evnts.sctp_partial_delivery_event = 1;
29 evnts.sctp_adaption_layer_event = 1;
30 Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof(evnts));

31 Listen(sock_fd, LISTENQ);
32 for (;;) {
```

```
33 len = sizeof(struct sockaddr_in);
34 rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof(readbuf),
35 (SA*)&cliaddr, &len, &sri, &msg_flags);
36 if (msg_flags & MSG_NOTIFICATION) {
37 print_notification(readbuf);
38 continue;
39 }
```

Подписка на уведомления

21-30 Сервер изменяет параметры подписки на события таким образом, чтобы получать все возможные уведомления.

Получение данных

31-35 Эта часть кода сервера осталась неизменной.

Обработка уведомлений

36-39 Сервер проверяет поле msg_flags. Если сообщение представляет собой уведомление, сервер вызывает рассмотренную ранее функцию sctp_print_notification и переходит к обработке следующего сообщения.

Запуск программы

Мы запускаем клиент и отправляем одно сообщение.

FreeBSD-lap: ./sctpclient01 10.1.1.5

[0]Hello

From str:1 seq:0 (assoc:c99e15a0):[0]Hello

Control-D

FreeBSD-lap:

Сервер отображает сообщения обо всех происходящих событиях (приеме входящего соединения, получении сообщения, завершении соединения).

FreeBSD-lap:./sctpserv06

SCTP_ADAPTION_INDICATION:0x504c5253

SCTP_ASSOC_CHANGE: COMMUNICATION UP, assoc=c99e2680h

SCTP_SHUTDOWN_EVENT; assoc=c99e2680h

SCTP_ASSOC_CHANGE: SHUTDOWN COMPLETE, assoc=c99e2680h

Control-C

Как видите, сервер действительно выводит сообщения обо всех происходящих событиях транспортного уровня.

23.5. Неупорядоченные данные

В обычном режиме SCTP обеспечивает надежную упорядоченную доставку данных. Кроме того, SCTP предоставляет и сервис надежной неупорядоченной доставки. Сообщение с флагом MSG_UNORDERED отправляется вне очереди и делается доступным для чтения сразу же после приема на удаленном узле. Такое сообщение может быть отправлено по любому потоку. Ему не присваивается порядковый номер внутри какого-либо потока. В листинге 23.6 представлены изменения кода клиента, позволяющие ему отправлять внеочередные запросы серверу.

Листинг 23.6. Функция sctp_strcli, отправляющая внеочередные данные

//sctp/sctp_strcli_un.c

```

18 out_sz = strlen(sendline);
19 Sctp_sendmsg(sock_fd, sendline, out_sz,
20 to, tolen, 0, MSG_UNORDERED, sri.sinfo_stream, 0, 0);

```

Отправка внеочередных данных

18-20 Функция `sctp_str_cli` практически не отличается от той, которую мы разработали в разделе 10.4. Единственное изменение произошло в строке 21: клиент передает флаг `MSG_UNORDERED`, включающий механизм частичной доставки. Обычно все сообщения внутри потока упорядочиваются по номерам. Флаг `MSG_UNORDERED` позволяет отправить сообщение без порядкового номера. Такое сообщение доставляется адресату сразу после получения его стеком SCTP, даже если другие внеочередные сообщения, отправленные ранее по тому же потоку, еще не были приняты.

23.6. Связывание с подмножеством адресов

Некоторым приложениям требуется связывать один сокет с некоторым конкретным подмножеством всех адресов узла. Протоколы TCP и UDP не позволяют выделить подмножество адресов. Системный вызов `bind` позволяет приложению связать сокет с единственным адресом или сразу со всеми адресами узла (то есть с универсальным адресом). Поэтому в SCTP был добавлен новый системный вызов `sctp_bindx`, который позволяет приложению связываться с произвольным количеством адресов. Все адреса должны иметь один и тот же номер порта, а если ранее вызывалась функция `bind`, то номер порта должен быть таким, как в вызове `bind`. Если указать не тот порт, вызов `sctp_bindx` завершится с ошибкой. В листинге 23.7 представлена функция, которую мы добавим к нашему серверу, чтобы получить возможность связывать сокет с адресами, передаваемыми в качестве аргументов командной строки.

Листинг 23.7. Функция, связывающая сокет с набором адресов

```

1 #include "unp.h"

2 int
3 sctp_bind_arg_list(int sock_fd, char **argv, int argc)
4 {
5     struct addrinfo *addr;
6     char *bindbuf, *p, portbuf[10];
7     int addrcnt=0;
8     int i;

9     bindbuf = (char*)calloc(argc, sizeof(struct sockaddr_storage));
10    p = bindbuf;
11    sprintf(portbuf, "%d", SERV_PORT);
12    for (i=0; i<argc; i++ ) {
13        addr = Host_serv(argv[i], portbuf, AF_UNSPEC, SOCK_SEQPACKET);
14        memcpy(p, addr->ai_addr, addr->ai_addrlen);
15        freeaddrinfo(addr);
16        addrcnt++;
17        p += addr->ai_addrlen;
18    }
19    Sctp_bindx(sock_fd, (SA*)bindbuf, addrcnt, SCTP_BINDX_ADD_ADDR);
20    free(bindbuf);
21    return(0);
22 }

```

Выделение памяти под аргументы `bind`

9-10 Наша новая функция начинает работу с выделения памяти под аргументы функции `sctp_bindx`. Обратите внимание, что функция `sctp_bindx` может принимать в качестве аргументов адреса IPv4 и IPv6 в

произвольных комбинациях. Для каждого адреса мы выделяем место под структуру `sockaddr_storage` несмотря на то, что соответствующий аргумент `sctp_bindx` представляет собой упакованный список адресов (см. рис. 9.3). В результате мы расходуем зря некоторый объем памяти, но зато функция работает быстрее, потому что ей не приходится вычислять точный объем памяти и лишний раз обрабатывать список аргументов.

Обработка аргументов

11-18 Мы подготавливаем `portbuf` к хранению номера порта в ASCII-представлении, имея в виду вызов нашей обертки для `getaddrinfo`, которая называется `host_serv`. Каждый адрес с номером порта мы передаем `host_serv`, указывая константы `AF_UNSPEC` (протоколы IPv4 и IPv6) и `SOCK_SEQPACKET` (протокол SCTP). Мы копируем первую возвращаемую структуру `sockaddr`, игнорируя все остальные. Поскольку аргументами этой функции должны быть адреса в строковом представлении, а не имена, с каждым из которых может быть связано несколько адресов, это не вызывает проблем. Мы освобождаем буфер, увеличиваем количество адресов на единицу и перемещаем указатель на следующий элемент в упакованном массиве структур `sockaddr`.

Вызов связывающей функции

19 Указатель устанавливается на начало буфера адресов, после чего вызывается функция `sctp_bindx`, в качестве аргументов которой используется раскодированный ранее набор адресов.

Успешное завершение

20-21 Если мы добрались до этого места, можно считать, что выполнение прошло успешно, поэтому мы освобождаем память и возвращаем управление вызвавшему процессу.

В листинге 23.8 представлен модифицированный эхо-сервер, связывающий сокет с набором адресов, передаваемых в командной строке. Мы слегка изменили код сервера, чтобы он отправлял эхо-сообщения по тем потокам, по которым были приняты исходные сообщения.

Листинг 23.8. Сервер, работающий с произвольным набором адресов

```
if (argc < 2)
    err_quit("Error, use %s [list of addresses to bind]\n", argv[0]);
sock_fd = Socket(AF_INET6, SOCK_SEQPACKET, IPPROTO_SCTP);

if (sctp_bind_arg_list(sock_fd, argv + 1, argc - 1))
    err_sys("Can't bind the address set");

bzero(&evnts, sizeof(evnts));
evnts.sctp_data_io_event = 1;
```

Работа с IPv6

14 Это тот же сервер, с которым мы работали во всех предыдущих разделах этой главы, но с незначительным изменением. Сервер создает сокет `AF_INET6`, что позволяет ему работать с протоколом IP обеих версий.

Вызов `sctp_bind_arg_list`

15-16 Сервер вызывает новую функцию `sctp_bind_arg_list` и передает ей список аргументов для обработки.

23.7. Получение адресов

Поскольку протокол SCTP ориентирован на многоинтерфейсные узлы, для определения адресов локального и удаленного узла не могут использоваться те же механизмы, что и в TCP. В этом разделе мы изменим код клиента, добавив в него подписку на уведомление о событии COMMUNICATION UP. В этом уведомлении клиент будет получать сведения об адресах, между которыми установлена ассоциация. В листингах 23.9 и 23.10 представлены изменения в коде клиента. Листинги 23.11 и 23.12 содержат добавления к коду клиента.

Листинг 23.9. Клиент включает уведомления

```
16 bzero(&evnts, sizeof(evnts));
17 evnts.sctp_data_io_event = 1;
18 evnts.sctp_association_event = 1;
19 Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof(evnts));

20 sctpstr_cli(stdin, sock_fd, (SA*)&servaddr, sizeof(servaddr));
```

Включение событий и вызов функции отправки сообщения

16-20 Функция main клиента претерпевает не слишком значительные изменения. Клиент явным образом подписывается на уведомления об изменении состояния ассоциации.

Посмотрим, что нам придется изменить в функции sctpstr_cli, чтобы она смогла вызывать нашу новую функцию обработки уведомлений.

Листинг 23.10. Функция sctp_strcli, способная работать с уведомлениями

```
21 do {
22     len = sizeof(peeraddr);
23     rd_sz = Sctp_recvmsg(sock_fd, recvline, sizeof(recvline),
24     (SA*)&peeraddr, &len, &sri, &msg_flags);
25     if (msg_flags & MSG_NOTIFICATION)
26         check_notification(sock_fd, recvline, rd_sz);
27 } while (msg_flags & MSG_NOTIFICATION);
28 printf("From str:%d seq:%d (assoc.0x%u) ",
29 sri.sinfo_stream, sri.sinfo_ssn, (u_int)sri.sinfo_assoc_id);
30 printf("%.*s", rd_sz.recvline);
```

Цикл ожидания сообщения

21-24 Клиент устанавливает переменную, в которой хранится длина адреса, и вызывает функцию sctp_recvmsg для получения эхо-ответа сервера на свое сообщение.

Проверка уведомлений

25-26 Клиент проверяет, не является ли полученное сообщение уведомлением. В последнем случае он вызывает функцию обработки уведомлений, представленную в листинге 23.11.

Переход на начало цикла

27 Если сообщение действительно было уведомлением, происходит переход на начало цикла ожидания сообщений.

Отображение сообщения

28-30 Клиент отображает сообщение и переходит к ожиданию пользовательского ввода.

Теперь мы можем изучить новую функцию `sctp_check_notification`, которая будет отображать адреса обоих конечных точек при получении уведомления об изменении состояния ассоциации.

Листинг 23.11. Обработка уведомлений

```
//sctp/sctp_check_notify.c
1 #include "unp.h"

2 void
3 check_notification(int sock_fd, char *recvline, int rd_len)
4 {
5     union sctp_notification *snp;
6     struct sctp_assoc_change *sac;
7     struct sockaddr_storage *sal, *sar;
8     int num_rem, num_loc;

9     snp = (union sctp_notification*)recvline;
10    if (snp->sn_header.sn_type == SCTP_ASSOC_CHANGE) {
11        sac = &snp->sn_assoc_change;
12        if ((sac->sac_state == SCTP_COMM_UP) ||
13            (sac->sac_state == SCTP_RESTART)) {
14            num_rem = sctp_getpaddrs(sock_fd, sac->sac_assoc_id, &sar);
15            printf("There are %d remote addresses and they are:\n", num_rem);
16            sctp_print_addresses(sar, num_rem);
17            sctp_freepaddrs(sar);

18            num_loc = sctp_getladdrs(sock_fd.sac->sac_assoc_id, &sal);
19            printf("There are %d local addresses and they are:\n", num_loc);
20            sctp_print_addresses(sal, num_loc);
21            sctp_freladdrs(sal);
22        }
23    }
24 }
```

Проверка типа уведомления

9-13 Функция преобразует буфер приема к типу универсального указателя на уведомления, чтобы определить тип полученного уведомления. Из всех уведомлений нас интересуют только уведомления об изменении ассоциации, а из них — уведомления о создании или перезапуске ассоциации (`SCTP_COMM_UP` и `SCTP_RESTART`). Все прочие уведомления нас не интересуют.

Получение и вывод адресов собеседника

14-17 Функция `sctp_getpaddrs` возвращает нам список удаленных адресов, которые мы выводим при помощи функции `sctp_print_addresses`, представленной в листинге 23.12. После работы с ней мы освобождаем ресурсы, выделенные `sctp_getpaddrs`, вызывая функцию `sctp_freepaddrs`.

Получение и вывод локальных адресов

18-21 Функция `sctp_getladdrs` возвращает нам список локальных адресов, которые мы выводим на экран вместе с их общим количеством. После завершения работы с адресами мы освобождаем память вызовом `sctp_freladdrs`.

Последняя из новых функций называется `sctp_print_addresses`. Она выводит на экран адреса из списка, возвращаемого функциями `sctp_getpaddrs` и `sctp_getladdrs`. Текст функции представлен в

листинге 23.12.

Листинг 23.12. Вывод списка адресов

```
//sctp/sctp_print_addrs.c
1 #include "unp.h"

2 void
3 sctp_print_addresses(struct sockaddr_storage *addrs, int num)
4 {
5     struct sockaddr_storage *ss;
6     int i, salen;

7     ss = addrs;
8     for (i=0; i<num; i++){
9         printf("%s\n", Sock_ntop((SA*)ss, salen));
10    #ifdef HAVE_SOCKADDR_SA_LEN
11        salen = ss->ss_len;
12    #else
13        switch (ss->ss_family) {
14            case AF_INET:
15                salen = sizeof(struct sockaddr_in);
16                break;
17            #ifdef IPV6
18            case AF_INET6:
19                salen = sizeof(struct sockaddr_in6);
20                break;
21            #endif
22            default:
23                err_quit("sctp_print_addresses: unknown AF");
24                break;
25        }
26    #endif
27    ss = (struct sockaddr_storage*)((char*)ss + salen);
28 }
29 }
```

Последовательная обработка адресов

7-8 Функция перебирает адреса в цикле. Общее количество адресов указывается вызывающим процессом.

Вывод адреса

9 Адрес преобразуется к удобочитаемому виду функцией `sock_ntop`, которая, как вы помните, должна работать со структурами адреса сокета всех форматов, поддерживаемых системой.

Определение размера адреса

10-26 Список адресов передается в упакованном формате. Это не просто массив структур `sockaddr_storage`. Дело в том, что структура `sockaddr_storage` достаточно велика, и ее нецелесообразно использовать при передаче адресов между ядром и пользовательскими процессами. В системах, где эта структура содержит внутреннее поле длины, обработка списка является делом тривиальным: достаточно извлекать длину из текущей структуры `sockaddr_storage`. В прочих системах длина определяется на

основании семейства адреса. Если семейство не определено, функция завершает работу с сообщением об ошибке.

Перемещение указателя

27 К указателю на элемент списка прибавляется размер адреса. Таким образом осуществляется перемещение по списку адресов.

Выполнение программы

Результат взаимодействия модифицированного клиента с сервером представлен ниже.

```
FreeBSD-lap: ./sctpclient01 10.1.1.5
[0]Hi
There are 2 remote addresses and they are:
10.1.1.5:9877
127.0.0.1:9877
There are 2 local addresses and they are:
10.1.1.5:1025
127.0.0.1:1025
From str:0 seq:0 (assoc:c99e2680):[0]Hi
Control-D
FreeBSD-lap:
```

23.8. Определение идентификатора ассоциации по IP-адресу

Модифицированный клиент из раздела 23.7 использовал уведомления в качестве сигнала для получения списков адресов. Это было достаточно удобно, поскольку идентификатор ассоциации, для которой требовалось получить адреса, содержался в уведомлении в поле `sac_assoc_id`. Но что если приложение не отслеживает идентификаторы ассоциаций, а ему вдруг понадобилось определить какой-либо идентификатор по адресу собеседника? В листинге 23.13 представлена простая функция, преобразующая адрес собеседника в идентификатор ассоциации. Эта функция будет использоваться сервером из раздела 23.10.

Листинг 23.13. Преобразование адреса в идентификатор ассоциации

```
//sctp/sctp_addr_to_associd.c
1 #include "unp.h"

2 sctp_assoc_t
3 sctp_address_to_associd(int sock_fd, struct sockaddr *sa, socklen_t salen)
4 {
5     struct sctp_paddrparams sp;
6     int siz;

7     siz = sizeof(struct sctp_paddrparams);
8     bzero(&sp, siz);
9     memcpy(&sp, spp_address, sa.salen);
10    sctp_opt_info(sock_fd, 0, SCTP_PEER_ADDR_PARAMS, &sp, &siz);
11    return(sp.spp_assoc_id);
12 }
```

Инициализация

7-8 Функция начинает работу с инициализации структуры `sctp_paddrparams`.

Копирование адреса

9 Мы копируем адрес в структуру `sctp_paddrparams`, используя переданную нам вызвавшим процессом информацию о длине этого адреса.

Вызов параметра сокета

10 При помощи параметра сокета `SCTP_PEER_ADDR_PARAMS` наша функция запрашивает параметры адреса собеседника. Обратите внимание, что мы используем `sctp_opt_info` вместо `getsockopt`, потому что параметр `SCTP_PEER_ADDR_PARAMS` требует копирования аргументов как в ядро, так и из ядра. Вызов, который мы делаем, возвратит нам текущий интервал проверки работоспособности соединения, максимальное количество попыток повторной передачи перед принятием решения о признании адреса собеседника отказавшим, и, что самое важное, идентификатор ассоциации. Мы не проверяем возвращаемое значение, потому что если вызов оказывается неудачным, мы хотим вернуть 0.

11 Функция возвращает идентификатор ассоциации. Если вызов `sctp_opt_info` оказался неудачным, обнуление структуры гарантирует, что вызвавший нашу функцию процесс получит 0. Идентификатор ассоциации нулевым быть не может. Это значение используется реализацией SCTP для указания на отсутствие ассоциации.

23.9. Проверка соединения и ошибки доступа

Механизм периодической проверки соединения, предоставляемый протоколом SCTP, основан на той же концепции, что и параметр поддержания соединения TCP `keep-alive`. Однако в SCTP этот механизм по умолчанию включен, тогда как в TCP он выключен. Приложение может устанавливать пороговое значение количества неудачных проверок при помощи того же параметра сокета, который использовался в разделе 23.8. Порог ошибок — это количество пропущенных проверочных пакетов и тайм-аутов повторной передачи, после которого адрес получателя считается недоступным. Когда доступность адреса восстанавливается (о чем сообщают все те же проверочные пакеты), он снова становится активным.

Приложение может отключить проверку соединения, но без нее SCTP не сможет узнать о доступности адреса собеседника, который ранее был признан недоступным. Без вмешательства пользователя такой адрес не сможет стать активным.

Параметр проверки соединения задается полем `spp_hbinterval` структуры `sctp_paddrparams`. Если приложение устанавливает это поле равным `SCTP_NO_NB` (эта константа имеет значение 0), проверка соединения отключается. Ненулевое значение устанавливает задержку проверки соединения в миллисекундах. К фиксированной задержке прибавляется текущее значение таймера повторной передачи и некоторое случайное число, в результате чего получается реальный промежуток времени между проверками соединения. В листинге 23.14 приводится небольшая функция, которая позволяет устанавливать задержку проверки соединения, или вовсе отключать этот механизм протокола SCTP для конкретного адресата. Обратите внимание, что если поле `spp_pathmaxrxxr` структуры `sctp_paddrparams` оставить равным нулю, текущее значение задержки останется неизменным.

Листинг 23.14. Управление периодической проверкой соединения

```
//sctp/sctp_modify_hb.c
1 #include "unp.h"

2 int
3 heartbeat_action(int sock_fd, struct sockaddr *sa, socklen_t salen,
4 u_int value)
5 {
6     struct sctp_paddrparams sp;
7     int siz;

8     bzero(&sp, sizeof(sp));
9     sp.spp_hbinterval = value;
10    memcpy((caddr_t)&sp, spp_address, sa.salen);
```

```
11 Setsockopt(sock_fd, IPPROTO_SCTP,
12     SCTP_PEER_ADDR_PARAMS, &sp, sizeof(sp));
13 return(0);
14 }
```

Обнуление структуры *sctp_paddrparams* и копирование аргумента

8-9 Мы обнуляем структуру *sctp_paddrparams*, чтобы случайно не изменить какой-нибудь параметр, который нас не интересует. Затем мы копируем в нее переданное пользователем значение задержки: *SCTP_ISSUE_HB*, *SCTP_NO_HB* или конкретное число.

Установка адреса

10 Функция подготавливает адрес и копирует его в структуру *sctp_paddrparams*, чтобы реализация SCTP знала, к какому адресу относятся устанавливаемые нами параметры периодической проверки соединения.

Выполнение действия

11-12 Наконец, функция делает вызов параметра сокета, чтобы выполнить запрошенную пользователем операцию.

23.10. Выделение ассоциации

Пока что мы занимались исключительно интерфейсом типа «один-ко-многим». Этот интерфейс имеет несколько преимуществ перед традиционным интерфейсом «один-к-одному»:

- программа работает с единственным дескриптором;
- программисту достаточно написать простой последовательный сервер;
- приложение может передавать данные в третьем и четвертом пакетах четырехэтажного рукопожатия, если для явной установки соединения используются функции *sendmsg* и *sctp_sendmsg*;
- отсутствует необходимость в отслеживании состояния на транспортном уровне. Другими словами, приложение просто запрашивает данные из дескриптора сокета, не вызывая традиционных функций *connect* и *accept* для получения сообщений.

Есть у этого интерфейса и недостатки. Самый существенный из них состоит в том, что интерфейс типа «один-ко-многим» затрудняет написание параллельного сервера (многопоточного или порождающего процессы). Для устранения этого недостатка была придумана функция *sctp_peeloff*. Она принимает в качестве аргумента дескриптор сокета типа «один-ко-многим» и идентификатор ассоциации, а возвращает новый дескриптор сокета типа «один-к-одному» с единственной ассоциацией (сохраняя все уведомления и данные, помещенные в очередь этой ассоциации). Исходный сокет остается открытым, причем все остальные ассоциации проведенной операцией извлечения никак не затрагиваются.

Выделенный сокет может быть передан потоку или дочернему процессу для обработки запросов клиента. Листинг 23.15 демонстрирует новую модифицированную версию сервера, который обрабатывает первое сообщение клиента, выделяет ассоциацию при помощи *sctp_peeloff*, порождает дочерний процесс и вызывает функцию *str_echo* для TCP, которая была написана в разделе 5.3. Адрес из полученного сообщения мы передаем нашей функции из раздела 23.8, которая по этому адресу определяет идентификатор ассоциации. Идентификатор хранится также в поле *sri*, *sinfo_assoc_id*. Наша функция служит лишь иллюстрацией использования альтернативного метода. Породив процесс, сервер переходит к обработке следующего сообщения.

Листинг 23.15. Параллельный сервер SCTP

```
//sctp/sctpserv_fork.c
23 for (;;) {
24     len = sizeof(struct sockaddr_in);
25     rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof(readbuf),
```

```

26     (SA*)&cliaddr, &len, &sri, &msg_flags);
27 Sctp_sendmsg(sock_fd, readbuf, rd_sz,
28     (SA*)&cliaddr, len,
29     sri.sinfo_ppid,
30     sri.sinfo_flags, sn.sinfo_stream, 0, 0);
31 assoc = sctp_address_to_associd(sock_fd, (SA*)&cliaddr, len);
32 if ((int)assoc == 0) {
33     err_ret("Can't get association id");
34     continue;
35 }
36 connfd = sctp_peeloff(sock_fd, assoc);
37 if (connfd == -1) {
38     err_ret("sctp_peeloff fails");
39     continue;
40 }
41 if ((childpid = fork()) == 0) {
42     Close(sock_fd);
43     str_echo(connfd);
44     exit(0);
45 } else {
46     Close(connfd);
47 }
48 }
```

Получение и обработка первого сообщения

26-30 Сервер получает и обрабатывает первое сообщение клиента.

Преобразование адреса в идентификатор ассоциации

31-35 Сервер вызывает функцию из листинга 23.13 для получения идентификатора ассоциации по ее адресу. Если что-то мешает серверу получить идентификатор, он не делает попыток породить дочерний процесс, а просто переходит к обработке следующего сообщения.

Выделение ассоциации

36-40 Сервер выделяет ассоциацию в отдельный дескриптор сокета при помощи `sctp_peeloff`. Полученный сокет типа «один-к-одному» может быть без проблем передан написанной ранее для TCP функции `str_echo`.

Передача работы дочернему процессу

41-47 Сервер порождает дочерний процесс, который и выполняет всю обработку по конкретному дескриптору.

23.11. Управление таймерами

Протокол SCTP имеет множество численных пользовательских параметров. Все они устанавливаются через параметры сокетов, рассмотренные в разделе 7.10. Далее мы займемся рассмотрением нескольких параметров, определяющих задержку перед объявлением об отказе ассоциации или адреса собеседника.

Время обнаружения отказа в SCTP определяется семью переменными (табл. 23.1).

Таблица 23.1. Поля таймеров SCTP

Поле	Описание	По умолчанию	Единицы
srto_min	Минимальный тайм-аут повторной передачи	1000	Мс
srto_max	Максимальный тайм-аут повторной передачи	60000	Мс
srto_initial	Начальный тайм-аут повторной передачи	3000	Мс
sinit_max_init_timeo	Максимальный тайм-аут повторной передачи сегмента INIT	3000	Мс
sinit_max_attempts	Максимальное количество повторных передач сегмента INIT	8	попыток
spp_pathmaxrxt	Максимальное количество повторных передач по адресу	5	попыток
sasoc_asocmaxrxt	Максимальное количество повторных передач на ассоциацию	10	попыток

Эти параметры можно воспринимать как регуляторы, укорачивающие и удлиняющие время обнаружения отказа. Рассмотрим два сценария.

1. Конечная точка SCTP пытается открыть ассоциацию с собеседником, отключившимся от сети.

2. Две многоинтерфейсные конечные точки SCTP обмениваются данными. Одна из них отключается от сети питания в момент передачи данных. Сообщения ICMP фильтруются защитными экранами и потому не достигают второй конечной точки.

В сценарии 1 система, пытающаяся открыть соединение, устанавливает таймер RTO равным srto_initial (3000 мс). После первой повторной передачи пакета INIT таймер устанавливается на значение 6000 мс. Это продолжается до тех пор, пока не будет сделано sinit_max_attempts попыток (9 штук), между которыми пройдут семь тайм-аутов. Удвоение таймера закончится на величине sinit_max_init_timeo, равной 60 000 мс. Таким образом, через $3 + 6 + 12 + 24 + 48 + 60 + 60 + 60 = 273$ с стек SCTP объявит потенциального собеседника недоступным.

Вращением нескольких «ручек» мы можем удлинять и укорачивать это время. Начнем с двух параметров, позволяющих уменьшить общую задержку. Сократим количество повторных передач, изменив переменную sinit_max_attempts. Альтернативное изменение может состоять в уменьшении максимального тайм-аута для пакета INIT (переменная srto_max_init_timeo). Если количество попыток снизить до 4, время детектирования резко упадет до 45 с (одна шестая первоначального значения). Однако у этого метода есть недостаток: из-за проблем в сети или перегруженности собеседника мы можем объявить его недоступным, даже если это состояние является лишь временным.

Другой подход состоит в уменьшении srto_max_init_timeo до 20 с. При этом задержка до обнаружения недоступности сократится до 121 с — менее половины исходной величины. Однако и это решение является компромиссным. Если мы выберем слишком низкое значение тайм-аута, при большой сетевой задержке мы будем отправлять гораздо больше пакетов INIT, чем это требуется на самом деле.

Перейдем теперь к сценарию 2, описывающему взаимодействие двух многоинтерфейсных узлов. Одна конечная точка имеет адреса IP-А и IP-В, другая IP-Х и IP-Ү. Если одна из них становится недоступна, а вторая отправляет какие-то данные, последней придется делать повторные передачи по каждому из адресов с задержкой, начинающейся с srto_min (по умолчанию 1 с) и последовательно удваивающейся до значения srto_max (по умолчанию 60 с). Повторные передачи будут продолжаться до тех пор, пока не будет достигнуто ограничение на их количество sasoc_asocmaxrxt (по умолчанию 10 повторных передач).

В нашем сценарии последовательность тайм-аутов будет иметь вид 1(IP-А) + 1(IP-В) + 2(IP-А) + 2(IP-В) + 4(IP-А) + 4(IP-В) + 8(IP-А) + 8(IP-В) + 16(IP-А) + 16(IP-В), что в общей сложности составит 62 с. Параметр srto_max не влияет на работу многоинтерфейсного узла, если его значение совпадает с установленным по умолчанию, потому что ограничение на количество передач для ассоциации sasoc_asocmaxrxt действует раньше, чем srto_max. Опять-таки, у нас есть два параметра влияющих на длительность тайм-аутов и эффективность обнаружения отказов. Мы можем уменьшить количество попыток, изменив значение sasoc_asocmaxrxt (по умолчанию 10), или снизить максимальное значение тайм-аута, изменив значение srto_max (по умолчанию 60 с). Если мы сделаем srto_max равным 10 с, время обнаружения отказа собеседника снизится на 12 с и станет равным 50 с. Альтернативой может быть уменьшение количества повторных передач до 8; при этом время обнаружения снизится до 30 с. Изложенные ранее соображения относятся и к этому сценарию: кратковременные неполадки в сети и перегрузка удаленной системы могут привести к обрыву работоспособного соединения.

Одну из множества альтернатив мы не рассматриваем в качестве рекомендуемой. Это снижение минимального тайм-аута (`srtomin`). При передаче данных через Интернет снижение этого значения приведет к неприятным последствиям: наш узел будет передавать повторные пакеты слишком часто, перегружая инфраструктуру Интернета. В частной сети снижение этого значения допустимо, но для большинства приложений в этом просто нет необходимости.

Для каждого приложения выбор конкретных значений параметров повторной передачи должен определяться несколькими факторами:

- Насколько быстро нужно приложению обнаруживать отказы?

■ Будет ли приложение выполнять в частных сетях, где условия передачи заранее известны и меняются не так резко, как в Интернете?

- Каковы последствия неправильного обнаружения отказа?

Только внимательно подумав над ответами на эти вопросы, программист может правильно настроить параметры тайм-аутов SCTP.

23.12. Когда SCTP оказывается предпочтительнее TCP

Изначально протокол SCTP разрабатывался для управления сигналами и реализации интернет-телефонии. Однако в процессе разработки область применения этого протокола значительно расширилась. Фактически он превратился в общесетевой транспортный протокол. SCTP поддерживает почти все функции TCP и значительно расширяет их новыми сервисами транспортного уровня. Маловероятно, чтобы сетевое приложение ничего не выиграло от перехода на SCTP. Так в каких же случаях следует использовать этот протокол? Начнем с перечисления его достоинств.

1. Протокол SCTP обеспечивает явную поддержку многоинтерфейсных узлов. Конечная точка может передавать данные по нескольким сетям для повышения надежности. Никаких особых действий, кроме перехода на SCTP, для использования новых сервисов SCTP предпринимать не требуется. Подробнее об SCTP для многоинтерфейсных узлов читайте в [117, раздел 7.4].

2. Протокол SCTP устраняет блокирование очереди. Приложение может передавать данные параллельно по нескольким потокам одной ассоциации. Потеря пакета в одном потоке не приведет к задержке передачи по другим потокам той же ассоциации (см. раздел 10.5 настоящей книги).

3. Границы сообщений уровня приложения сохраняются протоколом SCTP. Многие приложения не нуждаются в отправке потока байтов. Им удобнее работать с сообщениями. SCTP сохраняет границы сообщений и тем самым упрощает задачу программисту-разработчику, которому больше не приходится отмечать границы сообщений внутри потока байтов и писать специальные функции для реконструкции сообщений из этого потока.

4. SCTP предоставляет сервис неупорядоченной доставки. Некоторые приложения не нуждаются в сохранении порядка сообщений при передаче их по сети. Раньше такому приложению, использующему TCP для обеспечения надежности, приходилось мириться с задержками, вызванными блокированием очереди и необходимостью упорядоченной доставки (хотя само приложение в ней не нуждалось). SCTP предоставляет таким приложениям именно тот тип сервиса, который им нужен.

5. Некоторые реализации SCTP предоставляют сервис частичной надежности. Отправитель получает возможность указывать время жизни каждого сообщения в поле `sinfo_timetolive` структуры `sctp_sndrcvinfo`. (Это время жизни отличается от TTL IPv4 и ограничения на количество прыжков IPv6 тем, что оно на самом деле измеряется в единицах времени.) Если частичная надежность поддерживается обоими узлами, не доставленные вовремя данные могут сбрасываться транспортным уровнем, а не приложением, даже если они были переданы и утеряны. Таким образом оптимизируется передача данных в условиях загруженных линий.

6. Легкость перехода с TCP на SCTP обеспечивается сокетами типа «один-к-одному». Сокеты этого типа предоставляют типичный для TCP интерфейс, так что приложение может быть перенесено на новый протокол с самыми незначительными изменениями.

7. Многие функции TCP поддерживаются и SCTP: уведомление о приеме, повторная передача утерянных данных, сохранение последовательности данных, оконное управление передачей, медленное начало и алгоритмы предотвращения перегрузки линий, а также выборочные уведомления. Есть и два исключения: состояние неполного закрытия и срочные данные.

8. SCTP позволяет приложению настраивать транспортный уровень по своим потребностям, причем настройка выполняется для каждой ассоциации в отдельности. Эта гибкость в сочетании с универсальным

набором значений по умолчанию (для приложений, не нуждающихся в тонкой настройке транспортного уровня) дает приложению нечто большее, нежели оно могло получить при работе с TCP.

SCTP лишен двух особенностей TCP. Одной из них является состояние неполного (половинного) закрытия соединения. Это состояние возникает, когда приложение закрывает свой конец соединения, но разрешает собеседнику отправлять данные, а само принимает их (мы обсуждали это состояние в разделе 6.6). Приложение входит в это состояние для того, чтобы сообщить собеседнику, что отправка данных завершена. Приложения очень редко используют эту возможность, поэтому при разработке SCTP решено было не заботиться об ее поддержке. Приложениям, которым нужна эта функция, с переходом на SCTP придется изменять протокол уровня приложения, чтобы отправлять сигнал в потоке данных. В некоторых случаях изменения могут быть далеко не тривиальными.

SCTP не поддерживает и такую функцию TCP, как обработка внеочередных данных (*urgent data*). Для доставки срочных данных в SCTP можно использовать отдельный поток, однако это не позволяет в точности воспроизвести поведение TCP.

Для приложений, ориентированных на передачу потока байтов, переход на SCTP может оказаться невыгодным. К таким приложениям относятся telnet, rlogin, rsh и ssh. TCP сегментирует поток байтов на пакеты IP более эффективно, чем SCPT, который пытается сохранять границы сообщений, из-за чего могут получаться блоки, не помещающиеся целиком в IP-дейтаграммы и вызывающие избыточные накладные расходы на передачу.

В заключение следует сказать, что многим программистам стоит задуматься о переносе своих приложений на SCTP, когда этот протокол станет доступен на их Unix-платформе. Однако чтобы эффективно использовать специальные функции SCTP, нужно хорошо разбираться в них. Пока этот протокол не будет распространен повсеместно, вам может быть выгоднее не уходить от TCP.

23.13. Резюме

В этой главе мы изучили функцию автоматического закрытия ассоциации SCTP и исследовали, каким образом она может быть использована для ограничения неактивных соединений через сокет типа «один-ко-многим». Мы написали простую функцию, при помощи которой приложение может получать большие сообщения, используя механизм частичной доставки. Мы узнали, каким образом приложение может декодировать уведомления о событиях, происходящих на транспортном уровне. Мы достаточно коротко рассказали о том, как процесс может отправлять неупорядоченные данные, связывать сокет с подмножеством адресов, получать адреса собеседника и свои собственные, а также преобразовывать IP-адрес в идентификатор ассоциации.

Периодическая проверка соединения для ассоциаций SCTP включена по умолчанию. Мы научились управлять этой функцией посредством простой подпрограммы, которую сами же написали. Мы научились отделять ассоциацию при помощи системного вызова `sctp_peeloff` и написали параллельно-последовательный сервер, использующий эту возможность. Мы обсудили проблему настройки тайм-аутов повторной передачи SCTP, а также раскрыли преимущества и недостатки перехода на SCTP.

Упражнения

1. Напишите клиент для тестирования интерфейса частичной доставки из раздела 23.3.
2. Каким образом можно задействовать механизм частичной доставки, если не отправлять очень больших сообщений?
3. Перепишите сервер, использующий механизм частичной доставки, таким образом, чтобы он умел обрабатывать соответствующие уведомления.
4. Каким приложениям пригодится механизм передачи неупорядоченных данных? А каким он не нужен? Поясните.
5. Каким образом можно протестировать сервер, связывающийся с подмножеством IP-адресов узла?
6. Предположим, ваше приложение работает в частной сети, причем конечные точки находятся в одной локальной сети. Все серверы и клиенты являются многоинтерфейсными узлами. Каким образом следует настроить параметры повторной передачи, чтобы обнаруживать отказ узла не более, чем за 2 с?

Глава 24

Внеполосные данные

24.1. Введение

Ко многим транспортным уровням применима концепция *внеполосных данных* (*out-of-band data*), которые иногда называются *срочными данными* (*expedited data*). Суть этой концепции заключается в том, что если на одном конце соединения происходит какое-либо важное событие, то требуется быстро сообщить об этом собеседнику. В данном случае «быстро» означает, что сообщение должно быть послано прежде, чем будут посланы какие-либо обычные данные (называемые иногда *данными из полосы пропускания*), которые уже помещены в очередь для отправки, то есть внеполосные данные имеют более высокий приоритет, чем обычные данные. Для передачи внеполосных данных не создается новое соединение, а используется уже существующее.

К сожалению, когда мы переходим от общих концепций к реальной ситуации, почти в каждом транспортном протоколе имеется своя реализация внеполосных данных. В качестве крайнего примера можно привести UDP, где внеполосных данных нет вовсе. В этой главе мы уделим основное внимание модели внеполосных данных TCP. Мы приведем различные примеры обработки внеполосных данных в API сокетов и опишем, каким образом внеполосные данные используются приложениями Telnet, Rlogin и FTP. За пределами очерченного круга удаленных интерактивных приложений найти применение внеполосным данным довольно сложно.

24.2. Внеполосные данные протокола TCP

В протоколе TCP нет настоящих *внеполосных данных*. Вместо этого в TCP предусмотрен так называемый *срочный режим*^[4] (*urgent mode*), к рассмотрению которого мы сейчас и приступим. Предположим, процесс записал N байт данных в сокет протокола TCP, и эти данные образуют очередь в буфере отправки сокета и ожидают отправки собеседнику. Ситуацию иллюстрирует рис. 24.1. Байты данных пронумерованы от 1 до N .



Рис. 24.1. Буфер отправки сокета, содержащий данные для отправки

Теперь процесс отправляет один байт внеполосных данных, содержащий символ ASCII a, используя функцию send с флагом MSG_OOB:

```
send(fd, "a", 1, MSG_OOB);
```

TCP помещает данные в следующую свободную позицию буфера отправки сокета и устанавливает указатель на срочные данные (или просто *срочный указатель*^[5] — *urgent pointer*) для этого соединения на первую свободную позицию. Этот буфер показан на рис. 24.2, а байт, содержащий внеполосные данные, помечен буквами OOB.



Рис. 24.2. Буфер отправки сокета, в который добавлен один байт внеполосных данных

ПРИМЕЧАНИЕ

Срочный указатель TCP указывает на байт данных, который следует за последним байтом внеполосных данных (то есть данных, снабженных флагом MSG_OOB). В книге [111] на с. 292–296 говорится, что это исторически сложившаяся особенность, которая теперь эмулируется во

всех реализациях. Если посылающий и принимающий протоколы TCP одинаково интерпретируют срочный указатель TCP, беспокоиться не о чем.

Если состояние буфера таково, как показано на рис. 24.2, то в заголовке TCP следующего отправленного сегмента будет установлен флаг URG, а поле смещения срочных данных (или просто *поле срочного смещения* [9]) будет указывать на байт, следующий за байтом с внеполосными данными. Но этот сегмент может содержать байт, помеченный как ОOB, а может и не содержать его. Будет ли послан этот байт, зависит от количества предшествующих ему байтов в буфере отправки сокета, от размера сегмента, который TCP пересыпает собеседнику, и от текущего размера окна, объявленного собеседником.

Выше мы использовали термины «срочный указатель» (urgent pointer) и «срочное смещение» (urgent offset). На уровне TCP эти термины имеют различные значения. Величина, представленная 16 битами в заголовке TCP, называется срочным смещением и должна быть прибавлена к полю последовательного номера в заголовке TCP для получения 32-разрядного последовательного номера последнего байта срочных данных (то есть срочного указателя). TCP использует срочное смещение, только если в заголовке установлен другой бит, называемый флагом URG. Программисту можно не заботиться об этом различии и работать только со срочным указателем TCP.

Важная характеристика срочного режима TCP заключается в следующем: заголовок TCP указывает на то, что отправитель вошел в срочный режим (то есть флаг URG установлен вместе со срочным смещением), но фактической отправки байта данных, на который указывает срочный указатель, не требуется. Действительно, если поток данных TCP остановлен функциями управления потоком (когда буфер приема сокета получателя заполнен и TCP получателя объявил нулевое окно для отправляющего TCP), то срочное уведомление отправляется без каких-либо данных [128, с. 1016–1017], как показано в листингах 24.8 и 24.9. Это одна из причин, по которой в приложениях используется срочный режим TCP (то есть внеполосные данные): срочное уведомление всегда отсылается собеседнику, даже если поток данных остановлен функциями управления потоком TCP.

Что произойдет, если мы отправим несколько байтов внеполосных данных, как в следующем примере?

```
send(fd, "abc", 3, MSG_OOB);
```

В этом примере срочный указатель TCP указывает на байт, следующий за последним байтом, и таким образом, последний байт (c) считается байтом внеполосных данных.

Посмотрим теперь, как выглядит процесс отправки внеполосных данных с точки зрения принимающей стороны.

1. Когда TCP получает сегмент, в котором установлен флаг URG, срочный указатель проверяется для выяснения того, указывает ли он на *новые* внеполосные данные. Иначе говоря, проверяется, впервые ли этот конкретный байт передается в срочном режиме TCP. Дело в том, что часто отправляющий TCP посыпает несколько сегментов (обычно в течение короткого промежутка времени), содержащих флаг URG, в которых срочный указатель указывает на один и тот же байт данных. Только первый из этих сегментов фактически уведомляет принимающий процесс о прибытии новых внеполосных данных.

2. Принимающий процесс извещается о том, что прибыли новые внеполосные данные. Сначала владельцу сокета посыпается сигнал SIGURG. При этом предполагается, что для установления владельца сокета была вызвана функция fcntl или ioctl (см. табл. 7.9) и что для данного сигнала процессом был установлен обработчик сигнала. Затем, если процесс блокирован в вызове функции select, которая ждет возникновения исключительной ситуации для дескриптора сокета, происходит возврат из этой функции.

Эти два уведомления действуют в том случае, когда прибывает новый срочный указатель, вне зависимости от того, принят ли байт, на который он указывает.

В потоке данных может быть только одна отметка ОOB (один срочный указатель). Если новый срочный байт отправляется до того, как будет принят старый, последний просто сбрасывается и перестает быть срочным.

3. Когда байт данных, на который указывает срочный указатель, фактически прибывает на принимающий TCP, этот байт может быть помещен отдельно или оставлен вместе с другими данными. По умолчанию параметр сокета SO_OOBINLINE не установлен, поэтому внеполосный байт не размещается в приемном буфере сокета. Вместо этого содержащиеся в нем данные помещаются в отдельный внеполосный буфер размером в один байт, предназначенный специально для этого соединения [128, с. 986–988]. Для процесса единственным способом прочесть данные из этого специального однобайтового буфера является вызов функции recv, recvfrom или recvmsg с заданием флага MSG_OOB. Если новый срочный байт прибывает до того, как будет считан старый, новое значение записывается в буфер поверх прежнего.

Однако если процесс устанавливает параметр сокета `SO_OOBINLINE`, то байт данных, на который указывает срочный указатель TCP, остается в обычном буфере приема сокета. В этом случае процесс не может задать флаг `MSG_OOB` для считывания данных, содержащихся во внеполосном байте. Процесс сможет распознать этот байт, только когда дойдет до него и проверит *отметку внеполосных данных (out-of-band mark)* для данного соединения, как показано в разделе 24.3. Возможны следующие ошибки:

1. Если процесс запрашивает внеполосные данные (то есть устанавливает флаг `MSG_OOB`), но собеседник таких данных не послал, возвращается `EINVAL`.
2. Если процесс был уведомлен о том, что собеседник послал содержащий внеполосные данные байт (например, с помощью функции `select` или сигнала `SIGURG`), и пытается считать эти данные, когда указанный байт еще не прибыл, возвращается ошибка `EWOULDBLOCK`. В такой ситуации все, что может сделать процесс, — это считать данные из приемного буфера сокета (возможно, сбрасывая данные, если отсутствует свободное место для их хранения), чтобы освободить место в буфере для приема байта внеполосных данных, посыпаемых собеседником.
3. Если процесс пытается считать одни и те же внеполосные данные несколько раз, возвращается ошибка `EINVAL`.
4. Если процесс установил параметр сокета `SO_OOBINLINE`, а затем пытается считать внеполосные данные, задавая флаг `MSG_OOB`, возвращается `EINVAL`.

Простой пример использования сигнала SIGURG

Теперь мы рассмотрим тривиальный пример отправки и получения внеполосных данных. В листинге 24.1^[1] показана программа отправки этих данных.

Листинг 24.1. Простая программа отправки внеполосных данных

```
//oob/tcpsend01.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;

6     if (argc != 3)
7         err_quit("usage: tcpsend01 <host> <port#>");
8     sockfd = Tcp_connect(argv[1], argv[2]);
9     Write(sockfd, "123", 3);
10    printf("wrote 3 bytes of normal data\n");
11    sleep(1);

12    Send(sockfd, "4", 1, MSG_OOB);
13    printf("wrote 1 byte of OOB data\n");
14    sleep(1);

15    Write(sockfd, "56", 2);
16    printf("wrote 2 bytes of normal data\n");
17    sleep(1);

18    Send(sockfd, "7", 1, MSG_OOB);
19    printf("wrote 1 byte of OOB data\n");
20    sleep(1);

21    Write(sockfd, "89", 2);
22    printf("wrote 2 bytes of normal data\n");
23    sleep(1);

24    exit(0);
```

```
25 }
```

Отправлены 9 байт, промежуток между операциями по отправке установлен с помощью функции sleep равным одной секунде. Назначение этого промежутка в том, чтобы данные каждой из функций write или send были переданы и получены на другом конце как отдельный сегмент TCP. Несколько позже мы обсудим некоторые вопросы согласования во времени при пересылке внеполосных данных. После выполнения данной программы мы видим вполне предсказуемый результат:

```
macosx % tcpsend01 freebsd 9999
```

```
wrote 3 bytes of normal data
wrote 1 byte of OOB data
wrote 2 bytes of normal data
wrote 1 byte of OOB data
wrote 2 bytes of normal data
```

В листинге 24.2 показана принимающая программа.

Листинг 24.2. Простая программа для получения внеполосных данных

```
//oob/tcprecv01.c
1 #include "unp.h"

2 int listenfd, connfd;

3 void sig_urg(int);

4 int
5 main(int argc, char **argv)
6 {
7     int n;
8     char buff[100];

9     if (argc == 2)
10     listenfd = Tcp_listen(NULL, argv[1], NULL);
11 else if (argc == 3)
12     listenfd = Tcp_listen(argv[1], argv[2], NULL);
13 else
14     err_quit("usage: tcprecv01 [ <host> ] <port#>");

15 connfd = Accept(listenfd, NULL, NULL);

16 Signal(SIGURG, sig_urg);
17 Fcntl(connfd, F_SETOWN, getpid());

18 for (;;) {
19     if ((n = Read(connfd, buff, sizeof(buff) - 1)) == 0) {
20         printf("received EOF\n");
21         exit(0);
22     }
23     buff[n] = 0; /* завершающий нуль */
24     printf("read bytes: %s\n", n, buff);
25 }
26 }

27 void
28 sig_urg(int signo)
29 {
30     int n;
31     char buff[100];

32     printf("SIGURG received\n");
```

```
33 n = Recv(connfd, buff, sizeof(buff) - 1, MSG_OOB);
34 buff[n] = 0; /* завершающий нуль */
35 printf("read OOB byte: %s\n", n, buff);
36 }
```

Установка обработчика сигнала и владельца сокета

16-17 Устанавливается обработчик сигнала SIGURG и функция fcntl задает владельца сокета для данного соединения.

ПРИМЕЧАНИЕ

Обратите внимание, что мы не задаем обработчик сигнала, пока не завершается функция accept. Существует небольшая вероятность того, что внеполосные данные могут прибыть после того, как TCP завершил трехэтапное рукопожатие, но до завершения функции accept. Внеполосные данные мы в этом случае потеряем. Допустим, что мы установили обработчик сигнала перед вызовом функции accept, а также задали владельца прослушиваемого сокета (который затем стал бы владельцем присоединенного сокета). Тогда, если внеполосные данные прибудут до завершения функции accept, наш обработчик сигналов еще не получит значения для дескриптора connfd. Если данный сценарий важен для приложения, следует инициализировать connfd, «вручную» присвоив этому дескриптору значение -1, добавить в обработчик проверку равенства connfd == -1 и при истинности этого условия просто установить флаг, который будет проверяться в главном цикле после вызова accept. За счет этого главный цикл сможет узнать о поступлении внеполосных данных и считать их. Можно заблокировать сигнал на время вызова accept, но при этом программа будет страдать от всех возможных ситуаций гонок, описанных в разделе 20.5.

18-25 Процесс считывает данные из сокета и выводит каждую строку, которая возвращается функцией read. После того как отправитель разрывает соединение, то же самое делает и получатель.

Обработчик сигнала SIGURG

27-36 Наш обработчик сигнала вызывает функцию printf, считывает внеполосные данные, устанавливая флаг MSG_OOB, а затем выводит полученные данные. Обратите внимание, что при вызове функции recv мы запрашиваем до 100 байт, но, как мы вскоре увидим, всегда возвращается только один байт внеполосных данных.

ПРИМЕЧАНИЕ

Как сказано ранее, вызов ненадежной функции printf из обработчика сигнала не рекомендуется. Мы делаем это просто для того, чтобы увидеть, что произойдет с нашей программой.

Ниже приведен результат, который получается, когда мы запускаем эту программу, а затем — программу для отправки внеполосных данных, приведенную в листинге 24.1.

```
freebsd % tcprecv01 9999
read 3 bytes: 123
SIGURG received
read 1 OOB byte: 4
read 2 bytes: 56
SIGURG received
read 1 OOB byte: 7
```

```
read 2 bytes: 89
received EOF
```

Результаты оказались такими, как мы и ожидали. Каждый раз, когда отправитель посыпает внеполосные данные, для получателя генерируется сигнал SIGURG, после чего получатель считывает один байт, содержащий внеполосные данные.

Простой пример использования функции select

Теперь мы переделаем код нашего получателя внеполосных данных и вместо сигнала SIGURG будем использовать функцию `select`. В листинге 24.3 показана принимающая программа.

Листинг 24.3. Принимающая программа, в которой (ошибочно) используется функция `select` для уведомления о получении внеполосных данных

```
//oob/tcprecv02.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int listenfd, connfd, n;
6     char buff[100];
7     fd_set rset, xset;

8     if (argc == 2)
9         listenfd = Tcp_listen(NULL, argv[1], NULL);
10    else if (argc ==3)
11        listenfd = Tcp_listen(argv[1], argv[2], NULL);
12    else
13        err_quit("usage: tcprecv02 [ <host> ] <port#>");

14    connfd = Accept(listenfd, NULL, NULL);

15    FD_ZERO(&rset);
16    FD_ZERO(&xset);
17    for (;;) {
18        FD_SET(connfd, &rset);
19        FD_SET(connfd, &xset);

20        Select(connfd + 1, &rset, NULL, &xset, NULL);

21        if (FD_ISSET(connfd, &xset)) {
22            n = Recv(connfd, buff, sizeof(buff) - 1, MSG_OOB);
23            buff[n] =0; /* завершающий нуль */
24            printf("read OOB byte: %s\n", n, buff);
25        }
26        if (FD_ISSET(connfd, &rset)) {
27            if ((n = Read(connfd, buff, sizeof(buff) - 1)) == 0) {
28                printf("received EOF\n");
29                exit(0);
30            }
31            buff[n] = 0; /* завершающий нуль */
32            printf("read bytes: %s\n", n, buff);
33        }
34    }
35 }
```

15-20 Процесс вызывает функцию `select`, которая ожидает получения либо обычных данных (набор дескрипторов для чтения, `rset`), либо внеполосных (набор дескрипторов для обработки исключений, `xset`). В обоих случаях полученные данные выводятся.

Если мы запустим эту программу, а затем — программу для отправки, которая приведена в листинге 24.1, то столкнемся со следующей ошибкой:

```
freebsd4 % tcprecv02 9999
read 3 bytes: 123
read 1 00B byte: 4
recv error: Invalid argument
```

Проблема заключается в том, что функция `select` будет сообщать об исключительной ситуации, пока процесс не считает данные, находящиеся за отметкой внеполосных данных (то есть после них [128, с. 530-531]). Мы не можем считывать внеполосные данные больше одного раза, так как после первого же их считывания ядро очищает буфер, содержащий один байт внеполосных данных. Когда мы вызываем функцию `recv`, устанавливая флаг `MSG_OOB` во второй раз, она возвращает ошибку `EINVAL`.

Чтобы решить эту проблему, нужно вызывать функцию `select` для проверки на наличие исключительной ситуации только после того, как будут приняты все обычные данные. В листинге 24.4 показана модифицированная версия принимающей программы из листинга 24.3. В этой версии описанный сценарий обрабатывается корректно.

Листинг 24.4. Модификация программы, приведенной в листинге 24.3. Функция `select` применяется для проверки исключительной ситуации корректным образом

```
//oob/tcprecv03.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int listenfd, connfd, n, justreadoob = 0;
6     char buff[100];
7     fd_set rset, xset;

8     if (argc == 2)
9         listenfd = Tcp_listen(NULL, argv[1], NULL);
10    else if (argc == 3)
11        listenfd = Tcp_1listen(argv[1], argv[2], NULL);
12    else
13        err_quit("usage: tcprecv03 [ <host> ] <port#>");

14    connfd = Accept(listenfd, NULL, NULL);

15    FD_ZERO(&rset);
16    FD_ZERO(&xset);
17    for (;;) {
18        FD_SET(connfd, &rset);
19        if (justreadoob == 0)
20            FD_SET(connfd, &xset);
21        Select(connfd + 1, &rset, NULL, &xset, NULL);

22        if (FD_ISSET(connfd, &xset)) {
23            n = Recv(connfd, buff, sizeof(buff) - 1, MSG_OOB);
24            buff[n] = 0; /* завершающий нуль */
25            printf("read %d 00B byte: %s\n", n, buff);
26            justreadoob = 1;
27            FD_CLR(connfd, &xset);
28        }
29        if (FD_ISSET(connfd, &rset)) {
30            if ((n = Read(connfd, buff, sizeof(buff) - 1)) == 0) {
```

```

31     printf("received EOF\n");
32     exit(0);
33 }
34 buff[n] = 0; /* завершающий нуль */
35 printf("read %d bytes: %s\n", n, buff);
36 justreadoob = 0;
37 }
38 }
39 }
```

5 Мы объявляем новую переменную с именем `justreadoob`, которая указывает, какие данные мы считываем — внеполосные или обычные. Этот флаг определяет, нужно ли вызывать функцию `select` для проверки на наличие исключительной ситуации.

26-27 Когда мы устанавливаем флаг `justreadoob`, мы также должны выключить бит соответствующего дескриптора в наборе для проверки исключительных ситуаций.

Теперь программа работает так, как мы ожидали.

24.3. Функция `sockatmark`

С приемом внеполосных данных всегда связана так называемая *отметка внеполосных данных (out-of-bandmark)*. Это позиция в потоке обычных данных *на стороне отправителя*, соответствующая тому моменту; когда посылающий процесс отправляет байт, содержащий внеполосные данные. Считывая данные из сокета, принимающий процесс путем вызова функции `sockatmark` определяет, находится ли он в данный момент на этой отметке.

```
#include <sys/socket.h>
```

```
int sockatmark(int sockfd);
```

Возвращает: 1, если находится на отметке внеполосных данных; 0, если не на отметке; -1 в случае ошибки

ПРИМЕЧАНИЕ

Эта функция появилась в POSIX. Разработчики стандарта POSIX стремятся заменить отдельными функциями все вызовы `ioctl` с различными параметрами.

В листинге 24.5 показана реализация этой функции с помощью поддерживаемого в большинстве систем параметра `SIOCATMARK` функции `ioctl`.

Листинг 24.5. Функция `sockatmark` реализована с использованием функции `ioctl`

```
//lib/sockatmark.c
1 #include "unp.h"

2 int
3 sockatmark(int fd)
4 {
5     int flag;

6     if (ioctl(fd, SIOCATMARK, &flag) < 0)
7         return (-1);
8     return (flag != 0 ? 1 : 0);
9 }
```

Отметка внеполосных данных применима независимо от того, как принимающий процесс получает внеполосные данные: вместе с обычными данными (параметр сокета `SO_OOBINLINE`) или отдельно (флаг `MSG_OOB`). Отметка внеполосных данных часто используется для того, чтобы принимающий процесс мог интерпретировать получаемые данные специальным образом до тех пор, пока он не дойдет до этой отметки.

Пример: особенности отметки внеполосных данных

Далее мы приводим простой пример, иллюстрирующий следующие две особенности отметки внеполосных данных:

1. Отметка внеполосных данных всегда указывает на один байт дальше конечного байта обычных данных. Это означает, что, когда внеполосные данные получены вместе с обычными, функция `sockatmark` возвращает 1, если следующий считываемый байт был послан с флагом `MSG_OOB`. Если параметр `SO_OOBINLINE` не включен (состояние по умолчанию), то функция `sockatmark` возвращает 1, когда следующий байт данных является первым байтом, посланным следом за внеполосными данными.

2. Операция считывания всегда останавливается на отметке внеполосных данных [128, с. 519–520]. Это означает, что если в приемном буфере сокета 100 байт, но только 5 из них расположены перед отметкой внеполосных данных, то когда процесс выполнит функцию `read`, запрашивая 100 байт, возвратится только 5 байт, расположенные до этой отметки. Эта вынужденная остановка на отметке позволяет процессу вызвать функцию `sockatmark`, которая определит, находится ли указатель буфера на отметке внеполосных данных.

В листинге 24.6 показана наша программа отправки. Она посыпает три байта обычных данных, один байт внеполосных данных, а затем еще один байт обычных данных. Паузы между этими операциями отсутствуют.

В листинге 24.7 показана принимающая программа. В ней не используется ни функция `select`, ни сигнал `SIGURG`. Вместо этого в ней вызывается функция `sokatmark`, определяющая положение байта внеполосных данных.

Листинг 24.6. Программа отправки

```
//oob/tcpse04.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;

6     if (argc != 3)
7         err_quit("usage: tcpsend04 <host> <port#>");

8     sockfd = Tcp_connect(argv[1], argv[2]);

9     Write(sockfd, "123", 3);
10    printf("wrote 3 bytes of normal data\n");

11    Send(sockfd, "4", 1, MSG_OOB);
12    printf("wrote 1 byte of OOB data\n");

13    Write(sockfd, "5", 1);
14    printf("wrote 1 byte of normal data\n");

15    exit(0);
16 }
```

Листинг 24.7. Принимающая программа, в которой вызывается функция `sokatmark`

```
//oob/tcprecv04.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int listenfd, connfd, n, on = 1;
6     char buff[100];

7     if (argc == 2)
```

```

8   listenfd = Tcp_listen(NULL, argv[1], NULL);
9   else if (argc == 3)
10  listenfd = Tcp_listen(argv[1], argv[2], NULL);
11 else
12   err_quit("usage- tcprecv04 [ <host> ] <port#>");
13 Setsockopt(listenfd, SOL_SOCKET, SO_OOBINLINE, &on, sizeof(on));
14 connfd = Accept(listenfd, NULL, NULL);
15 sleep(5);

16 for (;;) {
17   if (Sockatmark(connfd))
18     printf("at OOB mark\n");

19   if ((n = Read(connfd, buff, sizeof(buff) - 1)) == 0) {
20     printf("received EOF\n");
21     exit(0);
22   }
23   buff[n] = 0; /* завершающий нуль */
24   printf("read %d bytes: %s\n", n, buff);
25 }
26 }
```

Включение параметра сокета SO_OOBINLINE

13 Мы хотим принимать внеполосные данные вместе с обычными данными, поэтому нам нужно включить параметр `SO_OOBINLINE`. Но если мы будем ждать, когда выполнится функция `accept` и установит этот параметр для присоединенного сокета, трехэтапное рукопожатие завершится и внеполосные данные могут уже прибыть. Поэтому нам нужно установить этот параметр еще для прослушиваемого сокета, помня о том, что все параметры прослушиваемого сокета наследуются присоединенным сокетом (см. раздел 7.4).

Вызов функции `sleep` после вызова функции `accept`

14-15 После того как выполнена функция `accept`, получатель переходит в спящее состояние, что позволяет получить все данные, посланные отправителем. Это позволяет нам продемонстрировать, что функция `read` останавливается на отметке внеполосных данных, даже если в приемном буфере сокета имеются дополнительные данные.

Считывание всех отправленных данных

16-25 В программе имеется цикл, в котором вызывается функция `read` и выводятся полученные данные. Но перед вызовом функции `read` функция `sockatmark` проверяет, находится ли указатель буфера на отметке внеполосных данных.

После выполнения этой программы мы получаем следующий результат:

```
freebsd4 % tcprecv04 6666
read 3 bytes: 123
at OOB mark
read 2 bytes: 45
received EOF
```

Хотя принимающий TCP получил все посланные данные, первый вызов функции `read` возвращает только три байта, так как была обнаружена отметка внеполосных данных. Следующий считанный байт —

это байт, содержащий внеполосные данные (его значение равно 4), так как мы дали ядру указание поместить внеполосные данные вместе с обычными.

Пример: дополнительные свойства внеполосных данных

Теперь мы покажем другой столь же простой пример, иллюстрирующий две дополнительные особенности внеполосных данных, о которых мы уже упоминали ранее.

1. TCP посыпает уведомление об отправке внеполосных данных (их срочный указатель), даже если поток данных остановлен функциями управления потоком.

2. Принимающий процесс может получить уведомление о том, что отправитель отоспал внеполосные данные (с помощью сигнала SIGURG или функции select) до того, как эти данные фактически прибудут. Если после получения этого уведомления процесс вызывает функцию recv, задавая флаг MSG_OOB, а внеполосные данные еще не прибыли, то будет возвращена ошибка EWOULDBLOCK.

В листинге 24.8 приведена программа отправки.

Листинг 24.8. Программа отправки

```
//oob/tcpsend05.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd, size;
6     char buff[16384];

7     if (argc != 3)
8         err_quit("usage: tcpsend04 <host> <port#>");

9     sockfd = Tcp_connect(argv[1], argv[2]);

10    size = 32768;
11    Setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &size, sizeof(size));

12    Write(sockfd, buff, 16384);
13    printf("wrote 16384 bytes of normal data\n");
14    sleep(5);

15    Send(sockfd, "a", 1, MSG_OOB);
16    printf("wrote 1 byte of OOB data\n");

17    Write(sockfd, buff, 1024);
18    printf("wrote 1024 bytes of normal data\n");

19    exit(0);
20 }
```

9-19 Этот процесс устанавливает размер буфера отправки сокета равным 32 768 байт, записывает 16 384 байт обычных данных, а затем на 5 с переходит в спящее состояние. Чуть ниже мы увидим, что приемник устанавливает размер приемного буфера сокета равным 4096 байт, поэтому данные, отправленные отсылающим TCP, с гарантией заполнят приемный буфер сокета получателя. Затем отправитель посыпает один байт внеполосных данных, за которым следуют 1024 байт обычных данных, и, наконец, закрывает соединение.

В листинге 24.9 представлена принимающая программа.

Листинг 24.9. Принимающая программа

```
//oob/tcprecv05.c
1 #include "unp.h"
```

```

2 int listenfd, connfd;

3 void sig_urg(int);

4 int
5 main(int argc, char **argv)
6 {
7     int size;

8     if (argc == 2)
9         listenfd = Tcp_listen(NULL, argv[1], NULL);
10    else if (argc == 3)
11        listenfd = Tcp_listen(argv[1], argv[2], NULL);
12    else
13        err_quit("usage: tcprecv05 [ <host> ] <port#>");

14    size = 4096;
15    Setsockopt(listenfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));

16    connfd = Accept(listenfd, NULL, NULL);

17    Signal(SIGURG, sig_urg);
18    Fcntl(connfd, F_SETOWN, getpid());

19    for (;;)
20        pause();
21 }

22 void
23 sig_urg(int signo)
24 {
25     int n;
26     char buff[2048];

27     printf("SIGURG received\n");
28     n = Recv(connfd, buff, sizeof(buff) - 1, MSG_OOB);
29     buff[n] = 0; /* завершающий пустой байт */
30     printf("read %d OOB byte\n", n);
31 }

```

14-20 Принимающий процесс устанавливает размер приемного буфера сокета приемника равным 4096 байт. Этот размер наследуется присоединенным сокетом после установления соединения. Затем процесс вызывает функцию accept, задает обработчик для сигнала SIGURG и задает владельца сокета. В основном цикле (бесконечном) вызывается функция pause.

22-31 Обработчик сигнала вызывает функцию recv для считывания внеполосных данных.

Если мы запускаем сначала принимающую программу, а затем программу отправки, то получаем следующий результат выполнения программы отправки:

```

macosx % tcpsend05 freebsd 5555
wrote 16384 bytes of normal data
wrote 1 byte of OOB data
wrote 1024 bytes of normal data

```

Как и ожидалось, все данные помещаются в буфер отправки сокета отправителя, и программа завершается. Ниже приведен результат работы принимающей программы:

```

freebsd4 % tcprecv05 5555
SIGURG received
recv error: Resource temporarily unavailable

```

Сообщение об ошибке, которое выдает наша функция `err_sys`, соответствует ошибке `EAGAIN`, которая в FreeBSD аналогична ошибке `EWOULDBLOCK`. TCP посыпает уведомление об отправке внеполосных данных принимающему TCP, который в результате генерирует сигнал `SIGURG` для принимающего процесса. Но когда вызывается функция `recv` и задается флаг `MSG_OOB`, байт с внеполосными данными не может быть прочитан.

Для решения этой проблемы необходимо, чтобы получатель освобождал место в своем приемном буфере, считывая поступившие обычные данные. В результате TCP объявит для отправителя окно ненулевого размера, что в конечном счете позволит отправителю передать байт, содержащий внеполосные данные.

ПРИМЕЧАНИЕ

В реализациях, происходящих от Беркли [128, с. 1016-1017], можно отметить две близких проблемы. Во-первых, даже если приемный буфер сокета заполнен, ядро всегда принимает от процесса внеполосные данные для отправки собеседнику. Во-вторых, когда отправитель посыпает байт с внеполосными данными, немедленно посыпается сегмент TCP, содержащий срочное уведомление. Все обычные проверки вывода TCP (алгоритм Нагла, предотвращение синдрома «глупого окна») при этом блокируются.

Пример: единственность отметки внеполосных данных в TCP

Нашим очередным примером мы иллюстрируем тот факт, что для данного соединения TCP существует всего одна отметка внеполосных данных, и если новые внеполосные данные прибудут прежде, чем принимающий процесс начнет считывать пришедшие ранее внеполосные данные, то предыдущая отметка будет утеряна.

В листинге 24.10 показана посылающая программа, аналогичная программе, приведенной в листинге 24.6. Отличие заключается в том, что сейчас мы добавили еще одну функцию `send` для отправки внеполосных данных и еще одну функцию `write` для записи обычных данных.

Листинг 24.10. Отправка двух байтов внеполосных данных друг за другом

```
//oob/tcpsend06.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;

6     if (argc != 3)
7         err_quit("usage: tcpsend04 <host> <port#>");

8     sockfd = Tcp_connect(argv[1], argv[2]);

9     Write(sockfd, "123", 3);
10    printf("wrote 3 bytes of normal data\n");

11    Send(sockfd, "4", 1, MSG_OOB);
12    printf("wrote 1 byte of OOB data\n");

13    Write(sockfd, "5", 1);
14    printf("wrote 1 byte of normal data\n");

15    Send(sockfd, "6", 1, MSG_OOB);
16    printf("wrote 1 byte of OOB data\n");
```

```

17 Write(sockfd, "7", 1);
18 printf("wrote 1 byte of normal data\n");

19 exit(0);
20 }

```

В данном случае отправка данных происходит без пауз, что позволяет быстро переслать данные собеседнику.

Принимающая программа идентична программе, приведенной в листинге 24.7, где вызывается функция `sleep`, которая после установления соединения переводит получателя в спящее состояние на 5 с, чтобы позволить данным прибыть на принимающий TCP. Ниже приводится результат выполнения этой программы:

```

freebsd4 % tcprecv06 5555
read 5 bytes: 12345
at 00B mark
read 2 bytes: 67
received EOF

```

Прибытие второго байта внеполосных данных (6) изменяет отметку, которая ассоциировалась с первым прибывшим байтом внеполосных данных (4). Как мы сказали, для конкретного соединения TCP допускается только одна отметка внеполосных данных.

24.4. Резюме по теме внеполосных данных TCP

Все приведенные до сих пор примеры, иллюстрирующие использование внеполосных данных, были весьма тривиальны. К сожалению, когда мы начинаем учитывать возможные проблемы, связанные с согласованием во времени при пересылке внеполосных данных, ситуация заметно усложняется. В первую очередь, нужно осознать, что концепция внеполосных данных подразумевает передачу получателю трех различных фрагментов информации:

1. Сам факт того, что отправитель вошел в срочный режим. Принимающий процесс получает уведомление об этом либо с помощью сигнала `SIGURG`, либо с помощью функции `select`. Это **уведомление** передается сразу же после того, как отправитель посыпает байт внеполосных данных, поскольку, как показано в листинге 24.9, TCP посыпает уведомление, даже если поток каких-либо данных от сервера к клиенту остановлен функциями управления потоком. В результате получения такого уведомления получатель может входить в определенный специальный режим обработки последующих данных.

2. *Позиция* байта, содержащего внеполосные данные, то есть расположение этого байта по отношению к остальным данным, посланным отправителем, иначе говоря, *отметка* внеполосных данных.

3. *Фактическое значение* внеполосного байта. Поскольку TCP является потоковым протоколом, который не интерпретирует данные, посланные приложением, это может быть любое 8-разрядное значение.

Говоря о срочном режиме TCP, мы можем рассматривать флаг `URG` как уведомление, а срочный указатель как внеполосную отметку.

Проблемы, связанные с концепцией внеполосных данных, сформулированы в следующих пунктах:

1. Для каждого соединения имеется только один срочный указатель.

2. Для каждого соединения допускается только одна отметка внеполосных данных.

3. Для каждого соединения имеется только один однобайтовый буфер, предназначенный для внеполосных данных (это имеет значение, только если внеполосные данные не считаются вместе с обычными данными).

В листинге 24.10 показано, что вновь прибывающая отметка внеполосных данных отменяет все предыдущие отметки, до которых принимающий процесс еще не дошел. Если внеполосные данныечитываются вместе с обычными данными, то в случае прибытия новых внеполосных данных предыдущие не теряются, но теряются их отметки.

Типичный пример использования внеполосных данных — протокол Rlogin, действующий эту концепцию в ситуации, когда клиент прерывает программу, выполняемую на стороне сервера [111, с. 393–394]. Сервер должен сообщить клиенту, что нужно сбросить все данные, принятые от сервера, буферизованные и предназначенные для вывода на терминал. Сервер посыпает клиенту специальный байт внеполосных данных, указывая тем самым, что необходимо сбросить все полученные данные. Когда клиент получает сигнал `SIGURG`, он просто считывает данные из сокета, пока не встречает отметку внеполосных данных, после чего он сбрасывает все данные вплоть до этой отметки. (В [111, с. 398–401]

показан пример подобного использования внеполосных данных вместе с выводом программы `tcpdump`.) Если в этом сценарии сервер посыпает несколько внеполосных байтов, следующих с небольшими промежутками друг за другом, то такая последовательность не оказывает влияния на клиента, поскольку тот просто сбрасывает все данные, расположенные до последней отметки внеполосных данных.

В итоге можно сказать, что польза применения внеполосных данных зависит от того, для каких целей они служат в приложении. Если их назначение в том, чтобы сообщить собеседнику о необходимости сбросить все обычные данные, расположенные до отметки, то утрата промежуточных внеполосных данных и их отметок не повлечет никаких последствий. Но если потеря внеполосных данных недопустима, то эти данные следует получать вместе с обычными данными. Более того, байты, посланные как внеполосные данные, требуется каким-то образом отличать от обычных данных, так как промежуточные отметки могут быть перезаписаны при получении новых внеполосных данных. Telnet, например, посыпает свои собственные команды в потоке обычных данных между клиентом и сервером, но ставит перед этими командами байт, содержащий 255 (поэтому для отправки этого значения требуется послать последовательно два байта, содержащих 255). Эти байты позволяют отличить команды сервера от обычных пользовательских данных, но при этом для обнаружения команд сервера требуется, чтобы клиент и сервер обрабатывали каждый байт данных.

24.5. Резюме

В TCP не существует настоящих внеполосных данных. Вместо этого при переходе отправителя в срочный режим собеседнику отсылается в TCP-заголовке срочный указатель. Получение этого указателя на другом конце соединения служит уведомлением для процесса о том, что отправитель вошел в срочный режим, а указатель указывает на последний байт внеполосных (срочных) данных. Но эти данные отсылаются через то же соединение и подчиняются обычным функциям управления потоком данных TCP.

В API сокетов срочный режим TCP сопоставляется внеполосным данным. Отправитель входит в срочный режим, задавая флаг `MSG_OOB` при вызове функции `send`. Последний байт данных, переданных с помощью этой функции, считается внеполосным байтом. Приемник получает уведомление о том, что его TCP получил новый срочный указатель. Это происходит либо с помощью сигнала `SIGURG`, либо с помощью функции `select`, которая указывает, что на сокете возникла исключительная ситуация. По умолчанию TCP извлекает байт с внеполосными данными и помещает его в специальный однобайтовый буфер для внеполосных данных, откуда принимающий процесс считывает его с помощью вызова функции `recv` с флагом `MSG_OOB`. Имеется другой вариант — получатель может включить параметр сокета `SO_OOBINLINE`, и тогда внеполосный байт остается в потоке обычных данных. Независимо от того, какой метод используется принимающей стороной, уровень сокета поддерживает отметку внеполосных данных в потоке данных, и операция считывания остановится, когда дойдет до этой отметки. Чтобы определить, достигнута ли эта отметка, принимающий процесс использует функцию `sockatmark`.

Внеполосные данные применяются не очень широко. Они используются в протоколах Telnet и Rlogin, а также FTP. Во всех случаях внеполосные данные уведомляют собеседника об исключительной ситуации (например, прерывании на стороне клиента), после чего собеседник сбрасывает все принятые данные до отметки внеполосных данных.

Упражнения

1. Есть ли разница между одним вызовом функции
`send(fd, "ab", 2, MSG_OOB);`
и двумя последовательными вызовами
`send(fd, "a", 1, MSG_OOB);`
`send(fd, "b", 1, MSG_OOB);`
?
2. Переделайте программу, приведенную в листинге 24.4, так, чтобы использовать функцию `poll` вместо функции `select`.

Глава 25

Управляемый сигналом ввод-вывод

25.1. Введение

Ввод-вывод, управляемый сигналом, подразумевает, что мы указываем ядру проинформировать нас сигналом, если что-либо произойдет с дескриптором. Исторически такой ввод-вывод называли **асинхронным вводом-выводом**, но в действительности описанный далее управляемый сигналом ввод-вывод асинхронным не является. Последний обычно определяется как операция ввода-вывода с немедленным возвратом управления процессу после инициирования операции в ядре. Процесс продолжает выполняться во время того, как производится ввод-вывод. Когда операция ввода-вывода завершается или обнаруживается некоторая ошибка, процесс некоторым образом оповещается. В разделе 6.2 проводилось сравнение всех возможных типов ввода-вывода и было показано различие между вводом-выводом, управляемым сигналом, и асинхронным вводом-выводом.

Следует отметить, что неблокируемый ввод-вывод, описанный в главе 16, также не является асинхронным. При неблокируемом вводе-выводе ядро не возвращает управление после инициирования операции ввода-вывода. Управление возвращается немедленно, только если операция не может быть выполнена без блокирования процесса.

ПРИМЕЧАНИЕ

Стандарт POSIX обеспечивает истинный асинхронный ввод-вывод с помощью функций aio_XXX. Эти функции позволяют процессу решить, генерировать ли при завершении ввода-вывода сигнал, и какой именно.

Беркли-реализации поддерживают ввод-вывод, управляемый сигналом, для сокетов и устройств вывода с помощью сигнала SIGIO. SVR4 поддерживает ввод-вывод, управляемый сигналом, для устройств STREAMS с помощью сигнала SIGPOLL, который в данном случае приравнивается к SIGIO.

25.2. Управляемый сигналом ввод-вывод для сокетов

Для использования управляемого сигналом ввода-вывода с сокетом (SIGIO) необходимо, чтобы процесс выполнил три следующих действия:

1. Установил обработчик сигнала SIGIO.
2. Задал владельца сокета. Обычно это выполняется с помощью команды F_SETOWN функции fcntl (см. табл. 7.9).
3. Разрешил управляемый сигналом ввод-вывод для данного сокета, что обычно выполняется с помощью команды F_SETFL функции fcntl или путем включения флага O_ASYNC (см. табл. 7.9).

ПРИМЕЧАНИЕ

Флаг O_ASYNC был добавлен в POSIX относительно поздно. Его поддержка пока реализована в небольшом количестве систем. Для разрешения управляемого сигналом ввода-вывода в листинге 25.2 вместо этого флага мы используем функцию ioctl с флагом FIOASYNC. Следует отметить, что разработчики POSIX выбрали не самое удачное имя для нового флага: ему больше подходит имя O_SIGIO.

Обработчик сигнала должен быть установлен до того, как будет задан владелец сокета. В Беркли-реализациях порядок вызова этих функций не имеет значения, поскольку по умолчанию сигнал SIGIO игнорируется. Поэтому если изменить порядок вызова функций на противоположный, появится небольшая вероятность того, что сигнал будет сгенерирован после вызова функции fcntl, но перед вызовом функции signal. Однако если это произойдет, то сигнал просто не будет учитываться. В SVR4 SIGIO определяется в заголовочном файле <sys/signal.h>

как SIGPOLL, а действием по умолчанию для SIGPOLL является прерывание процесса. Таким образом, в SVR4 желательно быть уверенным в том, что обработчик сигнала установлен до задания владельца сокета.

Перевести сокет в режим ввода-вывода, управляемого сигналом, несложно. Сложнее определить условия, которые должны приводить к генерации сигнала SIGIO для владельца сокета. Это зависит от транспортного протокола.

Сигнал SIGIO и сокеты UDP

Использовать ввод-вывод, управляемый сигналом, с сокетами UDP довольно легко. Сигнал генерируется в следующих случаях:

- на сокет прибывает дейтаграмма;
- на сокете возникает асинхронная ошибка.

Таким образом, когда мы перехватываем сигнал SIGIO для сокета UDP, вызывается функция `recvfrom` как для чтения дейтаграммы, так и для получения асинхронной ошибки. Асинхронные ошибки, касающиеся UDP-сокетов, обсуждались в разделе 8.9. Напомним, что эти сигналы генерируются, только если сокет UDP является присоединенным (создан с помощью вызова функции `connect`).

ПРИМЕЧАНИЕ

Сигнал SIGIO генерируется для этих двух условий путем вызова макрока `sorwakeu`, описываемого в книге [128, с. 775, с. 779, с. 784].

Сигнал SIGIO и сокеты TCP

К сожалению, использовать управляемый сигналом ввод-вывод для сокетов TCP почти бесполезно. Проблема состоит в том, что сигнал генерируется слишком часто, а само по себе возникновение сигнала не позволяет выяснить, что произошло. Как отмечается в [128, с. 439], генерацию сигнала SIGIO для TCP-сокета вызывают все нижеперечисленные ситуации (при условии, что управляемый сигналом ввод-вывод разрешен):

- на прослушиваемом сокете выполнен запрос на соединение;
- инициирован запрос на отключение;
- запрос на отключение выполнен;
- половина соединения закрыта;
- данные доставлены на сокет;
- данные отправлены с сокета (то есть в буфере отправки имеется свободное место);
- произошла асинхронная ошибка.

Например, если одновременно осуществляются и чтение, и запись в TCP-сокет, то сигнал SIGIO генерируется, когда поступают новые данные и когда подтверждается прием ранее записанных данных, а обработчик сигнала не имеет возможности различить эти сигналы. Если используется сигнал SIGIO, то для предотвращения блокирования при выполнении функции `read` или `write` TCP-сокет должен находиться в режиме неблокируемого ввода-вывода. Следует использовать сигнал SIGIO лишь с прослушиваемым сокетом TCP, поскольку для прослушиваемого сокета этот сигнал генерируется только при завершении установления нового соединения.

Единственное реальное применение управляемого сигналом ввода-вывода с сокетами, которое удалось обнаружить автору, — это сервер NTP (Network Time Protocol — сетевой протокол синхронизации времени), использующий протокол UDP. Основной цикл этого сервера получает дейтаграмму от клиента и посыпает ответ. Но обработка клиентского запроса на этом сервере требует некоторого ненулевого количества времени (больше, чем для нашего тривиального эхо-сервера). Серверу важно записать точные отметки времени для каждой принимаемой дейтаграммы, поскольку это значение возвращается клиенту и используется им для вычисления времени обращения к серверу (RTT). На рис. 25.1 показаны два варианта построения такого UDP-сервера.

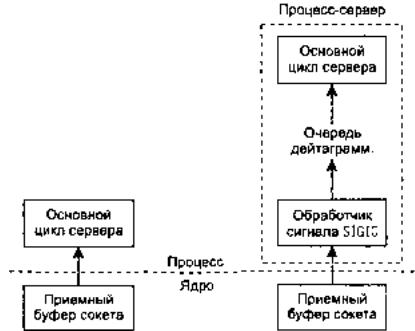


Рис. 25.1. Два варианта построения UDP-сервера

Большинство UDP-серверов (включая наш эхо-сервер, описанный в главе 8) построены так, как показано на рисунке слева. Однако NTP-сервер использует способ, показанный справа: когда прибывает новая дейтаграмма, она читается обработчиком сигнала SIGIO, который также записывает время прибытия дейтаграммы. Далее дейтаграмма помещается в другую очередь внутри процесса, из которой она будет извлечена, а затем обработана основным циклом сервера. Это усложняет код сервера, но зато обеспечивает точные отметки времени прибытия дейтаграмм.

ПРИМЕЧАНИЕ

Вспомните листинг 22.3: процесс может установить параметр сокета IP_RECVDSTADDR, чтобы получить адрес получателя пришедшей UDP-дейтаграммы. Можно возразить, что вместе с полученной дейтаграммой UDP должны быть возвращены два дополнительных фрагмента информации — интерфейс, на котором была получена дейтаграмма (этот интерфейс может отличаться от адреса получателя, если узел использует более типичную модель системы с гибкой привязкой), и время прибытия дейтаграммы.

Для IPv6 интерфейс, на котором была получена дейтаграмма, можно получить, если включен параметр сокета IPV6_PKTINFO (см. раздел 22.8). Аналогичный параметр сокета IP_RECVIF для IPv4 описывался в разделе 22.2.

В FreeBSD также предусмотрен параметр сокета SO_TIMESTAMP, возвращающий время получения дейтаграммы как вспомогательные данные в структуре timeval. В Linux существует флаг SIOCGSTAMP для функции ioctl, которая возвращает структуру timeval, содержащую время прибытия дейтаграммы.

25.3. Эхо-сервер UDP с использованием сигнала SIGIO

В этом разделе мы приведем пример, аналогичный правой части рис. 25.1: UDP-сервер, использующий сигнал SIGIO для получения приходящих дейтаграмм. Этот пример также иллюстрирует использование надежных сигналов стандарта POSIX.

В данном случае клиент совсем не изменен по сравнению с листингами 8.3 и 8.4, а функция сервера main не изменилась по сравнению с листингом 8.1. Единственные внесенные изменения касаются функции dg_echo, которая будет приведена в следующих четырех листингах. В листинге 25.1^[1] представлены глобальные объявления.

Листинг 25.1. Глобальные объявления

```
//sigio/dgecho01.c
1 #include "unp.h"

2 static int sockfd;

3 #define QSIZE     8 /* размер входной очереди */
4 #define MAXDG 4096 /* максимальный размер дейтаграммы */

5 typedef struct {
6     void *dg_data;           /* указатель на текущую дейтаграмму */
```

```

7 size_t dg_len; /* длина дейтаграммы */
8 struct sockaddr *dg_sa; /* указатель на sockaddr{} с адресом клиента */
9 socklen_t dg_salen; /* длина sockaddr{} */
10 } DG;
11 static DG dg[QSIZE]; /* очередь дейтаграмм для обработки */
12 static long cntread[QSIZE + 1]; /* диагностический счетчик */
13 static int ige; /* следующий элемент для обработки в основном цикле */
14 static int iput; /* следующий элемент для считывания обработчиком
    сигналов */
15 static int nqueue; /* количество дейтаграмм в очереди на обработку
    в основном цикле */
16 static socklen_t clilen; /* максимальная длина sockaddr{} */
17 static void sig_io(int);
18 static void sig_hup(int);

```

Очередь принимаемых дейтаграмм

3-12 Обработчик сигнала SIGIO помещает приходящие дейтаграммы в очередь. Эта очередь является массивом структур DG, который интерпретируется как кольцевой буфер. Каждая структура содержит указатель на принятую дейтаграмму, ее длину и указатель на структуру адреса сокета, содержащую адрес протокола клиента и размер адреса протокола. В памяти размещается столько этих структур, сколько указано в QSIZE (в данном случае 8), и в листинге 25.2 будет видно, что функция dg_echo для размещения в памяти всех структур дейтаграмм и адресов сокетов вызывает функцию malloc. Также происходит выделение памяти под диагностический счетчик cntread, который будет рассмотрен чуть ниже. На рис. 25.2 приведен массив структур, при этом предполагается, что первый элемент указывает на 150-байтовую дейтаграмму, а длина связанного с ней адреса сокета равна 16.

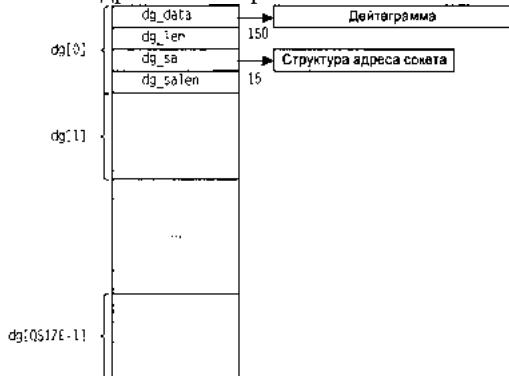


Рис. 25.2. Структуры данных, используемые для хранения прибывающих дейтаграмм и структур адресов их сокетов

Индексы массивов

13-15 Переменная ige является индексом следующего элемента массива для обработки в основном цикле, а переменная iput — это индекс следующего элемента массива, в котором сохраняется результат действия обработчика сигнала. Переменная nqueue обозначает полное количество дейтаграмм, предназначенных для обработки в основном цикле.

В листинге 25.2 показан основной цикл сервера — функция dg_echo.

Листинг 25.2. Функция dg_echo: основной обрабатывающий цикл сервера

```

//sigio/dgecho01.c
19 void
20 dg_echo(int sockfd_arg, SA *pcliaddr, socklen_t clilen_arg)
21 {

```

```

22 int i;
23 const int on = 1;
24 sigset_t zeromask, newmask, oldmask;

25 sockfd = sockfd_arg;
26 clilen = clilen_arg;

27 for (i = 0; i < QSIZE; i++) { /* инициализация очереди */
28     dg[i].dg_data = Malloc(MAXDG);
29     dg[i].dg_sa = Malloc(clilen);
30     dg[i].dg_salen = clilen;
31 }
32 igure = input = nqueue = 0;

33 Signal(SIGHUP, sig_hup);
34 Signal(SIGIO, sig_io);
35 Fcntl(sockfd, F_SETOWN, getpid());
36 Ioctl(sockfd, FIOASYNC, &on);
37 Ioctl(sockfd, FIONBIO, &on);

38 Sigemptyset(&zeromask); /* инициализация трех наборов сигналов */
39 Sigemptyset(&oldmask);
40 Sigemptyset(&newmask);
41 Sigaddset(&newmask, SIGIO); /* сигнал, который хотим блокировать*/

42 Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
43 for (;;) {
44     while (nqueue == 0)
45         sigsuspend(&zeromask); /* ждем дейтаграмму для обработки */

46     /* разблокирование SIGIO */
47     Sigprocmask(SIG_SETMASK, &oldmask, NULL);

48     Sendto(sockfd, dg[igure].dg_data, dg[igure].dg_len, 0,
49             dg[igure].dg_sa, dg[igure].dg_salen);

50     if (++igure >= QSIZE)
51         igure = 0;

52     /* блокировка SIGIO */
53     Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
54     nqueue--;
55 }
56 }

```

Инициализация очереди принятых дейтаграмм

27-32 Дескриптор сокета сохраняется в глобальной переменной, поскольку он необходим обработчику сигналов. Происходит инициализация очереди принятых дейтаграмм.

Установка обработчиков сигналов и флагов сокетов

33-37 Для сигналов SIGHUP (он используется для диагностических целей) и SIGIO устанавливаются обработчики. С помощью функции fcntl задается владелец сокета, а с помощью функции ioctl

устанавливаются флаги ввода-вывода, управляемого сигналом, и неблокируемого ввода-вывода.

ПРИМЕЧАНИЕ

Ранее отмечалось, что для разрешения ввода-вывода, управляемого сигналом, в POSIX применяется флаг `O_ASYNC` функции `fcntl`, но поскольку большинство систем пока его не поддерживают, мы используем функцию `ioctl`. Поскольку большинство систем не поддерживают флаг `O_NONBLOCK` для включения неблокируемого ввода-вывода, здесь также рассмотрен вариант использования функции `ioctl`.

Инициализация наборов сигналов

38-41 Инициализируется три набора сигналов: `zeromask` (никогда не изменяется), `oldmask` (хранит старую маску сигнала, когда `SIGIO` блокируется) и `newmask`. Функция `sigaddset` включает в набор `newmask` бит, соответствующий `SIGIO`.

Блокирование SIGIO и ожидание дальнейших действий

42-45 Функция `sigprocmask` сохраняет текущую маску сигналов процесса в `oldmask`, а затем выполняет логическое сложение, сравнивая `newmask` с текущей маской сигналов. Такие действия блокируют сигнал `SIGIO` и возвращают текущую маску сигналов. Далее мы заходим в цикл `for` и проверяем счетчик `nqueue`. Пока этот счетчик равен нулю, ничего делать не нужно, и мы вызываем функцию `sigsuspend`. Эта функция POSIX, сохранив в одной из локальных переменных текущую маску сигналов, присваивает текущей маске значение аргумента `zeromask`. Так как `zeromask` является пустым набором сигналов, то разрешается доставка любых сигналов. Как только перехватывается сигнал и завершается обработчик, функция `sigsuspend` также завершается. (Это необычная функция, поскольку она всегда возвращает ошибку `EINTR`.) Прежде чем завершиться, функция `sigsuspend` всегда устанавливает такое значение маски сигналов, которое предшествовало ее вызову (в данном случае `newmask`). Таким образом гарантируется, что, когда функция `sigsuspend` возвращает значение, сигнал `SIGIO` блокирован. Именно поэтому можно проверять счетчик `nqueue`, поскольку известно, что пока он проверяется, сигнал `SIGIO` не может быть доставлен.

ПРИМЕЧАНИЕ

А что произойдет, если сигнал `SIGIO` не будет блокирован во время проверки переменной `nqueue`, используемой совместно основным циклом и обработчиком сигналов? Может случиться так, что проверка `nqueue` покажет нулевое значение, а сразу после проверки возникнет сигнал и `nqueue` станет равна 1. Далее мы вызовем функцию `sigsuspend` и перейдем в режим ожидания, в результате чего пропустим сигнал. После вызова функции `sigsuspend` мы не выйдем из режима ожидания, пока не поступит другой сигнал. Это похоже на ситуацию гонок, описанную в разделе 20.5

Разблокирование SIGIO и отправка ответа

46-51 Разблокируем сигнал `SIGIO` с помощью вызова `sigprocmask`, чтобы вернуть маске сигналов процесса значение, сохраненное ранее (`oldmask`). В этом случае ответ посыпается с помощью функции `sendto`. Индекс `iget` увеличился на 1, и если его значение совпадает с количеством элементов массива, он снова обнуляется. Массив используется как кольцевой буфер. Обратите внимание, что нет необходимости блокировать сигнал `SIGIO` во время изменения переменной `iget`, поскольку этот индекс используется только в основном цикле и никогда не изменяется обработчиком сигнала.

Блокирование SIGIO

52-54 Сигнал SIGIO блокируется, а значение переменной `nqueue` уменьшается на 1. Во время изменения данной переменной необходимо заблокировать сигнал, поскольку она используется совместно основным циклом и обработчиком сигнала. Также необходимо, чтобы сигнал SIGIO был заблокирован, когда в начале цикла происходит проверка переменной `nqueue`.

Альтернативным способом является удаление обоих вызовов функции `sigprocmask`, находящихся внутри цикла `for`, что предотвращает разблокирование сигнала и его последующее блокирование. Однако проблема состоит в следующем: в такой ситуации весь цикл выполняется при блокированном сигнале, что уменьшает быстроту реагирования обработчика сигнала. При этом дейтаграммы не будут теряться (если, конечно, буфер приема сокета достаточно велик), но выдача сигнала процессу будет задерживаться на время блокирования сигнала. Одной из задач при создании приложений, производящих обработку сигналов, должна быть минимизация времени блокирования сигнала.

Листинг 25.3. Обработчик сигнала SIGIO

```
//sigio/dgecho01.c
57 static void
58 sig_io(int signo)
59 {
60     ssize_t len;
61     int nread;
62     DG *ptr;
63     for (nread = 0;;) {
64         if (nqueue >= QSIZE)
65             err_quit("receive overflow");
66
67         ptr = &dg[iput];
68         ptr->dg_salen = clilen;
69         len = recvfrom(sockfd, ptr->dg_data, MAXDG, 0,
70                         ptr->dg_sa, &ptr->dg_salen);
71         if (len < 0) {
72             if (errno == EWOULDBLOCK)
73                 break; /* все сделано; очередь на чтение отсутствует */
74             else
75                 err_sys("recvfrom error");
76         }
77         ptr->dg_len = len;
78
79         nread++;
80         nqueue++;
81         if (++iput >= QSIZE)
82             iput = 0;
83     }
82     cntread[nread]++;
83     /* гистограмма количества дейтаграмм,
84      считанных для каждого сигнала */
83 }
```

Во время создания этих обработчиков сигналов была обнаружена следующая проблема: в стандарте POSIX сигналы обычно не помещаются в очередь. Это означает, что если во время пребывания внутри обработчика сигналов (при этом сигнал заведомо заблокирован) возникает еще два сигнала, то сигнал доставляется еще один раз.

ПРИМЕЧАНИЕ

В стандарте POSIX предусмотрено несколько сигналов реального времени, для которых обеспечивается буферизация, однако ряд других сигналов, в том числе и SIGIO, обычно не буферизуются, то есть не помещаются в очередь на доставку.

Рассмотрим следующий сценарий. Прибывает дейтаграмма и выдается сигнал. Обработчик сигнала считывает дейтаграмму и помещает ее в очередь к основному циклу. Но во время работы обработчика сигнала приходят еще две дейтаграммы, вызывая генерацию сигнала еще дважды. Поскольку сигнал блокирован, то когда обработчик сигналов возвращает управление после обработки первого сигнала, он запустится снова всего лишь один раз. После второго запуска обработчик считывает вторую дейтаграмму, а третья будет оставлена в очереди приходящих дейтаграмм сокета. Эта третья дейтаграмма будет прочитана, только если (и только когда) придет четвертая. Когда придет четвертая дейтаграмма, считана и поставлена в очередь на обработку основным циклом будет именно третья, а не четвертая дейтаграмма.

Поскольку сигналы не помещаются в очередь, дескриптор, установленный для управляемого сигналом ввода-вывода, обычно переводится в неблокируемый режим. Обработчик сигнала SIGIO мы кодируем таким образом, чтобы он считывал дейтаграммы в цикле, который прерывается, только когда при считывании возвращается ошибка EWOULDBLOCK.

Проверка переполнения очереди

64-65 Если очередь переполняется, происходит завершение работы. Для обработки такой ситуации существуют и другие способы (например, можно размещать в памяти дополнительные буферы), но для данного примера достаточно простого завершения.

Чтение дейтаграммы

66-76 На неблокируемом сокете вызывается функция recvfrom. Элемент массива, обозначенный индексом iput, — это то место, куда записывается дейтаграмма. Если нет дейтаграмм, которые нужно считывать, мы выходим из цикла for с помощью оператора break.

Увеличение счетчиков и индекса на единицу

77-80 Переменная nread является диагностическим счетчиком количества дейтаграмм, читаемых на один сигнал. Переменная nqueue — это количество дейтаграмм для обработки основным циклом.

82 Прежде чем обработчик сигналов возвращает управление, он увеличивает счетчик на единицу в соответствии с количеством дейтаграмм, прочитанных за один сигнал. Этот массив распечатывается программой в листинге 25.4 и представляет собой диагностическую информацию для обработки сигнала SIGHUP.

Последняя функция (листинг 25.4) представляет собой обработчик сигнала SIGHUP, который выводит массив cntread. Он считает количество дейтаграмм, прочитанных за один сигнал.

Листинг 25.4. Обработчик сигнала SIGHUP

```
//sigio/dgecho01.c
84 static void
85 sig_hup(int signo)
86 {
87     int i;

88     for (i = 0; i <= QSIZE; i++)
89         printf("cntread[%d] = %ld\n", i, cntread[i]);
90 }
```

Чтобы проиллюстрировать, что сигналы не буферизуются и что в дополнение к установке флага, указывающего на управляемый сигналом ввод-вывод, необходимо перевести сокет в неблокируемый режим, запустим этот сервер с шестью клиентами одновременно. Каждый клиент посыпает серверу 3645 строк (для отражения). При этом каждый клиент запускается из сценария интерпретатора в фоновом

режиме, так что все клиенты стартуют приблизительно одновременно. Когда все клиенты завершены, серверу посыпается сигнал SIGHUP, в результате чего сервер выводит получившийся массив cntread:

```
linux % udpserv01
cntread[0] = 2
cntread[1] = 21838
cntread[2] = 12
cntread[3] = 1
cntread[4] = 0
cntread[5] = 1
cntread[6] = 0
cntread[7] = 0
cntread[8] = 0
```

Большую часть времени обработчик сигналов читает только одну дейтаграмму, но бывает, что готово больше одной дейтаграммы. Ненулевое значение счетчика cntread[0] получается потому, что сигнал генерируется в процессе выполнения клиента. Мы считываем дейтаграммы в цикле обработчика сигнала. Дейтаграмма, прибывшая во время считывания других дейтаграмм, будет считана вместе с этими дейтаграммами (в том же вызове обработчика), а сигнал об ее прибытии будет отложен и доставлен процессу после завершения обработчика. Это приведет к повторному вызову обработчика, но считывать ему будет нечего (отсюда $\text{cntread}[0] > 0$). Наконец, можно проверить, что взвешенная сумма элементов массива ($21 \cdot 838 \times 1 + 12 \times 2 + 1 \times 3 + 1 \times 5 = 21870$) равна 6×3645 (количество клиентов \times количество строк клиента).

25.4. Резюме

При управляемом сигнальном вводе-выводе ядро уведомляет процесс сигналом SIGIO, если «что-нибудь» происходит на сокете.

- Для присоединенного TCP-сокета существует множество ситуаций, которые вызывают такое уведомление, что делает эту возможность практически бесполезной.
- Для прослушиваемого TCP-сокета уведомление приходит процессу только в случае готовности принятия нового соединения.
- Для UDP такое уведомление означает, что либо пришла дейтаграмма, либо произошла асинхронная ошибка: в обоих случаях вызывается recvfrom.

С помощью метода, аналогичного применяемому для сервера NTP, был изменен эхо-сервер UDP для работы с вводом-выводом, управляемым сигналом: мы стремимся выполнить чтение дейтаграммы как можно быстрее после ее прибытия, чтобы получить точную отметку времени прибытия и поставить дейтаграмму в очередь для дальнейшей обработки.

Упражнения

1. Далее приведен альтернативный вариант цикла, рассмотренного в листинге 25.2:

```
for (;;) {
    Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
    while (nqueue == 0)
        sigsuspend(&zeromask); /* ожидание дейтаграммы для обработки */
    nqueue--;
    /* разблокирование SIGIO */
    Sigprocmask(SIG_SETMASK, &oldmask, NULL);

    Sendto(sockfd, dg[iget].dg_data, dg[iget].dg_len, 0,
            dg[iget].dg_sa, dg[iget].dg_salen);
    if (++iget >= QSIZE)
        iget = 0;
}
```

Верна ли такая модификация?

Глава 26

Программные потоки

26.1. Введение

Согласно традиционной модели Unix, когда процессу требуется, чтобы некое действие было выполнено каким-либо другим объектом, он порождает дочерний процесс, используя функцию `fork`, и этим порожденным процессом выполняется необходимое действие. Большинство сетевых серверов под Unix устроены именно таким образом, как мы видели при рассмотрении примера параллельного (*concurrent*) сервера: родительский процесс осуществляет соединение с помощью функции `accept` и порождает дочерний процесс, используя функцию `fork`, а затем дочерний процесс занимается обработкой клиентского запроса.

Хотя эта концепция с успехом использовалась на протяжении многих лет, с функцией `fork` связаны определенные неудобства.

■ Стоимость функции `fork` довольно высока, так как при ее использовании требуется скопировать все содержимое памяти из родительского процесса в дочерний, продублировать все дескрипторы и т.д. Текущие реализации используют технологию, называемую *копированием при записи* (*copy-on-write*), при которой копирование пространства данных из родительского процесса в дочерний происходит лишь тогда, когда дочернему процессу требуется своя собственная копия. Но несмотря на эту оптимизацию, стоимость функции `fork` остается высокой.

■ Для передачи данных между родительским и дочерним процессами *после* вызова функции `fork` требуется использовать средства взаимодействия процессов (IPC). Передача информации перед вызовом `fork` не вызывает затруднений, так как при запуске дочерний процесс получает от родительского копию пространства данных и копии всех родительских дескрипторов. Но возвращение информации из дочернего процесса в родительский требует большей работы.

Обе проблемы могут быть разрешены путем использования *программных потоков* (*threads*). Программные потоки иногда называются *облегченными процессами* (*lightweight processes*), так как поток проще, чем процесс. В частности, создание потока требует в 10–100 раз меньше времени, чем создание процесса.

Все потоки одного процесса совместно используют его глобальные переменные, поэтому им легко обмениваться информацией, но это приводит к необходимости синхронизации.

Однако общими становятся не только глобальные переменные. Все потоки одного процесса разделяют:

- инструкции процесса;
- большую часть данных;
- открытые файлы (например, дескрипторы);
- обработчики сигналов и вообще настройки для работы с сигналами (действие сигнала);
- текущий рабочий каталог;
- идентификаторы пользователя и группы пользователей.

У каждого потока имеются собственные:

- идентификатор потока;
- набор регистров, включая счетчик команд и указатель стека;
- стек (для локальных переменных и адресов возврата);
- переменная `errno`;
- маска сигналов;
- приоритет.

ПРИМЕЧАНИЕ

Как сказано в разделе 11.18, можно рассматривать обработчик сигнала как некую разновидность потока. В традиционной модели Unix у нас имеется основной поток выполнения и обработчик сигнала (другой поток). Если в основном потоке в момент возникновения сигнала происходит корректировка связного списка и обработчик сигнала также пытается изменить

связный список, обычно начинается путаница. Основной поток и обработчик сигнала совместно используют одни и те же глобальные переменные, но у каждого из них имеется свой собственный стек.

В этой книге мы рассматриваем потоки POSIX, которые также называются *Pthreads* (POSIX threads). Они были стандартизованы в 1995 году как часть POSIX.1c и будут поддерживаться большинством версий Unix. Мы увидим, что все названия функций Pthreads начинаются с символов `pthread_`. Эта глава является введением в концепцию потоков, необходимым для того, чтобы в дальнейшем мы могли использовать потоки в наших сетевых приложениях. Более подробную информацию вы можете найти в [15].

26.2. Основные функции для работы с потоками: создание и завершение потоков

В этом разделе мы рассматриваем пять основных функций для работы с потоками, а в следующих двух разделах мы используем эти функции для написания потоковой модификации клиента и сервера TCP.

Функция `pthread_create`

Когда программа запускается с помощью функции `exec`, создается один поток, называемый *начальным* (*initial*) или *главным* (*main*). Дополнительные потоки создаются функцией `pthread_create`.

```
#include <pthread.h>
```

```
int pthread_create(pthread_t* tid, const pthread_attr_t *attr,
                  void *(*func)(void*), void *arg);
```

Возвращает: 0 в случае успешного выполнения, положительное значение Exxx в случае ошибки

Каждый поток процесса обладает собственным *идентификатором потока* (*thread ID*), относящимся к типу данных `pthread_t` (как правило, это `unsigned int`). При успешном создании нового потока его идентификатор возвращается через указатель `tid`.

У каждого потока имеется несколько *атрибутов*: его приоритет, исходный размер стека, указание на то, должен ли этот поток являться демоном или нет, и т.д. При создании потока мы можем задать эти атрибуты, инициализируя переменную типа `pthread_attr_t`, что позволяет заменить значение, заданное по умолчанию. Обычно мы используем значение по умолчанию, в этом случае мы задаем аргумент `attr` равным пустому указателю.

Наконец, при создании потока мы должны указать, какую функцию будет выполнять этот поток. Выполнение потока начинается с вызова заданной функции, а завершается либо явно (вызовом `pthread_exit`), либо неявно (когда вызванная функция возвращает управление). Адрес функции задается аргументом `func`, и она вызывается с единственным аргументом-указателем `arg`. Если этой функции необходимо передать несколько аргументов, следует поместить их в некоторую структуру и передать адрес этой структуры как единственный аргумент функции.

Обратите внимание на объявления `func` и `arg`. Функции передается один аргумент — универсальный указатель `void*`. Это позволяет нам передавать потоку с помощью единственного указателя все, что требуется, и точно так же поток возвращает любые данные, используя этот указатель.

Возвращаемое значение функций Pthreads — это обычно 0 в случае успешного выполнения или ненулевая величина в случае ошибки. Но в отличие от функций сокетов и большинства системных вызовов, для которых в случае ошибки возвращается -1 и переменной `errno` присваивается некоторое положительное значение (код ошибки), функции Pthreads возвращают сам код ошибки. Например, если функция `pthread_create` не может создать новый поток, так как мы превысили допустимый системный предел количества потоков, функция возвратит значение `EAGAIN`. Функции Pthreads не присваивают переменной `errno` никаких значений. Соглашение о том, что 0 является индикатором успешного выполнения, а ненулевое значение — индикатором ошибки, не приводит к противоречию, так как все значения `Exxx`, определенные в заголовочном файле `<sys/errno.h>`, являются положительными. Ни одному из имен ошибок `Exxx` не сопоставлено нулевое значение.

Функция `pthread_join`

Мы можем приостановить выполнение текущего потока и ждать завершения выполнения какого-либо другого потока, используя функцию `pthread_join`. Сравнивая потоки и процессы Unix, можно сказать, что функция `pthread_create` аналогична функции `fork`, а функция `pthread_join` — функции `waitpid`.

```
#include <pthread.h>
```

```
int pthread_join(pthread_t tid, void **status);
```

Возвращает: 0 в случае успешного выполнения, положительное значение Exxx в случае ошибки

Следует указать идентификатор `tid` того потока, завершения которого мы ждем. К сожалению, нет способа указать, что мы ждем завершения любого потока данного процесса (тогда как при работе с процессами мы могли с помощью функции `waitpid` ждать завершения любого процесса, задав аргумент идентификатора процесса, равный `-1`). Мы вернемся к этой проблеме при обсуждении листинга 26.11.

Если указатель `status` непустой, то значение, возвращаемое потоком (указатель на некоторый объект), хранится в ячейке памяти, на которую указывает `status`.

Функция `pthread_self`

Каждый поток снабжен идентификатором, уникальным в пределах данного процесса. Идентификатор потока возвращается функцией `pthread_create` и, как мы видели, используется функцией `pthread_join`. Поток может узнать свой собственный идентификатор с помощью вызова `pthread_self`.

```
#include <pthread.h>
```

```
pthread_t pthread_self(void);
```

Возвращает: идентификатор вызывающего потока

Сравнивая потоки и процессы Unix, можно отметить, что функция `pthread_self` аналогична функции `getpid`.

Функция `pthread_detach`

Поток может быть либо *присоединяемым* (*joinable*), каким он является по умолчанию, либо *отсоединенным* (*detached*). Когда присоединяемый поток завершает свое выполнение, его статус завершения и идентификатор сохраняются, пока другой поток данного процесса не вызовет функцию `pthread_join`. В свою очередь, отсоединенный поток напоминает процесс-демон: когда он завершается, все занимаемые им ресурсы освобождаются и мы не можем отслеживать его завершение. Если один поток должен знать, когда завершится выполнение другого потока, нам следует оставить последний присоединяемым.

Функция `pthread_detach` изменяет состояние потока, превращая его из присоединяемого в отсоединеный.

```
#include <pthread.h>
```

```
int pthread_detach(pthread_t tid);
```

Возвращает: 0 в случае успешного выполнения, положительное значение Exxx в случае ошибки

Эта функция обычно вызывается потоком при необходимости изменить собственный статус в следующем формате:

```
pthread_detach(pthread_self());
```

Функция `pthread_exit`

Одним из способов завершения потока является вызов функции `pthread_exit`.

```
#include <pthread.h>
```

```
void pthread_exit(void *status);
```

Ничего не возвращает вызвавшему потоку

Если поток не является отсоединенными, идентификатор потока и статус завершения сохраняются до того момента, пока какой-либо другой поток данного процесса не вызовет функцию `pthread_join`.

Указатель `status` не должен указывать на объект, локальный по отношению к вызывающему потоку, так как этот объект будет уничтожен при завершении потока.

Существуют и другие способы завершения потока.

■ Функция, которая была вызвана потоком (третий аргумент функции `pthread_create`), может возвратить управление в вызывающий процесс. Поскольку, согласно своему объявлению, эта функция возвращает указатель `void`, возвращаемое ею значение играет роль статуса завершения данного потока.

■ Если функция `main` данного процесса возвращает управление или любой поток вызывает функцию `exit`, процесс завершается вместе со всеми своими потоками.

26.3. Использование потоков в функции str_cli

В качестве первого примера использования потоков мы перепишем нашу функцию `str_cli`. В листинге 16.6 была представлена версия этой функции, в которой использовалась функция `fork`. Напомним, что были также представлены и некоторые другие версии этой функции: изначально в листинге 5.4 функция блокировалась в ожидании ответа и была, как мы показали, далека от оптимальной в случае пакетного ввода; в листинге 6.2 применяется блокируемый ввод-вывод и функция `select`; версии, показанные в листинге 16.1 и далее, используют неблокируемый ввод-вывод.

На рис. 26.1 показана структура очередной версии функции `str_cli`, на этот раз использующей потоки, а в листинге 26.1^[1] представлен код этой функции.

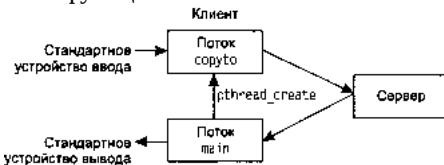


Рис. 26.1. Измененная функция `str_cli`, использующая потоки

Листинг 26.1. Функция `str_cli`, использующая потоки

```
//threads/strclithread.c
1 #include "unpthread.h"

2 void *copyto(void *);

3 static int sockfd; /* глобальная переменная, доступная обоим потокам */
4 static FILE *fp;

5 void
6 str_cli(FILE *fp_arg, int sockfd_arg)
7 {
8     char recvline[MAXLINE];
9     pthread_t tid;

10    sockfd = sockfd_arg; /* копирование аргументов во внешние переменные */
11    fp = fp_arg;

12    Pthread_create(&tid, NULL, copyto, NULL);

13    while (Readline(sockfd, recvline, MAXLINE) > 0)
14        Fputs(recvline, stdout);
15    }

16 void*
17 copyto(void *arg)
18 {
19     char sendline[MAXLINE];

20    while (Fgets(sendline, MAXLINE, fp) != NULL)
```

```
21     Writen(sockfd, sendline, strlen(sendline));  
22     Shutdown(sockfd, SHUT_WR); /* признак конца файла в стандартном  
23     потоке ввода, отправка сегмента FIN */  
24     return (NULL);  
25     /* завершение потока происходит, когда в стандартном потоке ввода  
26     встречается признак конца файла */  
27 }
```

Заголовочный файл *unpthread.h*

1 Мы впервые встречаемся с заголовочным файлом *unpthread.h*. Он включает наш обычный заголовочный файл *unp.h*, затем — заголовочный файл POSIX *<pthread.h>*, и далее определяет прототипы наших потоковых функций-оберток для *pthread_XXX* (см. раздел 1.4), название каждой из которых начинается с *Pthread_*.

Сохранение аргументов во внешних переменных

10-11 Для потока, который мы собираемся создать, требуются значения двух аргументов функции *str_cli*: *fp* — указатель на структуру *FILE* для входного файла, и *sockfd* — сокет TCP, связанный с сервером. Для простоты мы храним эти два значения во внешних переменных. Альтернативой является запись этих двух значений в структуру, указатель на которую затем передается в качестве аргумента создаваемому потоку.

Создание нового потока

12 Создается поток, и значение нового идентификатора потока сохраняется в *tid*. Функция, выполняемая новым потоком, — это *copyto*. Никакие аргументы потоку не передаются.

Главный цикл потока: копирование из сокета в стандартный поток вывода

13-14 В основном цикле вызываются функции *readline* и *fputs*, которые осуществляют копирование из сокета в стандартный поток вывода.

Завершение

15 Когда функция *str_cli* возвращает управление, функция *main* завершается при помощи вызова функции *exit* (см. раздел 5.4). При этом завершаются все потоки данного процесса. В обычном сценарии второй поток уже должен завершиться в результате считывания признака конца файла из стандартного потока ввода. Но в случае, когда сервер преждевременно завершил свою работу (см. раздел 5.12), при вызове функции *exit* завершается также и второй поток, чего мы и добиваемся.

Поток *copyto*

16-25 Этот поток осуществляет копирование из стандартного потока ввода в сокет. Когда он считывает признак конца файла из стандартного потока ввода, на сокете вызывается функция *shutdown* и отсылается сегмент FIN, после чего поток возвращает управление. При выполнении оператора *return* (то есть когда функция, запустившая поток, возвращает управление) поток также завершается.

В конце раздела 16.2 мы привели результаты измерений времени выполнения для пяти различных реализаций функции *str_cli*. Мы отметили, что многопоточная версия выполняется всего 8,5 с —

немногим быстрее, чем версия, использующая функцию `fork` (как мы и ожидали), но медленнее, чем версия с неблокируемым вводом-выводом. Тем не менее, сравнивая устройство версии с неблокируемым вводом-выводом (см. раздел 16.2) и версии с использованием потоков, мы заметили, что первая гораздо сложнее. Поэтому мы рекомендуем использовать именно версию с потоками, а не с неблокируемым вводом-выводом.

26.4. Использование потоков в эхо-сервере TCP

Теперь мы перепишем эхо-сервер TCP, приведенный в листинге 5.1, используя для каждого клиента по одному потоку вместо одного процесса. Кроме того, с помощью нашей функции `tcp_listen` мы сделаем эту версию не зависящей от протокола. В листинге 26.2 показан код сервера.

Листинг 26.2. Эхо-сервер TCP, использующий потоки

```
//threads/tcperv01.c
1 #include "unpthread.h"

2 static void *doit(void*); /* каждый поток выполняет эту функцию */

3 int
4 main(int argc, char **argv)
5 {
6     int listenfd, connfd;
7     pthread_t tid;
8     socklen_t addrlen, len;
9     struct sockaddr *cliaddr;

10    if (argc == 2)
11        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
12    else if (argc == 3)
13        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
14    else
15        err_quit("usage: tcpserv01 [ <host> ] <service or port>");

16    cliaddr = Malloc(addrlen);

17    for (;;) {
18        len = addrlen;
19        connfd = Accept(listenfd, cliaddr, &len);
20        Pthread_create(&tid, NULL, &doit, (void*)connfd);
21    }
22 }

23 static void*
24 doit(void *arg)
25 {
26     Pthread_detach(pthread_self());
27     str_echo((int)arg); /* та же функция, что и раньше */
28     Close((int)arg); /* мы закончили с присоединенным сокетом */
29     return (NULL);
30 }
```

Создание потока

17-21 Когда функция `accept` возвращает управление, мы вызываем функцию `Pthread_create` вместо функции `fork`. Мы передаем функции `doit` единственный аргумент — дескриптор присоединенного сокета `connfd`.

ПРИМЕЧАНИЕ

Мы преобразуем целочисленный дескриптор сокета к универсальному указателю (`void`). В ANSI C не гарантируется, что такое преобразование будет выполнено корректно, — мы можем быть уверены лишь в том, что оно сработает в тех системах, в которых размер целого числа не превышает размера указателя. К счастью, большинство реализаций Unix обладают этим свойством (см. табл. 1.5). Далее мы поговорим об этом подробнее.

Функция потока

23-30 `doit` — это функция, выполняемая потоком. Поток отделяет себя с помощью функции `pthread_detach`, так как нет причины, по которой главному потоку имело бы смысл ждать завершения каждого созданного им потока. Функция `str_echo` не изменилась и осталась такой же, как в листинге 5.2. Когда эта функция завершается, следует вызвать функцию `close` для того, чтобы закрыть присоединенный сокет, поскольку этот поток использует все дескрипторы совместно с главным потоком. При использовании функции `fork` дочерний процесс не должен специально закрывать присоединенный сокет, так как при завершении дочернего процесса все открытые дескрипторы закрываются (см. упражнение 26.2).

Обратите также внимание на то, что главный поток не закрывает присоединенный сокет, что всегда происходило, когда параллельный сервер вызывал функцию `fork`. Это объясняется тем, что все потоки внутри процесса совместно используют все дескрипторы, поэтому если главному потоку потребуется вызвать функцию `close`, это приведет к закрытию соединения. Создание нового потока не влияет на счетчики ссылок для открытых дескрипторов, в отличие от того, что происходит при вызове функции `fork`.

В этой программе имеется одна неявная ошибка, о которой рассказывается в разделе 26.5. Можете ли вы ее обнаружить? (См. упражнение 26.5.)

Передача аргументов новым потокам

Мы уже упомянули, что в листинге 26.2 мы преобразуем целочисленную переменную `connfd` к указателю на неопределенный тип (`void`), но этот способ не работает в некоторых системах. Для корректной обработки данной ситуации требуются дополнительные усилия.

В первую очередь, заметим, что мы не можем просто передать адрес `connfd` нового потока, то есть следующий код не будет работать:

```
int main(int argc, char **argv) {
    int listenfd, connfd;
    ...

    for (;;) {
        len = addrlen;
        connfd = Accept(listenfd, cliaddr, &len);

        Pthread_create(&tid, NULL, &doit, &connfd);
    }
}

static void* doit(void *arg) {
    int connfd;

    connfd = *((int*)arg);
    Pthread_detach(pthread_self());
    str_echo(connfd); /* та же функция, что и прежде */
    Close(connfd);    /* мы закончили с присоединенным сокетом */
    return(NULL);
}
```

С точки зрения ANSI C здесь все в порядке: мы гарантированно можем преобразовать целочисленный указатель к типу `void*` и затем обратно преобразовать получившийся указатель на неопределенный тип к целочисленному указателю. Проблема заключается в другом — на что именно он будет указывать?

В главном потоке имеется одна целочисленная переменная `connfd`, и при каждом вызове функции `accept` значение этой переменной меняется на новое (в соответствии с новым присоединенным сокетом). Может сложиться следующая ситуация:

- Функция `accept` возвращает управление, записывается новое значение переменной `connfd` (допустим, новый дескриптор равен 5) и в главном потоке вызывается функция `pthread_create`. Указатель на `connfd` (а не фактическое его значение!) является последним аргументом функции `pthread_create`.

- Создается новый поток, и начинает выполняться функция `doit`.

- Готово другое соединение, и главный поток снова начинает выполнять (прежде, чем начнется выполнение вновь созданного потока). Завершается функция `accept`, записывается новое значение переменной `connfd` (например, значение нового дескриптора равно 6) и главный поток вновь вызывает функцию `pthread_create`.

Хотя созданы два новых потока, оба они будут работать с одним и тем же последним значением переменной `connfd`, которое, согласно нашему предположению, равно 6. Проблема заключается в том, что несколько потоков получают доступ к совместно используемой переменной (целочисленному значению, хранящемуся в `connfd`) при отсутствии синхронизации. В листинге 26.2 мы решаем эту проблему, передавая значение переменной `connfd` функции `pthread_create`, вместо того чтобы передавать указатель на это значение. Этот метод работает благодаря тому способу, которым целочисленные значения в С передаются вызываемой функции (копия значения помещается в стек вызванной функции).

В листинге 26.3 показано более удачное решение описанной проблемы.

Листинг 26.3. Эхо-сервер TCP, использующий потоки с более переносимой передачей аргументов

```
//threads/tcpser02.c
1 #include "unpthread.h"

2 static void *doit(void*); /* каждый поток выполняет эту функцию */

3 int
4 main(int argc, char **argv)
5 {
6     int listenfd, *iptr;
7     thread_t tid;
8     socklen_t addrlen, len;
9     struct sockaddr *cliaddr;

10    if (argc == 2)
11        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
12    else if (argc == 3)
13        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
14    else
15        err_quit("usage: tcpser01 [ <host> ] <service or port>");

16    cliaddr = Malloc(addrlen);

17    for (;;) {
18        len = addrlen;
19        iptr = Malloc(sizeof(int));
20        *iptr = Accept(listenfd, cliaddr, &len);
21        Pthread_create(&tid, NULL, &doit, iptr);
22    }
23 }

24 static void*
25 doit(void *arg)
26 {
```

```

27 int connfd;
28 connfd = *((int*)arg);
29 free(arg);

30 Pthread_detach(pthread_self());
31 str_echo(connfd); /* та же функция, что и раньше */
32 Close(connfd); /* мы закончили с присоединенным сокетом */
33 return (NULL);
34 }

```

17-22 Каждый раз перед вызовом функции accept мы вызываем функцию malloc и выделяем в памяти пространство для целочисленной переменной (дескриптора присоединенного сокета). Таким образом каждый поток получает свою собственную копию этого дескриптора.

28-29 Поток получает значение дескриптора присоединенного сокета, а затем освобождает занимаемую им память с помощью функции free.

Исторически функции malloc и free не допускали повторного вхождения. Это означает, что при вызове той или иной функции из обработчика сигнала в то время, когда главный поток выполняет одну из них, возникает большая путаница, так как эти функции оперируют статическими структурами данных. Как же мы можем вызывать эти две функции в листинге 26.3? Дело в том, что в POSIX требуется, чтобы эти две функции, так же как и многие другие, были *безопасными в многопоточной среде (thread-safe)*. Обычно это достигается с помощью некоторой разновидности синхронизации, осуществляющейся внутри библиотечных функций и являющейся для нас прозрачной (то есть незаметной).

Функции, безопасные в многопоточной среде

Стандарт POSIX.1 требует, чтобы все определенные в нем функции, а также функции, определенные в стандарте ANSI C, были безопасными в многопоточной среде. Исключения из этого правила приведены в табл. 26.1.

К сожалению, в POSIX.1 ничего не сказано о безопасности в многопоточной среде по отношению к функциям сетевого API. Последние пять строк в этой таблице появились благодаря Unix 98. В разделе 11.18 мы говорили о том, что функции gethostbyname и gethostbyaddr не допускают повторного вхождения. Как уже отмечалось, некоторые производители определяют версии этих функций, обладающие свойством безопасности в многопоточной среде (их названия заканчиваются на _r), но поскольку они не стандартизованы, лучше от них отказаться. Все функции getXXX, не допускающие повторного вхождения, были приведены в табл. 11.5.

Таблица 26.1. Функции, безопасные в многопоточной среде

Могут не быть безопасными в многопоточной среде	Должны быть безопасными в многопоточной среде	Комментарии
Asctime	asctime_r	Безопасна в многопоточной среде только в случае непустого аргумента
	ctermid	
Ctime	ctime_r	
getc_unlocked		
getchar_unlocked		
Getgrid	getgrid_r	
Getgrnam	getgrnam_r	
Getlogin	getlogin_r	
Getpwnam	getpwnam_r	
Getpwuid	getpwuid_r	
Gmtime	gmtime_r	
Localtime	localtime_r	
putc_unlocked		

putchar_unlocked		
Rand	rand_r	
Readdir	readdir_r	
Strtostock	strtostock_r	
	tmpnam	Безопасна в многопоточной среде только в случае непустого аргумента
Ttynname	ttynname_r	
GethostXXX		
GetnetXXX		
GetprotoXXX		
GetservXXX		
inet_ntoa		

Приведенная таблица позволяет заключить, что общим способом сделать функцию допускающей повторное вхождение является определение новой функции с названием, оканчивающимся на `_r`. Обе функции будут безопасными в многопоточной среде, только если вызывающий процесс выделяет в памяти место для результата и передает соответствующий указатель как аргумент функции.

26.5. Собственные данные потоков

При преобразовании существующих функций для использования в многопоточной среде часто возникают проблемы, связанные со статическими переменными. Функция, сохраняющая состояние в собственном буфере или возвращающая результат в виде указателя на статический буфер, не является безопасной в многопоточной среде, поскольку несколько потоков не могут использовать один и тот же буфер для хранения разных данных. Такая проблема имеет несколько решений.

1. Использование собственных данных потоков (thread-specific data). Это нетривиальная задача, и функция при этом преобразуется к такому виду, что может использоваться только в системах, поддерживающих потоки. Преимущество этого подхода заключается в том, что не меняется вызывающая последовательность, и все изменения связаны с библиотечной функцией, а не с приложениями, которые вызывают эту функцию. Позже в этом разделе мы покажем безопасную в многопоточной среде версию функции `readline`, созданную с применением собственных данных потоков.

2. Изменение вызывающей последовательности таким образом, чтобы вызывающий процесс упаковывал все аргументы в некую структуру, а также записывал в нее статические переменные из листинга 3.12. Это также было сделано, и в листинге 26.4 показана новая структура и новые прототипы функций.

Листинг 26.4. Структура данных и прототип функции для версии функции `readline`, допускающей повторное вхождение

```
typedef struct {
    int      read_fd;    /* дескриптор, указывающий, откудачитываются данные */
    char   *read_ptr;    /* буфер, куда передаются данные */
    size_t  read maxlen; /* максимальное количество байтов, которое может быть считано */
    /* следующие три элемента для внутреннего использования функцией */
    int      rl_cnt;     /* инициализируется нулем */
    char   *rl_bufptr;   /* инициализируется значением rl_buf */
    char   rl_buf[MAXLINE];
} Rline;

void readline_rinit(int, void*, size_t, Rline* );
ssize_t readline_r(Rline* );
ssize_t Readline_r(Rline* );
```

Эти новые функции могут использоваться как в системах с поддержкой потоков, так и в тех, где потоки не поддерживаются, но все приложения,зывающие функцию `readline`, должны быть изменены.

3. Реструктуризация интерфейса для исключения статических переменных и обеспечения безопасности функции в многопоточной среде. Для `readline` это будет означать отказ от увеличения быстродействия, достигнутого в листинге 3.12, и возвращение к более старой версии, представленной в

листинге 3.11. Поскольку мы назвали старую версию «ужасно медленной», это решение не всегда пригодно на практике.

Использование собственных данных потоков — это распространенный способ сделать существующую функцию безопасной в многопоточной среде. Прежде чем описывать функции Pthread, работающие с такими данными, мы опишем саму концепцию и возможный способ реализации, так как эти функции кажутся более сложными, чем являются на самом деле.

Частично осложнения возникают по той причине, что во всех книгах, где идет речь о потоках, описание собственных данных потоков дается по образцу стандарта Pthreads. Пары ключ-значение и ключи рассматриваются в них как непрозрачные объекты. Мы описываем собственные данные потоков в терминах индексов и указателей, так как обычно в реализациях в качестве ключей используются небольшие положительные целые числа (индексы), а значение, ассоциированное с ключом, — это просто указатель на область памяти, выделяемую потоку с помощью функции malloc.

В каждой системе поддерживается ограниченное количество объектов собственных данных потоков. В POSIX требуется, чтобы этот предел не превышал 128 (на каждый процесс), и в следующем примере мы используем именно это значение. Система (вероятно, библиотека потоков) поддерживает один массив структур (которые мы называем структурами Key) для каждого процесса, как показано на рис. 26.2.

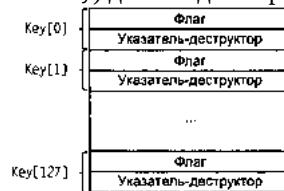


Рис. 26.2. Возможная реализация собственных данных потока

Флаг в структуре Key указывает, используется ли в настоящий момент данный элемент массива. Все флаги инициализируются как указывающие на то, что элемент не используется. Когда поток вызывает функцию pthread_key_create для создания нового элемента собственных данных потока, система отыскивает в массиве структур Key первую структуру, не используемую в настоящий момент. Индекс этой структуры, который может иметь значение от 0 до 127, называется ключом и возвращается вызывающему потоку как результат выполнения функции. О втором элементе структуры Key, так называемом *указателе-деструкторе*, мы поговорим чуть позже.

В дополнение к массиву структур Key, общему для всего процесса, система хранит набор сведений о каждом потоке процесса в структуре Pthread. Частью этой структуры является массив указателей, состоящий из 128 элементов, который мы называем rkey. Это показано на рис. 26.3.

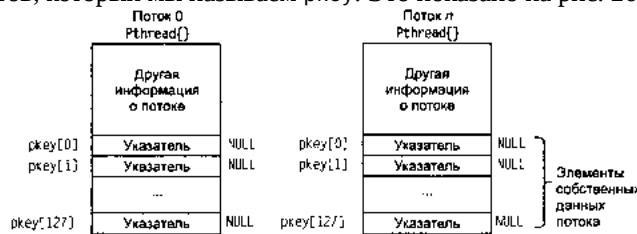


Рис. 26.3. Информация, хранящаяся в системе для каждого потока

Все элементы массива rkey инициализируются пустыми указателями. Эти 128 указателей являются «значениями», ассоциированными с каждым из 128 «ключей» процесса.

Когда мы с помощью функции pthread_key_create создаем ключ, система сообщает нам фактическое значение ключа (индекс). Затем каждый поток может сохранить значение (указатель), связанное с этим ключом, и, как правило, каждый поток получает этот указатель в виде возвращаемого значения функции malloc. Частично путаница с собственными данными потока обусловлена тем, что указатель в паре ключ-значение играет роль значения, но сами собственные данные потока — это то, на что указывает данный указатель.

Теперь мы перейдем к примеру применения собственных данных потока, предполагая, что наша функция readline использует их для хранения информации о состоянии каждого потока при последовательных обращениях к ней. Вскоре мы покажем код, выполняющий эту задачу, в котором функция readline модифицирована так, чтобы реализовать представленную далее последовательность шагов.

1. Запускается процесс, и создается несколько потоков.

2. Один из потоков вызовет функцию `readline` первой, а та, в свою очередь, вызовет функцию `pthread_key_create`. Система отыщет первую неиспользуемую структуру `Key` (см. рис. 26.2) и возвратит вызывающему процессу ее индекс. В данном примере мы предполагаем, что индекс равен 1.

Мы будем использовать функцию `pthread_once`, чтобы гарантировать, что функция `pthread_key_create` вызывается только первым потоком, вызвавшим функцию `readline`.

3. Функция `readline` вызывает функцию `pthread_getspecific`, чтобы получить значение `rkey[1]` («указатель» на рис. 26.3 для ключа, имеющего значение 1) для данного потока, но эта функция возвращает пустой указатель. Тогда функция `readline` вызывает функцию `malloc` для выделения памяти, которая необходима для хранения информации о каждом потоке при последовательных вызовах функции `readline`. Функция `readline` инициализирует эти области памяти по мере надобности и вызывает функцию `pthread_setspecific`, чтобы установить указатель собственных данных потока (`rkey[1]`), соответствующий данному ключу, на только что выделенную область памяти. Мы показываем этот процесс на рис. 26.4, предполагая, что вызывающий поток — это поток с номером 0 в данном процессе.

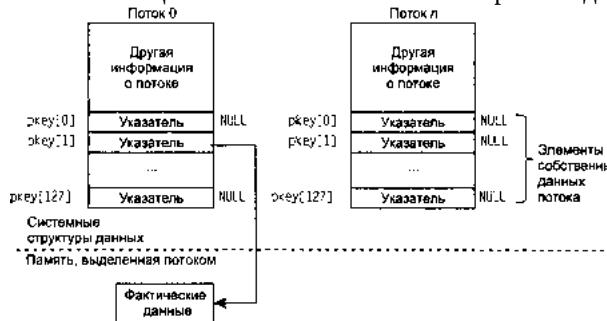


Рис. 26.4. Соответствие между областью памяти, выделенной функцией `malloc`, и указателем собственных данных потока

На этом рисунке мы отмечаем, что структура Pthread поддерживается системой (вероятно, библиотекой потоков), но фактически собственные данные потока, которые мы размещаем в памяти с помощью функции `malloc`, поддерживаются нашей функцией (в данном случае `readline`). Все, что делает функция `pthread_setspecific`, — это установка указателя для данного ключа в структуре Pthread на выделенную область памяти. Аналогично, действие функции `pthread_getspecific` сводится к возвращению этого указателя.

4. Другой поток, например поток с номером n , вызывает функцию `readline`, возможно, в тот момент, когда поток с номером 0 все еще находится в стадии выполнения функции `readline`.

Функция `readline` вызывает функцию `pthread_once`, чтобы инициализировать ключ этого элемента собственных данных, но так как эта функция уже была однажды вызвана, то больше она не выполняется.

5. Функция `readline` вызывает функцию `pthread_getspecific` для получения значения указателя `rkey[1]` для данного потока, но возвращается пустой указатель. Тогда поток вызывает функцию `malloc` и функцию `pthread_setspecific`, как и в случае с потоком номер 0, инициализируя элемент собственных данных потока, соответствующий этому ключу (1). Этот процесс иллюстрирует рис. 26.5.

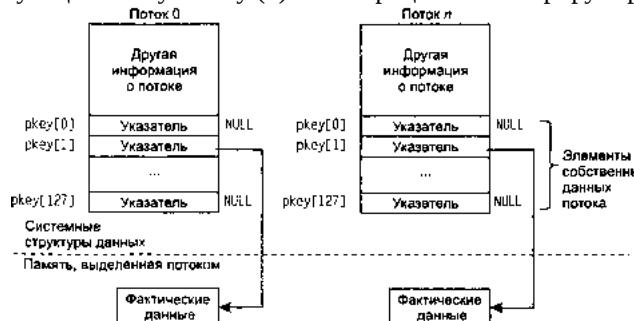


Рис. 26.5. Структуры данных после того, как поток n инициализировал свои собственные данные

6. Поток номер n продолжает выполнять функцию `readline`, используя и модифицируя свои собственные данные.

Один вопрос, который мы пока не рассмотрели, заключается в следующем: что происходит, когда поток завершает свое выполнение? Если поток вызвал функцию `readline`, эта функция выделила в памяти

область, которая должна быть освобождена по завершении выполнения потока. Для этого используется *указатель-деструктор*, показанный на рис. 26.2. Когда поток, создающий элемент собственных данных, вызывает функцию `pthread_key_create`, одним из аргументов этой функции является указатель на *функцию-деструктор*. Когда выполнение потока завершается, система перебирает массив `rkey` для данного потока, вызывая соответствующую функцию-деструктор для каждого непустого указателя `rkey`. Под «соответствующим деструктором» мы понимаем указатель на функцию, хранящийся в массиве `Key` с рис. 26.2. Таким образом осуществляется освобождение памяти, занимаемой собственными данными потока, когда выполнение потока завершается.

Первые две функции, которые обычно вызываются при работе с собственными данными потока, — это `pthread_once` и `pthread_key_create`.

```
#include <pthread.h>
```

```
int pthread_once(pthread_once_t *onceptr, void (*init)(void));
int pthread_key_create(pthread_key_t *keyptr, void (*destructor)(void *value));
```

Обе функции возвращают: 0 в случае успешного выполнения, положительное значение Errxx в случае ошибки

Функция `pthread_once` обычно вызывается при вызове функции, манипулирующей собственными данными потока, но `pthread_once` использует значение переменной, на которую указывает `onceptr`, чтобы гарантировать, что функция `init` вызывается для каждого процесса только один раз.

Функция `pthread_key_create` должна вызываться только один раз для данного ключа в пределах одного процесса. Значение ключа возвращается с помощью указателя `keyptr`, а функция-деструктор (если аргумент является непустым указателем) будет вызываться каждым потоком по завершении его выполнения, если этот поток записывал какое-либо значение, соответствующее этому ключу.

Обычно эти две функции используются следующим образом (если игнорировать возвращение ошибок):

```
pthread_key_t rl_key;
pthread_once_t rl_once = PTHREAD_ONCE_INIT;

void readline_destructor(void *ptr) {
    free(ptr);
}

void readline_once(void) {
    pthread_key_create(&rl_key, readline_destructor);
}

ssize_t readline(...) {
    ...
    pthread_once(&rl_once, readline_once);

    if ((ptr = pthread_getspecific(rl_key)) == NULL) {
        ptr = Malloc(...);
        pthread_setspecific(rl_key, ptr);
        /* инициализация области памяти, на которую указывает ptr */
    }
    ...
    /* используются значения, на которые указывает ptr */
}
```

Каждый раз, когда вызывается функция `readline`, она вызывает функцию `pthread_once`. Эта функция использует значение, на которое указывает ее аргумент-указатель `onceptr` (содержащийся в переменной `rl_once`), чтобы удостовериться, что функция `init` вызывается только один раз. Функция инициализации `readline_once` создает ключ для собственных данных потока, который хранится в `rl_key` и который функция `readline` затем использует в вызовах функций `pthread_getspecific` и `pthread_setspecific`.

Функции `pthread_getspecific` и `pthread_setspecific` используются для того, чтобы получать и задавать значение, ассоциированное с данным ключом. Это значение представляет собой тот указатель,

который показан на рис. 26.3. На что указывает этот указатель — зависит от приложения, но обычно он указывает на динамически выделяемый участок памяти.

```
#include <pthread.h>
```

```
void *pthread_getspecific(pthread_key_t key);
```

Возвращает: указатель на собственные данные потока (возможно, пустой указатель)

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

Возвращает: 0 в случае успешного выполнения, положительное значение Errxx в случае ошибки

Обратите внимание на то, что аргументом функции `pthread_key_create` является указатель на ключ (поскольку эта функция хранит значение, присвоенное ключу), в то время как аргументами функций `get` и `set` являются сами ключи (которые, скорее всего, представляют собой небольшие целые числа, как уже говорилось).

Пример: функция readline, использующая собственные данные потока

В этом разделе мы приводим полный пример использования собственных данных потока, преобразуя оптимизированную версию функции `readline` из листинга 3.12 к виду, безопасному в многопоточной среде, не изменяя последовательность вызовов.

В листинге 26.5 показана первая часть функции: переменные `pthread_key_t` и `pthread_once_t`, функции `readline_destructor` и `readline_once` и наша структура `Rline`, которая содержит всю информацию, нужную нам для каждого потока.

Листинг 26.5. Первая часть функции `readline`, безопасной в многопоточной среде

```
//threads/readline.c
1 #include "unpthread.h"

2 static pthread_key_t rl_key;
3 static pthread_once_t rl_once = PTHREAD_ONCE_INIT;

4 static void
5 readline_destructor(void *ptr)
6 {
7     free(ptr);
8 }

9 static void
10 readline_once(void)
11 {
12     Pthread_key_create(&rl_key, readline_destructor);
13 }

14 typedef struct {
15     int rl_cnt;          /* инициализируется нулем */
16     char *rl_bufptr;    /* инициализируется значением rl_buf */
17     char rl_buf[MAXLINE];
18 } Rline;
```

Деструктор

4-8 Наша функция-деструктор просто освобождает всю память, которая была выделена для данного потока.

«Одноразовая» функция

9-13 Мы увидим, что наша «одноразовая» (то есть вызываемая только один раз) функция вызывается однократно из функции `pthread_once` и создает ключ, который затем используется в функции `readline`.

Структура Rline

14-18 Наша структура `Rline` содержит три переменные, которые, будучи объявленными как статические (`static`) в листинге 3.12, привели к возникновению описанных далее проблем. Такая структура динамически выделяется в памяти для каждого потока, а по завершении выполнения этого потока она освобождается функцией-деструктором.

В листинге 26.6 показана сама функция `readline`, а также функция `my_read`, которую она вызывает. Этот листинг является модификацией листинга 3.12.

Листинг 26.6. Вторая часть функции `readline`, безопасной в многопоточной среде

```
//threads/readline.c
19 static ssize_t
20 my_read(Rline *tsd, int fd, char *ptr)
21 {
22     if (tsd->rl_cnt <= 0) {
23         again:
24         if ((tsd->rl_cnt = read(fd, tsd->rl_buf, MAXLINE)) < 0) {
25             if (errno == EINTR)
26                 goto again;
27             return (-1);
28         } else if (tsd->rl_cnt == 0)
29             return (0);
30         tsd->rl_bufptr = tsd->rl_buf;
31     }
32     tsd->rl_cnt--;
33     *ptr = *tsd->rl_bufptr++;
34     return (1);
35 }

36 ssize_t
37 readline(int fd, void *vptr, size_t maxlen)
38 {
39     int n, rc;
40     char c, *ptr;
41     Rline *tsd;

42     Pthread_once(&rl_once, readline_once);
43     if ((tsd = pthread_getspecific(rl_key)) == NULL) {
44         tsd = Calloc(1, sizeof(Rline)); /* инициализируется нулем */
45         Pthread_setspecific(rl_key, tsd);
46     }

47     ptr = vptr;
48     for (n = 1; n < maxlen; n++) {
49         if ((rc = my_read(tsd, fd, &c)) == 1) {
50             *ptr++ = c;
51             if (c == '\n')
52                 break;
53         } else if (rc == 0) {
54             *ptr = 0;
55             return (n-1); /* EOF, данные не были считаны */
56         } else
57             return (-1); /* ошибка, errno устанавливается функцией read() */
```

```
58     }
59     *ptr = 0;
60     return (n);
61 }
```

Функция *my_read*

19-35 Первым аргументом функции теперь является указатель на структуру Rline, которая была размещена в памяти для данного потока (и содержит собственные данные этого потока).

Размещение собственных данных потока в памяти

42 Сначала мы вызываем функцию *pthread_once*, так чтобы первый поток, вызывающий функцию *readline* в этом процессе, вызвал бы функцию *readline_once* для создания ключа собственных данных потока.

Получение указателя на собственные данные потока

43-46 Функция *pthread_getspecific* возвращает указатель на структуру Rline для данного потока. Но если это первый вызов функции *readline* данным потоком, то возвращаемым значением будет пустой указатель. В таком случае мы выделяем в памяти место для структуры Rline, а элемент *r1_cnt* этой структуры инициализируется нулем с помощью функции *calloc*. Затем мы записываем этот указатель для данного потока, вызывая функцию *pthread_setspecific*. Когда этот поток вызовет функцию *readline* в следующий раз, функция *pthread_getspecific* возвратит этот указатель, который был только что записан.

26.6. Веб-клиент и одновременное соединение (продолжение)

Вернемся к нашему примеру с веб-клиентом из раздела 16.5 и перепишем его с использованием потоков вместо неблокируемой функции *connect*. Мы можем оставить сокеты в их заданном по умолчанию виде — блокирующими, и создать один поток на каждое соединение. Каждый поток может блокироваться в вызове функции *connect*, так как ядро будет просто выполнять какой-либо другой поток, готовый к работе.

В листинге 26.7 показана первая часть нашей программы, глобальные переменные и начало функции *main*.

Листинг 26.7. Глобальные переменные и начало функции *main*

```
//threads/web01.c
1 #include "unpthread.h"
2 #include <thread.h> /* потоки Solaris */

3 #define MAXFILES 20
4 #define SERV      "80" /* номер порта или имя службы */

5 struct file {
6     char      *f_name; /* имя файла */
7     char      *f_host; /* имя узла или IP-адрес */
8     int       f_fd;   /* дескриптор */
9     int       f_flags; /* F_xxx ниже */
10    pthread_t f_tid; /* идентификатор потока */
11 } file[MAXFILES];
12 #define F_CONNECTING 1 /* функция connect() в процессе
                           выполнения */
13 #define F_READING 2    /* функция connect() завершена;
                           выполняется считывание */
```

```

14 #define F_DONE 4      /* все сделано */

15 #define GET_CMD "GET %s HTTP/1.0\r\n\r\n"

16 int nconn, nfiles, nlefttoconn, nlefttoread;

17 void *do_get_read(void*); 
18 void home_page(const char*, const char*); 
19 void write_get_cmd(struct file*); 

20 int
21 main(int argc, char **argv)
22 {
23     int i, n, maxnconn;
24     pthread_t tid;
25     struct file *fptr;

26     if (argc < 5)
27         err_quit("usage: web <#conns> <IPaddr> <homepage> file1 ...");
28     maxnconn = atoi(argv[1]);

29     nfiles = min(argc - 4, MAXFILES);
30     for (i = 0; i < nfiles; i++) {
31         file[i].f_name = argv[i + 4];
32         file[i].f_host = argv[2];
33         file[i].f_flags = 0;
34     }
35     printf("nfiles = %d\n", nfiles);

36     home_page(argv[2], argv[3]);

37     nlefttoread = nlefttoconn = nfiles;
38     nconn = 0;

```

Глобальные переменные

1-16 Мы подключаем заголовочный файл `<thread.h>` в добавок к обычному `<pthread.h>`, так как нам требуется использовать потоки Solaris в дополнение к потокам Pthreads, как мы вскоре покажем.

10 Мы добавили к структуре `file` один элемент — идентификатор потока `f_tid`. Остальная часть этого кода аналогична коду в листинге 16.9. В этой версии нам не нужно использовать функцию `select`, а следовательно, не нужны наборы дескрипторов и переменная `maxfd`.

36 Функция `home_page` не изменилась относительно листинга 16.10. В листинге 26.8 показан основной рабочий цикл потока `main`.

Листинг 26.8. Основной рабочий цикл потока `main`

```

//threads/web01.c
39     while (nlefttoread > 0) {
40         while (nconn < maxnconn && nlefttoconn > 0) {
41             /* находим файл для считывания */
42             for (i = 0; i < nfiles; i++)
43                 if (file[i].f_flags == 0)
44                     break;
45             if (i == nfiles)
46                 err_quit("nlefttoconn = %d but nothing found", nlefttoconn);

47             file[i].f_flags = F_CONNECTING;

```

```

48     Pthread_create(&tid, NULL, &do_get_read, &file[i]);
49     file[i].f_tid = tid;
50     nconn++;
51     nlefttoconn--;
52 }

53     if ((n = thr_join(0, &tid, (void**)&fptr)) != 0)
54         errno = n, err_sys("thr_join error");

55     nconn--;
56     nlefttoread--;
57     printf("thread id %d for %s done\n", tid, fptr->f_name);
58 }

59 exit(0);
60 }

```

По возможности создаем другой поток

40-52 Если имеется возможность создать другой поток (nconn меньше, чем maxconn), мы так и делаем. Функция, которую выполняет каждый новый поток, — это do_get_read, а ее аргументом является указатель на структуру file.

Ждем, когда завершится выполнение какого-либо потока

53-54 Мы вызываем функцию потоков thr_join Solaris с нулевым первым аргументом, чтобы дождаться завершения выполнения какого-либо из наших потоков. К сожалению, в Pthreads не предусмотрен способ, с помощью которого мы могли бы ждать завершения выполнения любого потока, и функция pthread_join требует, чтобы мы точно указали, завершения какого потока мы ждем. В разделе 26.9 мы увидим, что решение этой проблемы в случае применения технологии Pthreads оказывается сложнее и требует использования условной переменной для сообщения главному потоку о завершении выполнения дополнительного потока.

ПРИМЕЧАНИЕ

Показанное здесь решение, в котором используется функция потоков thr_join Solaris, не является, вообще говоря, совместимым со всеми системами. Тем не менее мы приводим здесь эту версию веб-клиента, использующую потоки, чтобы не осложнять обсуждение рассмотрением условных переменных и взаимных исключений (mutex). К счастью, в Solaris допустимо смешивать потоки Pthreads и потоки Solaris.

В листинге 26.9 показана функция do_get_read, которая выполняется каждым потоком. Эта функция устанавливает соединение TCP, посыпает серверу команду HTTP GET и считывает ответ сервера.

Листинг 26.9. Функция do_get_read

```

//threads/web01.c
61 void*
62 do_get_read(void *vptr)
63 {
64     int fd, n;
65     char line[MAXLINE];
66     struct file *fptr;

67     fptr = (struct file*)vptr;

```

```

68 fd = Tcp_connect(fptra->f_host, SERV);
69 fptra->f_fd = fd;
70 printf("do_get_read for %s, fd %d, thread %d\n",
71 fptra->f_name, fd, fptra->f_tid);
72 write_get_cmd(fptra);

73 /* Чтение ответа сервера */
74 for (;;) {
75   if ((n = Read(fd, line, MAXLINE)) == 0)
76     break; /* сервер закрывает соединение */
77   printf ("read %d bytes from %s\n", n, fptra->f_name);
78 }
79 printf("end-of-file on %s\n", fptra->f_name);
80 Close(fd);
81 fptra->f_flags = F_DONE; /* сбрасываем F_READING */

82 return (fptra); /* завершение потока */
83 }

```

Создание сокета TCP, установление соединения

68-71 Создается сокет TCP, и с помощью функции `tcp_connect` устанавливается соединение. В данном случае используется обычный блокируемый сокет, поэтому поток будет блокирован при вызове функции `connect`, пока не будет установлено соединение.

Отправка запроса серверу

72 Функция `write_get_cmd` формирует команду HTTP `GET` и отсылает ее серверу. Мы не показываем эту функцию заново, так как единственным отличием от листинга 16.12 является то, что в версии, использующей потоки, не вызывается макрос `FD_SET` и не используется `maxfd`.

Чтение ответа сервера

73-82 Затем считывается ответ сервера. Когда соединение закрывается сервером, устанавливается флаг `F_DONE` и функция возвращает управление, завершая выполнение потока.

Мы также не показываем функцию `home_page`, так как она полностью повторяет версию, приведенную в листинге 16.10.

Мы вернемся к этому примеру, заменив функцию Solaris `thr_join` на более переносимую функцию семейства Pthreads, но сначала нам необходимо обсудить взаимные исключения и условные переменные.

26.7. Взаимные исключения

Обратите внимание на то, что в листинге 26.8 при завершении выполнения очередного потока в главном цикле уменьшаются на единицу и `nconn`, и `nlefttoread`. Мы могли бы поместить оба эти оператора уменьшения в одну функцию `do_get_read`, что позволило бы каждому потоку уменьшать эти счетчики непосредственно перед тем, как выполнение потока завершается. Но это привело бы к возникновению трудноуловимой серьезной ошибки параллельного программирования.

Проблема, возникающая при помещении определенного кода в функцию, которая выполняется каждым потоком, заключается в том, что обе эти переменные являются глобальными, а не собственными переменными потока. Если один поток в данный момент уменьшает значение переменной и это действие приостанавливается, чтобы выполнился другой поток, который также станет уменьшать на единицу эту переменную, может произойти ошибка. Предположим, например, что компилятор C осуществляет

уменьшение переменной на единицу в три этапа: загружает информацию из памяти в регистр, уменьшает значение регистра, а затем сохраняет значение регистра в памяти. Рассмотрим возможный сценарий.

1. Выполняется поток А, который загружает в регистр значение переменной `nconn` (равное 3).
2. Система переключается с выполнения потока А на выполнение потока В. Регистры потока А сохранены, регистры потока В восстановлены.
3. Поток В выполняет три действия, составляющие оператор декремента в языке С (`nconn--`), сохранив новое значение переменной `nconn`, равное 2.
4. Впоследствии в некоторый момент времени система переключается на выполнение потока А. Восстанавливаются регистры потока А, и он продолжает выполняться с того места, на котором остановился, а именно начиная со второго этапа из трех, составляющих оператор декремента. Значение регистра уменьшается с 3 до 2, и значение 2 записывается в переменную `nconn`.

Окончательный результат таков: значение `nconn` равно 2, в то время как оно должно быть равным 1. Это ошибка.

Подобные ошибки параллельного программирования трудно обнаружить по многим причинам. Во-первых, они возникают нечасто. Тем не менее это ошибки, которые по закону Мэрфи вызывают сбои в работе программ. Во-вторых, ошибки такого типа возникают не систематически, так как зависят от недетерминированного совпадения нескольких событий. Наконец, в некоторых системах аппаратные команды могут быть атомарными. Это значит, что имеется аппаратная команда уменьшения значения целого числа на единицу (вместо трехступенчатой последовательности, которую мы предположили выше), а аппаратная команда не может быть прервана до окончания своего выполнения. Но это не гарантировано для всех систем, так что код может работать в одной системе и не работать в другой.

Программирование с использованием потоков является *параллельным (parallel)*, или *одновременным (concurrent)*, программированием, так как несколько потоков могут выполняться параллельно (одновременно), получая доступ к одним и тем же переменным. Хотя ошибочный сценарий, рассмотренный нами далее, предполагает систему с одним центральным процессором, вероятность ошибки также присутствует, если потоки А и В выполняются в одно и то же время на разных процессорах в многопроцессорной системе. В обычном программировании под Unix мы не сталкиваемся с подобными ошибками, так как при использовании функции `fork` родительский и дочерний процессы не используют совместно ничего, кроме дескрипторов. Тем не менее мы столкнемся с ошибками этого типа при обсуждении совместного использования памяти несколькими процессами.

Эту проблему можно с легкостью продемонстрировать на примере потоков. В листинге 26.11 показана программа, которая создает два потока, после чего каждый поток увеличивает некоторую глобальную переменную 5000 раз.

Мы повысили вероятность ошибки за счет того, что потребовали от программы получить текущее значение переменной `counter`, вывести это значение и записать его. Если мы запустим эту программу, то получим результат, представленный в листинге 26.10.

Листинг 26.10. Результат выполнения программы, приведенной в листинге 26.11

```
4: 1
4: 2
4: 3
4: 4
    продолжение выполнения потока номер 4
4: 517
4: 518
5: 518 теперь выполняется поток номер 5
5: 519
5: 520
    продолжение выполнения потока номер 5
5: 926
5: 927
4: 519 теперь выполняется поток номер 4, записывая неверные значения
4: 520
```

Листинг 26.11. Два потока, которые неверно увеличивают значение глобальной переменной

```
//threads/example01.c
```

```
1 #include "unpthread.h"
```

```

2 #define NLOOP 5000

3 int counter; /* потоки должны увеличивать значение этой переменной */

4 void *doit(void*);

5 int
6 main(int argc, char **argv)
7 {
8     pthread_t tidA, tidB;

9     Pthread_create(&tidA, NULL, &doit, NULL);
10    Pthread_create(&tidB, NULL, &doit, NULL);

11    /* ожидание завершения обоих потоков */
12    Pthread_join(tidA, NULL);
13    Pthread_join(tidB, NULL);

14    exit(0);
15 }

16 void*
17 doit(void *vptr)
18 {
19     int i, val;

20     /* Каждый поток получает, выводит и увеличивает на
21      * единицу переменную counter NLOOP раз. Значение
22      * переменной должно увеличиваться монотонно.
23      */
24
25     for (i = 0; i < NLOOP; i++) {
26         val = counter;
27         printf("%d: %d\n", pthread_self(), val + 1);
28         counter = val + 1;
29     }
30 }
```

Обратите внимание на то, что в первый раз ошибка происходит при переключении системы с выполнения потока номер 4 на выполнение потока номер 5: каждый поток в итоге записывает значение 518. Это происходит множество раз на протяжении 10 000 строк вывода.

Недетерминированная природа ошибок такого типа также будет очевидна, если мы запустим программу несколько раз: каждый раз результат выполнения программы будет отличаться от предыдущего. Также, если мы переадресуем вывод результатов в файл на диск, эта ошибка иногда не будет возникать, так как программа станет работать быстрее, что приведет к уменьшению вероятности переключения системы между потоками. Наибольшее количество ошибок возникнет в случае, если программа будет работать интерактивно, записывая результат на медленный терминал, но при этом также сохраняя результат в файл при помощи программы Unix script (которая описана в главе 19 книги [110]).

Только что описанная проблема, возникающая, когда несколько потоков изменяют значение одной переменной, является самой простой из проблем параллельного программирования. Для решения этой проблемы используются так называемые *взаимные исключения* (*mutex — mutual exclusion*), с помощью которых контролируется доступ к переменной. В терминах Pthreads взаимное исключение — это переменная типа *pthread_mutex_t*, которая может быть заблокирована и разблокирована с помощью следующих двух функций:

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mptr);
int pthread_mutex_unlock(pthread_mutex_t *mptr);
Обе функции возвращают: 0 в случае успешного выполнения, положительное значение Exxx в
случае ошибки
```

Если некоторый поток попытается блокировать взаимное исключение, которое уже блокировано каким-либо другим потоком (то есть принадлежит ему в данный момент времени), этот поток окажется заблокированным до освобождения взаимного исключения.

Если переменная-исключение размещена в памяти статически, следует инициализировать ее константой PTHREAD_MUTEX_INITIALIZER. В разделе 30.8 мы увидим, что если мы размещаем исключение в совместно используемой (разделяемой) памяти, мы должны инициализировать его во время выполнения программы путем вызова функции pthread_mutex_init.

ПРИМЕЧАНИЕ

Некоторые системы (например, Solaris) определяют константу PTHREAD_MUTEX_INITIALIZER как 0. Если данная инициализация будет опущена, это ни на что не повлияет, так как статически размещаемые переменные все равно автоматически инициализируются нулем. Но для других систем такой гарантии дать нельзя — например, в Digital Unix константа инициализации ненулевая.

В листинге 26.12 приведена исправленная версия листинга 26.11, в которой используется одно взаимное исключение для блокирования счетчика при работе с двумя потоками.

Листинг 26.12. Исправленная версия листинга 26.11, использующая взаимное исключение для защиты совместно используемой переменной

```
//threads/example01.c
1 #include "unpthread.h"

2 #define NLOOP 5000

3 int counter; /* увеличивается потоками */
4 pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;

5 void *doit(void*);

6 int
7 main(int argc, char **argv)
8 {
9     pthread_t tidA, tidB;

10    Pthread_create(&tidA, NULL, &doit, NULL);
11    Pthread_create(&tidB, NULL, &doit, NULL);

12    /* ожидание завершения обоих потоков */
13    Pthread_join(tidA, NULL);
14    Pthread_join(tidB, NULL);

15    exit(0);
16 }

17 void*
18 doit(void *vptr)
19 {
20     int i, val;
```

```

21  /*
22   * Каждый поток считывает, выводит и увеличивает счетчик NLOOP раз.
23   * Значение счетчика должно возрастать монотонно.
24  */
25 for (i = 0; i < NLOOP; i++) {
26     Pthread_mutex_lock(&counter_mutex);
27
28     val = counter;
29     printf("%d\n", pthread_self(), val + 1);
30     counter = val + 1;
31 }
32 return(NULL);
33 }
```

Мы объявляем взаимное исключение с именем `counter_mutex`. Это исключение должно быть заблокировано потоком на то время, когда он манипулирует переменной `counter`. Когда мы запускали эту программу, результат всегда был правильным: значение переменной увеличивалось монотонно, а ее окончательное значение всегда оказывалось равным 10 000.

Насколько серьезной является дополнительная нагрузка, связанная с использованием взаимных исключений? Мы изменили программы, приведенные в листингах 26.11 и 26.12, заменив значение `NLOOP` на 50 000 (вместо исходного значения 5000), и засекли время, направив вывод на устройство `/dev/null`. Время работы центрального процессора в случае корректной версии, использующей взаимное исключение, увеличилось относительно времени работы некорректной версии без взаимного исключения на 10 %. Это означает, что использование взаимного исключения не связано со значительными издержками.

26.8. Условные переменные

Взаимное исключение позволяет предотвратить одновременный доступ к совместно используемой (разделяемой) переменной, но для того чтобы перевести поток в состояние ожидания (спящее состояние) до момента выполнения некоторого условия, необходим другой механизм. Продемонстрируем сказанное на следующем примере. Вернемся к нашему веб-клиенту из раздела 26.6 и заменим функцию Solaris `thr_join` на `pthread_join`. Но мы не можем вызвать функцию `pthread_join` до тех пор, пока не будем знать, что выполнение потока завершилось. Сначала мы объявляем глобальную переменную, которая служит счетчиком количества завершившихся потоков, и организуем управление доступом к ней с помощью взаимного исключения.

```

int ndone; /* количество потоков, завершивших выполнение */
pthread_mutex_t ndone_mutex = PTHREAD_MUTEX_INITIALIZER;

void* do_get_read(void *vptr) {
    ...
    Pthread_mutex_lock(&ndone_mutex);
    ndone++;
    Pthread_mutex_unlock(&ndone_mutex);

    return(fptr); /* завершение выполнения потока */
}
```

Но каким при этом получается основной цикл? Взаимное исключение должно быть постоянно блокировано основным циклом, который проверяет, какие потоки завершили свое выполнение.

```

while (nlefttoread > 0) {
    while (nconn < maxnconn && nlefttoconn > 0) {
        /* находим файл для чтения */
```

```

    ...
}

/* Проверяем, не завершен ли поток */
Pthread_mutex_lock(&ndone_mutex);
if (ndone > 0) {
    for (i = 0; i < nfiles; i++) {
        if (file[i].f_flags & F_DONE) {
            Pthread_join(file[i].f_tid, (void**)&fptr);
            /* обновляем file[i] для завершенного потока */
            ...
        }
    }
    Pthread_mutex_unlock(&ndone_mutex);
}

```

Это означает, что главный поток *никогда* не переходит в спящее состояние, а просто входит в цикл, проверяя каждый раз значение переменной `ndone`. Этот процесс называется *опросом (polling)* и рассматривается как пустая траты времени центрального процессора.

Нам нужен метод, с помощью которого главный цикл мог бы входить в состояние ожидания, пока один из потоков не оповестит его о том, что какая-либо задача выполнена. Эта возможность обеспечивается использованием *условной переменной (conditional variable)* вместе со взаимным исключением. Взаимное исключение используется для реализации блокирования, а условная переменная обеспечивает сигнальный механизм.

В терминах Pthreads условная переменная — это переменная типа `pthread_cond_t`. Такие переменные используются в следующих двух функциях:

```
#include <pthread.h>
```

```
int pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);
int pthread_cond_signal(pthread_cond_t *cptr);
```

Обе функции возвращают: 0 в случае успешного выполнения, положительное значение Errxx в случае ошибки

Слово `signal` в названии второй функции не имеет отношения к сигналам Unix `SIGxxx`.

Проще всего объяснить действие этих функций на примере. Вернемся к нашему примеру веб-клиента. Счетчик `ndone` теперь ассоциируется и с условной переменной, и с взаимным исключением:

```
int ndone;
pthread_mutex_t ndone_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t ndone_cond = PTHREAD_COND_INITIALIZER;
```

Поток оповещает главный цикл о своем завершении, увеличивая значение счетчика, пока взаимное исключение принадлежит данному потоку (блокировано им), и используя условную переменную для сигнализации.

```
Pthread_mutex_lock(&ndone_mutex);
ndone++;
Pthread_cond_signal(&ndone_cond);
Pthread_mutex_unlock(&ndone_mutex);
```

Затем основной цикл блокируется в вызове функции `pthread_cond_wait`, ожидая оповещения о завершении выполнения потока:

```
while (nlefttoread > 0) {
    while (nconn < maxnconn && nlefttoconn > 0) {
        /* находим файл для чтения */
        ...
    }

    /* Ждем завершения выполнения какого-либо потока */
    Pthread_mutex_lock(&ndone_mutex);
    while (ndone == 0)
        Pthread_cond_wait(&ndone_cond, &ndone_mutex);
```

```

for (i = 0; i < nfiles; i++) {
    if (file[i].f_flags & F_DONE) {
        Pthread_join(file[i].f_tid, (void**)&fptr);

        /* обновляем file[i] для завершенного потока */
        ...
    }
}
Pthread_mutex_unlock(&ndone_mutex);
}

```

Обратите внимание на то, что переменная `ndone` по-прежнему проверяется, только если потоку принадлежит взаимное исключение. Тогда, если не требуется выполнять какое-либо действие, вызывается функция `pthread_cond_wait`. Таким образом, вызывающий поток переходит в состояние ожидания, и разблокируется взаимное исключение, которое принадлежало этому потоку. Кроме того, когда управление возвращается потоку функцией `pthread_cond_wait` (после того как поступил сигнал от какого-либо другого потока), он снова блокирует взаимное исключение.

Почему взаимное исключение всегда связано с условной переменной? «Условие» обычно представляет собой значение некоторой переменной, используемой совместно несколькими потоками. Взаимное исключение требуется для того, чтобы различные потоки могли задавать и проверять значение условной переменной. Например, если в примере кода, приведенном ранее, отсутствовало бы взаимное исключение, то проверка в главном цикле выглядела бы следующим образом:

```

/* Ждем завершения выполнения одного или нескольких потоков */
while (ndone == 0)
    Pthread_cond_wait(&ndone_cond, &ndone_mutex);

```

Но при этом существует вероятность, что последний поток увеличивает значение переменной `ndone` после проверки главным потоком условия `ndone == 0`, но перед вызовом функции `pthread_cond_wait`. Если это происходит, то последний «сигнал» теряется, и основной цикл оказывается заблокированным навсегда, так как он будет ждать события, которое никогда не произойдет.

По этой же причине при вызове функции `pthread_cond_wait` поток должен блокировать соответствующее взаимное исключение, после чего эта функция разблокирует взаимное исключение и помещает вызывающий поток в состояние ожидания, выполняя эти действия как одну атомарную операцию. Если бы эта функция не разблокировала взаимное исключение и не заблокировала его снова после своего завершения, то выполнять эти операции пришлось бы потоку, как показано в следующем фрагменте кода:

```

/* Ждем завершения выполнения одного или нескольких потоков */
Pthread_mutex_lock(&ndone_mutex);
while (ndone == 0) {
    Pthread_mutex_unlock(&ndone_mutex);
    Pthread_cond_wait(&ndone_cond, &ndone_mutex);
    Pthread_mutex_lock(&ndone_mutex);
}

```

Существует вероятность того, что по завершении выполнения поток увеличит на единицу значение переменной `ndone` и это произойдет между вызовом функций `pthread_mutex_unlock` и `pthread_cond_wait`.

Обычно функция `pthread_cond_signal` выводит из состояния ожидания один поток, на который указывает условная переменная. Существуют ситуации, когда некоторый поток знает, что из состояния ожидания должны быть выведены несколько потоков. В таком случае используется функция `pthread_cond_broadcast`, выводящая из состояния ожидания все потоки, которые блокированы условной переменной.

```

#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cptr);
int pthread_cond_timedwait(pthread_cond_t *cptr, pthread_mutex_t *mptr,
    const struct timespec *abstime);

```

Обе функции возвращают: 0 в случае успешного выполнения, положительное значение Errxx в случае ошибки

Функция `pthread_cond_timedwait` позволяет потоку задать предельное время блокирования. Аргумент `abstime` представляет собой структуру `timespec` (определенную в разделе 6.9 при рассмотрении функции `pselect`), которая задает системное время для момента, когда функция должна возвратить управление, даже если к этому моменту условная переменная не подала сигнал. Если возникает такая ситуация, возвращается ошибка `ETIME`.

В данном случае значение времени является *абсолютным значением времени*, в отличие от относительного значения *разницы во времени* (*time delta*) между некоторыми событиями. Иными словами, `abstime` — это системное время, то есть количество секунд и наносекунд, прошедших с 1 января 1970 года (UTC) до того момента, когда эта функция должна вернуть управление. Здесь имеется различие как с функцией `pselect`, так и с функцией `select`, задающими количество секунд (и наносекунд в случае `pselect`) до некоторого момента в будущем, когда функция должна вернуть управление. Обычно для этого вызывается функция `gettimeofday`, которая выдает текущее время (в виде структуры `timeval`), а затем оно копируется в структуру `timespec` и к нему добавляется требуемое значение:

```
struct timeval tv;
struct timespec ts;

if ( gettimeofday(&tv, NULL) < 0)
    err_sys("gettimeofday error");
ts.tv_sec = tv.tv_sec + 5; /* 5 с в будущем */
ts.tv_nsec = tv.tv_usec * 1000; /* микросекунды переводим в наносекунды */

pthread_cond_timedwait( , &ts);
```

Преимущество использования абсолютного времени (в противоположность относительному) заключается в том, что функция может завершиться раньше (возможно, из-за перехваченного сигнала). Тогда функцию можно вызвать снова, не меняя содержимое структуры `timespec`. Недостаток этого способа заключается в необходимости вызывать дополнительно функцию `gettimeofday` перед тем, как в первый раз вызывать функцию `pthread_cond_timedwait`.

ПРИМЕЧАНИЕ

В POSIX определена новая функция `clock_gettime`, возвращающая текущее время в виде структуры `timespec`.

26.9. Веб-клиент и одновременный доступ

Изменим код нашего веб-клиента из раздела 26.6: уберем вызов функции Solaris `thr_join` и заменим его вызовом функции `pthread_join`. Как сказано в разделе 26.6, теперь нам нужно точно указать, завершения какого потока мы ждем. Для этого мы используем условную переменную, описанную в разделе 26.8.

Единственным изменением в отношении глобальных переменных (см. листинг 26.7) является добавление нового флага и условной переменной:

```
#define F_JOINED 8 /* количество потоков */

int ndone; /* количество завершившихся потоков */
```

```
pthread_mutex_t ndone_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_cond_t ndone_cond = PTHREAD_COND_INITIALIZER;
```

Единственным изменением функции `do_get_read` (см. листинг 26.9) будет увеличение на единицу

значения переменной `ndone` и оповещение главного цикла о завершении выполнения потока:

```
printf("end-of-file on %s\n", fptr->f_name);
Close(fd);

Pthread_mutex_lock(&ndone_mutex);
fptr->f_flags = F_DONE; /* сбрасывает флаг F_READING */
ndone++;
Pthread_cond_signal(&ndone_cond);
```

```
Pthread_mutex_unlock(&ndone_mutex);

return(fptr); /* завершение выполнения потока */
}
```

Большинство изменений касается главного цикла, представленного в листинге 26.8. Новая версия показана в листинге 26.13.

Листинг 26.13. Основной рабочий цикл функции main

```
//threads/web03.c
43 while (nlefttoread > 0) {
44     while (nconn < maxnconn && nlefttoconn > 0) {
45         /* находим файл для считывания */
46         for (i = 0; i < nfiles; i++)
47             if (file[i].f_flags == 0)
48                 break;
49         if (i == nfiles)
50             err_quit("nlefttoconn = %d but nothing found", nlefttoconn);

51         file[i].f_flags = F_CONNECTING;
52         Pthread_create(&tid, NULL, &do_get_read, &file[i]);
53         file[i].f_tid = tid;
54         nconn++;
55         nlefttoconn--;
56     }

57     /* Ждем завершения выполнения одного из потоков */
58     Pthread_mutex_lock(&ndone_mutex);
59     while (ndone == 0)
60         Pthread_cond_wait(&ndone_cond, &ndone_mutex);

61     for (i = 0; i < nfiles; i++) {
62         if (file[i].f_flags & F_DONE) {
63             Pthread_join(file[i].f_tid, (void**)&fptr);

64             if (&file[i] != fptr)
65                 err_quit("file[i] != fptr");
66             fptr->f_flags = F_JOINED; /* clears F_DONE */
67             ndone--;
68             nconn--;
69             nlefttoread--;
70             printf("thread %d for %s done\n", fptr->f_tid, fptr->f_name);
71         }
72     }
73     Pthread_mutex_unlock(&ndone_mutex);
74 }

75 exit(0);
76 }
```

По возможности создаем новый поток

44-56 Эта часть кода не изменилась.

Ждем завершения выполнения потока

57-60 Мы ждем завершения выполнения потоков, отслеживая, когда значение `ndone` станет равно нулю. Как сказано в разделе 26.8, эта проверка должна быть проведена перед тем, как взаимное исключение будет блокировано, а переход потока в состояние ожидания осуществляется функцией `pthread_cond_wait`.

Обработка завершенного потока

61-73 Когда выполнение потока завершилось, мы перебираем все структуры `file`, отыскивая соответствующий поток, вызываем `pthread_join`, а затем устанавливаем новый флаг `F_JOINED`.

В табл. 16.1 показано, сколько времени требует выполнение этой версии веб-клиента, а также версии, использующей неблокируемую функцию `connect`.

26.10. Резюме

Создание нового потока обычно требует меньше времени, чем порождение нового процесса с помощью функции `fork`. Одно это уже является большим преимуществом использования потоков на активно работающих сетевых серверах. Многопоточное программирование, однако, представляет собой отдельную технологию, требующую большей аккуратности при использовании.

Все потоки одного процесса совместно используют глобальные переменные и дескрипторы, тем самым эта информация становится доступной всем потокам процесса. Но совместное использование информации вносит проблемы, связанные с синхронизацией доступа к разделяемым переменным, и поэтому нам следует использовать примитивы синхронизации технологии Pthreads — взаимные исключения и условные переменные. Синхронизация доступа к совместно используемым данным — необходимое условие почти для любого приложения, работающего с потоками.

При разработке функций, которые могут быть вызваны таким приложением, нужно учитывать требование безопасности в многопоточной среде. Это требование выполнимо при использовании собственных данных потоков (thread-specific data), пример которых мы показали при рассмотрении функции `readline` в этой главе.

К модели потоков мы вернемся в главе 30, где сервер при запуске создает пул потоков. Для обслуживания очередного клиентского запроса используется любой свободный поток.

Упражнения

1. Сравните использование дескриптора в случае, когда в коде сервера применяется функция `fork`, и в случае, когда используются потоки. Предполагается, что одновременно обрабатывается 100 клиентов.

2. Что произойдет в листинге 26.2, если поток при завершении функции `str_echo` не вызовет функцию `close` для закрытия сокета?

3. В листингах 5.4 и 6.2 мы выводили сообщение `Server terminated prematurely` (Сервер завершил работу преждевременно), когда мы ждали от сервера прибытия отраженной строки, а вместо этого получали признак конца файла (см. раздел 5.12). Модифицируйте листинг 26.1 таким образом, чтобы в соответствующих случаях также выдавалось аналогичное сообщение.

4. Модифицируйте листинги 26.5 и 26.6 таким образом, чтобы программы можно было компилировать в системах, не поддерживающих потоки.

5. Чтобы увидеть ошибку в функции `readline`, приведенной в листинге 26.2, запустите эту программу на стороне сервера. Затем измените эхо-клиент TCP из листинга 6.2, корректно работающий в пакетном режиме. Возьмите какой-либо большой текстовый файл в своей системе и трижды запустите клиент в пакетном режиме, чтобы он считывал текст из этого файла и записывал результат во временный файл. Если есть возможность, запустите клиенты на другом узле (не на том, на котором запущен сервер). Если все три клиента выполняют работу правильно (часто они зависают), посмотрите на файлы с результатом и сравните их с исходным файлом.

Теперь создайте версию сервера, используя корректную версию функции `readline` из раздела 26.5. Повторите тест, используя три эхо-клиента. Теперь все три клиента должны работать исправно. Также поместите функцию `printf` в функции `readline_destructor`, `readline_once` и в вызов функции `malloc` в

`readline`. Это даст вам возможность увидеть, что ключ создается только один раз, но для каждого потока выделяется область памяти и вызывается функция-деструктор.

Глава 27

Параметры IP

27.1. Введение

В IPv4 допускается, чтобы после фиксированного 20-байтового заголовка шли до 40 байт, отведенных под различные параметры. Хотя всего определено десять параметров, чаще всего используется параметр маршрута от отправителя (*source route option*). Доступ к этим параметрам осуществляется через параметр сокета `IP_OPTIONS`, что мы покажем именно на примере использования маршрутизации от отправителя.

В IPv6 допускается наличие расширяющих заголовков (extension headers) между фиксированным 40-байтовым заголовком IPv6 и заголовком транспортного уровня (например, ICMPv6, TCP или UDP). В настоящее время определены 6 различных расширяющих заголовков. В отличие от подхода, использованного в IPv4, доступ к расширяющим заголовкам IPv6 осуществляется через функциональный интерфейс, что не требует от пользователя понимания фактических деталей того, как именно эти заголовки расположены в пакете IPv6.

27.2. Параметры IPv4

На рис. A.1 мы показываем параметры, расположенные после 20-байтового заголовка IPv4. Как отмечено при рассмотрении этого рисунка, 4-разрядное поле длины ограничивает общий размер заголовка IPv4 до 15 32-разрядных слов (что составляет 60 байт), так что на параметры IPv4 остается 40 байт. Для IPv4 определено 10 различных параметров.

1. NOP (no-operation — нет действий). Этот однобайтовый параметр используется для выравнивания очередного параметра по 4-байтовой границе.
2. EOL (end-of-list — конец списка параметров). Этот однобайтовый параметр обозначает конец списка параметров. Поскольку суммарный размер параметров IP должен быть кратным 4 байтам, после последнего параметра добавляются байты EOL.
3. LSRR (Loose Source and Record Route — гибкая маршрутизация от отправителя с записью) (см. раздел 8.5 [111]). Пример использования этого параметра мы вскоре продемонстрируем.
4. SSRR (Strict Source and Record Route — жесткая маршрутизация от отправителя с записью) (см. раздел 8.5 [111]). Пример использования этого параметра мы также вскоре продемонстрируем.
5. Отметка времени (timestamp) (см. раздел 7.4 [111]).
6. Запись маршрута (record route) (см. раздел 7.3 [111]).
7. Основной параметр обеспечения безопасности (устаревший параметр) (basic security).
8. Расширенный параметр обеспечения безопасности (устаревший параметр) (extended security).
9. Идентификатор потока (устаревший параметр) (stream identifier).
10. Извещение маршрутизатора (router alert). Этот параметр описан в RFC 2113 [60]. Он включается в дейтаграмму IP, для того чтобы все пересылающие эту дейтаграмму маршрутизаторы обрабатывали ее содержимое.

В главе 9 книги [128] приводится более подробное рассмотрение первых шести параметров, а в указанных далее разделах [111] имеются примеры их использования.

Функции `getsockopt` и `setsockopt` (с аргументом `level`, равным `IPPROTO_IP`, а аргументом `optname` — `IP_OPTIONS`) предназначены соответственно для получения и установки параметров IP. Четвертый аргумент функций `getsockopt` и `setsockopt` — это указатель на буфер (размер которого не превосходит 44 байт), а пятый аргумент — это размер буфера. Причина, по которой размер буфера может на 4 байт превосходить максимальный суммарный размер параметров, заключается в способе обработки параметров маршрута от отправителя, как мы вскоре увидим. Все остальные параметры помещаются в буфер именно в том виде, в котором они потом упаковываются в заголовок дейтаграммы.

Когда параметры IP задаются с использованием функции `setsockopt`, указанные параметры включаются во все дейтаграммы, отсылаемые с данного сокета. Этот принцип работает для сокетов TCP, UDP и для символьных сокетов. Для отмены какого-либо параметра следует вызвать функцию `setsockopt` и задать либо пустой указатель в качестве четвертого аргумента, либо нулевое значение в качестве пятого аргумента (длина).

ПРИМЕЧАНИЕ

Установка параметров IP для символьного сокета IP работает не во всех реализациях, если уже установлен параметр IP_HDRINCL (который мы обсудим в последующих главах). Многие Беркли-реализации не отсылают параметры, установленные с помощью IP_OPTIONS, если включен параметр IP_HDRINCL, так как приложение может устанавливать собственные параметры в формируемом им заголовке IP [128, с. 1056–1057]. В других системах (например, в FreeBSD) приложение может задавать свои параметры IP, либо используя параметр сокета IP_OPTIONS, либо установив параметр IP_HDRINCL и включив требуемые параметры в создаваемый им заголовок IP, но одновременное применение обоих этих способов не допускается.

При вызове функции `getsockopt` для получения параметров IP присоединенного сокета TCP, созданного функцией `accept`, возвращается лишь обращенный параметр маршрута от отправителя, полученный вместе с клиентским сегментом SYN на прослушиваемом сокете [128, с. 931]. TCP автоматически обращает маршрут от отправителя, поскольку маршрут, указанный клиентом, — это маршрут от клиента к серверу, а сервер должен использовать для отсылаемых им дейтаграмм обратный маршрут. Если вместе с сегментом SYN не был получен маршрут от отправителя, то значение пятого аргумента (этот аргумент типа «значение-результат», как было указано ранее, задает длину буфера) при завершении функции `getsockopt` будет равно нулю. Для прочих сокетов TCP, всех сокетов UDP и всех символьных сокетов IP при вызове функции `getsockopt` вы просто получите копию тех параметров IP, которые были установлены для этих сокетов с помощью функции `setsockopt`. Заметим, что для символьных сокетов IP полученный заголовок IP, включая все параметры IP, всегда возвращается всеми входными функциями, поэтому полученные параметры IP всегда доступны.

ПРИМЕЧАНИЕ

В Беркли-ядрах полученный маршрут от отправителя, так же как и другие параметры IP, никогда не возвращается для сокетов UDP. Показанный на с. 775 [128] код, предназначенный для получения параметров IP, существовал со времен 4.3BSD Reno, но так как он не работал, его всегда приходилось превращать в комментарий. Таким образом, для сокетов UDP невозможно использовать обращенный маршрут от отправителя полученной дейтаграммы, чтобы отослать ответ.

27.3. Параметры маршрута от отправителя IPv4

Маршрут от отправителя (*source route*) — это список IP-адресов, указанных отправителем дейтаграммы IP. Если маршрут является *жестким* (*строгим, strict*), то дейтаграмма должна передаваться только между указанными узлами и пройти их все. Иными словами, все узлы, перечисленные в маршруте от отправителя, должны быть соседними друг для друга. Но если маршрут является *свободным*, или *гибким* (*loose*), дейтаграмма должна пройти все перечисленные в нем узлы, но может при этом пройти и еще какие-то узлы, не входящие в список.

ПРИМЕЧАНИЕ

Маршрутизация от отправителя (*source routing*) в IPv4 является предметом споров и сомнений. В [20] пропагандируется отказ от поддержки этой функции на всех маршрутизаторах, и многие организации и провайдеры действительно следуют этому принципу. Один из наиболее разумных способов использования маршрутизации от отправителя — это обнаружение с помощью программы traceroute асимметричных маршрутов, как показано на с. 108–109 [111], но в настоящее время даже этот способ становится непопулярен. Тем не менее определение и получение маршрута от отправителя — это часть API сокетов, и поэтому заслуживает описания.

Параметры IPv4, связанные с маршрутизацией от отправителя, называются *параметрами маршрутизации от отправителя с записью* (Loose Source and Record Routes — LSRR в случае свободной маршрутизации и Strict Source and Record Routes — SSRR в случае жесткой маршрутизации), так как при проходе дейтаграммы через каждый из перечисленных в списке узлов происходит замена указанного адреса на адрес интерфейса для исходящих дейтаграмм. Это позволяет получателю дейтаграммы обратить полученный список, превратив его в маршрут, по которому будет послан ответ отправителю. Примеры этих двух маршрутов от отправителя вместе с соответствующим выводом программы `tcpdump`, приведены в разделе 8.5 книги [111].

Маршрут от отправителя мы определяем как массив адресов IPv4, которому предшествуют три однобайтовых поля, как показано на рис. 27.1. Это формат буфера, который передается функции `setsockopt`.

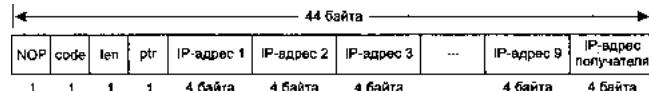


Рис. 27.1. Передача маршрута от отправителя ядру

Перед параметром маршрута от отправителя мы поместили параметр NOP (нет действий), чтобы все IP-адреса были выровнены по 4-байтовой границе. Это не обязательно, но желательно, поскольку в результате мы выравниваем адреса, не расходуя дополнительной лишней памяти (все IP-параметры обычно выравниваются, чтобы в итоге занимать место, кратное 4 байтам).

На рис. 27.1 показано, что маршрут состоит из 10 адресов, но первый приведенный адрес удаляется из параметра маршрута от отправителя и становится адресом получателя, когда дейтаграмма IP покидает узел отправителя. Хотя в 40-байтовом пространстве, отведенном под данный параметр IP, хватает места только для 9 адресов (не забудьте о 3-байтовом заголовке параметра, который мы вскоре опишем), фактически в заголовке IPv4 у нас имеется 10 IP-адресов, так как к 9 адресам узлов добавляется адрес получателя.

Поле `code` — это либо `0x83` для параметра LSRR, либо `0x89` для параметра SSRR. Задаваемое нами значение поля `len` — это размер параметра в байтах, включая 3-байтовый заголовок и дополнительный адрес получателя, приведенный в конце списка. Для маршрута, состоящего из одного IP-адреса, это значение будет равно 11, для двух адресов — 15, и т.д. вплоть до максимального значения 43. Параметр NOP не является частью обсуждаемого параметра, и его длина не включается в значение поля `len`, но она входит в размер буфера, который мы сообщаем функции `setsockopt`. Когда первый адрес в списке удаляется из параметра маршрута от отправителя и добавляется в поле адреса получателя в заголовок IP, значение поля `len` уменьшается на 4 (см. рис. 9.32 и 9.33 [128]). Поле `ptr` — это указатель, или сдвиг, задающий положение следующего IP-адреса из списка, который должен быть обработан. Мы инициализируем это поле значением 4, что соответствует первому адресу IP. Значение этого поля увеличивается на 4 каждый раз, когда дейтаграмма обрабатывается одним из перечисленных в маршруте узлов.

Теперь мы переходим к определению трех функций, с помощью которых мы инициализируем, создаем и обрабатываем параметр маршрута от отправителя. Наши функции предназначены для работы только с этим параметром. Хотя в принципе возможно объединить параметр маршрута от отправителя с другими параметрами IP (такими как параметр извещения маршрутизатора), но на практике параметры редко комбинируются. В листинге 27.1^[1] приведена функция `inet_srcrt_init`, а также некоторые статические переменные, используемые при составлении параметра.

Листинг 27.1. Функция `inet_srcrt_init`: инициализация перед записью маршрута от отправителя

```
//ipopts/sourceroute.c
1 #include "unp.h"
2 #include <netinet/in_system.h>
3 #include <netinet/ip.h>

4 static u_char *optr; /* указатель на формируемые параметры */
5 static u_char *lenptr; /* указатель на длину параметра SRR */
6 static int      ocnt; /* количество адресов */

7 u_char*
8 inet_srcrt_init(int type)
9 {
10    optr = Malloc(44); /* NOP, код параметра. len, указатель + до 10
```

```

    адресов */
11 bzero(optr, 44); /* гарантирует наличие EOL на конце */
12 ocnt = 0;
13 *optr++ = IPOPT_NOP; /* выравнивающие NOP */
14 *optr++ = type ? IPOPT_SSRR : IPOPT_LSRR;
15 lenptr = optr++; /* поле длины заполняется позже */
16 *optr++ = 4; /* сдвиг на первый адрес */

17 return(optr - 4); /* указатель для setsockopt() */
18 }

```

Инициализация

10-17 Мы выделяем в памяти буфер, максимальный размер которого — 44 байт, и обнуляем его содержимое. Значение параметра EOL равно нулю, так что тем самым параметр инициализируется байтами EOL. Затем мы подготавливаем заголовок для маршрутизации от источника. Как показано на рис. 27.1, сначала мы обеспечиваем выравнивание при помощи параметра NOP, после чего указываем тип маршрута (гибкий, жесткий), длину и значение указателя. Мы сохраняем указатель в поле len. Это значение мы будем записывать при поочередном добавлении адресов к списку. Указатель на параметр возвращается вызывающему процессу, а затем передается как четвертый аргумент функции setsockopt.

Следующая функция, `inet_srcrt_add`, добавляет один IPv4-адрес к создаваемому маршруту от отправителя.

Листинг 27.2. Функция `inet_srcrt_add`: добавление одного IPv4-адреса к маршруту от отправителя

```

//ipopts/sourceroute.c
19 int
20 inet_srcrt_add(char *hostptr)
21 {
22     int len;
23     struct addrinfo *ai;
24     struct sockaddr_in *sin;

25     if (ocnt > 9)
26         err_quit("too many source routes with: %s", hostptr);

27     ai = Host_serv(hostptr, NULL, AF_INET, 0);
28     sin = (struct sockaddr_in*)ai->ai_addr;
29     memcpy(optr, &sin->sin_addr, sizeof(struct in_addr));
30     freeaddrinfo(ai);

31     optr += sizeof(struct in_addr);
32     ocnt++;
33     len = 3 + (ocnt * sizeof(struct in_addr));
34     *lenptr = len;
35     return(len + 1); /* размер для setsockopt() */
36 }

```

Аргумент

19-20 Аргумент функции указывает либо на имя узла, либо на адрес IP в точечно- десятичной записи.

Проверка переполнения

25-26 Мы проверяем количество переданных адресов и выполняем инициализацию, если обрабатывается первый адрес.

Получение двоичного IP-адреса и запись маршрута

29-37 Функция `host_serv` обрабатывает имя узла или его IP-адрес, а возвращаемый ей адрес в двоичной форме мы помещаем в список. Мы обновляем поле `len` и возвращаем полный размер буфера (с учетом параметров NOP), который вызывающий процесс затем передаст функции `setsockopt`.

Когда полученный маршрут от отправителя возвращается приложению функцией `getsockopt`, формат этого параметра отличается от того, что было показано на рис. 27.1. Формат полученного параметра маршрута от отправителя показан на рис. 27.2.

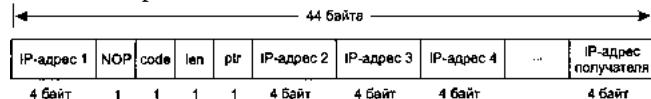


Рис. 27.2. Формат параметра маршрута от отправителя, возвращаемого функцией `getsockopt`

В первую очередь, мы можем отметить, что порядок следования адресов изменен ядром на противоположный относительно полученного маршрута от отправителя. Имеется в виду следующее: если в полученном маршруте содержались адреса A, B, C и D в указанном порядке, то под противоположным порядком подразумевается следующий: D, C, B, A. Первые 4 байта содержат первый IP-адрес из списка, затем следует однобайтовый параметр NOP (для выравнивания), затем — 3-байтовый заголовок параметра маршрута от отправителя, и далее остальные IP-адреса. После 3-байтового заголовка может следовать до 9 IP-адресов, и максимальное значение поля `len` в возвращенном заголовке равно 39. Поскольку параметр NOP всегда присутствует, длина буфера, возвращаемая функцией `getsockopt`, всегда будет равна значению, кратному 4 байтам.

ПРИМЕЧАНИЕ

Формат, приведенный на рис. 27.2, определен в заголовочном файле `<netinet/ip_var.h>` в виде следующей структуры:

```
#define MAX_IPOPTLEN 40

struct ipoption {
    struct in_addr ipopt_dst; /* адрес первого получателя */
    char ipopt_list[MAX_IPOPTLEN]; /* соответствующие параметры */
};
```

В листинге 27.3 мы анализируем эти данные, не используя указанную структуру.

Возвращаемый формат отличается от того, который был передан функции `setsockopt`. Если нам было бы нужно преобразовать формат, показанный на рис. 27.2, к формату, показанному на рис. 27.1, нам следовало бы поменять местами первые и вторые 4 байта и изменить значение поля `len`, добавив к имеющемуся значению 4. К счастью, нам не нужно этого делать, так как Беркли-реализации автоматически используют обращенный маршрут от получателя для сокета TCP. Иными словами, данные, возвращаемые функцией `getsockopt` (представленные на рис. 27.2), носят чисто информативный характер. Нам не нужно вызывать функцию `setsockopt`, чтобы указать ядру на необходимость использования данного маршрута для дейтаграмм IP, отсылаемых по соединению TCP, — ядро сделает это само. Подобный пример с нашим сервером TCP мы вскоре увидим.

Следующей из рассматриваемых нами функций, связанных с параметром маршрутизации, полученный маршрут от отправителя передается в формате, показанном на рис. 27.2. Затем она выводит соответствующую информацию. Эту функцию `inet_srcrt_print` мы показываем в листинге 27.3.

Листинг 27.3. Функция `inet_srcrt_print`: вывод полученного маршрута от отправителя

```
//ipopts/sourceroute.c
37 void
38 inet_srcrt_print(u_char *ptr, int len)
39 {
```

```

40 u_char c;
41 char str[INET_ADDRSTRLEN];
42 struct in_addr hop1;

43 memcpy(&hop1, ptr, sizeof(struct in_addr));
44 ptr += sizeof(struct in_addr);

45 while ((c = *ptr++) == IPOPT_NOP); /* пропуск параметров NOP */

46 if (c == IPOPT_LSRR)
47 printf("received LSRR: ");
48 else if (c == IPOPT_SSRR)
49 printf("received SSRR: ");
50 else {
51 printf("received option type %d\n", c);
52 return;
53 }
54 printf("%s ", Inet_ntop(AF_INET, &hop1, str, sizeof(str)));

55 len = *ptr++ - sizeof(struct in_addr); /* вычитаем адрес получателя */
56 ptr++; /* пропуск указателя */
57 while (len > 0) {
58 printf("%s ", Inet_ntop(AF_INET, ptr, str, sizeof(str)));
59 ptr += sizeof(struct in_addr);
60 len -= sizeof(struct in_addr);
61 }
62 printf("\n");
63 }

```

Сохраняем первый адрес IP, пропускаем все параметры NOP

43-45 Первый IP-адрес в буфере сохраняется, а все следующие за ним параметры NOP мы пропускаем.

Проверяем параметр маршрута от отправителя

46-62 Мы выводим информацию о маршруте и проверяем значение поля code, содержащегося в 3-байтовом заголовке, получаем значение поля len и пропускаем указатель ptr. Затем мы выводим все IP-адреса, следующие за 3-байтовым заголовком, кроме IP-адреса получателя.

Пример

Теперь мы модифицируем наш эхо-сервер TCP таким образом, чтобы выводить полученный маршрут от отправителя, а эхо-клиент TCP — так, чтобы маршрут от отправителя можно было задавать. В листинге 27.4 показан код эхо-клиента TCP.

Листинг 27.4. Эхо-клиент TCP, задающий маршрут от отправителя

```

//ipopts/tcpcli01.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5 int c, sockfd, len = 0;
6 u_char *ptr = NULL;

```

```

7 struct addrinfo *ai;

8 if (argc < 2)
9   err_quit("usage: tcpcli01 [ -[gG] <hostname> ... ] <hostname>");

10 opterr = 0; /* отключаем запись сообщений getopt() в stderr */
11 while ((c = getopt(argc, argv, "gG")) != -1) {
12   switch (c) {
13     case 'g': /* свободный маршрут от отправителя */
14       if (ptr)
15         err_quit("can't use both -g and -G");
16       ptr = inet_srcrt_init(0);
17       break;
18
19     case 'G': /* жесткий маршрут от отправителя */
20       if (ptr)
21         err_qint("can't use both -g and -G");
22       ptr = inet_srcrt_init(1);
23       break;
24
25     case '?':
26       err_quit("unrecognized option: %c", c);
27   }
28
29   if (ptr)
30     while (optind < argc-1)
31       len = inet_srcrt_add(argv[optind++]);
32   else if (optind < argc-1)
33     err_quit("need -g or -G to specify route");
34
35   if (optind != argc-1)
36     err_quit("missing <hostname>");
37
38   ai = Host_serv(argv[optind], SERV_PORT_STR, AF_INET, SOCK_STREAM);
39
40   sockfd = Socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
41
42   if (ptr) {
43     len = inet_srcrt_add(argv[optind]); /* получатель в конце */
44     Setsockopt(sockfd, IPPROTO_IP, IP_OPTIONS, ptr, len);
45     free(ptr);
46   }
47
48   Connect(sockfd, ai->ai_addr, ai->ai_addrlen);
49
50   str_cli(stdin, sockfd); /* вызов рабочей функции */
51
52   exit(0);
53 }
```

Обработка аргументов командной строки

12-26 Мы вызываем нашу функцию `inet_srcrt_init`, чтобы инициализировать маршрут от отправителя. Тип маршрутизации указывается при помощи параметра `-g` (свободная) или `-G` (жесткая).

27-33 Если указатель `ptr` установлен, значит, был указан параметр маршрутизации от отправителя, и все указанные промежуточные узлы добавляются к маршруту, подготовленному на предыдущем этапе функцией `inet_srcrt_add`. Если же `ptr` не установлен, но в командной строке еще есть аргументы, значит, пользователь задал маршрут, но не указал его тип. В этом случае программа завершает работу с сообщением об ошибке.

Обработка адреса получателя и создание сокета

34-35 Последний аргумент командной строки — это имя узла или адрес сервера в точечно-десятичной записи, который обрабатывается нашей функцией `host_serv`. Мы не можем вызвать функцию `tcp_connect`, так как должны задать маршрут от отправителя между вызовом функций `socket` и `connect`. Последняя инициирует трехэтапное рукопожатие, а нам нужно, чтобы сегмент SYN отправителя и все последующие пакеты проходили по одному и тому же маршруту.

36-42 Если маршрут от отправителя задан, следует добавить IP-адрес сервера в конец списка адресов (см. рис. 27.1). Функция `setsockopt` устанавливает маршрут от отправителя для данного сокета. Затем мы вызываем функцию `connect`, а потом — нашу функцию `str_cli` (см. листинг 5.4).

Наш TCP-сервер имеет много общего с кодом, показанным в листинге 5.9, но содержит следующие изменения.

Во-первых, мы выделяем место для параметров:

```
int len;
u_char *opts;
```

```
opts = Malloc(44);
```

Во-вторых, мы получаем параметры IP после вызова функции `accept`, но перед вызовом функции `fork`:

```
len = 44;
Getsockopt(connfd, IPPROTO_IP, IP_OPTIONS, opts, &len);
if (len > 0) {
    printf("received IP options, len = %d\n", len);
    inet_srcrt_print(opts, len);
}
```

Если сегмент SYN, полученный от клиента, не содержит никаких параметров IP, переменная `len` по завершении функции `getsockopt` будет иметь нулевое значение (эта переменная относится к типу «значение-результат»). Как уже упоминалось, нам не нужно предпринимать какие-либо шаги для того, чтобы на стороне сервера использовался обращенный маршрут от отправителя: это делается автоматически без нашего участия [128, с. 931]. Вызывая функцию `getsockopt`, мы просто получаем копию обращенного маршрута от отправителя. Если мы не хотим, чтобы TCP использовал этот маршрут, то после завершения функции `accept` следует вызвать функцию `setsockopt` и задать нулевую длину (последний аргумент), тем самым удалив все используемые в текущий момент параметры IP. Но маршрут от отправителя тем не менее уже был использован в процессе трехэтапного рукопожатия при пересылке второго сегмента. Если мы уберем параметры маршрутизации, IP составит и будет использовать для пересылки последующих пакетов какой-либо другой маршрут.

Теперь мы покажем пример клиент-серверного взаимодействия при заданном маршруте от отправителя. Мы запускаем наш клиент на узле `freebsd` следующим образом:

```
freebsd % tcpcli01 -g macosx freebsd4 macosx
```

Тем самым дейтаграммы IP отсылаются с узла `freebsd` на узел `macosx`, обратно на узел `freebsd4`, и наконец, на `macosx`, где запущен наш сервер. Две промежуточные системы `freebsd4` и `macosx` должны переправлять дейтаграммы и принимать дейтаграммы с маршрутизацией от отправителя, чтобы этот пример работал.

Когда соединение устанавливается, на стороне сервера выдается следующий результат:

```
macosx % tcpser01
received IP options, len = 16
received LSRR, 172.24.37.94 172.24.37.78 172.24.37.94
```

Первый выведенный IP-адрес — это первый узел обратного маршрута (`freebsd4`, как показано на рис. 27.2), а следующие два адреса идут в том порядке, который используется сервером для отправки

дейтаграмм назад клиенту. Если мы понаблюдаем за процессом взаимодействия клиента и сервера с помощью программы `tcpdump`, мы увидим, как используется параметр маршрутизации для каждой дейтаграммы в обоих направлениях.

ПРИМЕЧАНИЕ

К сожалению, действие параметра сокета `IP_OPTIONS` никогда не было документировано, поэтому вы можете увидеть различные вариации поведения в системах, не происходящих от исходного кода Беркли. Например, в системе Solaris 2.5 первый адрес, возвращаемый функцией `getsockopt` (см. рис. 27.2) — это не первый адрес в обращенном маршруте, а адрес собеседника. Тем не менее обратный маршрут, используемый TCP, будет корректен. Кроме того, в Solaris 2.5 всем параметрам маршрутизации предшествует четыре параметра `NOP`, что ограничивает параметр маршрутизации восемью IP-адресами, а не девятью, которые реально могли бы поместиться.

Уничтожение полученного маршрута от отправителя

К сожалению, использование параметра маршрутизации образует брешь в системе обеспечения безопасности программ, выполняющих аутентификацию по IP-адресам (сейчас такая проверка считается недостаточной). Если хакер отправляет пакеты, используя один из доверенных адресов в качестве адреса отправителя, но указывая в качестве одного из промежуточных адресов маршрута от отправителя свой собственный адрес, возвращаемые по обратному маршруту пакеты будут попадать к хакеру, а «отправитель», чьим адресом хакер прикрывался, никогда не узнает об этом. Начиная с выпуска Net/1 (1989), серверы `rlogind` и `rshd` использовали код, аналогичный следующему:

```
u_char buf[44];
char lbuf[BUFSIZ];
int optsize;

optsize = sizeof(buf);
if (getsockopt(0, IPPROTO_IP, IP_OPTIONS,
    buf, &optsize) == 0 && optsize != 0) {
    /* форматируем параметры как шестнадцатеричные числа для записи в lbuf[] */
    syslog(LOG_NOTICE,
        "Connection received using IP options (ignored):%s", lbuf);
    setsockopt(0, IPPROTO, IP_OPTIONS, NULL, 0);
}
```

Если устанавливается соединение с какими-либо параметрами IP (значение переменной `optsize`, возвращенное функцией `getsockopt`, не равно нулю), то с помощью функции `syslog` делается запись соответствующего сообщения и вызывается функция `setsockopt` для очистки всех параметров. Таким образом предотвращается отправка последующих сегментов TCP для данного соединения по обращенному маршруту от отправителя. Сейчас уже известно, что этой технологии недостаточно, так к моменту установления соединения трехэтапное рукопожатие TCP будет уже завершено и второй сегмент (сегмент SYN-ACK на рис. 2.5) будет уже отправлен по обращенному маршруту от отправителя к клиенту. Даже если этот сегмент не успеет дойти до клиента, то во всяком случае он дойдет до некоторого промежуточного узла, входящего в маршрут от отправителя, где, возможно, затаился хакер. Так как предполагаемый хакер видел порядковые номера TCP в обоих направлениях, даже если никаких других пакетов по маршруту от отправителя послано не будет, он по-прежнему сможет отправлять серверу сообщения с правильным порядковым номером.

Единственным решением этой возможной проблемы является запрет на прием любых соединений TCP, приходящих по обращенному маршруту от отправителя, когда вы используете IP-адрес от отправителя для какой-либо формы подтверждения (как, например, в случае с `rlogin` или `rshd`). Вместо вызова функции `setsockopt` во фрагменте кода, приведенном ранее, закройте только что принятое соединение и завершите только что порожденный процесс сервера. Второй сегмент трехэтапного рукопожатия отправится, но соединение не останется открытym и не будет использоваться далее.

27.4. Заголовки расширения IPv6

Мы не показываем никаких параметров в заголовке IPv6 на рис. A.2 (который всегда имеет длину 40 байт), но следом за этим заголовком могут идти заголовки расширения^[2] (*extension headers*).

1. Параметры для транзитных узлов (*hop-by-hop options*) должны следовать непосредственно за 40-байтовым заголовком IPv6. В настоящее время не определены какие-либо параметры для транзитных узлов, которые могли бы использоваться в приложениях.

2. Параметры получателя (*destination options*). В настоящее время не определены какие-либо параметры получателя, которые могли бы использоваться в приложениях.

3. Заголовок маршрутизации. Этот параметр маршрутизации от отправителя аналогичен по своей сути тем, которые мы рассматривали в случае IPv4 в разделе 27.3.

4. Заголовок фрагментации. Этот заголовок автоматически генерируется узлом при фрагментации дейтаграммы IPv6, а затем обрабатывается получателем при сборке дейтаграммы из фрагментов.

5. Заголовок аутентификации (AH — *authentication header*). Использование этого заголовка документировано в RFC 2402 [65].

6. Заголовок шифрования (ESH — *encapsulating security payload header*). Использование этого заголовка документировано в RFC 2406 [66].

Мы уже говорили о том, что заголовок фрагментации целиком обрабатывается ядром, как и заголовки AH и ESP, обработка которых управляется согласно базе данных соглашений о безопасности (о сокетах управления ключами читайте в главе 9). Остаются еще три параметра, которые мы обсудим в следующем разделе. Интерфейс этих параметров определен в RFC 3542 [114].

27.5. Параметры транзитных узлов и параметры получателя IPv6

Параметры для транзитных узлов и параметры получателя IPv6 имеют одинаковый формат, показанный на рис. 27.3. Восьмиразрядное поле *следующий заголовок* (*next header*) идентифицирует следующий заголовок, который следует за данным заголовком. Восьмиразрядное поле *длина заголовка расширения* (*header extension length*) содержит длину заголовка расширения в условных единицах (1 у.е. = 8 байт), но не учитывает первые 8 байт заголовка. Например, если заголовок занимает всего 8 байт, то значение поля длины будет равно нулю. Если заголовок занимает 16 байт, то соответственно значение этого поля будет равно 1, и т.д. Оба заголовка заполняются таким образом, чтобы длина каждого была кратна 8 байтам. Это достигается либо с помощью параметра *pad1*, либо с помощью параметра *padN*, которые мы вскоре рассмотрим.

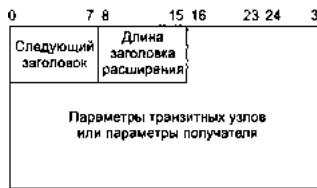


Рис. 27.3. Формат параметра для транзитных узлов и параметра получателя

Заголовок параметра транзитных узлов и заголовок параметра получателя могут содержать произвольное количество отдельных параметров, как показано на рис. 27.4.



Рис. 27.4. Формат отдельных параметров, входящих в заголовок параметра транзитных узлов и заголовок параметра получателя

Этот формат иногда называется TLV, так как для каждого отдельного параметра указывается его тип, длина и значение (*type, length, value*). Восьмиразрядное поле *типа* (*type*) указывает тип параметра. В дополнение к этому два старших разряда указывают, что именно узел IPv6 будет делать с этим параметром в том случае, если он не сможет в нем разобраться:

- 00 — пропустить параметр и продолжить обработку заголовка.
- 01 — игнорировать пакет.
- 10 — игнорировать пакет и отослать отправителю сообщение об ошибке ICMP типа 2 (см. табл. А.6), независимо от того, является ли адрес получателя групповым адресом.

- 11 — игнорировать пакет и отослать отправителю сообщение об ошибке ICMP типа 2 (см. табл. А.6) но только в том случае, если адрес получателя пакета не является адресом многоадресной передачи.

Следующий разряд указывает, могут ли меняться данные, входящие в этот параметр, в процессе передачи пакета.

- 0 — данные параметра не могут быть изменены.
- 1 — данные параметра могут быть изменены.

Оставшиеся пять младших разрядов задают сам параметр. Заметьте, что код параметра определяется всеми восемью битами, младших пяти битов для этого недостаточно. Однако значения параметров выбираются таким образом, чтобы обеспечивать уникальность младших пяти битов как можно дольше.

8-разрядное поле длины задает длину данных этих параметров в байтах. Длина поля типа и длина самого поля длины не входят в это значение.

Два параметра заполнения (pad options) определены в RFC 2460 [27] и могут быть использованы как в заголовке параметров для транзитных узлов, так и в заголовке параметров получателя. Один из параметров транзитных узлов — параметр *размера увеличенного поля данных* (*jumbo pay load length option*) — определен в RFC 2675 [9]. Ядро генерирует этот параметр по мере необходимости и обрабатывает при получении. Новый параметр увеличенного объема данных для IPv6, аналогичный параметру *извещения маршрутизатора* (*router alert*), описан в RFC 2711 [87]. Эти параметры изображены на рис. 27.5. Есть и другие параметры (например, для Mobile-IPv6), но мы их на рисунке не показываем.

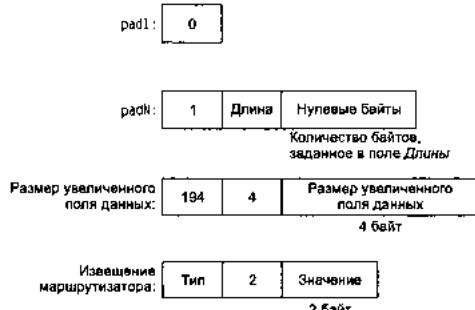


Рис. 27.5. Параметры IPv6 для транзитных узлов

Параметр *pad1* — это единственный параметр, для которого не указывается длина и значение. Его назначение — вставка одного пустого байта для заполнения. Параметр *padN* используется, когда требуется вставить 2 или более байта заполнения. Для 2 байт заполнения длина параметра будет иметь нулевое значение, а сам параметр будет состоять из поля типа и поля длины. В случае 3 байт заполнения длина будет равна 1, а следом за полем длины будет стоять один нулевой байт. Параметр размера увеличенного поля данных допускает увеличение поля размера дейтаграмм до 32 бит и используется, когда 16-разрядное поле размера, показанное на рис. А.2, оказывается недостаточно большим.

Мы показываем эти параметры схематически, потому что для всех параметров получателя и транзитных узлов действует так называемое *условие выравнивания* (*alignment requirement*), записываемое как $xn + y$. Это означает, что сдвиг данного параметра относительно начала заголовка равен числу, n раз кратному x байтам, к которому добавлено y байтов (то есть величина сдвига в байтах равна $xn + y$). Например, условие выравнивания для параметра размера увеличенного поля данных записывается как $4n + 2$. Это означает, что 4-байтовое значение параметра (длина размера увеличенного поля данных) будет выровнено по 4-байтовой границе. Причина, по которой значение y для этого параметра равно 2, заключается в том, что параметры транзитных узлов и получателя начинаются именно с двух байтов — один байт используется для указания типа, другой — для указания длины (см. рис. 27.4). Для параметра уведомления маршрутизатора условие выравнивания записывается как $2n + 0$, благодаря чему 2-байтовое значение параметра оказывается выровненным по 2-байтовой границе.

Параметры транзитных узлов и параметры получателя обычно задаются как вспомогательные данные в функции *sendmsg* и возвращаются функцией *recvmsg* также в виде вспомогательных данных. От приложения не требуется никаких специальных действий для отправки этих параметров — нужно только задать их при вызове функции *sendmsg*. Но для получения этих параметров должен быть включен соответствующий параметр сокета: *IPV6_RECVNOROPTS* для параметра транзитных узлов и *IPV6_RECVDSTOPTS* для параметров получателя. Например, чтобы можно было получить оба параметра, нужен следующий код:

```
const int on = 1;
```

```
setsockopt(sockfd, IPPROTO_IPV6, IPV6_RECVHOPTS, &on, sizeof(on));
setsockopt(sockfd, IPPROTO_IPV6, IPV6_RECVDSTOPTS, &on, sizeof(on));
```

На рис. 27.6 показан формат объектов вспомогательных данных, используемый для отправки и получения параметров транзитных узлов и параметров получателя.

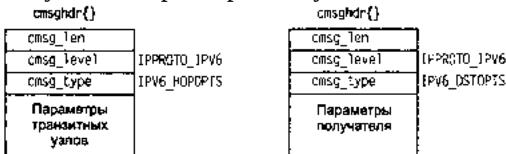


Рис. 27.6. Объекты вспомогательных данных, используемые для параметров транзитных узлов и параметров получателя

Чтобы уменьшить объем дублированного кода, определены семь функций, которые создают и обрабатывают эти вспомогательные объекты данных. Следующие четыре функции формируют отправляемый параметр.

```
#include <netinet/in.h>
```

```
int inet6_opt_init(void *extbuf, socklen_t extlen);
```

Возвращает: количество байтов для размещения пустого заголовка расширения, -1 в случае ошибки

```
int inet6_opt_append(void *extbuf, socklen_t extlen,
```

```
    int offset, uint8_t type, socklen_t len, uint_t align, void **databufp);
```

Возвращает: длину расширяющего заголовка после добавления параметра, -1 в случае ошибки

```
int inet6_opt_finish(void *extbuf, socklen_t extlen, int offset);
```

Возвращает: длину законченного заголовка расширения, -1 в случае ошибки

```
int inet6_opt_set_val(void *databuf, int offset,
    const void *val, socklen_t vallen);
```

Возвращает: новое смещение в буфере databuf

Функция `inet6_opt_init` возвращает количество байтов, необходимое для данного параметра. Если аргумент `extbuf` не является нулевым указателем, функция инициализирует заголовок расширения. Значение -1 возвращается при аварийном завершении работы в том случае, если аргумент `extlen` не кратен 8. (Все заголовки параметров транзитных узлов и получателя в IPv6 должны быть кратны 8.)

Функция `inet6_opt_append` возвращает общую длину заголовка расширения после добавления указанного при вызове параметра. Если аргумент `extbuf` не является нулевым указателем, функция дополнительно выполняет инициализацию параметра и вставляет необходимое заполнение. Значение -1 возвращается в случае аварийного завершения работы, если параметр не помещается в выделенный буфер. Аргумент `offset` представляет собой текущую полную длину, то есть значение, возвращенное при предыдущем вызове `inet6_opt_append` или `inet6_opt_init`. Аргументы `type` и `len` задают тип и длину параметра, они копируются непосредственно в его заголовок. Аргумент `align` указывает условие выравнивания, то есть значение x из выражения $xn + y$. Значение y вычисляется по `align` и `len`, поэтому указывать его явным образом необходимости нет. Аргумент `databufp` представляет собой адрес будущего указателя на значение параметра. Значение параметра копируется вызывающим процессом при помощи функции `inet6_opt_set_val` или любым другим методом.

Для завершения расширяющего заголовка вызывается функция `inet6_opt_finish`, которая добавляет в заголовок заполнение, делая его длину кратной 8 байтам. Как и раньше, заполнение добавляется в буфер только в том случае, если аргумент `extbuf` представляет собой непустой указатель. В противном случае функция вычисляет обновленное значение длины. Подобно `inet6_opt_append`, аргумент `offset` задает текущую полную длину (значение, возвращаемое `inet6_opt_append` и `inet6_opt_init`). Функция `inet6_opt_finish` возвращает полную длину возвращаемого заголовка или -1, если требуемое заполнение не помещается в предоставленный буфер.

Функция `inet6_opt_set_val` копирует значение параметра в буфер данных, возвращаемый `inet6_opt_append`. Аргумент `databuf` представляет собой указатель, возвращаемый `inet6_opt_append`. Аргумент `offset` представляет собой текущую длину внутри параметра, его необходимо инициализировать нулем для каждого параметра, а затем использовать возвращаемые `inet6_opt_set_val` значения по мере

построения параметра. Аргументы `val` и `vallen` определяют значение для копирования в буфер значения параметра.

Предполагается, что с помощью этих функций вы будете делать два прохода по списку параметров, которые вы предполагаете вставить: во время первого прохода будет вычисляться требуемая длина буфера, а во время второго прохода — выполняться фактическое построение буфера параметра. При первом проходе нужно вызвать `inet6_opt_init`, `inet6_opt_append` (один раз для каждого параметра) и `inet6_opt_finish`, передавая нулевой указатель и 0 в качестве аргументов `extbuf` и `extlen` соответственно. Затем можно динамически выделить буфер, использовав в качестве размера значение, возвращенное `inet6_opt_finish`. Этот буфер будет передаваться в качестве аргумента `extbuf` при втором проходе. Во время второго прохода вызываются функции `inet6_opt_init` и `inet6_opt_append`. Копирование значений параметров может выполняться как «вручную», так и при помощи функции `inet6_opt_set_val`. Наконец, мы должны вызвать `inet6_opt_finish`. Альтернативный вариант действий состоит в выделении буфера достаточно большого размера для нашего параметра. В этом случае первый проход можно не выполнять. Однако если изменение параметров приведет к переполнению выделенного буфера, в программе возникнет ошибка.

Оставшиеся три функции обрабатывают полученный параметр.

```
#include <netinet/in.h>
```

```
int inet6_opt_next(const void *extbuf, socklen_t extlen,
    int offset, uint8_t *typep, socklen_t *lenp, void **databufp);
```

Возвращает: смещение следующего параметра, -1 в случае достижения конца списка параметров или в случае ошибки

```
int inet6_opt_find(const void *extbuf, socklen_t extlen,
    int offset, uint8_t type, socklen_t *lenp, void **databufp);
```

Возвращает: смещение следующего параметра, -1 в случае достижения конца списка параметров или в случае ошибки

```
int inet6_opt_get_val(const void *databuf, int offset, void *val, socklen_t vallen);
```

Возвращает: новое значение смещения внутри буфера databuf

Функция `inet6_opt_next` обрабатывает следующий параметр в буфере. Аргументы `extbuf` и `extlen` определяют буфер, в котором содержится заголовок. Как и у `inet6_opt_append`, аргумент `offset` представляет собой текущее смещение внутри буфера. При первом вызове `inet6_opt_next` значение этого аргумента должно быть равно нулю, а при всех последующих — значению, возвращенному при предыдущем вызове функции. Аргументы `typep`, `lenp` и `databufp` предназначены для возвращения функцией типа, длины и значения параметра соответственно. Функция `inet6_opt_next` возвращает -1 в случае обработки заголовка с нарушенной структурой или в случае достижения конца буфера.

Функция `inet6_opt_find` аналогична предыдущей функции, но позволяет вызывающему процессу задать тип параметра, который следует искать (аргумент `type`), вместо того чтобы каждый раз возвращать следующий параметр.

Функция `inet6_opt_get_val` предназначена для извлечения значений из параметра по указателю `databuf`, возвращаемому предшествующим вызовом `inet6_opt_next` или `inet6_opt_find`. Как и для `inet6_opt_set_val`, аргумент `offset` должен начинаться с 0 для каждого параметра, а затем должен приравниваться значению, возвращаемому предшествующим вызовом `inet6_opt_get_val`.

27.6. Заголовок маршрутизации IPv6

Заголовок маршрутизации IPv6 используется для маршрутизации от отправителя в IPv6. Первые два байта заголовка маршрутизации такие же, как показанные на рис. 27.3: поле *следующего заголовка* (*next header*) и поле длины заголовка расширения (*header extension length*). Следующие два байта задают *тип маршрутизации* (*routing type*) и количество оставшихся сегментов (*number of segments left*) (то есть сколько из перечисленных узлов еще нужно пройти). Определен только один тип заголовка маршрутизации, обозначаемый как тип 0. Формат заголовка маршрутизации показан на рис. 27.7.

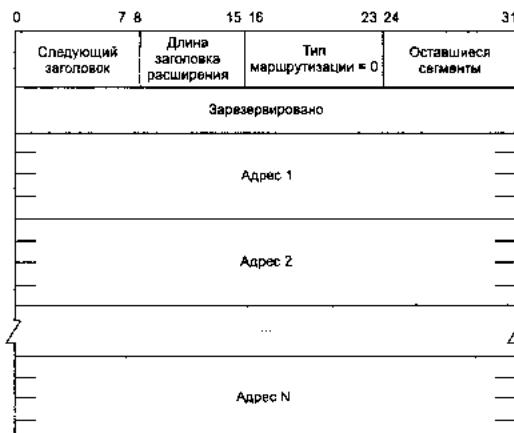


Рис. 27.7. Заголовок маршрутизации IPv6

В заголовке маршрутизации IPv6 может появиться неограниченное количество адресов (реальное ограничение накладывается длиной пакета), а количество оставшихся сегментов не должно превышать количество адресов в заголовке. Документ RFC 2460 [27] описывает подробности обработки этого заголовка при пересылке его в направлении получателя. Там же вы можете найти подробно рассмотренный пример.

Заголовок маршрутизации обычно задается как вспомогательные данные в функции `sendmsg` и возвращается в виде вспомогательных данных функцией `recvmsg`. Для отправки заголовка приложению не требуется выполнять какие-либо специальные действия — достаточно просто указать его при вызове функции `sendmsg`. Но для получения заголовка маршрутизации требуется, чтобы был включен параметр `IPV6_RECVRTHDR`:

```
const int on = 1;
```

```
setsockopt(sockfd, IPPROTO_IPV6, IPV6_RECVRTHDR, &on, sizeof(on));
```

На рис. 27.8 показан формат объекта вспомогательных данных, используемый для отправки и получения заголовка маршрутизации. Для создания и обработки заголовка маршрутизации определены шесть функций. Следующие три функции используются для создания отправляемого параметра.

```
#include <netinet/in.h>
```

```
socklen_t inet6_rth_space(int type, int segments);
```

Возвращает: положительное число, равное количеству байтов в случае успешного выполнения, 0 в случае ошибки

```
void *inet6_rth_init(void *rthbuf, socklen_t rthlen, int type, int segments);
```

Возвращает: непустой указатель в случае успешного выполнения, NULL в случае ошибки

```
int inet6_rth_add(void *rthbuf, const struct in6_addr *addr);
```

Возвращает: 0 в случае успешного выполнения, -1 в случае ошибки

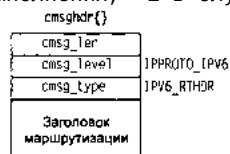


Рис. 27.8. Объект вспомогательных данных для заголовка маршрутизации IPv6

Функция `inet6_rth_space` возвращает количество байтов, необходимое для размещения объекта вспомогательных данных, содержащего заголовок маршрутизации указанного типа (обычно это `IPV6_RTHDR_TYPE_0`) с заданным количеством сегментов.

Функция `inet6_rth_init` инициализирует буфер, на который указывает аргумент `rthbuf`, для помещения заголовка маршрутизации типа `type` и заданного количества сегментов. Возвращаемое значение этой функции — указатель на буфер. Этот указатель используется как аргумент при вызове следующей

функции. Функция `inet6_rth_init` возвращает `NULL` в случае возникновения ошибок (например, при недостаточном размере предоставленного буфера).

Функция `inet6_rth_add` добавляет адрес IPv6, на который указывает аргумент `addr`, к концу составляемого заголовка маршрутизации. В случае успешного выполнения обновляется значение элемента `segleft` заголовка маршрутизации, чтобы учесть добавленный новый адрес.

Следующие три функции манипулируют полученным заголовком маршрутизации:

```
#include <netinet/in.h>
```

```
int inet6_rth_reverse(const void *in, void *out);
```

Возращает: 0 в случае успешного выполнения, -1 в случае ошибки

```
int inet6_rth_segments(const void *rthbuf);
```

Возращает: количество сегментов в заголовке маршрутизации в случае успешного выполнения, -1 в случае ошибки

```
struct in6_addr *inet6_rth_getaddr(const void *rthbuf, int index);
```

Возращает: непустой указатель в случае успешного выполнения, NULL в случае ошибки

Функция `inet6_rth_reverse` принимает в качестве аргумента заголовок маршрутизации, полученный в виде объекта вспомогательных данных (на который указывает аргумент `in`), и создает новый заголовок маршрутизации (в буфере, на который указывает аргумент `out`), отправляющий дейтаграммы по обратному маршруту. Указатели `in` и `out` могут указывать на один и тот же буфер.

Функция `inet6_rth_segments` возвращает количество сегментов в заголовке маршрутизации, на который указывает `rthbuf`. В случае успешного выполнения функции возвращаемое значение оказывается больше 0.

Функция `inet6_rth_getaddr` возвращает указатель на адрес IPv6, заданный через `index` в заголовке маршрутизации `rthbuf`. Аргумент `index` должен лежать в пределах от 1 до значения, возвращенного функцией `inet6_rth_segments`, включительно.

Чтобы продемонстрировать использование этих параметров, мы создали UDP-клиент и UDP-сервер. Клиент представлен в листинге 27.5. Он принимает маршрут от отправителя в командной строке подобно TCP-клиенту IPv4, представленному в листинге 27.4. Сервер печатает маршрут полученного сообщения и обращает этот маршрут для отправки сообщения в обратном направлении.

Листинг 27.5. UDP-клиент, использующий маршрутизацию от отправителя

```
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int c, sockfd, len = 0;
6     u_char *ptr = NULL;
7     void *rth;
8     struct addrinfo *ai;

9     if (argc < 2)
10        err_quit("usage: udpcli01 [ <hostname> ... ] <hostname>");

11    if (argc > 2) {
12        int i;

13        len = Inet6_rth_space(IPV6_RTHDR_TYPE_0, argc-2);
14        ptr = Malloc(len);
15        Inet6_rth_init(ptr, len, IPV6_RTHDR_TYPE_0, argc-2);
16        for (i = 1; i < argc-1; i++) {
17            ai = Host_serv(argv[i], NULL, AF_INET6, 0);
18            Inet6_rth_add(ptr,
19                &((struct sockaddr_in6*)ai->ai_addr)->sin6_addr);
20    }
```

```

21 }

22 ai = Host_serv(argv[argc-1], SERV_PORT_STR, AF_INET6, SOCK_DGRAM);

23 sockfd = Socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);

24 if (ptr) {
25     Setsockopt(sockfd, IPPROTO_IPV6, IPV6_RTHDR, ptr, len);
26     free(ptr);
27 }

28 dg_cli(stdin, sockfd, ai->ai_addr, ai->ai_addrlen); /* do it all */

29 exit(0);
30 }

```

Создание маршрута

11-21 Если при вызове программы было указано более одного аргумента, все параметры командной строки, за исключением последнего, формируют маршрут от отправителя. Сначала мы определяем, какой объем памяти займет заголовок маршрутизации, при помощи функции `inet6_rth_space`, затем выделяем буфер соответствующего размера вызовом `malloc`. После этого каждый адрес маршрута преобразуется в числовую форму функцией `host_serv` и добавляется к маршруту функцией `inet6_rth_add`. Примерно то же самое выполнял и TCP-клиент IPv4, за тем исключением, что здесь мы используем библиотечные функции, а не свои собственные.

Поиск адресата и создание сокета

22-23 Мы определяем адрес назначения при помощи `host_serv` и создаем сокет для отправки пакетов.

Установка «закрепленного» параметра IPV6_RTHDR и вызов рабочей функции

24-27 В разделе 27.7 будет показано, что не обязательно отправлять одни и те же вспомогательные данные с каждым пакетом. Вместо этого можно вызвать `setsockopt` таким образом, что один и тот же заголовок будет добавляться ко всем пакетам в рамках одного сеанса. Этот параметр устанавливается только в том случае, если указатель `ptr` не нулевой, то есть мы уже должны были выделить буфер под заголовок маршрутизации. На последнем этапе мы вызываем рабочую функцию `dg_cli`, которая не меняется с листинга 8.4.

Программа UDP-сервера не изменилась по сравнению с предыдущими примерами. Сервер открывает сокет и вызывает функцию `dg_echo`. В листинге 27.6 представлена функция `dg_echo`, печатающая информацию о маршруте от источника (если таковой был получен) и обращающая этот маршрут для отправки сообщения в обратном направлении.

Листинг 27.6. Функция `dg_echo`, печатающая маршрут

```

//ipopts/dgechoprintroute.c
1 #include "unp.h"

2 void
3 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
4 {
5     int n;
6     char mesg[MAXLINE];
7     int on;
8     char control[MAXLINE];

```

```

9 struct msghdr msg;
10 struct cmsghdr *cmsg;
11 struct iovec iov[1];

12 on = 1;
13 Setsockopt(sockfd, IPPROTO_IPV6, IPV6_RECVRTHDR, &on, sizeof(on));

14 bzero(&msg, sizeof(msg));
15 iov[0].iov_base = mesg;
16 msg.msg_name = pcliaddr;
17 msg.msg_iov = iov;
18 msg.msg_iovlen = 1;
19 msg.msg_control = control;
20 for (;;) {
21     msg.msg_namelen = clilen;
22     msg.msg_controllen = sizeof(control);
23     iov[0].iov_len = MAXLINE;
24     n = Recvmsg(sockfd, &msg, 0);

25     for (cmsg = CMSG_FIRSTHDR(&msg); cmsg != NULL;
26          cmsg = CMSG_NXTHDR(&msg, cmsg)) {
27         if (cmsg->cmsg_level == IPPROTO_IPV6 &&
28             cmsg->cmsg_type == IPV6_RTHDR) {
29             inet6_srcrt_print(CMSG_DATA(cmsg));
30             Inet6_rth_reverse(CMSG_DATA(cmsg), CMSG_DATA(cmsg));
31         }
32     }

33     iov[0].iov_len = n;
34     Sendmsg(sockfd, &msg, 0);
35 }
36 }
```

Включение параметра IPV6_RECVRTHDR и подготовка структуры msghdr

12-13 Чтобы получить информацию о маршруте, мы должны установить параметр сокета IPV6_RECVRTHDR. Кроме того, мы должны использовать функцию recvmsg, поэтому мы настраиваем поля структуры msghdr, которые не требуют изменения.

Настройка изменяемых полей и вызов recvmsg

21-24 Мы устанавливаем размер полей длины и вызываем recvmsg.

Поиск и обработка маршрута от отправителя

25-32 Мы перебираем вспомогательные данные, используя CMSG_FIRSTHDR и CMSG_NXTHDR. Несмотря на то, что мы ожидаем получить только один объект вспомогательных данных, выполнить такой перебор всегда полезно. Если мы обнаруживаем заголовок маршрутизации, он распечатывается функцией `inet6_srcrt_print` (листинг 27.7). Затем маршрут обращается функцией `inet6_rth_reverse` для последующего использования при возвращении пакета клиенту. В данном случае обращение производится без копирования в новый буфер, так что можно использовать старый объект вспомогательных данных для отправки пакета клиенту.

Отправка эхо-пакета

33-34 Мы устанавливаем длину пакета и передаем его клиенту вызовом sendmsg.

Благодаря наличию вспомогательных библиотечных функций IPv6 наша функция inet6_srcrt_print становится почти тривиальной.

Листинг 27.7. Функция inet6_srcrt_print: вывод маршрута

```
1 #include "unp.h"

2 void
3 inet6_srcrt_print(void *ptr)
4 {
5     int i, segments;
6     char str[INET6_ADDRSTRLEN];

7     segments = Inet6_rth_segments(ptr);
8     printf("received source route: ");
9     for (i = 0; i < segments; i++)
10    printf("%s ", Inet_ntop(AF_INET6, Inet6_rth_getaddr(ptr, i),
11        str, sizeof(str)));
12    printf("\n");
13 }
```

Определение количества сегментов маршрута

7 Количество сегментов маршрута определяется функцией inet6_rth_segments.

Перебор сегментов

9-11 Мы перебираем сегменты маршрута, вызывая для каждого из них inet6_rth_getaddr и преобразуя адреса в формат представления функцией inet_ntop.

Клиенту и серверу, работающим с маршрутами IPv6, не нужно ничего знать о формате этих маршрутов внутри пакета. Библиотечные функции интерфейса скрывают детали форматирования, но не мешают нам программировать с той же гибкостью, которая была в IPv4, где параметры нужно было строить вручную.

27.7. «Закрепленные» параметры IPv6

Мы рассмотрели использование вспомогательных данных с функциями sendmsg и recvmsg для отправки и получения следующих семи различных типов объектов вспомогательных данных:

1. Информация о пакете IPv6: структура in6_pktnfo, содержащая адрес получателя и индекс интерфейса для исходящих дейтаграмм либо адрес отправителя и индекс интерфейса для приходящих дейтаграмм (индекс принимающего интерфейса) (см. рис. 22.5).

2. Предельное количество транзитных узлов для исходящих или приходящих дейтаграмм (см. рис. 22.5).

3. Адрес следующего транзитного узла (см. рис. 22.5).

4. Класс исходящего или входящего трафика (см. рис. 22.5).

5. Параметры транзитных узлов (см. рис. 27.6).

6. Параметры получателя (см. рис. 27.6).

7. Заголовок маршрутизации (см. рис. 27.8).

В табл. 14.4 приведены значения полей cmsg_level и cmsg_type для этих объектов, а также значения для других объектов вспомогательных данных.

Вместо того чтобы отсылать эти параметры при каждом вызове функции sendmsg, мы можем установить соответствующие параметры сокета. Параметры сокета используют те же константы, что и вспомогательные данные, то есть уровень параметра всегда должен иметь значение IPPROTO_IPV6, а

название параметра может быть `IPV6_DSTOPTS`, `IPV6_HOPLIMIT`, `IPV6_HOPOPTS`, `IPV6_NEXTHOP`, `IPV6_PKTINFO`, `IPV6_RTHDR` или `IPV6_TCLASS`. Закрепленные параметры могут быть заменены для конкретного пакета в случае сокета UDP или символьного сокета IPv6, если при вызове функции `sendmsg` задать какие-либо другие параметры в качестве объектов вспомогательных данных. Если при вызове функции `sendmsg` указаны какие-либо вспомогательные данные, ни один из закрепленных параметров не будет послан с этим пакетом.

Концепция закрепленных параметров также может быть использована и в случае TCP, поскольку вспомогательные данные никогда не отсылаются и не принимаются с помощью функций `sendmsg` или `recvmsg` на сокете TCP. Вместо этого приложение TCP может установить соответствующий параметр сокета и указать любой из упомянутых в начале этого раздела семи объектов вспомогательных данных. Тогда эти параметры будут относиться ко всем пакетам, отсылаемым с данного сокета. Поведение при повторной передаче пакетов, первоначально переданных до изменения закрепленных параметров, не определено: могут использоваться как старые, так и новые значения параметров.

Не существует способа получить параметры, принятые в IP-пакете по TCP, потому что в этом протоколе отсутствует соответствие между пакетами и операциями чтения из сокета, выполняемыми пользователем.

27.8. История развития интерфейса IPv6

Документ RFC 2292 [113] определял более раннюю версию описываемого интерфейса, которая была реализована в некоторых системах. В этой версии для работы с параметрами получателя и транзитных узлов использовались функции `inet6_option_space`, `inet6_option_init`, `inet6_option_append`, `inet6_option_alloc`, `inet6_option_next` и `inet6_option_find`. Эти функции работали непосредственно с объектами типа `struct cmsghdr`, предполагая, что все параметры содержатся во вспомогательных данных. Для работы с заголовками маршрутизации были предназначены функции `inet6_rthdr_space`, `inet6_rthdr_init`, `inet6_rthdr_add`, `inet6_rthdr_lasthop`, `inet6_rthdr_reverse`, `inet6_rthdr_segments`, `inet6_rthdr_getaddr` и `inet6_rthdr_getflags`. Эти функции также работали непосредственно со вспомогательными данными.

В этом API закрепленные параметры устанавливались при помощи параметра сокета `IPV6_PKTOPTIONS`. Объекты вспомогательных данных при этом передавались в качестве данных параметра `IPV6_PKTOPTIONS`. Нынешние параметры сокета `IPV6_DSTOPTS`, `IPV6_HOPOPTS` и `IPV6_RTHDR` были флагами, позволявшими получать соответствующие заголовки во вспомогательных данных.

Подробнее обо всем этом вы можете прочесть в разделах 4–8 документа RFC 2292 [113].

27.9. Резюме

Из десяти определенных в IPv4 параметров наиболее часто используются параметры маршрутизации от отправителя, но в настоящее время их популярность падает из-за проблем, связанных с безопасностью. Доступ к параметрам заголовков IPv4 осуществляется с помощью параметра сокета `IP_OPTIONS`.

В IPv6 определены шесть заголовков расширения. Доступ к заголовкам расширения IPv6 осуществляется с помощью функционального интерфейса, что освобождает нас от необходимости углубляться в детали фактического формата пакета. Эти заголовки расширения записываются как вспомогательные данные функцией `sendmsg` и возвращаются функцией `recvmsg` также в виде вспомогательных данных.

Упражнения

1. Что изменится, если в нашем примере, приведенном в конце раздела 27.3, мы зададим каждый промежуточный узел с параметром `-G` вместо `-g`?
2. Размер буфера, указываемый в качестве аргумента функции `setsockopt` для параметра сокета `IP_OPTIONS`, должен быть кратен 4 байтам. Что бы нам пришлось делать, если бы мы не поместили параметр NOP в начало буфера, как показано на рис. 27.1?
3. Каким образом программа `ping` получает маршрут от отправителя, когда используется параметр IP Record Route (запись маршрута), описанный в разделе 7.3 [128]?

4. Почему в примере кода для сервера `rlogind`, приведенном в конце раздела 27.3, который предназначен для удаления полученного маршрута от отправителя, дескриптор сокета (первый аргумент функций `getsockopt` и `setsockopt`) имеет нулевое значение?

5. В течение долгого времени для удаления маршрута использовался код, несколько отличающийся от приведенного в конце раздела 27.3. Он выглядел следующим образом:

```
optsizе = 0;  
setsockopt(0, IPPROTO, IP_OPTIONS, NULL, &optsizе);
```

Что в этом фрагменте неправильно? Имеет ли это значение?

Глава 28

Символьные сокеты

28.1. Введение

Символьные, или неструктурированные, сокеты (*raw sockets*) обеспечивают три возможности, не предоставляемые обычными сокетами TCP и UDP.

1. Символьные сокеты позволяют читать и записывать пакеты ICMPv4, IGMPv4 и ICMPv6. Например, программа `ping` посыпает эхо-запросы ICMP и получает эхо-ответы ICMP. (Наша оригинальная версия программы `ping` приведена в разделе 28.5.) Демон маршрутизации многоадресной передачи `mrtouted` посыпает и получает пакеты IGMPv4.

2. Эта возможность также позволяет реализовывать как пользовательские процессы те приложения, которые построены с использованием протоколов ICMP и IGMP, вместо того чтобы помещать большее количество кода в ядро. Например, подобным образом построен демон обнаружения маршрутов (`in.rdisc` в системе Solaris 2.x). В приложении F книги [111] рассказывается, как можно получить исходный код открытой версии. Этот демон обрабатывает два типа сообщений ICMP, о которых ядро ничего не знает (извещение маршрутизатора и запрос маршрутизатору).

С помощью символьных сокетов процесс может читать и записывать IPv4-дейтаграммы с полем протокола IPv4, которое не обрабатывается ядром. Посмотрите еще раз на 8-разрядное поле протокола IPv4, изображенное на рис. А.1. Большинство ядер обрабатывают дейтаграммы, содержащие значения поля протокола 1 (ICMP), 2 (IGMP), 6 (TCP) и 17 (UDP). Но для этого поля определено гораздо большее количество значений, полный список которых приведен в реестре IANA «Номера протоколов» (Protocol Numbers). Например, протокол маршрутизации OSPF не использует протоколы TCP или UDP, а работает напрямую с протоколом IP, устанавливая в поле протокола значение 89 для IP-дейтаграмм. Программа `gated`, реализующая OSPF, должна использовать для чтения и записи таких IP-дейтаграмм символьный сокет, поскольку они содержат значение поля протокола, о котором ничего не известно ядру. Эта возможность также переносится в версию IPv6.

3. С помощью символьных сокетов процесс может построить собственный заголовок IPv4 при помощи параметра сокета `IP_HDRINCL`. Такую возможность имеет смысл использовать, например, для построения собственного пакета UDP или TCP. Подобный пример приведен в разделе 29.7.

В данной главе описывается создание символьных сокетов, а также операции ввода и вывода с этими сокетами. Далее приводятся версии программ `ping` и `traceroute`, работающие как с версией IPv4, так и с версией IPv6.

28.2. Создание символьных сокетов

При создании символьных сокетов выполняются следующие шаги:

1. Символьный сокет создается функцией `socket` со вторым аргументом `SOCK_RAW`. Третий аргумент (протокол) обычно ненулевой. Например, для создания символьного сокета IPv4 следует написать:

```
int sockfd;
sockfd = socket(AF_INET, SOCK_RAW, protocol);
```

где `protocol` — одна из констант `IPPROTO_xxx`, определенных в подключенному заголовочном файле `<netinet/in.h>`, например `IPPROTO_ICMP`.

Только привилегированный пользователь может создать символьный сокет. Такой подход предотвращает отправку IP-дейтаграмм в сеть обычными пользователями.

2. Параметр сокета `IP_HDRINCL` может быть установлен следующим образом:

```
const int on = 1;
if (setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0)
    обработка ошибки
```

В следующем разделе описывается действие этого параметра.

3. На символьном сокете можно вызвать функцию `bind`, но это делается редко. Эта функция устанавливает только локальный адрес: на символьном сокете нет понятия порта. Что касается вывода, вызов функции `bind` устанавливает IP-адрес отправителя, который будет использоваться для дейтаграмм,

отправляемых на символьном сокете (только если не установлен параметр сокета `IP_HDRINCL`). Если функция `bind` не вызывается, ядро использует в качестве IP-адреса отправителя основной IP-адрес исходящего интерфейса.

4. На символьном сокете можно вызвать функцию `connect`, но это делается редко. Эта функция устанавливает только внешний адрес, так как на символьном сокете нет понятия порта. О выводе можно сказать, что вызов функции `connect` позволяет нам вызвать функцию `write` или `send` вместо `sendto`, поскольку IP-адрес получателя уже определен.

28.3. Вывод на символьном сокете

Вывод на символьном сокете регулируется следующими правилами:

1. Стандартный вывод выполняется путем вызова функции `sendto` или `sendmsg` и определения IP-адреса получателя. Функции `write`, `writev` и `send` также можно использовать, если сокет был присоединен.

2. Если не установлен параметр сокета `IP_HDRINCL`, то начальный адрес данных, предназначенных для записи ядром, указывает на первый байт, следующий за IP-заголовком, поскольку ядро будет строить IP-заголовок и добавлять его к началу данных из процесса. Ядро устанавливает поле протокола создаваемого заголовка IPv4 равным значению третьего аргумента функции `socket`.

3. Если параметр сокета `IP_HDRINCL` установлен, то начальный адрес данных, предназначенных для записи ядром, указывает на первый байт IP-заголовка. Размер данных для записи должен включать размер IP-заголовка вызывающего процесса. Процесс полностью формирует IP-заголовок, за исключением того, что, во-первых, значение поля идентификации IPv4 может быть нулевым (что указывает ядру на необходимость самостоятельно установить это значение), во-вторых, ядро всегда вычисляет и сохраняет контрольную сумму заголовка IPv4, в-третьих, включает или не включает параметры IP (см. раздел 27.2).

4. Ядро фрагментирует символьные пакеты, превышающие значение MTU исходящего интерфейса.

ПРИМЕЧАНИЕ

Согласно документации, символьные сокеты должны предоставлять протоколу такой же интерфейс, как если бы он был реализован в ядре [74]. К сожалению, это означает, что некоторые части интерфейса зависят от ядра операционной системы. В частности, это относится к порядку байтов полей заголовка IP. В Беркли-ядрах все поля имеют порядок байтов сети, за исключением полей `ip_len` и `ip_off`, имеющих порядок байтов узла [128, с. 233, с. 1057]. В системах Linux и OpenBSD все поля имеют порядок байтов сети.

Параметр сокета `IP_HDRINCL` впервые был представлен в системе 4.3BSD Reno. До этого приложение имело единственную возможность определить свой собственный IP-заголовок в пакетах, отсылаемых на символьный сокет, — использовать заплату ядра (kernel patch), которая была представлена в 1988 году Van Jacobsonом (Van Jacobson) для поддержки программы `traceroute`. Эта заплата позволяла приложению создавать символьный IP-сокет, определяя протокол как `IPPROTO_RAW`, что соответствовало значению 255 (это значение является зарезервированным и никогда не должно появляться в поле протокола IP-заголовка).

Функции, осуществляющие ввод-вывод на символьном сокете, являются одними из простейших функций в ядре. Например, в книге [128, с. 1054–1057] каждая такая функция занимает около 40 строк кода на языке C. Для сравнения: функция ввода TCP содержит около 2000 строк, а функция вывода TCP около 700 строк.

Приводимое в этой книге описание параметра сокета `IP_HDRINCL` относится к системе 4.4BSD. В более ранних версиях, таких как Net/2, при использовании данного параметра заполнялось большее количество полей заголовка IP.

В протоколе IPv4 пользовательский процесс отвечает за вычисление и установку контрольной суммы любого заголовка, следующего за заголовком IPv4. Например, в нашей программе `ping` (см. листинг 28.10), прежде чем вызывать функцию `sendto`, мы должны вычислить контрольную сумму ICMPv4 и сохранить ее в заголовке ICMPv4.

Особенности символьного сокета версии IPv6

Для символьного сокета IPv6 существуют несколько отличий (RFC 3542 [114]).

■ Все поля в заголовках протоколов, отсылаемых или получаемых на символьном сокете IPv6, должны находиться в сетевом порядке байтов.

■ В IPv6 не существует параметров, подобных параметру `IP_HDRINCL` сокета IPv4. Полные пакеты IPv6 (включая дополнительные заголовки) не могут быть прочитаны или записаны через символьный сокет IPv6. Приложения имеют доступ ко всем полям заголовка IPv6 и дополнительных заголовков через параметры сокета или вспомогательные данные (см. упражнение 28.1). Если приложению все же необходимо полностью считать или записать IPv6-дейтаграмму, необходимо использовать доступ к канальному уровню (о нем речь пойдет в главе 29).

■ Как вскоре будет показано, на символьном сокете IPv6 по-другому обрабатываются контрольные суммы.

Параметр сокета `IPV6_CHECKSUM`

Для символьного сокета ICMPv6 ядро всегда вычисляет и сохраняет контрольную сумму в заголовке ICMPv6, тогда как для символьного сокета ICMPv4 приложение должно выполнять данную операцию самостоятельно (сравните листинги 28.10 и 28.12). И ICMPv4, и ICMPv6 требуют от отправителя вычисления контрольной суммы, но ICMPv6 включает в свою контрольную сумму псевдозаголовок (понятие псевдозаголовка обсуждается при вычислении контрольной суммы UDP в листинге 29.10). Одно из полей этого псевдозаголовка представляет собой IPv6-адрес отправителя, и обычно приложение оставляет ядру возможность выбирать это значение. Чтобы приложению не нужно было пытаться отыскать этот адрес только для вычисления контрольной суммы, проще разрешить вычислять контрольную сумму ядру.

Для других символьных сокетов IPv6 (при создании которых третий аргумент функции `socket` отличен от `IPPROTO_ICMPV6`) параметр сокета сообщает ядру, вычислять ли контрольную сумму и сохранять ли ее в исходящих пакетах, а также следует ли проверять контрольную сумму в приходящих пакетах. По умолчанию этот параметр выключен, а включается он путем присваивания неотрицательного значения параметра, как в следующем примере:

```
int offset = 2;
if (setsockopt(sockfd, IPPROTO_IPV6, IPV6_CHECKSUM,
    &offset, sizeof(offset)) < 0)
    обработка ошибки
```

Здесь не только разрешается вычисление контрольной суммы на данном сокете, но и сообщается ядру смещение 16-разрядной контрольной суммы в байтах: в данном примере оно составляет два байта от начала данных приложения. Чтобы отключить данный параметр, ему нужно присвоить значение -1. Если он включен, ядро будет вычислять и сохранять контрольную сумму для исходящих пакетов, посланных на данном сокете, а также проверять контрольную сумму для пакетов, получаемых данным сокетом.

28.4. Ввод через символьный сокет

Первый вопрос, на который следует ответить, говоря о символьных сокетах, следующий: какие из полученных IP-дейтаграмм ядро передает символьному сокету? Применяются следующие правила:

1. Получаемые пакеты UDP и TCP никогда не передаются на символьный сокет. Если процесс хочет считать IP-дейтаграмму, содержащую пакеты UDP или TCP, пакеты должны считываться на канальном уровне, как показано в главе 29.

2. Большинство ICMP-пакетов передаются на символьный сокет, после того как ядро заканчивает обработку ICMP-сообщения. Беркли-реализации посыпают все получаемые ICMP-пакеты на символьный сокет, кроме эхо-запроса, запроса отметки времени и запроса маски адреса [128, с. 302–303]. Эти три типа ICMP-сообщений полностью обрабатываются ядром.

3. Все IGMP-пакеты передаются на символьный сокет, после того как ядро заканчивает обработку IGMP-сообщения.

4. Все IP-дейтаграммы с таким значением поля протокола, которое не понимает ядро, передаются на символьный сокет. Для этих пакетов ядро выполняет только минимальную проверку некоторых полей IP-заголовка, таких как версия IP, контрольная сумма IPv4-заголовка, длина заголовка и IP-адрес получателя [128, с. 213–220].

5. Если дейтаграмма приходит фрагментами, символьному сокету ничего не передается, до тех пор, пока все фрагменты не прибудут и не будут собраны вместе.

Если у ядра есть IP-дейтаграмма для пересылки символьному сокету, в поисках подходящих сокетов проверяются все символьные сокеты всех процессов. Копия IP-дейтаграммы доставляется *каждому* подходящему сокету. Для каждого символьного сокета выполняются три перечисленных ниже проверки, и только в том случае, если все три проверки дают положительный результат, дейтаграмма направляется данному сокету.

1. Если при создании символьного сокета определено ненулевое значение *protocol* (третий аргумент функции *socket*), то значение поля протокола полученной дейтаграммы должно совпадать с этим ненулевым значением, иначе дейтаграмма не будет доставлена на данный сокет.

2. Если локальный IP-адрес связан с символьным сокетом функцией *bind*, IP-адрес получателя в полученной дейтаграмме должен совпадать с этим адресом, иначе дейтаграмма не посыпается данному сокету.

3. Если для символьного сокета был определен внешний адрес с помощью функции *connect*, IP-адрес отправителя в полученной дейтаграмме должен совпадать с этим адресом, иначе дейтаграмма не посыпается данному сокету.

Следует отметить, что если символьный сокет создан с нулевым значением аргумента *protocol* и не вызывается ни функция *bind*, ни функция *connect*, то сокет получает копии всех дейтаграмм, которые ядро направляет символьным сокетам.

Дейтаграммы IPv4 всегда передаются через символьные сокеты целиком, вместе с заголовками. В версии IPv6 символьному сокету передается все, кроме дополнительных заголовков (см., например, рис. 28.4 и 28.6).

ПРИМЕЧАНИЕ

В заголовке IPv4, передаваемом приложению, для *ip_len*, *ip_off* и *ip_id* установлен порядок байтов узла, а все остальные ноля имеют порядок байтов сети. В системе Linux все поля остаются в сетевом порядке байтов.

Как уже говорилось, интерфейс символьных сокетов определяется таким образом, чтобы работа со всеми протоколами, в том числе и не обрабатываемыми ядром, осуществлялась одинаково. Поэтому содержимое полей зависит от ядра операционной системы.

В предыдущем разделе мы отметили, что все ноля символьного сокета IPv6 остаются в сетевом порядке байтов.

Фильтрация по типу сообщений ICMPv6

Символьный сокет ICMPv4 получает большинство сообщений ICMPv4, полученных ядром. Но ICMPv6 является расширением ICMPv4, включающим функциональные возможности ARP и IGMP (см. раздел 2.2). Следовательно, символьный сокет ICMPv6 потенциально может принимать гораздо больше пакетов по сравнению с символьным сокетом ICMPv4. Но большинство приложений, использующих символьные сокеты, заинтересованы только в небольшом подмножестве всех ICMP-сообщений.

Для уменьшения количества пакетов, передаваемых от ядра к приложению через символьный ICMPv6-сокет, предусмотрен фильтр, связанный с приложением. Фильтр объявляется с типом данных *struct icmp6_filter*, который определяется в заголовочном файле *<netinet/icmp6.h>*. Для установки и получения текущего ICMPv6-фильтра для символьного сокета ICMPv6 используются функции *setsockopt* и *getsockopt* с аргументом *level*, равным *IPPROTO_ICMPV6*, и аргументом *optname*, равным *ICMP6_FILTER*.

Со структурой *icmp6_filter* работают шесть макросов.

```
#include <netinet/icmp6.h>

void ICMP6_FILTER_SETPASSALL(struct icmp6_filter *filt);
void ICMP6_FILTER_SETBLOCKALL(struct icmp6_filter *filt);
void ICMP6_FILTER_SETPASS(int msgtype, struct icmp6_filter *filt);
void ICMP6_FILTER_SETBLOCK(int msgtype, struct icmp6_filter *filt);
int ICMP6_FILTER_WILLPASS(int msgtype, const struct icmp6_filter *filt);
```

```
int ICMP6_FILTER_WILLBLOCK(int msgtype, const struct icmp6_filter *filt);
```

Все возвращают: 1, если фильтр пропускает (блокирует) сообщение данного типа, 0 в противном случае

Аргумент filt всех макрокоманд является указателем на переменную icmp6_filter, изменяемую первыми четырьмя макрокомандами и проверяемую последними двумя. Аргумент msgtype является значением в интервале от 0 до 255, определяющим тип ICMP-сообщения.

Макрокоманда SETPASSALL указывает, что все типы сообщений должны пересыпаться приложению, а макрокоманда SETBLOCKALL — что никакие сообщения не должны посыпаться приложению. По умолчанию при создании символьного сокета ICMPv6 подразумевается, что все типы ICMP-сообщений пересыпаются приложению.

Макрокоманда SETPASS определяет конкретный тип сообщений, который должен пересыпаться приложению, а макрокоманда SETBLOCK блокирует один конкретный тип сообщений. Макрокоманда WILLPASS возвращает значение 1, если определенный тип пропускается фильтром. Макрокоманда WILLBLOCK возвращает значение 1, если определенный тип блокирован фильтром, и нуль в противном случае.

В качестве примера рассмотрим приложение, которое будет получать только ICMPv6-извещения маршрутизатора:

```
struct icmp6_filter myfilt;  
  
fd = Socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);  
  
ICMP6_FILTER_SETBLOCKALL(&myfilt);  
ICMP6_FILTER_SETPASS(ND_ROUTER_ADVERT, &myfilt);  
Setsockopt(fd, IPPROTO_ICMPV6, ICMP6_FILTER, &myfilt, sizeof(myfilt));
```

Сначала мы блокируем все типы сообщений (поскольку по умолчанию все типы сообщений пересыпаются), а затем разрешаем пересыпать только извещения маршрутизатора. Несмотря на то, что мы используем фильтр, приложение должно быть готово к получению всех типов пакетов ICMPv6, потому что любые пакеты ICMPv6, полученные между вызовами socket и setsockopt, будут добавлены в очередь на сокете. Параметр ICMP6_FILTER — лишь средство оптимизации условий функционирования приложения.

28.5. Программа ping

В данном разделе приводится версия программы ping, работающая как с IPv4, так и с IPv6. Вместо того чтобы представить известный доступный исходный код, мы разработали оригинальную программу, и сделано это по двум причинам. Во-первых, свободно доступная программа ping страдает общей болезнью программирования, известной как «ползучий улучшисм» (стремление к постоянным ненужным усложнениям программы в погоне за мелкими улучшениями): она поддерживает 12 различных параметров. Наша цель при исследовании программы ping в том, чтобы понять концепции и методы сетевого программирования и не быть при этом сбитыми с толку ее многочисленными параметрами. Наша версия программы ping поддерживает только один параметр и занимает в пять раз меньше места, чем общедоступная версия. Во-вторых, общедоступная версия работает только с IPv4, а нам хочется показать версию, поддерживающую также и IPv6.

Действие программы ping предельно просто: по некоторому IP-адресу посыпается эхо-запрос ICMP, и этот узел отвечает эхо-ответом ICMP. Оба эти сообщения поддерживаются в обеих версиях — и в IPv4, и в IPv6. На рис. 28.1 приведен формат ICMP-сообщений.

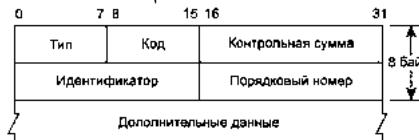


Рис. 28.1. Формат сообщений эхо-запроса и эхо-ответа ICMPv4 и ICMPv6

В табл. A.5 и A.6 приведены значения поля *тип* (*type*) для этих сообщений и говорится, что значение поля *код* (*code*) равно нулю. Далее будет показано, что в поле *идентификатор* (*identifier*) указывается идентификатор процесса ping, а значение поля порядковый номер (*sequence number*) увеличивается на 1 для каждого отправляемого пакета. В поле *дополнительные данные* (*optional data*) сохраняется 8-байтовая отметка времени отправки пакета. Правила ICMP-запроса требуют, чтобы *идентификатор*, *порядковый*

номер и все дополнительные данные возвращались в эхо-ответе. Сохранение отметки времени отправки пакета позволяет вычислить RTT при получении ответа.

В листинге 28.1^[1] приведены примеры работы нашей программы. В первом используется версия IPv4, а во втором IPv6. Обратите внимание, что мы установили для нашей программы ping флаг set-user-ID (установка идентификатора пользователя при выполнении), потому что для создания символьного сокета требуются права привилегированного пользователя.

Листинг 28.1. Примеры вывода программы ping

```
freebsd % ping www.google.com
PING www.google.com (216.239.57.99): 56 data bytes
64 bytes from 216.239.57.99: seq=0, ttl=53, rtt=5.611 ms
64 bytes from 216.239.57.99: seq=1, ttl=53, rtt=5.562 ms
64 bytes from 216.239.57.99: seq=2, ttl=53, rtt=5.589 ms
64 bytes from 216.239.57.99: seq=3, ttl=53, rtt=5.910 ms

freebsd % ping www.kame.net
PING orange.kame.net (2001:200:0:4819:203:47ff:fea5:3085): 56 data bytes
64 bytes from 2001:200:0:4819:203:47ff:fea5:3085: seq=0, hlim=52, rtt=422.066 ms
64 bytes from 2001:200:0:4819:203:47ff:fea5:3085: seq=1, hlim=52, rtt=417.398 ms
64 bytes from 2001:200:0:4819:203:47ff:fea5:3085: seq=2, hlim=52, rtt=416.528 ms
64 bytes from 2001:200:0:4819:203:47ff:fea5:3085: seq=3, hlim=52, rtt=429.192 ms
```

На рис. 28.2 приведен обзор функций, составляющих программу ping.

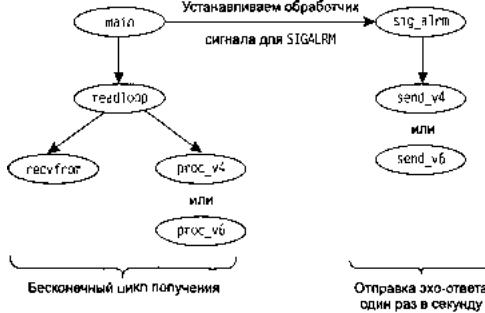


Рис. 28.2. Обзор функций программы ping

Данная программа состоит из двух частей: одна половина читает все, что приходит на символьный сокет, и выводит эхо-ответы ICMP, а другая половина один раз в секунду посылает эхо-запросы ICMP. Вторая половина запускается один раз в секунду сигналом SIGALRM.

В листинге 28.2 приведен заголовочный файл ping.h, подключаемый во всех файлах программы.

Листинг 28.2. Заголовочный файл ping.h

```
//ping/ping.h
1 #include "unp.h"
2 #include <netinet/in_systm.h>
3 #include <netinet/in.h>
4 #include <netinet/ip_icmp.h>

5 #define BUFSIZE 1500

6 /* глобальные переменные */
7 char sendbuf[BUFSIZE];

8 int datalen; /* размер данных после заголовка ICMP */
9 char *host;

10 int nsent; /* увеличиваем на 1 для каждого sendto() */
11 pid_t pid; /* наш PID */
12 int sockfd;
13 int verbose;
```

```

14 /* прототипы функций */
15 void init_v6(void);
16 void proc_v4(char*, ssize_t, struct msghdr*, struct timeval*);
17 void proc_v6(char*, ssize_t, struct msghdr*, struct timeval*);
18 void send_v4(void);
19 void send_v6(void);
20 void readloop(void);
21 void sig_alarm(int);
22 void tv_sub(struct timeval*, struct timeval*);

23 struct proto {
24     void (*fproc)(char*, ssize_t, struct msghdr*, struct timeval*);
25     void (*fsend)(void);
26     void (*finit)(void);
27     struct sockaddr *sasend; /* структура sockaddr{} для отправки,
                                полученная от getaddrinfo */
28     struct sockaddr *sarecv; /* sockaddr{} для получения */
29     socklen_t salen; /* длина sockaddr{} */
30     int icmpproto; /* значение IPPROTO_XXX для ICMP */
31 } *pr;

32 #ifdef IPV6
33 #include <netinet/ip6.h>
34 #include <netinet/icmp6.h>

35 #endif

```

Подключение заголовочных файлов IPv4 и ICMPv4

1-22 Подключаются основные заголовочные файлы IPv4 и ICMPv4, определяются некоторые глобальные переменные и прототипы функций.

Определение структуры proto

23-31 Для обработки различий между IPv4 и IPv6 используется структура proto. Данная структура содержит два указателя на функции, два указателя на структуры адреса сокета, размер структуры адреса сокета и значение протокола для ICMP. Глобальный указатель pr будет указывать на одну из этих структур, которая будет инициализироваться для IPv4 или IPv6.

Подключение заголовочных файлов IPv6 и ICMPv6

32-35 Подключаются два заголовочных файла, определяющие структуры и константы IPv6 и ICMPv6 (RFC 3542 [114]).

Функция main приведена в листинге 28.3.

Листинг 28.3. Функция main

```

//ping/main.c
1 #include "ping.h"

2 struct proto proto_v4 =
3 { proc_v4, send_v4, NULL, NULL, NULL, 0, IPPROTO_ICMP };

4 #ifdef IPV6
5 struct proto proto_v6 =

```

```
6 { proc_v6, send_v6, init_v6, NULL, NULL, 0, IPPROTO_ICMPV6 };
7 #endif

8 int datalen = 56; /* размер данных в эхо-запросе ICMP */

9 int
10 main(int argc, char **argv)
11 {
12     int c;
13     struct addrinfo *ai;
14     char *h;

15     opterr = 0; /* отключаем запись сообщений getopt() в stderr */
16     while ((c = getopt(argc, argv, "v")) != -1) {
17         switch (c) {
18             case 'v':
19                 verbose++;
20                 break;

21             case '?':
22                 err_quit("unrecognized option %c", c);
23             }
24         }

25     if (optind != argc-1)
26         err_quit("usage: ping [ -v ] <hostname>");
27     host = argv[optind];

28     pid = getpid() & 0xffff; /* поле идентификатора ICMP имеет размер 16 бит */
29     Signal(SIGALRM, sig_alarm);

30     ai = Host_serv(host, NULL, 0, 0);

31     h = Sock_ntop_host(ai->ai_addr, ai->ai_addrlen);
32     printf("PING %s (%s): %d data bytes\n",
33     ai->ai_canonname ? ai->ai_canonname : h, h, datalen);

34     /* инициализация в соответствии с протоколом */
35     if (ai->ai_family == AF_INET) {
36         pr = &proto_v4;
37     #ifdef IPV6
38     } else if (ai->ai_family == AF_INET6) {
39         pr = &proto_v6;
40         if (IN6_IS_ADDR_V4MAPPED(&((struct sockaddr_in6*)
41             ai->ai_addr)->sin6_addr)))
42             err_quit("cannot ping IPv4-mapped IPv6 address");
43     #endif
44     } else
45         err_quit("unknown address family %d", ai->ai_family);

46     pr->sasend = ai->ai_addr;
47     pr->sarecv = Calloc(1, ai->ai_addrlen);
48     pr->salen = ai->ai_addrlen;

49     readloop();
```

```
50 exit(0);
51 }
```

Определение структуры proto для IPv4 и IPv6

2-7 Определяется структура proto для IPv4 и IPv6. Указатели структуры адреса сокета инициализируются как нулевые, поскольку еще не известно, какая из версий будет использоваться — IPv4 или IPv6.

Длина дополнительных данных

8 Устанавливается количество дополнительных данных (56 байт), которые будут посыпаться с эхо-запросом ICMP. При этом полная IPv4-дейтаграмма будет иметь размер 84 байта (20 байт на IPv4-заголовок и 8 байт на ICMP-заголовок), а IPv6-дейтаграмма будет иметь длину 104 байта. Все данные, посылаемые с эхо-запросом, должны быть возвращены в эхо-ответе. Время отправки эхо-запроса будет сохраняться в первых 8 байтах области данных, а затем, при получении эхо-ответа, будет использоваться для вычисления и вывода времени RTT.

Обработка параметров командной строки

15-24 Единственный параметр командной строки, поддерживаемый в нашей версии, это параметр -v, в результате использования которого большинство ICMP-сообщений будут выводиться на консоль. (Мы не выводим эхо-ответы, принадлежащие другой запущенной копии программы ping.) Для сигнала SIGALRM устанавливается обработчик, и мы увидим, что этот сигнал генерируется один раз в секунду и вызывает отправку эхо-запросов ICMP.

Обработка аргумента, содержащего имя узла

31-48 Стока, содержащая имя узла или IP-адрес, является обязательным аргументом и обрабатывается функцией host_serv. Возвращаемая структура addrinfo содержит семейство протоколов — либо AF_INET, либо AF_INET6. Глобальный указатель rg устанавливается на требуемую в конкретной ситуации структуру proto. Также с помощью вызова функции IN6_IS_ADDR_V4MAPPED мы убеждаемся, что адрес IPv6 на самом деле не является адресом IPv4, преобразованным к виду IPv6, поскольку даже если возвращаемый адрес является адресом IPv6, узлу будет отправлен пакет IPv4. (Если такая ситуация возникнет, можно переключиться и использовать IPv4.) Структура адреса сокета, уже размещенная в памяти с помощью функции getaddrinfo, используется для отправки, а другая структура адреса сокета того же размера размещается в памяти для получения.

Обработка ответов осуществляется функцией readloop, представленной в листинге 28.4.

Листинг 28.4. Функция readloop

```
//ping/readloop.c
1 #include "ping.h"

2 void
3 readloop(void)
4 {
5     int size;
6     char recvbuf[BUFSIZE];
7     char controlbuf[BUFSIZE];
8     struct msghdr msg;
9     struct iovec iov;
10    ssize_t n;
11    struct timeval tval;
```

```

12 sockfd = Socket(pr->sasend->sa_family, SOCK_RAW, pr->icmpproto);
13 setuid(getuid()); /* права привилегированного пользователя
                      больше не нужны */
14 if (pr->finit)
15   (*pr->finit)();
16 size = 60 * 1024; /* setsockopt может завершиться с ошибкой */
17 setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));
18 sig_alrm(SIGALRM); /* отправка первого пакета */
19 iov.iov_base = recvbuf;
20 iov.iov_len = sizeof(recvbuf);
21 msg.msg_name = pr->sarecv;
22 msg.msg_iov = &iov;
23 msg.msg_iovlen = 1;
24 msg.msg_control = controlbuf;
25 for (;;) {
26   msg.msg_namelen = pr->salen;
27   msg.msg_controllen = sizeof(controlbuf);
28   n = recvmsg(sockfd, &msg, 0);
29   if (n < 0) {
30     if (errno == EINTR)
31       continue;
32     else
33       err_sys("recvmsg error");
34 }
35 Gettimeofday(&tval, NULL);
36 (*pr->fproc)(recvbuf, n, &msg, &tval);
37 }
38 }

```

Создание сокета

12-13 Создается символьный сокет, соответствующий выбранному протоколу. В вызове функции `setuid` нашему эффективному идентификатору пользователя присваивается фактический идентификатор пользователя. Для создания символьных сокетов программа должна иметь права привилегированного пользователя, но когда символьный сокет уже создан, от этих прав можно отказаться. Всегда разумнее отказаться от лишних прав, если в них нет необходимости, например на тот случай, если в программе есть скрытая ошибка, которой кто-либо может воспользоваться.

Выполнение инициализации для протокола

14-15 Мы выполняем функцию инициализации для выбранного протокола. Для IPv6 такая функция представлена в листинге 28.7.

Установка размера приемного буфера сокета

16-17 Пытаемся установить размер приемного буфера сокета, равный 61 440 байт (60×1024) — этот размер больше задаваемого по умолчанию. Это делается в расчете на случай, когда пользователь проверяет качество связи с помощью программы `ping`, обращаясь либо к широковещательному адресу IPv4, либо к

групповому адресу. В обоих случаях может быть получено большое количество ответов. Увеличивая размер буфера, мы уменьшаем вероятность того, что приемный буфер переполнится.

Отправка первого пакета

18 Запускаем обработчик сигнала, который, как мы увидим, посыпает пакет и создает сигнал SIGALRM один раз в секунду. Обычно обработчик сигналов не запускается напрямую, как у нас, но это можно делать. Обработчик сигналов является обычной функцией языка C, просто в нормальных условиях он асинхронно запускается ядром.

Подготовка msghdr для recvmsg

19-24 Мы записываем значения в неизменяемые поля структур msghdr и iovec, которые будут передаваться функции recvmsg.

Бесконечный цикл для считывания всех ICMP-сообщений

25-37 Основной цикл программы является бесконечным циклом, считающим все пакеты, возвращаемые на символьный сокет ICMP. Вызывается функция gettimeofday для регистрации времени получения пакета, а затем вызывается соответствующая функция протокола (proc_v4 или proc_v6) для обработки ICMP-сообщения.

В листинге 28.5 приведена функция tv_sub, вычисляющая разность двух структур timeval и сохраняющая результат в первой из них.

Листинг 28.5. Функция tv_sub: вычитание двух структур timeval

```
//lib/tv_sub.c
1 #include "unp.h"

2 void
3 tv_sub(struct timeval *out, struct timeval *in)
4 {
5     if ((out->tv_usec -= in->tv_usec) < 0) { /* out -= in */
6         --out->tv_sec;
7         out->tv_usec += 1000000;
8     }
9     out->tv_sec -= in->tv_sec;
10 }
```

В листинге 28.6 приведена функция proc_v4, обрабатывающая все принимаемые сообщения ICMPv4. Можно также обратиться к рис. А.1, на котором изображен формат заголовка IPv4. Кроме того, следует осознавать, что к тому моменту, когда процесс получает на символьном сокете ICMP-сообщение, ядро уже проверило, что основные поля в заголовке IPv4 и в сообщении ICMPv4 действительны [128, с. 214, с. 311].

Листинг 28.6. Функция proc_v4: обработка сообщений ICMPv4

```
//ping/prov_v4.c
1 #include "ping.h"

2 void
3 proc_v4(char *ptr, ssize_t len, struct msghdr *msg, struct timeval *tvrecv)
4 {
5     int hlen1, icmplen;
6     double rtt;
7     struct ip *ip;
8     struct icmp *icmp;
9     struct timeval *tvsend;
```

```

10 ip = (struct ip*)ptr; /* начало IP-заголовка */
11 hlen1 = ip->ip_hl << 2; /* длина IP-заголовка */
12 if (ip->ip_p != IPPROTO_ICMP)
13     return; /* не ICMP */

14 icmp = (struct icmp*)(ptr + hlen1); /* начало ICMP-заголовка */
15 if ((icmplen = len - hlen1) < 8)
16     return; /* плохой пакет */

17 if (icmp->icmp_type == ICMP_ECHOREPLY) {
18     if (icmp->icmp_id != pid)
19         return; /* это не ответ на наш ECHO_REQUEST */
20     if (icmplen < 16)
21         return; /* недостаточно данных */

22 tvsend = (struct timeval*)icmp->icmp_data;
23 tv_sub(tvrecv, tvsend);
24 rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;

25 printf("%d bytes from %s: seq=%u, ttl=%d, rtt=%.3f ms\n",
26        icmplen, Sock_ntop_host(pr->sarecv, pr->salen),
27        icmp->icmp_seq, ip->ip_ttl, rtt);

28 } else if (verbose) {
29     printf(" %d bytes from %s: type = %d, code = %d\n",
30            icmplen, Sock_ntop_host(pr->sarecv, pr->salen),
31            icmp->icmp_type, icmp->icmp_code);
32 }
33 }

```

Извлечение указателя на ICMP-заголовок

10-16 Значение поля длины заголовка IPv4, умноженное на 4, дает размер заголовка IPv4 в байтах. (Следует помнить, что IPv4-заголовок может содержать параметры.) Это позволяет нам установить указатель icmp так, чтобы он указывал на начало ICMP-заголовка. Мы проверяем, относится ли данный пакет к протоколу ICMP и имеется ли в нем достаточно данных для проверки временной отметки, включенной нами в эхо-запрос. На рис. 28.3 приведены различные заголовки, указатели и длины, используемые в коде.

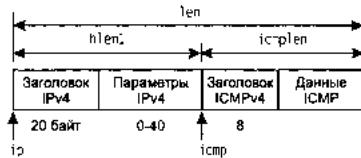


Рис. 28.3. Заголовки, указатели и длина при обработке ответов ICMPv4

Проверка эхо-ответа ICMP

17-21 Если сообщение является эхо-ответом ICMP, то необходимо проверить поле идентификатора, чтобы выяснить, относится ли этот ответ к посланному данным процессом запросу. Если программа ping запущена на одном узле несколько раз, каждый процесс получает копии всех полученных ICMP-сообщений.

22-27 Путем вычитания времени отправки сообщения (содержащегося в части ICMP-ответа, отведенной под дополнительные данные) из текущего времени (на которое указывает аргумент функции tvrecv) вычисляется значение RTT. Время RTT преобразуется из микросекунд в миллисекунды и

выводится на экран вместе с полем порядкового номера и полученным значением TTL. Поле порядкового номера позволяет пользователю проследить, не были ли пакеты пропущены, переупорядочены или дублированы, а значение TTL показывает количество транзитных узлов между двумя узлами.

Выход всех полученных ICMP-сообщений при включении параметра verbose

28-32 Если пользователем указан параметр командной строки -v, также выводятся поля типа и кода из всех других полученных ICMP-сообщений.

Обработка сообщений ICMPv6 управляется функцией proc_v6, приведенной в листинге 28.8. Она аналогична функции proc_v4, представленной в листинге 28.6. Однако поскольку символьные сокеты IPv6 не передают процессу заголовок IPv6, ограничение на количество транзитных узлов приходится получать в виде вспомогательных данных. Для этого нам приходится подготавливать сокет функцией init_v6, представленной в листинге 28.7.

Листинг 28.7. Функция init_v6: подготовка сокета

```
1 void
2 init_v6()
3 {
4 #ifdef IPV6
5     int on = 1;
6
7     if (verbose == 0) {
8         /* установка фильтра, пропускающего только пакеты ICMP6_ECHO_REPLY. если
9            не включен параметр verbose (вывод всех ICMP-сообщений) */
10        struct icmp6_filter myfilt;
11        ICMP6_FILTER_SETBLOCKALL(&myfilt);
12        ICMP6_FILTER_SETPASS(ICMP6_ECHO_REPLY, &myfilt);
13        setsockopt(sockfd, IPPROTO_IPV6, ICMP6_FILTER, &myfilt,
14                    sizeof(myfilt));
15        /* игнорируем ошибку, потому что фильтр - необязательная оптимизация */
16    }
17
18    /* следующую ошибку тоже игнорируем; придется обойтись без вывода
19       ограничения на количество транзитных узлов */
20 #ifdef IPV6_RECVHOPLIMIT
21    /* RFC 3542 */
22    setsockopt(sockfd, IPPROTO_IPV6, IPV6_RECVHOPLIMIT, &on, sizeof(on));
23 #else
24    /* RFC 2292 */
25    setsockopt(sockfd, IPPROTO_IPV6, IPV6_HOPLIMIT, &on, sizeof(on));
26 #endif
27 #endif
28 }
```

Приведенная в листинге 28.8 функция proc_v6 обрабатывает входящие пакеты.

Листинг 28.8. Функция proc_v6: обработка сообщений ICMPv6

```
//ping/proc_v6.c
1 #include "ping.h"

2 void
3 proc_v6(char *ptr, ssize_t len, struct msghdr *msg, struct timeval* tvrecv)
4 {
5 #ifdef IPV6
6     double rtt;
7     struct icmp6_hdr *icmp6;
8     struct timeval *tvsend;
9     struct cmsghdr *cmsg;
```

```

10 int hlim;

11 icmp6 = (struct icmp6_hdr*)ptr;
12 if (len < 8)
13     return; /* плохой пакет */

14 if (icmp6->icmp6_type == ICMP6_ECHO_REPLY) {
15     if (icmp6->icmp6_id != pid)
16         return; /* это не ответ на наш ECHO_REQUEST */
17     if (len < 16)
18         return; /* недостаточно данных */

19 tvsend = (struct timeval*)(icmp6 + 1);
20 tv_sub(tvrecv, tvsend);
21 rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;

22 hlim = -1;
23 for (cmsg = CMSG_FIRSTHDR(msg); cmsg != NULL;
24      cmsg = CMSG_NXTHDR(msg, cmsg)) {
25     if (cmsg->cmsg_level == IPPROTO_IPV6 &&
26         cmsg->cmsg_type == IPV6_HOPLIMIT) {
27         hlim = *(u_int32_t*)CMSG_DATA(cmsg);
28         break;
29     }
30 }
31 printf("%d bytes from %s; seq=%u, hlim=%u",
32        len, Sock_ntop_host(pr->sarecv, pr->salen), icmp6->icmp6_seq);
33 if (hlim == -1)
34     printf("???\n"); /* отсутствуют вспомогательные данные */
35 else
36     printf("%d", hlim);
37     printf(", rtt=%f ms\n", rtt);
38 } else if (verbose) {
39     printf(" %d bytes from type = %d, code = %d\n",
40            len, Sock_ntop_host(pr->sarecv, pr->salen));
41     icmp6->icmp6_type, icmp6->icmp6_code);
42 }
43 #endif /* IPV6 */
44 }

```

Извлечение указателя на заголовок ICMPv6

11-13 Заголовок ICMPv6 возвращается внутри данных при чтении из сокета. (Напомним, что дополнительные заголовки IPv6, если они присутствуют, всегда возвращаются не как стандартные данные, а как вспомогательные.) На рис. 28.4 приведены различные заголовки, указатели и длина, используемые в коде.

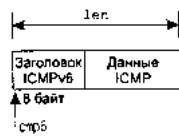


Рис. 28.4. Заголовки, указатели и длина при обработке ответов ICMPv6

Проверка эхо-ответа ICMP

14-37 Если ICMP-сообщение является эхо-ответом, то чтобы убедиться, что ответ предназначен для нас, мы проверяем поле идентификатора. Если это подтверждается, то вычисляется значение RTT, которое затем выводится вместе с порядковым номером и предельным количеством транзитных узлов IPv4. Ограничение на количество транзитных узлов мы получаем из вспомогательных данных IPV6_HOPLIMIT.

Выход всех полученных ICMP-сообщений при включении параметра verbose

38-42 Если пользователь указал параметр командной строки `-v`, выводятся также поля типа и кода всех остальных получаемых ICMP-сообщений.

Обработчиком сигнала SIGALRM является функция `sig_alrm`, приведенная в листинге 28.9. В листинге 28.4 функция `readloop` вызывает обработчик сигнала один раз для отправки первого пакета. Эта функция в зависимости от протокола вызывает функцию `send_v4` или `send_v6` для отправки эхо-запроса ICMP и далее программирует запуск другого сигнала SIGALRM через 1 с.

Листинг 28.9. Функция `sig_alrm`: обработчик сигнала SIGALRM

```
//ping/sig_alrm.c
1 #include "ping.h"

2 void
3 sig_alrm(int signo)
4 {
5     (*pr->fsend)();

6     alarm(1);
7     return;
8 }
```

Функция `send_v4`, приведенная в листинге 28.10, строит ICMPv4 сообщение эхо-запроса и записывает его в символьный сокет.

Листинг 28.10. Функция `send_v4`: построение эхо-запроса ICMPv4 и его отправка

```
//ping/send_v4.c
1 #include "ping.h"

2 void
3 send_v4(void)
4 {
5     int len;
6     struct icmp *icmp;

7     icmp = (struct icmp*)sendbuf;
8     icmp->icmp_type = ICMP_ECHO;
9     icmp->icmp_code = 0;
10    icmp->icmp_id = pid;
11    icmp->icmp_seq = nsent++;
12    memset(icmp->icmp_data, 0xa5, datalen); /* заполнение по шаблону */
13    Gettimeofday((struct timeval*)icmp->icmp_data, NULL);
14    len = 8 + datalen; /* контрольная сумма по заголовку и данным */
15    icmp->icmp_cksum = 0;
16    icmp->icmp_cksum = in_cksum((u_short*)icmp, len);
17    Sendto(sockfd, sendbuf, len, 0, pr->sasend, pr->salen);
18 }
```

Формирование ICMP-сообщения

7-13 ICMPv4 сообщение сформировано. В поле идентификатора установлен идентификатор нашего процесса, а порядковый номер установлен как глобальная переменная `nset`, которая затем увеличивается на

1 для следующего пакета. Текущее время сохраняется в части данных ICMP-сообщения.

Вычисление контрольной суммы ICMP

14-16 Для вычисления контрольной суммы ICMP значение поля контрольной суммы устанавливается равным 0, затем вызывается функция `in_cksum`, а результат сохраняется в поле контрольной суммы. Контрольная сумма ICMPv4 вычисляется по ICMPv4-заголовку и всем следующим за ним данным.

Отправка дейтаграммы

17 ICMP-сообщение отправлено на символьный сокет. Поскольку параметр сокета `IP_HDRINCL` не установлен, ядро составляет заголовок IPv4 и добавляет его в начало нашего буфера.

Контрольная сумма Интернета является суммой обратных кодов 16-разрядных значений. Если длина данных является нечетным числом, то для вычисления контрольной суммы к данным дописывается один нулевой байт. Перед вычислением контрольной суммы поле контрольной суммы должно быть установлено в 0. Такой алгоритм применяется для вычисления контрольных сумм IPv4, ICMPv4, IGMPv4, ICMPv6, UDP и TCP. В RFC 1071 [12] содержится дополнительная информация и несколько числовых примеров. В разделе 8.7 книги [128] более подробно рассказывается об этом алгоритме, а также приводится более эффективная его реализация. В нашем случае контрольную сумму вычисляет функция `in_cksum`, приведенная в листинге 28.11.

Листинг 28.11. Функция `in_cksum`: вычисление контрольной суммы Интернета

```
//libfree/in_cksum.c
1 uint16_t
2 in_cksum(uint16_t *addr, int len)
3 {
4     int nleft = len;
5     uint32_t sum = 0;
6     uint16_t *w = addr;
7     uint16_t answer = 0;

8     /*
9      * Наш алгоритм прост: к 32-разрядному аккумулятору sum мы добавляем
10     * 16-разрядные слова, а затем записываем все биты переноса из старших
11     * 16 разрядов в младшие 16 разрядов.
12     */
13    while (nleft > 1) {
14        sum += *w++;
15        nleft -= 2;
16    }

17    /* при необходимости добавляем четный байт */
18    if (nleft == 1) {
19        *(unsigned char*)(&answer) = *(unsigned char*)w;
20        sum += answer;
21    }

22    /* перемещение битов переноса из старших 16 разрядов в младшие */
23    sum = (sum >> 16) + (sum & 0xffff); /* добавление старших 16 к младшим */
24    sum += (sum >> 16); /* добавление переноса */
25    answer = ~sum; /* обрезаем по 16 разрядам */
26    return(answer);
27 }
```

Алгоритм вычисления контрольной суммы Интернета

1-27 Первый цикл `while` вычисляет сумму всех 16-битовых значений. Если длина нечетная, то к сумме добавляется конечный байт. Алгоритм, приведенный в листинге 28.11, является простым алгоритмом, подходящим для программы `ping`, но неудовлетворительным для больших объемов вычислений контрольных сумм, производимых ядром.

ПРИМЕЧАНИЕ

Эта функция взята из общедоступной версии программы `ping`, написанной Майком Миуссом (Mike Muuss).

Последней функцией нашей программы `ping` является функция `send_v6`, приведенная в листинге 28.12, которая формирует и посыпает эхо-запросы ICMPv6.

Функция `send_v6` аналогична функции `send_v4`, но обратите внимание, что она не вычисляет контрольную сумму. Как отмечалось ранее, поскольку для вычисления контрольной суммы ICMPv6 используется адрес отправителя из IPv6-заголовка, данная контрольная сумма вычисляется для нас ядром, после того как ядро выяснит адрес отправителя.

Листинг 28.12. Функция `send_v6`: построение и отправка ICMPv6-сообщения эхо-запроса

```
//ping/send_v6.c
1 #include "ping.h"

2 void
3 send_v6()
4 {
5 #ifdef IPV6
6     int len;
7     struct icmp6_hdr *icmp6;

8     icmp6 = (struct icmp6_hdr*)sendbuf,
9     icmp6->icmp6_type = ICMP6_ECHO_REQUEST;
10    icmp6->icmp6_code = 0;
11    icmp6->icmp6_id = pid;
12    icmp6->icmp6_seq = nsent++;
13    memset((icmp6 + 1), 0xa5, datalen); /* заполнение по шаблону */
14    Gettimeofday((struct timeval*)(icmp6 + 1), NULL);

15    len = 8 + datalen; /* 8-байтовый заголовок ICMPv6 */

16    Sendto(sockfd, sendbuf, len, 0, pr->sasend, pr->salen);
17    /* ядро вычисляет и сохраняет контрольную сумму само */
18 #endif /* IPV6 */
19 }
```

28.6. Программа traceroute

В этом разделе мы приведем собственную версию программы `traceroute`. Как и в случае с программой `ping`, приведенной в предыдущем разделе, мы представляем нашу собственную, а не общедоступную версию. Это делается для того, чтобы во-первых, получить версию, поддерживающую как IPv4, так и IPv6, а во-вторых, не отвлекаться на множество параметров, не относящихся к обсуждению сетевого программирования.

Программа `traceroute` позволяет нам проследить путь IP-дейтаграмм от нашего узла до получателя. Ее действие довольно просто, а в главе 8 книги [111] оно детально описано со множеством примеров.

В версии IPv6 программа traceroute использует поле TTL (в версии IPv4) или поле предельного количества транзитных узлов (называемое также полем ограничения пересылок), а также два типа ICMP-сообщений. Эта программа начинает свою работу с отправки UDP-дейтаграммы получателю, причем полю TTL (ограничения пересылок) присваивается значение 1. Такая дейтаграмма вынуждает первый маршрутизатор отправить ICMP-сообщение об ошибке «Time exceeded in transit» (Превышено время передачи). Затем значение TTL увеличивается на 1 и посыпается следующая UDP-дейтаграмма, которая достигает следующего маршрутизатора. Когда UDP-дейтаграмма достигает конечного получателя, необходимо заставить узел вернуть ICMP-ошибку Port unreachable (Порт недоступен). Для этого UDP-дейтаграмма посыпается на случайный порт, который (как можно надеяться) не используется на данном узле.

Ранние версии программы traceroute могли устанавливать поле TTL в заголовке IPv4 только с помощью параметра сокета IP_HDRINCL путем построения своего собственного заголовка. Однако современные системы поддерживают параметр сокета IP_TTL, позволяющий определить значение TTL для исходящих дейтаграмм. (Данный параметр сокета впервые был представлен в выпуске 4.3BSD Reno.) Потом установить данный параметр сокета, чем полностью формировать IPv4-заголовок (хотя в разделе 29.7 показано, как строить собственные заголовки IPv4 и UDP). Параметр сокета IPv6 IPV6_UNICAST_HOPS позволяет контролировать поле предельного количества транзитных узлов (ограничения пересылок) в дейтаграммах IPv6.

В листинге 28.13 приведен заголовочный файл trace.h, подключаемый ко всем файлам нашей программы.

Листинг 28.13. Заголовочный файл trace.h

```
//traceroute/trace.h
1 #include "unp.h"
2 #include <netinet/in_systm.h>
3 #include <netinet/ip.h>
4 #include <netinet/ip_icmp.h>
5 #include <netinet/udp.h>

6 #define BUFSIZE 1500

7 struct rec { /* структура данных UDP */
8     u_short rec_seq; /* порядковый номер */
9     u_short rec_ttl; /* значение TTL, с которым пакет отправляется */
10    struct timeval rec_tv; /* время отправки пакета */
11};

12 /* глобальные переменные */
13 char recvbuf[BUFSIZE];
14 char sendbuf[BUFSIZE];

15 int datalen; /* размер данных в байтах после заголовка ICMP */
16 char *host;
17 u_short sport, dport;
18 int nsent; /* добавляет 1 для каждого вызова sendto() */
19 pid_t pid; /* идентификатор нашего процесса PID */
20 int probe, nprobes;
21 int sendfd, recvfd; /* посыпает на сокет UDP. читает на
                           символном сокете ICMP */
22 int ttl, max_ttl;
23 int verbose;

24 /* прототипы функций */
25 char *icmpcode_v4(int);
26 char *icmpcode_v6(int);
27 int recv_v4(int, struct timeval*);
28 int recv_v6(int, struct timeval*);
```

```

29 void sig_alrm(int);
30 void traceloop(void);
31 void tv_sub(struct timeval*, struct timeval*);

32 struct proto {
33     char (*icmpcode)(int);
34     int (*recv)(int, struct timeval*);
35     struct sockaddr *sasend; /* структура sockaddr{} для отправки.
36                               получена из getaddrinfo */
37     struct sockaddr *sarecv; /* структура sockaddr{} для получения */
38     struct sockaddr *salast; /* последняя структура sockaddr{} для получения */
39     struct sockaddr *sabind; /* структура sockaddr{} для связывания
39                               порта отправителя*/
40     socklen_t salen; /* длина структур sockaddr{}s */
41     int icmpproto; /* значение IPPROTO_XXX для ICMP */
42     int ttl_level; /* значение аргумента level функции
42                      setsockopt() для задания TTL */
43     int ttloptname; /* значение аргумента name функции
43                      setsockopt() для задания TTL */
44 } *pr;

44 #ifdef IPV6

45 #include "ip6.h" /* должно быть <netinet/ip6.h> */
46 #include "icmp6.h" /* должно быть <netinet/icmp6.h> */

47 #endif
1-11 Подключаются стандартные заголовочные файлы IPv4, определяющие структуры и константы
IPv4, ICMPv4 и UDP. Структура rec определяет часть посылаемой UDP-дейтаграммы, содержащую
собственно данные, но, как мы увидим дальше, нам никогда не придется исследовать эти данные. Они
отсылаются в основном для целей отладки.

```

Определение структуры proto

32-43 Как и в программе ping, описанной в предыдущем разделе, мы обрабатываем различие между протоколами IPv4 и IPv6, определяя структуру proto, которая содержит указатели на функции, указатели на структуры адресов сокетов и другие константы, различные для двух версий IP. Глобальная переменная pr будет установлена как указатель на одну из этих структур, инициализированных либо для IPv4, либо для IPv6, после того как адрес получателя будет обработан функцией main (поскольку именно адрес получателя определяет, какая версия используется — IPv4 или IPv6).

Подключение заголовочных файлов IPv6

44-47 Подключаются заголовочные файлы, определяющие структуры и константы IPv6 и ICMPv6. Функция main приведена в листинге 28.14. Она обрабатывает аргументы командной строки, инициализирует указатель pr либо для IPv4, либо для IPv6 и вызывает нашу функцию traceloop.

Листинг 28.14. Функция main программы traceroute

```

//traceroute/main.c
1 #include "trace.h"

2 struct proto proto_v4 =
3 {icmpcode_v4, recv_v4, NULL, NULL, NULL, NULL, 0,
4 IPPROTO_ICMP, IPPROTO_IP, IP_TTL};

```

```

5 #ifdef IPV6
6 struct proto proto_v6 =
7 {icmpcode_v6, recv_v6, NULL, NULL, NULL, NULL, 0,
8 IPPROTO_ICMPV6, IPPROTO_IPV6, IPV6_UNICAST_HOPS};
9#endif

10 int datalen = sizeof(struct rec); /* значения по умолчанию */
11 int max_ttl = 30;
12 int nprobes = 3;
13 u_short dport = 32768 + 666;

14 int
15 main(int argc, char **argv)
16 {
17 int c;
18 struct addrinfo *ai;

19 opterr = 0; /* чтобы функция getopt() не записывала в stderr */
20 while ((c = getopt(argc, argv, "m:v")) != -1) {
21 switch (c) {
22 case 'm':
23 if ((max_ttl = atoi(optarg)) <= 1)
24 err_quit("invalid -m value");
25 break;

26 case 'v':
27 verbose++;
28 break;

29 case '?':
30 err_quit("unrecognized option: %c", c);
31 }
32 }

33 if (optind != argc - 1)
34 err_quit("usage: traceroute [ -m <maxttl> -v ] <hostname>");
35 host = argv[optind];

36 pid = getpid();
37 Signal(SIGALRM, sig_alarm);

38 ai = Host_serv(host, NULL, 0, 0);

39 printf("traceroute to %s (%s): %d hops max, %d data bytes\n",
40 ai->ai_canonname,
41 Sock_ntop_host(ai->ai_addr, ai->ai_addrlen);
42 max_ttl, datalen);

43 /* инициализация в зависимости от протокола */
44 if (ai->ai_family == AF_INET) {
45 pr = &proto_v4;
46#endif IPV6
47 } else if (ai->ai_family == AF_INET6) {
48 pr = &proto_v6;
49 if (IN6_IS_ADDR_V4MAPPED
50 (&((struct sockaddr_in6*)ai->ai_addr)->sin6_addr)))

```

```

51     err_quit("cannot traceroute IPv4-mapped IPv6 address");
52 #endif
53 } else
54     err_quit("unknown address family %d", ai->ai_family);
55     pr->sasend = ai->ai_addr; /* содержит адрес получателя */
56     pr->sarecv = Calloc(1, ai->ai_addrlen);
57     pr->salast = Calloc(1, ai->ai_addrlen);
58     pr->sabind = Calloc(1, ai->ai_addrlen);
59     pr->salen = ai->ai_addrlen;

60     traceloop();

61     exit(0);
62 }

```

Определение структуры proto

2-9 Определяются две структуры *proto*, одна для IPv4 и другая для IPv6, хотя указатели на структуры адреса сокета не размещаются в памяти до окончания выполнения данной функции.

Установка значений по умолчанию

10-13 Максимальное значение поля TTL или поля предельного количества транзитных узлов, используемое в программе, по умолчанию равно 30. Предусмотрен параметр командной строки *-m*, чтобы пользователь мог поменять это значение. Для каждого значения TTL посыпается три пробных пакета, но их количество также может быть изменено с помощью параметра командной строки. Изначально используется номер порта получателя 32 768 + 666, и каждый раз, когда посыпается новая дейтаграмма UDP, это значение увеличивается на 1. Мы можем надеяться, что порты с такими номерами не используются на узле получателя в тот момент, когда приходит дейтаграмма, однако гарантий здесь нет.

Обработка аргументов командной строки

19-37 Параметр командной строки *-v* позволяет вывести все остальные ICMP-сообщения.

Обработка имени узла или IP-адреса и завершение инициализации

38-58 Имя узла получателя или IP-адрес обрабатывается функцией *host_serv*, возвращающей указатель на структуру *addrinfo*. В зависимости от типа возвращенного адреса (IPv4 или IPv6) заканчивается инициализация структуры *proto*, сохраняется указатель в глобальной переменной *pr*, а также размещается в памяти дополнительная структура адреса сокета соответствующего размера.

Функция *traceloop*, приведенная в листинге 28.15, отправляет дейтаграммы и читает вернувшиеся ICMP-сообщения. Это основной цикл программы.

Листинг 28.15. Функция *traceloop*: основной цикл обработки

```

//traceroute/traceloop.c
1 #include "trace.h"

2 void
3 traceloop(void)
4 {
5     int seq, code, done;
6     double rtt;
7     struct rec *rec;

```

```

8 struct timeval tvrecv;

9 recvfd = Socket(pr->sasend->sa_family, SOCK_RAW, pr->icmppproto);
10 setuid(getuid()); /* права привилегированного пользователя больше
не нужны */

11 #ifdef IPV6
12 if (pr->sasend->sa_family == AF_INET6 && verbose == 0) {
13     struct icmp6_filter myfilt;
14     ICMP6_FILTER_SETBLOCKALL(&myfilt);
15     ICMP6_FILTER_SETPASS(ICMP6_TIME_EXCEEDED, &myfilt);
16     ICMP6_FILTER_SETPASS(ICMP6_DST_UNREACH, &myfilt);
17     setsockopt(recvfd, IPPROTO_IPV6, ICMP6_FILTER,
18                 &myfilt, sizeof(myfilt));
19 }
20#endif

21 sendfd = Socket(pr->sasend->sa_family, SOCK_DGRAM, 0);

22 pr->sabind->sa_family = pr->sasend->sa_family;
23 sport = (getpid() & 0xffff) | 0x8000; /* UDP-порт отправителя # */
24 sock_set_port(pr->sabind, pr->salen, htons(sport));
25 Bind(sendfd, pr->sabind, pr->salen);

26 sig_alrm(SIGALRM);

27 seq = 0;
28 done = 0;
29 for (ttl = 1; ttl <= max_ttl && done == 0; ttl++) {
30     Setsockopt(sendfd, pr->ttllevel, pr->ttloptname, &ttl, sizeof(int));
31     bzero(pr->salast, pr->salen);

32     printf("%2d ", ttl);
33     fflush(stdout);

34     for (probe = 0; probe < nprobes; probe++) {
35         rec = (struct rec*)sendbuf;
36         rec->rec_seq = ++seq;
37         rec->rec_ttl = ttl;
38         Gettimeofday(&rec->rec_tv, NULL);

39         sock_set_port(pr->sasend, pr->salen, htons(dport + seq));
40         Sendto(sendfd, sendbuf, datalen, 0, pr->sasend, pr->salen);
41         if ((code = (*pr->recv)(seq, &tvrecv)) == -3)
42             printf(" *"); /* тайм-аут, ответа нет */
43         else {
44             char str[NI_MAXHOST];

45             if (sock_cmp_addr(pr->sarecv, pr->salast, pr->salen) != 0) {
46                 if (getnameinfo(pr->sarecv, pr->salen, str, sizeof(str),
47                                 NULL, 0, 0) == 0)
48                     printf(" %s (%s)", str,
49                           Sock_ntop_host(pr->sarecv, pr->salen));
50             else
51                 printf(" %s", Sock_ntop_host(pr->sarecv, pr->salen));
52             memcpy(pr->salast, pr->sarecv, pr->salen);

```

```

53     }
54     tv_sub(&tvrecv, &rec->rec_tv);
55     rtt = tvrecv.tv_sec * 1000.0 + tvrecv.tv_usec / 1000.0;
56     printf(" %.3f ms", rtt);

57     if (code == -1) /* порт получателя недоступен */
58         done++;
59     else if (code >= 0)
60         printf("(ICMP %s)", (*pr->icmpcode)(code));
61     }
62     fflush(stdout);
63 }
64 printf("\n");
65 }
66 }
```

Создание двух сокетов

9-10 Нам необходимо два сокета: символьный сокет, на котором мы читаем все вернувшиеся ICMP-сообщения, и UDP-сокет, на который мы посылаем пробные пакеты с увеличивающимся значением поля TTL. После создания символьного сокета мы заменяем наш эффективный идентификатор пользователя на фактический, поскольку более нам не понадобятся права привилегированного пользователя.

Установка фильтра ICMPv6

11-20 Если мы отслеживаем маршрут к адресату IPv6 и параметр командной строки **-V** указан не был, можно установить фильтр, который будет блокировать все ICMP-сообщения, за исключением тех, которые нас интересуют: «Time exceeded» и «Destination unreachable». Это сократит число пакетов, получаемых на данном сокете.

Связывание порта отправителя UDP-сокета

21-25 Осуществляется связывание порта отправителя с UDP-сокетом, который используется для отправки пакетов. При этом берется 16 младших битов из идентификатора нашего процесса, а старшему биту присваивается 1. Поскольку несколько копий программы traceroute могут работать одновременно, нам необходима возможность определить, относится ли поступившее ICMP-сообщение к одной из наших дейтаграмм или оно пришло в ответ на дейтаграмму, посланную другой копией программы. Мы используем порт отправителя в UDP-заголовке для определения отправляющего процесса, поскольку возвращаемое ICMP-сообщение всегда содержит UDP-заголовок дейтаграммы, вызвавшей ICMP-ошибку.

Установка обработчика сигнала SIGALRM

26 Мы устанавливаем нашу функцию **sig_alarm** в качестве обработчика сигнала **SIGALRM**, поскольку каждый раз, когда мы посылаем UDP-дейтаграмму, мы ждем 3 с, прежде чем послать следующий пробный пакет.

Основной цикл: установка TTL или предельного количества транзитных узлов и отправка трех пробных пакетов

27-38 Основным циклом функции является двойной вложенный цикл **for**. Внешний цикл стартует со значениями TTL или предельного количества транзитных узлов, равного 1, и увеличивает это значение на 1, в

то время как внутренний цикл посыпает три пробных пакета (UDP-дейтаграммы) получателю. Каждый раз, когда изменяется значение TTL, мы вызываем `setsockopt` для установки нового значения, используя параметр сокета `IP_TTL` или `IPV6_UNICAST_HOPS`.

Каждый раз во внешнем цикле мы инициализируем нулем структуру адреса сокета, на которую указывает `salast`. Данная структура будет сравниваться со структурой адреса сокета, возвращенной функцией `recvfrom`, при считывании ICMP-сообщения, и если эти две структуры будут различны, на экран будет выведен IP-адрес из новой структуры. При использовании этого метода для каждого значения TTL выводится IP-адрес, соответствующий первому пробному пакету, а если для данного значения TTL IP-адрес изменится (то есть во время работы программы изменится маршрут), то будет выведен новый IP-адрес.

Установка порта получателя и отправка UDP-дейтаграммы

39-40 Каждый раз, когда посыпается пробный пакет, порт получателя в структуре адреса сокета `sasend` меняется с помощью вызова функции `sock_set_port`. Причина, по которой порт меняется для каждого пробного пакета, заключается в том, что когда мы достигаем конечного получателя, все три пробных пакета посыпаются на разные порты, чтобы увеличить шансы на обращение к неиспользуемому порту. Функция `sendto` посыпает UDP-дейтаграмму.

Чтение ICMP-сообщения

41-42 Одна из функций `recv_v4` или `recv_v6` вызывает функцию `recvfrom` для чтения и обработки вернувшихся ICMP-сообщений. Обе эти функции возвращают значение -3 в случае истечения времени ожидания (сообщая, что следует послать следующий пробный пакет, если для данного значения TTL еще не посланы все три пакета), значение -2, если приходит ICMP-ошибка о превышении времени передачи, и значение -1, если получена ICMP-ошибка «Port unreachable» (Порт недоступен), то есть достигнут конечный получатель. Если же приходит какая-либо другая ICMP-ошибка недоступности получателя («Destination unreachable»), эти функции возвращают неотрицательный ICMP-код.

Выход ответа

43-63 Как отмечалось выше, в случае первого ответа для данного значения TTL, а также если для данного TTL меняется IP-адрес узла, посыпающего ICMP-сообщение, выводится имя узла и IP-адрес (или только IP-адрес, если вызов функции `getnameinfo` не возвращает имени узла). Время RTT вычисляется как разность между временем отправки пробного пакета и временем возвращения и вывода ICMP-сообщения.

Функция `recv_v4` приведена в листинге 28.16.

Листинг 28.16. Функция `recv_v4`: чтение и обработка сообщений ICMPv4

```
//traceroute/recv_v4
1 #include "trace.h"

2 extern int gotalarm;

3 /* Возвращает:
4  * -3 при тайм-ауте
5  * -2 при сообщении ICMP time exceeded in transit (продолжаем поиск)
6  * -1 при сообщении ICMP port unreachable (цель достигнута)
7  * неотрицательные значения соответствуют всем прочим ошибкам ICMP
8 */

9 int
10 recv_v4(int seq, struct timeval *tv)
11 {
12     int hlen1, hlen2, icmplen, ret;
```

```

13  socklen_t len;
14  ssize_t n;
15  struct ip *ip, *hip;
16  struct icmp *icmp;
17  struct udphdr *udp;

18  gotalarm = 0;
19  alarm(3);
20  for (;;) {
21    if (gotalarm)
22      return(-3); /* истек таймер */
23    len = pr->salen;
24    n = recvfrom(recvfd, recvbuf, sizeof(recvbuf), 0, pr->sarecv, &len);
25    if (n < 0) {
26      if (errno == EINTR)
27        continue;
28      else
29        err_sys("recvfrom error");
30    }

31    ip = (struct ip*)recvbuf; /* начало IP-заголовка */
32    hlen1 = ip->ip_hl << 2; /* длина IP-заголовка */

33    icmp = (struct icmp*)(recvbuf + hlen1); /* начало ICMP-заголовка */
34    if ((icmplen = n - hlen1) < 8)
35      continue; /* недостаточно данных для проверки ICMP-заголовка */

36    if (icmp->icmp_type == ICMP_TIMXCEED &&
37        icmp->icmp_code == ICMP_TIMXCEED_INTRANS) {
38      if (icmplen < 8 + sizeof(struct ip))
39        continue; /* недостаточно данных для проверки внутреннего IP */

40    hip = (struct ip*)(recvbuf + hlen1 + 8);
41    hlen2 = hip->ip_hl << 2;
42    if (icmplen < 8 + hlen2 + 4)
43      continue; /* недостаточно данных для проверки UDP-порта */

44    udp = (struct udphdr*)(recvbuf + hlen1 + 8 + hlen2);
45    if (hip->ip_p == IPPROTO_UDP &&
46        udp->uh_sport == htons(sport) &&
47        udp->uh_dport == htons(dport + seq)) {
48      ret = -2; /* ответил промежуточный маршрутизатор */
49      break;
50    }

51  } else if (icmp->icmp_type == ICMP_UNREACH) {
52    if (icmplen < 8 + sizeof(struct ip))
53      continue; /* недостаточно данных для проверки внутреннего IP */

54    hip = (struct ip*)(recvbuf + hlen1 + 8);
55    hlen2 = hip->ip_hl << 2;
56    if (icmplen < 8 + hlen2 + 4)
57      continue; /* недостаточно данных для проверки UDP-портов */

58    udp = (struct udphdr*)(recvbuf + hlen1 + 8 + hlen2);
59    if (hip->ip_p == IPPROTO_UDP &&

```

```

60     udp->uh_sport == htons(sport) &&
61     udp->uh_dport == htons(dport + seq)) {
62     if (icmp->icmp_code == ICMP_UNREACH_PORT)
63         ret = -1; /* цель достигнута */
64     else
65         ret = icmp->icmp_code; /* 0, 1, 2, ... */
66     break;
67 }
68 }
69 if (verbose) {
70     printf(" (from %s: type = %d, code - %d)\n",
71         Sock_ntop_host(pr->sarecv, pr->salen),
72         icmp->icmp_type, icmp->icmp_code);
73 }
74 /* другая ICMP-ошибка, нужно снова вызвать recvfrom() */
75 }
76 alarm(0); /* отключаем таймер */
77 Gettimeofday(tv, NULL); /* время получения пакета */
78 return(ret);
79 }

```

Установка таймера и прочтение каждого ICMP-сообщения

17-27 Таймер устанавливается на 3 с, и функция входит в цикл, вызывающий `recvfrom`, считывая каждое ICMPv4-сообщение, возвращаемое на символьный сокет.

ПРИМЕЧАНИЕ

Эта функция не создает ситуации гонок, описанной в разделе 20.5, благодаря использованию глобального флага.

Извлечение указателя на ICMP-заголовок

31-35 Указатель `ip` указывает на начало IPv4-заголовка (напомним, что операция чтения на символьном сокете всегда возвращает IP-заголовок), а указатель `icmp` указывает на начало ICMP-заголовка. На рис. 28.5 показаны различные заголовки, указатели и длины, используемые в данном коде.

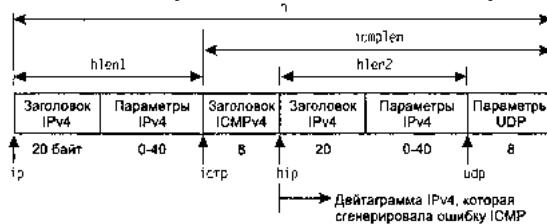


Рис. 28.5. Заголовки, указатели и длины при обработке ошибки

Обработка ICMP-сообщения о превышении времени передачи

36-50 Если ICMP-сообщение является сообщением «Time exceeded in transit» (Превышено время передачи), вероятно, оно является ответом на один из наших пробных пакетов. Указатель `hip` указывает на заголовок IPv4, который возвращается в ICMP-сообщении и следует сразу за 8-байтовым ICMP-заголовком. Указатель `udp` указывает на следующий далее UDP-заголовок. Если ICMP-сообщение было сгенерировано

UDP-дейтаграммой и если порты отправителя и получателя этой дейтаграммы совпадают с теми значениями, которые мы посылали, то тогда это ответ от промежуточного маршрутизатора на наш пробный пакет.

Обработка ICMP-сообщения о недоступности порта

51-68 Если ICMP-сообщение является сообщением «Destination unreachable» (Получатель недоступен), тогда, чтобы узнать, является ли это сообщение ответом на наш пробный пакет, мы смотрим на UDP-заголовок, возвращенный в данном ICMP-сообщении. Если это так и код означает сообщение «Port unreachable» (Порт недоступен), то возвращается значение -1, поскольку достигнут конечный получатель. Если же ICMP-сообщение является ответом на один из наших пробных пакетов, но не является сообщением типа «Destination unreachable» (Получатель недоступен), то тогда возвращается значение ICMP-кода. Обычным примером такого случая является ситуация, когда брандмауэр возвращает какой-либо другой код недоступности для получателя, на который посыпается пробный пакет.

Обработка других ICMP-сообщений

69-73 Все остальные ICMP-сообщения выводятся, если был задан параметр `-v`.

Следующая функция, `recv_v6`, приведена в листинге 28.18 и является IPv6-версией ранее описанной функции для IPv4. Эта функция почти идентична функции `recv_v4`, за исключением различий в именах констант и элементов структур. Кроме того, размер заголовка IPv6 является фиксированным и составляет 40 байт, в то время как для получения IP-параметров в заголовке IPv4 необходимо получить поле длины заголовка и умножить его на 4. На рис. 28.6 приведены различные заголовки, указатели и длины, используемые в коде.

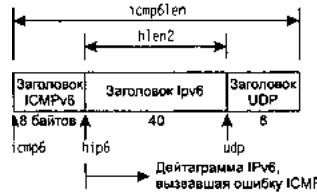


Рис. 28.6. Заголовки, указатели и длины, используемые при обработке ошибки ICMPv6

Мы определяем две функции `icmpcode_v4` и `icmpcode_v6`, которые можно вызывать в конце функции `traceroute` для вывода строки описания, соответствующей ICMP-ошибке недоступности получателя. В листинге 28.19 приведена IPv6-функция. IPv4-функция аналогична, хотя и длиннее, поскольку существует большее количество ICMPv4-кодов недоступности получателя (см. табл. А.5).

Последней функцией в нашей программе `traceroute` является обработчик сигнала `SIGALRM` — функция `sig_alrm`, приведенная в листинге 28.17. Эта функция лишь возвращает ошибку `EINTR` из функции `recvfrom`, как в случае функции `recv_v4`, так и в случае `recv_v6`.

Листинг 28.17. Функция `sig_alrm`

```
//traceroute/sig_alrm.c
1 #include "trace.h"

2 int gotalarm;
3 void
4 sig_alrm(int signo)
5 {
6     gotalarm = 1; /* установка флага, оповещающего о сигнале */
7     return; /* прерывается работа функции recvfrom() */
8 }
```

Листинг 28.18. Функция `recv_v6`: чтение и обработка сообщений ICMPv6

```
//traceroute/recv_v6
1 #include "trace.h"
```

```
2 extern int gotalarm;

3 /*
4  * Возвращает; -3 при тайм-ауте
5  * -2 для сообщения ICMP time exceeded in transit (продолжаем поиск
6  * маршрута)
7  * -1 для сообщения ICMP port unreachable (цель достигнута)
8  * неотрицательные значения соответствуют всем прочим ICMP-сообщениям
8 */

9 int
10 recv_v6(int seq, struct timeval *tv)
11 {
12 #ifdef IPV6
13     int hlen2, icmp6len, ret;
14     ssize_t n;
15     socklen_t len;
16     struct ip6_hdr *hip6;
17     struct icmp6_hdr *icmp6;
18     struct udphdr *udp;

19     gotalarm = 0;
20     alarm(3);
21     for (;;) {
22         if (gotalarm)
23             return(-3); /* истек таймер */
24         len = pr->salen;
25         n = recvfrom(recvfd, recvbuf, sizeof(recvbuf), 0, pr->sarecv, &len);
26         if (n < 0) {
27             if (errno == EINTR)
28                 continue;
29             else
30                 err_sys("recvfrom error");
31         }

32         icmp6 = (struct icmp6_hdr*)recvbuf; /* ICMP-заголовок */
33         if ((icmp6len = n) < 8)
34             continue; /* недостаточно для проверки ICMP-заголовка */

35         if (icmp6->icmp6_type == ICMP6_TIME_EXCEEDED &&
36             icmp6->icmp6_code == ICMP6_TIME_EXCEED_TRANSIT) {
37             if (icmp6len < 8 + sizeof(struct ip6_hdr) + 4)
38                 continue; /* недостаточно для проверки внутреннего заголовка */

39             hip6 = (struct ip6_hdr*)(recvbuf + 8);
40             hlen2 = sizeof(struct ip6_hdr);
41             udp = (struct udphdr*)(recvbuf + 8 + hlen2);
42             if (hip6->ip6_nxt == IPPROTO_UDP &&
43                 udp->uh_sport == htons(sport) &&
44                 udp->uh_dport == htons(dport + seq))
45                 ret = -2; /* ответил промежуточный маршрутизатор */
46             break;
47         } else if (icmp6->icmp6_type == ICMP6_DST_UNREACH) {
48             if (icmp6len < 8 + sizeof(struct ip6_hdr) + 4)
49                 continue; /* недостаточно для проверки внутреннего заголовка */

```

```

50     hip6 = (struct ip6_hdr*)(recvbuf + 8);
51     hlen2 = sizeof(struct ip6_hdr);
52     udp = (struct udphdr*)(recvbuf + 8 + hlen2);
53     if (hip6->ip6_nxt == IPPROTO_UDP &&
54         udp->uh_sport == htons(sport) &&
55         udp->uh_dport == htons(dport + seq)) {
56         if (icmp6->icmp6_code == ICMP6_DST_UNREACH_NOPORT)
57             ret = -1; /* цель достигнута */
58         else
59             ret = icmp6->icmp6_code; /* 0, 1, 2, ... */
60         break;
61     }
62 } else if (verbose) {
63     printf("(from %s: type = %d, code = %d)\n",
64            Sock_ntop_host(pr->sarecv, pr->salen);
65            icmp6->icmp6_type, icmp6->icmp6_code);
66 }
67 /* другая ICMP-ошибка. нужно вызвать recvfrom() */
68 }
69 alarm(0); /* отключаем таймер */
70 Gettimeofday(tv, NULL); /* get time of packet arrival */
71 return(ret);
72 #endif
73 }
```

Листинг 28.19. Возвращение строки, соответствующей коду недоступности ICMPv6

```

//traceroute/icmpcode_v6.c
1 #include "trace.h"

2 const char *
3 icmpcode_v6(int code)
4 {
5 #ifdef IPV6
6     static char errbuf[100];
7     switch (code) {
8     case ICMP6_DST_UNREACH_NOROUTE:
9         return("no route to host");
10    case ICMP6_DST_UNREACH_ADMIN:
11        return("administratively prohibited");
12    case ICMP6_DST_UNREACH_NOTNEIGHBOR:
13        return("not a neighbor");
14    case ICMP6_DST_UNREACH_ADDR:
15        return("address unreachable");
16    case ICMP6_DST_UNREACH_NOPORT:
17        return("port unreachable");
18    default:
19        sprintf(errbuf, "[unknown code %d]", . code);
20    }
21 #endif
22 }
```

Пример

Сначала приведем пример с Ipv4:

```

freebsd % traceroute www.unpbook.com
traceroute to www.unpbook.com (206.168.112.219): 30 hops max. 24 data bytes
1 12.106.32.1 (12.106.32.1) 0.799 ms 0.719 ms 0.540 ms
2 12.124.47.113 (12.124.47.113) 1.758 ms 1.760 ms 1.839 ms
3 gbr2-p27.sffca.ip.att.net (12.123.195.38) 2.744 ms 2.575 ms 2.648 ms
4 tbr2-p012701.sffca.ip.att.net (12.122.11.85) 3.770 ms 3.689 ms 3.848 ms
5 gbr3-p50.dvmco.ip.att.net (12.122.2.66) 26.202 ms 26.242 ms 26.102 ms
6 gbr2-p20.dvmco.ip.att.net (12.122.5.26) 26.255 ms 26.194 ms 26.470 ms
7 gar2-p370.dvmco.ip.att.net (12.123.36.141) 26.443 ms 26.310 ms 26.427 ms
8 att-46.den.internap.ip.att.net (12.124.158.58) 26.962 ms 27.130 ms 27.279 ms
9 border10 ge3-0-bbnet2.den.pnap.net (216.52.40.79) 27.285 ms 27.293 ms 26.860 ms
10 coop-2.border10.den.pnap.net (216.52.42.118) 28.721 ms 28.991 ms 30.077 ms
11 199.45.130.33 (199.45.130.33) 29.095 ms 29.055 ms 29.378 ms
12 border-to-141-netrack.boulder.co.coop.net (207.174.144.178) 30.875 ms 29.747 ms 30.142
ms
13 linux.unpbook.com (206.168.112.219) 31.713 ms 31.573 ms 33.952 ms
Ниже приведен пример с IPv6. Для лучшей читаемости длинные строки разбиты.
freebsd % traceroute www.kame.net
traceroute to orange.kame.net (2001:200:0:4819:203:47ff:fea5:3085): 30 hops max, 24 data
bytes
1 3ffe:b80:3:9ad1::1 (3ffe:b80:3:9ad1::1) 107.437 ms 99.341 ms 103.477 ms
2 Viagenie-gw.int.ipv6.ascc.net (2001:288:3b0::55)
    105.129 ms 89.418 ms 90.016 ms
3 gw-Viagenie.int.ipv6.ascc.net (2001:288:3b0::54)
    302.300 ms 291.580 ms 289.839 ms
4 c7513-gw.int.ipv6.ascc.net (2001:288:3b0::c)
    296.088 ms 298.600 ms 292.196 ms
5 m160-c7513.int.ipv6.ascc.net (2001:288:3b0::1e)
    296.266 ms 314.878 ms 302.429 ms
6 m20jp-ml60tw.int.ipv6.ascc.net (2001:288:3b0::1b)
    327.637 ms 326.897 ms 347.062 ms
7 hitachi1.otemachi.wide.ad.jp (2001:200:0:1800::9c4:2)
    420.140 ms 426.592 ms 422.756 ms
8 pc3.yagami.wide.ad.jp (2001:200:0:1c04::1000:2000)
    415.471 ms 418.308 ms 461.654 ms
9 gr2000.k2c.wide.ad.jp (2001:200:0:8002::2000:1)
    416.581 ms 422.430 ms 427.692 ms
10 2001:200:0:4819:203:47ff:fea5:3085 (2001:200:0:4819:203:47ff:fea5:3085)
    417.169 ms 434.674 ms 424.037 ms

```

28.7. Демон сообщений ICMP

Получение асинхронных ошибок ICMP на сокет UDP всегда было и продолжает оставаться проблемой. Ядро получает сообщения об ошибках ICMP, но они редко доставляются приложениям, которым необходимо о них знать. Мы видели, что для получения этих ошибок в API сокетов требуется присоединение сокета UDP к одному IP-адресу (см. раздел 8.11). Причина такого ограничения заключается в том, что единственная ошибка, возвращаемая функцией `recvfrom`, является целочисленным кодом `errno`, а если приложение посылает дейтаграммы по нескольким адресам, а затем вызывает `recvfrom`, то данная функция не может сообщить приложению, какая из дейтаграмм вызвала ошибку.

В данном разделе предлагается решение, не требующее никаких изменений в ядре. Мы предлагаем демон ICMP-сообщений `icmpd`, который создает символьный сокет ICMPv4 и символьный сокет ICMPv6 и получает все ICMP-сообщения, направляемые к ним ядром. Он также создает потоковый сокет домена Unix, связывает его (при помощи функции `bind`) с полным именем `/tmp/icmpd` и прослушивает входящие соединения (устанавливаемые при помощи функции `connect`) клиентов с этим сокетом. Схема соединений изображена на рис. 28.7.



Рис. 28.7. Демон icmpd: создание сокетов

Приложение UDP (являющееся клиентом для демона) сначала создает сокет UDP, для которого оно хочет получать асинхронные ошибки. Приложение должно связать (функция `bind`) с этим сокетом динамически назначаемый порт; для чего это делается, будет пояснено далее. Затем оно создает доменный сокет Unix и присоединяется (функция `connect`) к заранее известному полному имени файла демона. Это показано на рис. 28.8.

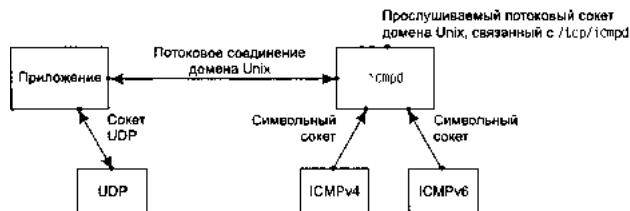


Рис. 28.8. Приложение создает свой сокет UDP и доменный сокет Unix

Далее приложение «передает» свой UDP-сокет демону через соединение домена Unix, используя технологию передачи дескрипторов, как показано в разделе 15.7. Такой подход позволяет демону получить копию сокета, так что он может вызвать функцию `getsockname` и получить номер порта, связанный с сокетом. На рис. 28.9 показана передача сокета.



Рис. 28.9. Пересылка сокета UDP демону через доменный сокет Unix

После того как демон получает номер порта, связанный с UDP-сокетом, он закрывает свою копию сокета, и мы возвращаемся к схеме, приведенной на рис. 28.8.

ПРИМЕЧАНИЕ

Если узел поддерживает передачу данных, идентифицирующих отправителя (см. раздел 15.8), приложение также может послать эти данные демону. Затем демон может проверить, можно ли допускать данного пользователя к данному устройству.

В таком случае в результате любой ошибки ICMP, полученной демоном в ответ на UDP-дейтаграмму, посланную с порта, который связан с UDP-сокетом приложения, демон посыпает приложению сообщение (о котором мы рассказываем чуть ниже) через доменный сокет Unix. Тогда приложение должно использовать функцию `select` или `poll`, чтобы обеспечить ожидание прибытия данных либо на UDP-сокет, либо на доменный сокет Unix.

Сначала рассмотрим исходный код приложения, использующего данный демон, а затем и сам демон. В листинге 28.20 приведен заголовочный файл, подключаемый к приложению, и к демону.

Листинг 28.20. Заголовочный файл unpricmpd.h

```

//icmpd/unpricmpd.h
1 #ifndef __unpricmp_h
2 #define __unpricmp_h
3
3 #include "unp.h"

```

```

4 #define ICMPD_PATH "/tmp/icmpd" /* известное имя сервера */

5 struct icmpd_err {
6 int icmpd_errno; /* EHOSTUNREACH, EMSGSIZE, ECONNREFUSED */
7 char icmpd_type; /* фактический тип ICMPv[46] */
8 char icmpd_code; /* фактический код ICMPv[46] */
9 socklen_t icmpd_len; /* длина последующей структуры sockaddr{} */
10 struct sockaddr_storage icmpd_dest; /* универсальная структура
                                         sockaddr_storage */
11};

12 #endif /* __unpicmp_h */

```

4-11 Определяются известное полное имя сервера и структура `icmpd_err`, передаваемая от сервера приложению сразу, как только получено ICMP-сообщение, которое должно быть передано данному приложению.

6-8 Проблема в том, что типы сообщений ICMPv4 отличаются численно (а иногда и концептуально) от типов сообщений ICMPv6 (см. табл. А.5 и А.6). Возвращаются реальные значения *типа* (*type*) и *кода* (*code*), но мы также отображаем соответствующие им значения *errno* (`icmpd_errno`), взятые из последнего столбца табл. А.5 и А.6. Приложение может использовать эти значения вместо зависящих от протокола значений ICMPv4 и ICMPv6. В табл. 28.1 показаны обрабатываемые сообщения ICMP и соответствующие им значения *errno*.

Таблица 28.1. Значения переменной `icmpd_errno`, сопоставляющей ошибки ICMPv4 и ICMPv6

icmpd_errno	Ошибка ICMPv4	Ошибка ICMPv6
ECONNREFUSED	Port unreachable (Порт недоступен)	Port unreachable (Порт недоступен)
EMSGSIZE	Fragmentation needed but DF bit set (Необходима фрагментация, но установлен бит DF)	Packet too big (Слишком большой пакет)
EHOSTUNREACH	Time exceeded (Превышено время передачи)	Time exceeded (Превышено время передачи)
EHOSTUNREACH	Source quench (Отключение отправителя)	Все другие сообщения о недоступности получателя (Destination unreachable)
EHOSTUNREACH	Все другие сообщения о недоступности получателя (Destination unreachable)	Все другие сообщения о недоступности получателя (Destination unreachable)

Демон возвращает пять типов ошибок ICMP:

1. «Port unreachable» (Порт недоступен) означает, что сокет не связан с портом получателя на IP-адресе получателя.

2. «Packet too big» (Слишком большой пакет) используется при определении транспортной MTU. В настоящее время нет определенного API, позволяющего UDP-приложениям осуществлять определение транспортной MTU. Если ядро поддерживает определение транспортной MTU для UDP, то обычно получение данной ошибки ICMP заставляет ядро записать новое значение транспортной MTU в таблицу маршрутизации ядра, но UDP-приложение, пославшее дейтаграмму, не извещается. Вместо этого приложение должно дождаться истечения тайм-аута и повторно послать дейтаграмму, и тогда ядро найдет новое (меньшее) значение MTU в своей таблице маршрутизации и фрагментирует дейтаграмму. Передача этой ошибки приложению позволяет ему ускорить повторную передачу дейтаграммы, и возможно, приложение сможет уменьшить размер посылаемой дейтаграммы.

3. Ошибка «Time exceeded» (Превышено время передачи) обычно возникает с кодом 0 и означает, что либо значение поля TTL (в случае IPv4), либо предельное количество транзитных узлов (в случае IPv6) достигло нуля. Обычно это свидетельствует о зацикливании маршрута, что, возможно, является временной ошибкой.

4. Ошибка «Source quench» (Отключение отправителя) ICMPv4 хотя и рассматривается в RFC 1812 [6] как устаревшая, может быть послана маршрутизаторами (или неправильно сконфигурированными узлами, действующими как маршрутизаторы). Такие ошибки означают, что пакет отброшен, и поэтому обрабатываются как ошибки недоступности получателя. Следует отметить, что в версии IPv6 нет ошибки отключения отправителя.

5. Все остальные ошибки недоступности получателя (Destination unreachable) означают, что пакет сброшен.

10 Элемент `icmpd_dest` является структурой адреса сокета, содержащей IP-адрес получателя и порта дейтаграммы, генерировавшей ICMP-ошибку. Этот элемент может быть структурой `sockaddr_in` для ICMPv4 либо структурой `sockaddr_in6` для ICMPv6. Если приложение посыпает дейтаграммы по нескольким адресам, оно, вероятно, имеет по одной структуре адреса сокета на каждый адрес. Возвращая эту информацию в структуре адреса сокета, приложение может сравнить ее со своими собственными структурами для поиска той, которая вызвала ошибку. Тип `sockaddr_storage` используется для того, чтобы в структуре можно было хранить адреса любого типа, поддерживаемого системой.

Эхо-клиент UDP, использующий демон `icmpd`

Теперь модифицируем наш эхо-клиент UDP (функцию `dg_cli`) для использования нашего демона `icmpd`. В листинге 28.21 приведена первая половина функции.

Листинг 28.21. Первая часть приложения `dg_cli`

```
//icmpd/dgcli01.c
1 #include "unpicmpd.h"

2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservadd, socklen_t servlen)
4 {
5     int icmpfd, maxfdp1;
6     char sendline[MAXLINE], recvline[MAXLINE + 1];
7     fd_set rset;
8     ssize_t n;
9     struct timeval tv;
10    struct icmpd_err icmpd_err;
11    struct sockaddr_un sun;

12    Sock_bind_wild(sockfd, pservaddr->sa_family);

13    icmpfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
14    sun.sun_family = AF_LOCAL;
15    strcpy(sun.sun_path, ICMPD_PATH);
16    Connect(icmpfd, (SA*)&sun, sizeof(sun));
17    Write_fd(icmpfd, "1", 1, sockfd);
18    n = Read(icmpfd, recvline, 1);
19    if (n != 1 || recvline[0] != '1')
20        err_quit("error creating icmp socket, n = %d, char = %c",
21        n, recvline[0]);

22    FD_ZERO(&rset);
23    maxfdp1 = max(sockfd, icmpfd) + 1;
```

2-3 Аргументы функции те же, что и во всех ее предыдущих версиях.

Связывание с универсальным адресом и динамически назначаемым портом

12 Вызываем функцию `sock_bind_wild` для связывания при помощи функции `bind` универсального IP-адреса и динамически назначаемого порта с UDP-сокетом. Таким образом копия сокета, который пересыпается демону, оказывается связана с портом, поскольку демону необходимо знать этот порт.

ПРИМЕЧАНИЕ

Демон также может произвести подобное связывание, если локальный порт не был связан с сокетом, который был передан демону, но это работает не во всех системах. В реализациях SVR4, таких как Solaris 2.5, сокеты не являются частью ядра, и когда один процесс связывает (bind) порт с совместно используемым сокетом, другой процесс при попытке использовать копию этого сокета получает ошибки. Простейшее решение — потребовать, чтобы приложение связывало локальный порт прежде, чем передавать сокет демону.

Установление доменного соединения Unix с демоном

13-16 Мы создаем сокет семейства AF_INET и подключаемся к известному имени сервера при помощи вызова connect.

Отправка UDP-сокета демону, ожидание ответа от демона

17-21 Вызываем функцию write_fd, приведенную в листинге 15.11 для отправки копии UDP-сокета демону. Мы также посылаем одиночный байт данных — символ "1", поскольку некоторые реализации не передают дескриптор без данных. Демон посылает обратно одиночный байт данных, состоящий из символа "1", для обозначения успешного выполнения. Любой другой ответ означает ошибку.

22-23 Инициализируем набор дескрипторов и вычисляем первый аргумент для функции select (максимальный из двух дескрипторов, увеличенный на единицу).

Вторая половина нашего клиента приведена в листинге 28.22. Это цикл, который считывает данные из стандартного ввода, посыпает строку серверу, считывает ответ сервера и записывает ответ в стандартный вывод.

Листинг 28.22. Вторая часть приложения dg_cli

```
//icmpd/dgcli01.c
24 while (Fgets(sendline, MAXLINE, fp) != NULL) {
25     Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

26     tv.tv_sec = 5;
27     tv.tv_usec = 0;
28     FD_SET(sockfd, &rset);
29     FD_SET(icmpfd, &rset);
30     if ((n = Select(maxfdp1, &rset, NULL, NULL, &tv)) == 0) {
31         fprintf(stderr, "socket timeout\n");
32         continue;
33     }

34     if (FD_ISSET(sockfd, &rset)) {
35         n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
36         recvline[n] = 0; /* завершающий нуль */
37         Fputs(recvline, stdout);
38     }

39     if (FD_ISSET(icmpfd, &rset)) {
40         if ((n = Read(icmpfd, &icmpd_err, sizeof(icmpd_err))) == 0)
41             err_quit("ICMP daemon terminated");
42         else if (n != sizeof(icmpd_err))
43             err_quit("n = %d, expected %d", n, sizeof(icmpd_err));
44         printf("ICMP error: dest = %s, %s, type = %d, code = %d\n",
45             Sock_ntop(&icmpd_err.icmpd_dest, icmpd_err.icmpd_len);
46             strerror(icmpd_err.icmpd_errno),
47             icmpd_err.icmpd_type, icmpd_err.icmpd_code);
48     }
49 }
```

Вызов функции *select*

26-33 Поскольку мы вызываем функцию *select*, мы можем легко установить время ожидания ответа от эхо-сервера. Задаем его равным 5 с, открываем оба дескриптора для чтения и вызываем функцию *select*. Если происходит превышение времени, выводится соответствующее сообщение и осуществляется переход в начало цикла.

Вывод ответа сервера

34-38 Если дейтаграмма возвращается сервером, она выводится в стандартный поток вывода.

Обработка ICMP-ошибки

39-48 Если наше доменное соединение Unix с демоном *icmpd* готово для чтения, мы пытаемся прочитать структуру *icmpd_err*. Если это удается, выводится соответствующая информация, возвращаемая демоном.

ПРИМЕЧАНИЕ

Функция *strerror* является примером простой, почти тривиальной функции, которая должна быть более переносимой, чем она есть. В ANSI C ничего не говорится об ошибках, возвращаемых этой функцией. В руководстве по операционной системе Solaris 2.5 говорится, что функция возвращает пустой указатель, если ее аргумент выходит за пределы допустимых значений. Это означает, что код наподобие следующего:

```
printf("%s", strerror(arg));
```

является некорректным, поскольку *strerror* может вернуть пустой указатель. Однако реализации FreeBSD, так же как и все реализации исходного кода, которые автор смог найти, обрабатывают неправильный аргумент, возвращая указатель на строку типа «Неизвестная ошибка». Это имеет смысл и означает, что приведенный выше код правильный. POSIX изменил ситуацию, утверждая, что поскольку не предусмотрено значение, сигнализирующее об ошибке, связанной с выходом аргумента за допустимые пределы, функция присваивает переменной *errno* значение EIVAL. (Ничего не сказано об указателе, возвращаемом в случае ошибки.) Это означает, что полностью правильный код должен обнулить *errno*, вызвать функцию *strerror*, проверить, не равняется ли значение *errno* величине EIVAL, и в случае ошибки вывести некоторое сообщение.

Примеры эхо-клиента UDP

Приведем несколько примеров работы данного клиента, прежде чем рассматривать исходный код демона. Сначала посыпаем дейтаграмму на IP-адрес, не связанный с Интернетом:

```
freebsd % udpcli01 192.0.2.5 echo
hi there
socket timeout
and hello
socket timeout
```

Мы считаем, что демон *icmpd* запущен, и ждем возвращения каким-либо маршрутизатором ICMP-ошибок недоступности получателя. Вместо этого наше приложение завершается по превышению времени ожидания. Мы показываем это, чтобы повторить, что время ожидания все еще необходимо, а генерация ICMP-сообщения о недоступности узла может и не произойти.

В следующем примере дейтаграмма отправляется на порт стандартного эхо- сервера узла, на котором этот сервер не запущен. Мы получаем ожидаемое ICMPv4-сообщение о недоступности порта.

```
freebsd % udpcli01 aix-4 echo  
hello  
ICMP error: dest = 192.168.42.2:7. Connection refused, type = 3, code = 1  
Выполнив ту же попытку с протоколом IPv6, мы получаем ICMPv6-сообщение о недоступности порта.  
freebsd % udpcli01 aix-6 echo hello, world  
ICMP error: dest = [3ffe:b80:1f8d:2:204:acff:fe17:bf38]:7. Connection refused, type = 1.  
code = 4
```

Демон icmpd

Начинаем описание нашего демона icmpd с заголовочного файла icmpd.h, приведенного в листинге 28.23.

Листинг 28.23. Заголовочный файл icmpd.h для демона icmpd

```
//icmpd/icmpd.h  
1 #include "unpicmpd.h"  
  
2 struct client {  
3     int connfd; /* потоковый доменный сокет Unix к клиенту */  
4     int family; /* AF_INET или AF_INET6 */  
5     int lport; /* локальный порт, связанный с UDP-сокетом клиента */  
6             /* сетевой порядок байтов */  
7 } client[FD_SETSIZE];  
  
8 /* глобальные переменные */  
9 int fd4, fd6, listenfd, maxi, maxfd, nready;  
10 fd_set rset, allset;  
11 struct sockaddr_un cliaddr;  
  
12 /* прототипы функций */  
13 int readable_conn(int);  
14 int readable_listen(void);  
15 int readable_v4(void);  
16 int readable_v6(void);
```

Mассив client

2-17 Поскольку демон может обрабатывать любое количество клиентов, для сохранения информации о каждом клиенте используется массив структур client. Они аналогичны структурам данных, которые использовались в разделе 6.8. Кроме дескриптора для доменного сокета Unix, через который осуществляется связь с клиентом, сохраняется также семейство адресов клиентского UDP-сокета AF_INET или AF_INET6 и номер порта, связанного с сокетом. Далее объявляются прототипы функций и глобальные переменные, совместно используемые этими функциями.

В листинге 28.24 приведена первая часть функции main.

Листинг 28.24. Первая часть функции main: создание сокетов

```
//icmpd/icmpd.c  
1 #include "icmpd.h"  
  
2 int  
3 main(int argc, char **argv)  
4 {  
5     int i, sockfd;  
6     struct sockaddr_un sun;
```

```

7 if (argc != 1)
8 err_quit("usage: icmpd");

9 maxi = -1; /* индекс массива client[] */
10 for (i = 0; i < FD_SETSIZE; i++)
11   client[i].connfd = -1; /* -1 означает свободный элемент */
12 FD_ZERO(&allset);

13 fd4 = Socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
14 FD_SET(fd4, &allset);
15 maxfd = fd4;

16 #ifdef IPV6
17 fd6 = Socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);
18 FD_SET(fd6, &allset);
19 maxfd = max(maxfd, fd6);
20#endif

21 listenfd = Socket(AF_UNIX, SOCK_STREAM, 0);
22 sun.sun_family = AF_LOCAL;
23 strcpy(sun.sun_path, ICMPD_PATH);
24 unlink(ICMPD_PATH);
25 Bind(listenfd, (SA*)&sun, sizeof(sun));
26 Listen(listenfd, LISTENQ);
27 FD_SET(listenfd, &allset);
28 maxfd = max(maxfd, listenfd);

```

Инициализация массива client

9-10 Инициализируется массив *client* путем присваивания значения -1 элементу присоединенного сокета.

Создание сокетов

12-28 Создаются три сокета: символьный сокет ICMPv4, символьный сокет ICMPv6 и потоковый доменный сокет Unix. Мы связываем при помощи функции *bind* свое заранее известное полное имя с сокетом и вызываем функцию *listen*. Это сокет, к которому клиенты присоединяются с помощью функции *connect*. Для функции *select* также вычисляется максимальный дескриптор, а для вызовов функции *accept* в памяти размещается структура адреса сокета.

В листинге 28.25 приведена вторая часть функции *main*. Она содержит бесконечный цикл, вызывающий функцию *select* в ожидании, когда будет готов к чтению какой-либо из дескрипторов демона.

Листинг 28.25. Вторая часть функции *main*: обработка готового к чтению дескриптора

```

//icmpd/icmpd.c
29 for (;;) {
30   rset = allset;
31   nready = Select(maxfd+1, &rset, NULL, NULL, NULL);

32   if (FD_ISSET(listenfd, &rset))
33     if (readable_listen() <= 0)
34       continue;

35   if (FD_ISSET(fd4, &rset))

```

```

36     if (readable_v4() <= 0)
37         continue;

38 #ifdef IPV6
39     if (FD_ISSET(fd6, &rset))
40         if (readable_v6() <= 0)
41             continue;
42 #endif

43     for (i = 0; i <= maxi; i++) { /* проверка всех клиентов */
44         if (sockfd = client[i].connfd) < 0)
45             continue;
46         if (FD_ISSET(sockfd, &rset))
47             if (readable_conn(i) <= 0)
48                 break; /* готовых дескрипторов больше нет */
49     }
50 }
51 exit(0);
52 }

```

Проверка прослушиваемого доменного сокета Unix

32-34 Прослушиваемый доменный сокет Unix проверяется в первую очередь, и если он готов, запускается функция `readable_listen`. Переменная `nready` — количество дескрипторов, которое функция `select` возвращает как готовые к чтению — является глобальной. Каждая из наших функций `readable_XXX` уменьшает ее значение на 1, и новое значение этой переменной является возвращаемым значением функции. Когда ее значение достигает нуля, это говорит о том, что все готовые к чтению дескрипторы обработаны, и поэтому функция `select` вызывается снова.

Проверка символьных сокетов ICMP

35-42 Проверяется символьный сокет ICMPv4, а затем символьный сокет ICMPv6.

Проверка присоединенных доменных сокетов Unix

43-49 Затем проверяется, готов ли для чтения какой-нибудь из присоединенных доменных сокетов Unix. Готовность для чтения какого-либо из таких сокетов обозначает, что клиент отоспал дескриптор или завершился.

В листинге 28.26 приведена функция `readable_listen`, вызываемая, когда прослушиваемый сокет готов для чтения. Это указывает на новое клиентское соединение.

Листинг 28.26. Обработка нового соединения клиента

```

//icmpd/readable_listen.c
1 #include "icmpd.h"

2 int
3 readable_listen(void)
4 {
5     int i, connfd;
6     socklen_t clilen;

7     clilen = sizeof(cliaddr);
8     connfd = Accept(listenfd, (SA*)&cliaddr, &clilen);
9     /* поиск первой свободной структуры в массиве client[] */

```

```

10  for (i = 0; i < FD_SETSIZE; i++) {
11    if (client[i].connfd < 0) {
12      client[i].connfd = connfd; /* сохранение дескриптора */
13      break;
14    }
15    if (i == FD_SETSIZE) {
16      close(connfd); /* невозможно обработать новый клиент */
17      return(--nready); /* грубое закрытие нового соединения */
18    }
19    printf("new connection, i = %d, connfd = %d\n", i, connfd);

20  FD_SET(connfd, &allset); /* добавление нового дескриптора в набор */
21  if (connfd > maxfd)
22    maxfd = connfd; /* для select() */
23  if (i > maxi)
24    maxi = i; /* максимальный индекс в массиве client[] */

25  return(--nready);
26 }

```

7-25 Принимается соединение и используется первый свободный элемент массива `client`. Код данной функции скопирован из начала кода, приведенного в листинге 6.4. Если свободных элементов в массиве нет, мы закрываем новое соединение и занимаемся обслуживанием уже имеющихся клиентов.

Когда присоединенный сокет готов для чтения, вызывается функция `readable_conn` (листинг 28.27), а ее аргументом является индекс данного клиента в массиве `client`.

Листинг 28.27. Считывание данных и, возможно, дескриптора от клиента

```

//icmpd/readable_conn.c
1 #include "icmpd.h"

2 int
3 readable_conn(int I)
4 {
5   int unixfd, recvfd;
6   char c;
7   ssize_t n;
8   socklen_t len;
9   struct sockaddr_storage ss;

10  unixfd = client[i].connfd;
11  recvfd = -1;
12  if ((n = Read_fd(unixfd, &c, 1, &recvfd)) == 0) {
13    err_msg("client %d terminated, recvfd = %d", i, recvfd);
14    goto clientdone; /* вероятно, клиент завершил работу */
15  }

16  /* данные от клиента, должно быть, дескриптор */
17  if (recvfd < 0) {
18    err_msg("read_fd did not return descriptor");
19    goto clienterr;
20  }

```

Считывание данных клиента и, возможно, дескриптора

13-18 Вызываем функцию `read_fd`, приведенную в листинге 15.9, для считывания данных и, возможно, дескриптора. Если возвращаемое значение равно нулю, клиент закрыл свою часть соединения, вероятно, завершив свое выполнение.

ПРИМЕЧАНИЕ

При написании кода пришлось выбирать, что использовать для связи между приложением и демоном — либо потоковый доменный сокет Unix, либо дейтаграммный доменный сокет Unix. Дескриптор сокета UDP может быть передан через любой доменный сокет Unix. Причина, по которой предпочтение было отдано потоковому сокету, заключается в том, что он позволяет определить момент завершения клиента. Все дескрипторы автоматически закрываются, когда клиент завершает работу, в том числе и доменный сокет Unix, используемый для связи с демоном, в результате чего данный клиент удаляется демоном из массива client. Если бы мы использовали сокет дейтаграмм, то не узнали бы, когда клиент завершил работу.

16-20 Если клиент не закрыл соединение, ждем получения дескриптора. Вторая часть функции readable_conn приведена в листинге 28.28.

Листинг 28.28. Получение номера порта, который клиент связал с UDP-сокетом

```
//icmpd/readable_conn.c
21 len = sizeof(ss);
22 if (getsockname(recvfd, (SA*)&ss, &len) < 0) {
23     err_ret("getsockname error");
24     goto clienterr;
25 }

26 client[i].family = ss.ss_family;
27 if ((client[i].lport = sock_get_port((SA*)&ss, len)) == 0) {
28     client[i].lport = sock_bind_wild(recvfd, client[i].family);
29     if (client[i].lport <= 0) {
30         err_ret("error binding ephemeral port");
31         goto clienterr;
32     }
33 }
34 Write(unixfd, "1", 1); /* сообщение клиенту об успехе */
35 Close(recvfd); /* работа с UDP-сокетом клиента завершена */
36 return(--nready);

37 clienterr:
38 Write(unixfd, "0", 1); /* сообщение клиенту об ошибке */
39 clientdone:
40 Close(unixfd);
41 if (recvfd >= 0)
42     Close(recvfd);
43 FD_CLR(unixfd, &allset);
44 client[i].connfd = -1;
45 return(--nready);
46 }
```

Получение номера порта, связанного с сокетом UDP

21-25 Вызывается функция getsockname, так что демон может получить номер порта, связанного с сокетом. Поскольку неизвестно, каков размер буфера, необходимого для размещения структуры адреса сокета, мы используем структуру sockaddr_storage, которая достаточно велика для структуры адреса сокета любого поддерживаемого системой типа и обеспечивает нужное выравнивание.

26-33 Семейство адресов сокета вместе с номером порта сохраняется в структуре client. Если номер порта равен нулю, мы вызываем функцию sock_bind_wild для связывания универсального адреса и динамически назначаемого порта с сокетом, но, как отмечалось ранее, такой подход не работает в реализациях SVR4.

Сообщение клиенту об успехе

34 Один байт, содержащий символ "1", отправляется обратно клиенту.

Закрытие UDP-сокета клиента

35 Заканчиваем работу с UDP-сокетом клиента и закрываем его с помощью функции `close`. Дескриптор был переслан нам клиентом и, таким образом, является копией; следовательно, UDP-сокет все еще открыт на стороне клиента.

Обработка ошибок и завершение работы клиента

37-45 Если происходит ошибка, клиент получает нулевой байт. Когда клиент завершается, наша часть доменного сокета Unix закрывается, и соответствующий дескриптор удаляется из набора дескрипторов для функции `select`. Полю `connfd` структуры `client` присваивается значение -1, что является указанием на ее освобождение.

Функция `readable_v4` вызывается, когда символьный сокет ICMPv4 открыт для чтения. Первая часть данной функции приведена в листинге 28.29. Этот код аналогичен коду для ICMPv4, приведенному ранее в листингах 28.6 и 28.15.

Листинг 28.29. Обработка полученныхдейтаграмм ICMPv4, первая часть

```
//icmpd/readable_v4.c
1 #include "icmpd.h"
2 #include <netinet/in_systm.h>
3 #include <netinet/ip.h>
4 #include <netinet/ip_icmp.h>
5 #include <netinet/udp.h>

6 int
7 readable_v4(void)
8 {
9     int i, hlen1, hlen2, icmpplen, sport;
10    char buf[MAXLINE];
11    char srcstr[INET_ADDRSTRLEN], dststr[INET_ADDRSTRLEN];
12    ssize_t n;
13    socklen_t len;
14    struct ip *ip, *hip;
15    struct icmp *icmp;
16    struct udphdr *udp;
17    struct sockaddr_in from, dest;
18    struct icmpd_err icmpd_err;

19    len = sizeof(from);
20    n = Recvfrom(fd4, buf, MAXLINE, 0, (SA*)&from, &len);

21    printf("%d bytes ICMPv4 from %s:", n, Sock_ntop_host((SA*)&from, len));

22    ip = (struct ip*)buf; /* начало IP-заголовка */
23    hlen1 = ip->ip_hl << 2; /* длина IP-заголовка */

24    icmp = (struct icmp*)(buf + hlen1); /* начало ICMP-заголовка */
25    if ((icmpplen = n - hlen1) < 8)
26        err_quit("icmpplen (%d) < 8", icmpplen);
```

```
27 printf(" type = %d, code = %d\n", icmp->icmp_type, icmp->icmp_code);
Функция выводит некоторую информацию о каждом получаемом сообщении ICMP. Это было сделано
для отладки при разработке демона, и вывод управляется аргументом командной строки.
```

В листинге 28.30 приведена вторая часть функции `readable_v4`.

Листинг 28.30. Обработка полученных дейтаграмм ICMPv4, вторая часть

```
//icmpd/readable_v4.c
28 if (icmp->icmp_type == ICMP_UNREACH ||
29     icmp->icmp_type == ICMP_TIMXCEED ||
30     icmp->icmp_type == ICMP_SOURCEQUENCH) {
31     if (icmplen < 8 + 20 + 8)
32         err_quit("icmplen (%d) < 8 + 20 + 8, icmplen");

33     hip = (struct ip*)(buf + hlen1 + 8);
34     hlen2 = hip->ip_hl << 2;
35     printf("\tsrcip = %s, dstip = %s, proto = %d\n",
36            Inet_ntop(AF_INET, &hip->ip_src, srcstr, sizeof(srcstr)),
37            Inet_ntop(AF_INET, &hip->ip_dst, dststr, sizeof(dststr)),
38            hip->ip_p);
39     if (hip->ip_p == IPPROTO_UDP) {
40         udp = (struct udphdr*)(buf + hlen1 + 8 + hlen2);
41         sport = udp->uh_sport;

42     /* поиск доменного сокета клиента, отправка заголовка */
43     for (i = 0; i <= maxi; i++) {
44         if (client[i].connfd >= 0 &&
45             client[i].family == AF_INET &&
46             client[i].lport == sport) {

47             bzero(&dest, sizeof(dest));
48             dest.sin_family = AF_INET;
49 #ifdef HAVE_SOCKADDR_SA_LEN
50             dest.sin_len = sizeof(dest);
51 #endif
52             memcpy(&dest.sin_addr, &hip->ip_dst,
53                   sizeof(struct in_addr));
54             dest.sin_port = udp->uh_dport;

55             icmpd_err.icmpd_type = icmp->icmp_type;
56             icmpd_err.icmpd_code = icmp->icmp_code;
57             icmpd_err.icmpd_len = sizeof(struct sockaddr_in);
58             memcpy(&icmpd_err.icmpd_dest, &dest, sizeof(dest));

59             /* преобразование кода и типа ICMP в значение errno */
60             icmpd_err.icmpd_errno = EHOSTUNREACH; /* по умолчанию */
61             if (icmp->icmp_type == ICMP_UNREACH) {
62                 if (icmp->icmp_code == ICMP_UNREACH_PORT)
63                     icmpd_err.icmpd_errno = ECONNREFUSED;
64                 else if (icmp->icmp_code == ICMP_UNREACH_NEEDFRAG)
65                     icmpd_err.icmpd_errno = EMSGSIZE;
66             }
67             Write(client[i].connfd, &icmpd_err, sizeof(icmpd_err));
68         }
69     }
70 }
71 }
72 return(--nready);
```

Проверка типа сообщения, уведомление приложения

29-31 ICMP-сообщения, которые посылаются приложениям, — это сообщения о недоступности порта, превышении времени и завершении клиента (см. табл. 28.1).

Проверка ошибки UDP, поиск клиента

34-42 Указатель `hip` указывает на IP-заголовок, который возвращается сразу после заголовка ICMP. Это IP-заголовок дейтаграммы, вызвавшей ICMP-ошибку. Мы убеждаемся, что эта IP-дейтаграмма является UDP-дейтаграммой, а затем извлекаем номер UDP-порта из UDP-заголовка, следующего за IP-заголовком.

43-55 По всем структурам `client` осуществляется поиск подходящего семейства адресов и порта. Если соответствие найдено, строится структура адреса сокета IPv4, которая содержит IP-адрес получателя и порт из UDP-дейтаграммы, вызвавшей ошибку.

Построение структуры icmpd_err

56-70 Строится структура `icmpd_err`, посылаемая клиенту через доменный сокет Unix. Тип и код сообщения ICMP сначала отображаются в значение `errno`, как показано в табл. 28.1.

Ошибки ICMPv6 обрабатываются функцией `readable_v6`, первая часть которой приведена в листинге 28.31. Обработка ошибок ICMPv6 аналогична коду, приведенному в листингах 28.7 и 28.16.

Листинг 28.31. Обработка полученной дейтаграммы ICMPv6, первая часть

```
//icmpd/readable_v6.c
1 #include "icmpd.h"
2 #include <netinet/in_systm.h>
3 #include <netinet/ip.h>
4 #include <netinet/ip_icmp.h>
5 #include <netinet/udp.h>

6 #ifdef IPV6
7 #include <netinet/ip6.h>
8 #include <netinet/icmp6.h>
9 #endif

10 int
11 readable_v6(void)
12 {
13 #ifdef IPV6
14 int i, hlen2, icmp6len, sport;
15 char buf[MAXLINE];
16 char srcstr[INET6_ADDRSTRLEN], dststr[INET6_ADDRSTRLEN];
17 ssize_t n;
18 socklen_t len;
19 struct ip6_hdr *ip6, *hip6;
20 struct icmp6_hdr *icmp6;
21 struct udphdr *udp;
22 struct sockaddr_in6 from, dest;
23 struct icmpd_err icmpd_err;

24 len = sizeof(from);
25 n = Recvfrom(fd6, buf, MAXLINE, 0, (SA*)&from, &len);
```

```

26 printf("%d bytes ICMPv6 from %s:", n, Sock_ntop_host((SA*)&from, len));
27 icmp6 = (struct icmp6_hdr*)buf; /* начало заголовка ICMPv6 */
28 if ((icmp6len = n) < 8)
29   err_quit("icmp6len (%d) < 8", icmp6len);

```

30 printf(" type = %d, code = %d\n", icmp6->icmp6_type, icmp6->icmp6_code);

Вторая часть функции `readable_v6` приведена в листинге 28.32. Код аналогичен приведенному в листинге 28.30: мы проверяем тип ICMP-ошибки, убеждаемся, что дейтаграмма, вызвавшая ошибку, является UDP-дейтаграммой, а затем строим структуру `icmpd_err`, которую отсылаем клиенту.

Листинг 28.32. Обработка полученной дейтаграммы ICMPv6, вторая часть

```

//icmpd/readable_v6.c
31 if (icmp6->icmp6_type == ICMP6_DST_UNREACH ||
32     icmp6->icmp6_type == ICMP6_PACKET_TOO_BIG ||
33     icmp6->icmp6_type == ICMP6_TIME_EXCEEDED) {
34   if (icmp6len < 8+8)
35     err_quit("icmp6len (%d) < 8 + 8", icmp6len);
36   hip6 = (struct ip6_hdr*)(buf + 8);
37   hlen2 = sizeof(struct ip6_hdr);
38   printf("\tsrcip = %s, dstip = %s, next hdr = %d\n",
39         Inet_ntop(AF_INET6, &hip6->ip6_src, srcstr, sizeof(srcstr)),
40         Inet_ntop(AF_INET6, &hip6->ip6_dst, dststr, sizeof(dststr)),
41         hip6->ip6_nxt);
42   if (hip6->ip6_nxt == IPPROTO_UDP) {
43     udp = (struct udphdr*)(buf + 8 + hlen2);
44     sport = udp->uh_sport;

45   /* поиск доменного сокета клиента, отправка заголовков */
46   for (i = 0; i <= maxi; i++) {
47     if (client[i].connfd >= 0 &&
48         client[i].family == AF_INET6 &&
49         client[i].lport == sport) {

50       bzero(&dest, sizeof(dest));
51       dest.sin6_family = AF_INET6;
52 #ifdef HAVE_SOCKADDR_SA_LEN
53       dest.sin6_len = sizeof(dest);
54 #endif
55       memcpy(&dest.sin6_addr, &hip6->ip6_dst,
56              sizeof(struct in6_addr));
57       dest.sin6_port = udp->uh_dport;

58       icmpd_err.icmpd_type = icmp6->icmp6_type;
59       icmpd_err.icmpd_code = icmp6->icmp6_code;
60       icmpd_err.icmpd_len = sizeof(struct sockaddr_in6);
61       memcpy(&icmpd_err.icmpd_dest, &dest, sizeof(dest));

62     /* преобразование типа и кода ICMPv6 к значению errno */
63     icmpd_err.icmpd_errno = EHOSTUNREACH; /* по умолчанию */
64     if (icmp6->icmp6_type == ICMP6_DST_UNREACH &&
65         icmp6->icmp6_code ICMP6_DST_UNREACH_NOPORT)
66       icmpd_err.icmpd_errno = ECONNREFUSED;
67     if (icmp6->icmp6_type == ICMP6_PACKET_TOO_BIG)
68       icmpd_err.icmpd_errno = EMSGSIZE;
69     Write(client[i].connfd, &icmpd_err, sizeof(icmpd_err));

```

```
70      }
71  }
72  }
73 }
74 return(--nready);
75 #endif
76 }
```

28.8. Резюме

Символьные сокеты обеспечивают три возможности:

1. Чтение и запись пакетов ICMPv4, IGMPv4 и ICMPv6.
2. Чтение и запись IP-дейтаграммы с полем протокола, которое не обрабатывается ядром.
3. Формирование своих собственных заголовков IPv4, обычно используемых в диагностических целях (или, к сожалению, хакерами).

Два традиционных диагностических средства — программы `ping` и `traceroute` — используют символьные сокеты. Мы разработали наши собственные версии этих программ, поддерживающие обе версии протокола — и IPv4, и IPv6. Также нами разработан наш собственный демон `icmprd`, который обеспечивает доступ к сообщениям об ошибках ICMP через сокет UDP. Данный пример также иллюстрирует передачу дескриптора через доменный сокет Unix между неродственными клиентом и сервером.

Упражнения

1. В этой главе говорилось, что почти все поля заголовка IPv6 и все дополнительные заголовки доступны приложению через параметры сокета или вспомогательные данные. Какая информация из дейтаграммы IPv6 *не* доступна приложению?
2. Что произойдет в листинге 28.30, если по какой-либо причине клиент перестанет производить считывание из своего доменного сокета Unix и демон `icmprd` накопит множество сообщений для данного клиента? В чем заключается простейшее решение этой проблемы?
3. Если задать нашей программе `ping` адрес широковещательной передачи, направленный в подсеть, она будет работать. То есть широковещательный эхо-запрос ICMP посыпается как широковещательный запрос канального уровня, даже если мы не установим параметр сокета `SO_BROADCAST`. Почему?
4. Что произойдет с программой `ping`, если мы запустим ее на узле с несколькими интерфейсами, а в качестве аргумента имени узла возьмем групповой адрес 224.0.0.1?

Глава 29

Доступ к канальному уровню

29.1. Введение

В настоящее время большинство операционных систем позволяют приложению получать доступ к канальному уровню. Это подразумевает следующие возможности:

1. Отслеживание пакетов, принимаемых на канальном уровне, что, в свою очередь, позволяет запускать такие программы, как `tcpdump`, на обычных компьютерных системах (а не только на специальных аппаратных устройствах для отслеживания пакетов). Если добавить к этому способность сетевого интерфейса работать в *смешанном режиме* (*promiscuous mode*), приложение сможет отслеживать все пакеты, проходящие по локальному кабелю, а не только предназначенные для того узла, на котором работает эта программа.

ПРИМЕЧАНИЕ

Эта возможность не так полезна в коммутируемых сетях, которые получили широкое распространение в последнее время. Дело в том, что коммутатор пропускает трафик на конкретный порт только в том случае, если этот трафик адресован конкретному устройству или устройствам, подключенным к этому порту, каким бы трафик ни был: направленным, широковещательным или многоадресным. Для того чтобы получать трафик, передаваемый через другие порты коммутатора, нужно сначала переключить свой порт коммутатора в режим контроля (*monitor mode* или *port mirroring*). Заметьте, что многие устройства, которые обычно не считаются коммутаторами, на самом деле являются таковыми. Например, двухскоростной концентратор 10/100 обычно является двухпортовым коммутатором: один порт для сетей, работающих на 100 Мбит/с, другой — для сетей на 10 Мбит/с.

2. Возможность запуска определенных программ как обычных приложений, а не как частей ядра. Скажем, большинство версий Unix сервера RARP — это обычные приложения, которые считывают запросы RARP с канального уровня (запросы RARP не являются дейтаграммами IP), а затем передают ответы также на канальный уровень.

Три наиболее распространенных средства получения доступа к канальному уровню в Unix — это пакетный фильтр BSD (BPF, BSD Packet Filter), DLPI в SVR4 (Datalink Provider Interface — интерфейс поставщика канального уровня) и интерфейс пакетных сокетов Linux (`SOCK_PACKET`). Мы приводим в этой главе обзор перечисленных средств, а затем описываем `libcap` — открытую для свободного доступа библиотеку, содержащую функции для захвата пакетов. Эта библиотека работает со всеми тремя перечисленными средствами, и использование библиотеки позволяет сделать наши программы не зависящими от фактического способа обеспечения доступа к канальному уровню, применяемому в данной операционной системе. Мы описываем эту библиотеку, разрабатывая программу, которая посыпает запросы серверу имен DNS (мы составляем свои собственные дейтаграммы UDP и записываем их в символьный сокет) и считывает ответ при помощи `libcap`, чтобы определить, добавляет ли сервер имен контрольную сумму в дейтаграммы UDP.

29.2. BPF: пакетный фильтр BSD

4.4BSD и многие другие Беркли-реализации поддерживают BPF — пакетный фильтр BSD (BSD Packet Filter). Реализация BPF описана в главе 31 [128]. История BPF, описание псевдопроцессора BPF и сравнение с пакетным фильтром SunOS 4.1.x NIT приведены в [72].

Каждый канальный уровень вызывает BPF сразу после получения пакета и непосредственно перед его передачей выше, как показано на рис. 29.1.

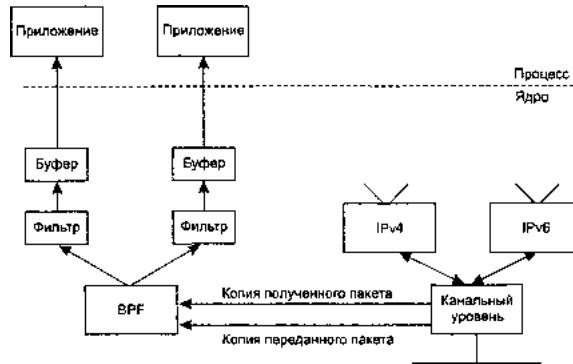


Рис. 29.1. Захват пакета с использованием BPF

Примеры подобных вызовов для интерфейса Ethernet приведены на рис. 4.11 и 4.19 в [128]. Вызов BPF должен произойти как можно скорее после получения пакета и как можно позже перед его передачей, так как это увеличивает точность временных отметок.

Организовать само по себе перехватывание пакетов из канального уровня не очень сложно, однако преимущество BPF заключается в возможности их фильтрации. Каждое приложение, открывающее устройство BPF, может загрузить свой собственный фильтр, который затем BPF применяет к каждому пакету. В то время как некоторые фильтры достаточно просты (например, при использовании фильтра `u64 or tcp` принимаются только пакеты UDP и TCP), другие фильтры позволяют исследовать значения определенных полей в заголовках пакетов. Например, фильтр

```
tcp and port 80 and tcp[13:1] & 0x7 != 0
```

использовался в главе 14 [128] для отбора сегментов TCP, направлявшихся к порту 80 или от него и содержащих флаги SYN, FIN или RST. Выражение `tcp [13:1]` соответствует однобайтовому значению, начинающемуся с 13-го байта от начала заголовка TCP.

В BPF реализован основанный на регистрах механизм фильтрации, который применяет специфические для приложений фильтры к каждому полученному пакету. Хотя можно написать свою программу фильтрации на машинном языке псевдопроцессора (он описан в руководстве по использованию BPF), проще всего будет компилировать строки ASCII (такие, как только что показанная строка, начинающаяся с `tcp`) в машинный язык с помощью функции `rcap_compile`, о которой мы рассказываем в разделе 29.7.

В технологии BPF применяются три метода, позволяющие уменьшить накладные расходы на ее использование.

1. Фильтрация BPF происходит внутри ядра, за счет чего минимизируется количество данных, которые нужно копировать из ядра в приложение. Копирование из пространства ядра в пользовательское пространство является довольно дорогостоящим. Если бы приходилось копировать каждый пакет, у BPF могли бы возникнуть проблемы при попытке взаимодействия с быстрыми каналами.

2. BPF передает приложению только часть каждого пакета. Здесь речь идет о *длине захвата* (*capture length*). Большинству приложений требуется только заголовок пакета, а не содержащиеся в нем данные. Это также уменьшает количество данных, которые BPF должен скопировать в приложение. В программе `tcpdump`, например, по умолчанию это значение равно 68 байт, и этого достаточно для размещения 14-байтового заголовка Ethernet, 20-байтового заголовка IP, 20-байтового заголовка TCP и 14 байт данных. Но для вывода дополнительной информации по другим протоколам (например, DNS или NFS) требуется, чтобы пользователь увеличил это значение при запуске программы `tcpdump`.

3. BPF буферизует данные, предназначенные для приложения, и этот буфер передается приложению только когда он заполнен или когда истекает заданное время ожидания для считывания (*read timeout*). Это время может быть задано приложением. Программа `tcpdump`, например, устанавливает время ожидания 1000 мс, а демон RARP задает нулевое время ожидания (поскольку пакетов RARP немного, а сервер RARP должен послать ответ сразу, как только он получает запрос). Назначением буферизации является уменьшение количества системных вызовов. При этом между BPF и приложением происходит обмен тем же количеством пакетов, но за счет того, что уменьшается количество системных вызовов, каждый из которых связан с дополнительными накладными расходами, уменьшается и общий объем этих расходов. Например, на рис. 3.1 [110] сравниваются накладные расходы, возникающие при системном вызове `read`, когда файл считывается в несколько приемов, причем размер фрагментов варьируется от 1 до 131 072 байт.

Хотя на рис. 29.1 мы показываем только один буфер, BPF поддерживает по два внутренних буфера для каждого приложения и заполняет один, пока другой копируется в приложение. Эта стандартная технология носит название *двойной буферизации* (*double buffering*).

На рис. 29.1 мы показываем только получение пакетов фильтром BPF: пакеты, приходящие на канальный уровень снизу (из сети) и сверху (IP). Приложение также может записывать в BPF, в результате чего пакеты будут отсылаться по канальному уровню, но большая часть приложений только считывает пакеты из BPF. У нас нет оснований использовать BPF для отправки дейтаграмм IP, поскольку параметр сокета `IP_HDRINCL` позволяет нам записывать дейтаграммы IP любого типа, включая заголовок IP. (Подобный пример мы показываем в разделе 29.7.) Записывать в BPF можно только с одной целью — чтобы отослать наши собственные сетевые пакеты, не являющиеся дейтаграммами IP. Например, демон RARP делает это для отправки ответов RARP, которые не являются дейтаграммами IP.

Для получения доступа к BPF необходимо открыть (вызвав функцию `open`) еще не открытое каким-либо другим процессом устройство BPF. Скажем, можно попробовать `/dev/bpf0`, и если будет возвращена ошибка `EBUSY`, то — `/dev/bpf1`, и т.д. Когда устройство будет открыто, потребуется выполнить примерно 12 команд `iocctl` для задания характеристик устройства, таких как загрузка фильтра, время ожидания для считывания, размер буфера, присоединение канального уровня к устройству BPF, включение смешанного режима, и т.д. Затем с помощью функций `read` и `write` осуществляется ввод и вывод.

29.3. DLPI: интерфейс поставщика канального уровня

SVR4 обеспечивает доступ к канальному уровню через DLPI (Data Link Provider Interface — интерфейс поставщика канального уровня). DLPI — это не зависящий от протокола интерфейс, разработанный в AT&T и служащий средством связи с сервисами, обеспечиваемыми канальным уровнем [124]. Доступ к DLPI осуществляется посредством отправки и получения сообщений через потоки STREAMS.

Для подсоединения к канальному уровню приложение просто открывает устройство (например, `le0`) с помощью команды `open` и использует запрос `DL_ATTACH_REQ`. Но для эффективной работы используются два дополнительных модуля: `pfmod`, который осуществляет фильтрацию внутри ядра, и `bufmod`, буферизующий данные, предназначенные для приложения. Это показано на рис. 29.2.

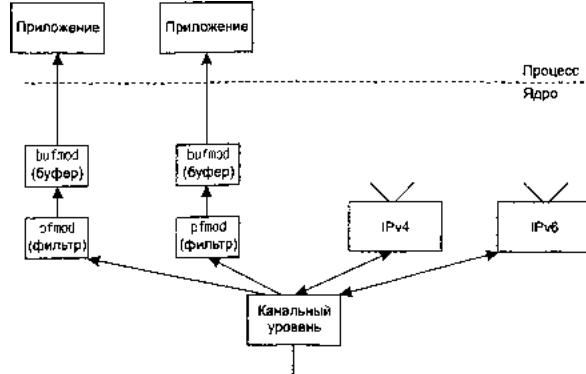


Рис. 29.2. Захват пакета с использованием DLPI, pfmod и bufmod

Концептуально DLPI аналогичен BPF. `pfmod` поддерживает фильтрацию внутри ядра, используя псевдопроцессор, а `bufmod` сокращает количество данных и системных вызовов, поддерживая длину захвата и время ожидания для считывания.

Одно интересное различие, тем не менее, заключается в том, что для BPF и фильтров `pfmod` используются разные типы псевдопроцессоров. Фильтр BPF — это *ориентированный ациклический граф управления потоком* (*acyclic control flow graph, CFG*), в то время как `pfmod` использует дерево булевых выражений. В первом случае естественным является отображение в код для вычислительной машины с регистровой организацией, а во втором — в код для машины со стековой организацией [72]. В статье [72] показано, что реализация CFG, используемая в BPF, обычно работает быстрее, чем дерево булевых выражений, в 3-20 раз в зависимости от сложности фильтра.

Еще одно отличие состоит в том, что BPF всегда выполняет фильтрацию перед копированием пакета, чтобы не копировать те пакеты, которые будутброшены фильтром. В некоторых реализациях DLPI пакеты сначала копируются в модуль `pfmod`, который затем может сбрасывать их.

29.4. Linux: SOCK_PACKET и PF_PACKET

Существует два метода получения пакетов канального уровня в Linux. Первоначальный метод получил более широкое распространение, но является менее гибким. Он состоит в создании сокета типа SOCK_PACKET. Новый метод, предоставляющий больше возможностей для настройки фильтров и оптимизации производительности, состоит в создании сокета семейства PF_PACKET. В любом случае мы должны обладать правами привилегированного пользователя (аналогичные необходимым для создания символьного сокета), а третий аргумент функции socket должен быть ненулевым значением, задающим тип кадра Ethernet. При использовании сокетов PF_PACKET второй аргумент socket может быть константой SOCK_DGRAM (для получения обработанных пакетов без заголовка канального уровня) или SOCK_RAW (для получения пакетов целиком). Сокеты SOCK_PACKET передают пакеты только целиком. Например, для получения всех кадров канального уровня мы пишем:

```
fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL)); /* в новых системах */
```

или

```
fd = socket(AF_INET, SOCK_PACKET, htons(ETH_P_ALL)); /* в старых системах */
```

В результате этого будут возвращены кадры для всех протоколов, получаемые канальным уровнем. Если нам нужны кадры IPv4, то вызов будет таким:

```
fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP)); /* в новых системах */
```

```
fd = socket(AF_INET, SOCK_PACKET, htons(ETH_P_IP)); /* в старых системах */
```

Другие константы, которые могут использоваться в качестве последнего аргумента, — это, например, ETH_P_ARP и ETH_P_IPV6.

Указывая протокол ETH_P_xxx, мы тем самым сообщаем канальному уровню, какой тип из получаемых канальным уровнем кадров передавать сокету. Если канальный уровень поддерживает смешанный режим (например, Ethernet), то устройство тоже должно работать в смешанном режиме. Это осуществляется при помощи параметра PACKET_ADD_MEMBERSHIP с использованием структуры packet_mreq. При этом необходимо указать конкретный интерфейс и задать тип действия PACKET_MR_PROMISC. В старых системах для этого нужно вызвать функцию ioctl с запросом SIOCGIFFLAGS для получения флагов, установить флаг IFF_PROMISC и далее сохранить флаги с помощью SIOCSIFFLAGS. К сожалению, при использовании этого метода программы, работающие в смешанном режиме, могут мешать друг другу, а если в одной из них содержатся ошибки, то она может и не отключить смешанный режим по завершении.

Сравнивая это средство Linux с BPF и DLPI, мы можем отметить некоторые различия.

1. В Linux не обеспечивается буферизация. Фильтрация на уровне ядра доступна только в новых системах (при помощи параметра SO_ATTACH_FILTER). Существует обычный буфер приема сокета, но отсутствует возможность буферизации и отправки приложению нескольких кадров с помощью одной операции считывания. Это увеличивает накладные расходы, связанные с копированием потенциально возможных больших объемов данных из ядра в приложение.

2. В Linux не предусмотрена фильтрация на уровне устройства. Сокеты PF_PACKET могут быть связаны с устройством функцией bind. Если в вызове функции socket указан аргумент ETH_P_IP, то все пакеты IPv4 со всех устройств (например, Ethernet, каналы PPP, каналы SLIP и закольцовка) будут переданы на сокет. Функция recvfrom возвращает общую структуру адреса сокета, а элемент sa_data содержит имя устройства (например, eth0). Тогда приложение само должно игнорировать данные с тех устройств, которые не представляют для него интереса. Здесь мы сталкиваемся фактически с той же проблемой: возможно, что приложение будет получать слишком много данных, особенно в случае наблюдения за высокоскоростной сетью.

29.5. Libcap: библиотека для захвата пакетов

Библиотека захвата пакетов libcap обеспечивает не зависящий от реализации доступ к средствам операционной системы, с помощью которых осуществляется этот захват. В настоящее время поддерживается только чтение пакетов (хотя добавление нескольких строк кода в библиотеку позволяет также записывать пакеты в некоторых системах). В следующем разделе приводится описание альтернативной библиотеки, которая не только дает возможность записывать пакеты на канальный уровень, но и позволяет конструировать пакеты произвольного типа.

Сейчас осуществляется поддержка BPF для Беркли-ядер, DLPI для Solaris 2.x, NIT для SunOS 4.1.x, пакетных сокетов (SOCK_PACKET, PF_PACKET) в Linux и нескольких других операционных системах.

Библиотека `libcap` используется программой `tcpdump`. Всего в библиотеке насчитывается порядка 25 функций, но вместо того чтобы просто описывать их, мы продемонстрируем их фактическое использование на примере, рассматриваемом в следующем разделе. Названия всех функций начинаются с `pcap_`. Они описаны более подробно на странице руководства, которая называется `pcap`.

ПРИМЕЧАНИЕ

Библиотека `libcap` находится в свободном доступе по адресу <http://www.tcpdump.org/>.

29.6. Libnet: библиотека создания и отправки пакетов

Библиотека `libnet` предоставляет интерфейс для создания и отправки в сеть пакетов произвольного содержимого. Она обеспечивает доступ на уровне символьных сокетов и доступ к канальному уровню в формате, не зависящем от реализации.

Библиотека скрывает большую часть деталей формирования заголовков IP, UDP и TCP и обеспечивает приложению простой и переносимый интерфейс для отправки пакетов канального уровня и IP-пакетов через символьные сокеты. Как и `libcap`, библиотека `libnet` содержит достаточно много функций. Мы приведем пример использования небольшой их части, предназначенной для работы с символьными сокетами, но в следующем разделе. Для сравнения там же будет приведен код, непосредственно работающий с символьными сокетами. Все функции библиотеки начинаются с префикса `libnet_`. За более подробным их описанием вы можете обратиться к странице руководства `libnet` или к доступной в Сети документации.

ПРИМЕЧАНИЕ

Библиотека `libnet` свободно доступна по адресу <http://www.packetfactory.net/libnet/>. Руководство находится по адресу <http://www.packetfactory.net/libnet/manual>. На момент написания этой книги в Сети имелось руководство только по устаревшей версии 1.0. Актуальная версия 1.1 имеет значительно отличающийся интерфейс. В нашем примере используется API версии 1.1.

29.7. Анализ поля контрольной суммы UDP

Теперь мы приступаем к рассмотрению примера, в котором отсылается дейтаграмма UDP, содержащая запрос UDP к серверу имен, а затем считывается ответ с помощью библиотеки захвата пакетов. Цель данного примера — установить, вычисляется на сервере имен контрольная сумма UDP или нет. В случае IPv4 вычисление контрольной суммы не является обязательным. В большинстве систем в настоящее время вычисление контрольных сумм по умолчанию включено, но, к сожалению, в более старых системах, в частности SunOS 4.1.x, оно по умолчанию отключено. В настоящее время все системы, а особенно система, в которой работает сервер имен, всегда должны работать с включенными контрольными суммами UDP, поскольку поврежденные (содержащие ошибки) дейтаграммы могут повредить базу данных сервера.

ПРИМЕЧАНИЕ

Включение и выключение контрольных сумм обычно осуществляется сразу для всей системы, как показано в приложении E [111].

Мы формируем дейтаграмму UDP (запрос DNS) и записываем ее в символьный сокет. Параллельно мы проделаем то же самое с помощью `libnet`. Для отправки запроса мы могли бы использовать обычный сокет UDP, но мы хотим показать, как использовать параметр сокета `IP_HDRINCL` для создания полной дейтаграммы IP.

Нет возможности получить контрольную сумму UDP при чтении из обычного сокета UDP, а также считывать пакеты UDP или TCP, используя символьный сокет (см. раздел 28.4). Следовательно, путем захвата пакетов нам нужно получить целую дейтаграмму UDP, содержащую ответ сервера имен.

Затем мы исследуем поле контрольной суммы UDP в заголовке UDP, и если оно равно нулю, это означает, что на сервере отключено вычисление контрольной суммы.

Действие нашей программы иллюстрирует рис. 29.3. Мы записываем наши собственные дейтаграммы UDP в символьный сокет и считываем ответы, используя библиотеку libpcap. Обратите внимание, что UDP также получает ответ сервера имен и отвечает сообщением о недоступности порта ICMP, так как ничего не знает о номере порта, выбранном нашим приложением. Сервер имен игнорирует эту ошибку ICMP. Также можно отметить, что написать подобную тестовую программу, использующую TCP, было бы сложнее, даже несмотря на то, что мы с легкостью можем записывать свои собственные сегменты TCP. Дело в том, что любой ответ на сегмент TCP, который мы генерируем, обычно инициирует отправку протоколом TCP ответного сегмента RST туда, куда был послан первый сегмент.



Рис. 29.3. Приложение, определяющее, включено ли на сервере вычисление контрольных сумм UDP

ПРИМЕЧАНИЕ

Указанную проблему можно обойти. Для этого нужно посыпать сегменты TCP с IP-адресом отправителя, который принадлежит присоединенной подсети, но в настоящий момент не присвоен никакому другому узлу. Нужно также добавить данные ARP на посылающем узле для этого нового IP-адреса, чтобы узел отвечал на запросы ARP для него. В результате стек IP на посылающем узле будет игнорировать пакеты, приходящие на этот IP-адрес, в предположении, что посылающий узел не является маршрутизатором.

На рис. 29.4 приведены функции, используемые в нашей программе.

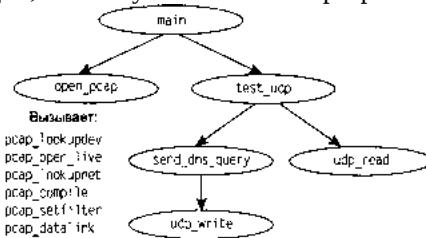


Рис. 29.4. Функции, которые используются в программе udpcsum

В листинге 29.1^[1] показан заголовочный файл udpcsum.h, в который включен наш базовый заголовочный файл unp.h, а также различные системные заголовки, необходимые для получения доступа к определениям структур для заголовков пакетов IP и UDP.

Листинг 29.1. Заголовочный файл udpcsum.h

```

//udpcsum/udpcsum.h
1 #include "unp.h"
2 #include <pcap.h>

3 #include <netinet/in_system.h> /* необходим для ip.h */
4 #include <netinet/in.h>
5 #include <netinet/ip.h>
6 #include <netinet/ip_var.h>

```

```

7 #include <netinet/udp.h>
8 #include <netinet/udp_var.h>
9 #include <net/if.h>
10 #include <netinet/if_ether.h>

11 #define TTL_OUT 64 /* исходящее TTL */

12 /* объявление глобальных переменных */
13 extern struct sockaddr *dest, *local;
14 extern socklen_t destlen, localallen;
15 extern int datalink;
16 extern char *device;
17 extern pcap_t *pd;
18 extern int rawfd;
19 extern int snaplen;
20 extern int verbose;
21 extern int zerosum;

```

22 /* прототипы функций */

```

23 void cleanup(int);
24 char *next_pcap(int*);
25 void open_output(void);
26 void open_pcap(void);
27 void send_dns_query(void);
28 void test_udp(void);
29 void udp_write(char*, int);
30 struct udphdr *udp_read(void);

```

3-10 Для работы с полями заголовков IP и UDP требуются дополнительные заголовочные файлы Интернета.

11-30 Мы определяем некоторые глобальные переменные и прототипы для своих собственных функций, которые вскоре покажем.

Первая часть функции main показана в листинге 29.2.

Листинг 29.2. Функция main: определения

```

//udpksum/main.c
1 #include "udpksum.h"

2 /* определение глобальных переменных */
3 struct sockaddr *dest, *local;
4 struct sockaddr_in locallookup;
5 socklen_t destlen, localallen;

6 int datalink; /* из pcap_datalink(), файл <net/bpf.h> */
7 char *device; /* устройство pcap */
8 pcap_t *pd; /* указатель на структуру захваченных пакетов */
9 int rawfd; /* символьный сокет */
10 int snaplen = 200; /* объем захваченных данных */
11 int verbose;
12 int zerosum; /* отправка UDP-запроса без контрольной суммы */

13 static void usage(const char*);

14 int
15 main(int argc, char *argv[])
16 {
17     int c, lopt=0;
18     char *ptr, localname[1024], *localport;

```

```

19 struct addrinfo *aip;
В следующей части функции main, представленной в листинге 29.3, обрабатываются аргументы
командной строки.
Листинг 29.3. Функция main: обработка аргументов командной строки
//udpcksum/main.c
20 opterr = 0; /* отключаем запись сообщений getopt() в stderr */
21 while ((c = getopt(argc, argv, "0i:l:v")) != -1) {
22     switch (c) {

23     case '0':
24         zerosum = 1;
25         break;

26     case 'i':
27         device = optarg; /* устройство pcap */
28         break;

29     case 'l'; /* локальный IP адрес и номер порта; a.b.c.d.p */
30     if ((ptr = strrchr(optarg, '.')) == NULL)
31         usage("invalid -l option");

32     *ptr++ = 0; /* нуль заменяет последнюю точку. */
33     local_port = ptr; /* имя сервиса или номер порта */
34     strncpy(localname, optarg, sizeof(localname));
35     lopt = 1;
36     break;

37     case 'v':
38         verbose = 1;
39         break;

40     case '?':
41         usage("unrecognized option");
42     }
43 }

```

Обработка аргументов командной строки

20-25 Мы вызываем функцию getopt для обработки аргументов командной строки. С помощью параметра -0 мы посылаем запросы UDP без контрольной суммы UDP, чтобы выяснить, обрабатываются ли эти дейтаграммы сервером иначе, чем дейтаграммы с контрольной суммой.

26 Параметр -i позволяет нам задать интерфейс, на котором будут приниматься ответы сервера. Если этот интерфейс не будет задан, библиотека для захвата пакетов выберет какой-либо интерфейс самостоятельно, но в случае узла с несколькими сетевыми интерфейсами этот выбор может оказаться некорректным. В этом заключается одно из различий между считыванием из обычного сокета и из устройства для захвата пакетов: в первом случае мы можем указать универсальный локальный адрес, что позволяет получать пакеты, прибывающие на любой из сетевых интерфейсов. Но во втором случае при работе с устройством для захвата пакетов мы можем получать пакеты, прибывающие только на конкретный интерфейс.

ПРИМЕЧАНИЕ

Можно отметить, что для пакетных сокетов Linux захват пакетов не ограничен одним устройством. Тем не менее библиотека libcap обеспечивает фильтрацию либо по умолчанию,

либо согласно заданному нами параметру -i.

29-36 Параметр -l позволяет нам задать IP-адрес отправителя и номер порта. В качестве номера порта (или названия службы) берется строка, следующая за последней точкой, а IP-адресом является все, что расположено перед последней точкой.

Последняя часть функции main показана в листинге 29.4.

Листинг 29.4. Функция main: преобразование имен узлов и названий служб, создание сокета

```
//udpcksum/main.c
44 if (optind != argc-2)
45 usage("missing <host> and/or <serv>");

46 /* преобразование имени получателя и службы */
47 aip = Host_serv(argv[optind], argv[optind+1], AF_INET, SOCK_DGRAM);
48 dest = aip->ai_addr; /* не освобождаем память при помощи freeaddrinfo() */
49 destlen = aip->ai_addrlen;

50 /*
51 * Нужен локальный IP-адрес для указания в UDP-дейтаграммах.
52 * Нельзя задать 0 и предоставить выбор уровню IP,
53 * потому что адрес нужен для вычисления контрольной суммы.
54 * Если указан параметр -1, используем заданные при вызове значения.
55 * в противном случае соединяем сокет UDP с адресатом и определяем
56 * правильный адрес отправителя.
57 */
58 if (lopt) {
59 /* преобразование локального имени и сервиса */
60 aip = Host_serv(localname, localport, AF_INET, SOCK_DGRAM);
61 local = aip->ai_addr; /* не вызываем freeaddrinfo() */
62 locallen = aip->ai_addrlen;
63 } else {
64 int s;
65 s = Socket(AF_INET, SOCK_DGRAM, 0);
66 Connect(s, dest, destlen);
67 /* ядро выбирает правильный локальный адрес */
68 locallen = sizeof(locallookup);
69 local = (struct sockaddr*)&locallookup;
70 Getsockname(s, local, &localen);
71 if (locallookup.sin_addr.s_addr == htonl(INADDR_ANY))
72 err_quit("Can't determine local address - use -l\n");
73 close(s);
74 }

75 open_output(); /* открываем поток вывода (символьный сокет или libnet) */
76 open_pcap(); /* открываем устройство захвата пакетов */

77 setuid(getuid()); /* права привилегированного пользователя больше
не нужны */

78 Signal(SIGTERM, cleanup);
79 Signal(SIGINT, cleanup);
80 Signal(SIGHUP, cleanup);

81 test_udp();

82 cleanup(0);
83 }
```

Обработка имени узла и порта получателя, затем локального имени узла и порта

46-49 Мы убеждаемся, что остается ровно два аргумента командной строки: имя узла получателя и название службы. Мы вызываем функцию `host_serv` для преобразования их в структуру адреса сокета, указатель на которую мы сохраняем в переменной `dest`.

Обработка локального имени и порта

50-74 Если в командной строке был указан соответствующий параметр, мы преобразуем имя локального узла и номер порта, сохраняя указатель на структуру адреса сокета под именем `local`. В противном случае для определения локального IP-адреса мы подключаемся через дейтаграммный сокет к нужному адресату и сохраняем полученный при этом локальный адрес под тем же именем `local`. Поскольку мы формируем собственные заголовки IP и UDP, мы должны знать IP-адрес отправителя при записи дейтаграммы UDP. Нельзя оставить адрес нулевым и предоставить уровню IP выбрать его самостоятельно, потому что адрес является частью псевдозаголовка UDP (о котором мы вскоре расскажем), используемого при вычислении контрольной суммы UDP.

Создаем символьный сокет и открываем устройство для захвата пакетов

75-76 Функция `open_output` выбирает метод отправки пакетов (символьный сокет или `libnet`). Функция `open_pcap` открывает устройство захвата пакетов. Она будет рассмотрена далее.

Изменение прав и установка обработчиков сигналов

77-80 Для создания символьного сокета необходимо иметь права привилегированного пользователя. Обычно такие привилегии нужны нам для того, чтобы открыть устройство для захвата пакетов, но это зависит от реализации. Например, в случае BPF администратор может установить разрешения для устройств `/dev/bpf` любым способом в зависимости от того, что требуется для данной системы. Здесь мы не используем эти дополнительные разрешения, предполагая, что для файла программы установлен бит SUID. Процесс выполняется с правами привилегированного пользователя, а когда они становятся не нужны, при вызове функции `setuid` фактический идентификатор пользователя (real user ID), эффективный идентификатор пользователя (effective user ID) и сохраненный SUID принимают значение фактического идентификатора пользователя (`getuid`). Мы устанавливаем обработчики сигналов на тот случай, если пользователь завершит программу раньше, чем будут изменены права.

Выполнение теста и очистка

81-82 Функция `test_udp` (см. листинг 29.6) выполняет тестирование и возвращает управление. Функция `cleanup` (см. листинг 29.14) выводит итоговую статистику библиотеки захвата пакетов, а затем завершает процесс.

В листинге 29.5 показана функция `open_pcap`, которую мы вызвали из функции `main`, чтобы открыть устройство для захвата пакетов.

Листинг 29.5. Функция `open_pcap`: открытие и инициализация устройства для захвата пакетов

```
//udpksum/pcap.c
1 #include "udpksum.h"

2 #define CMD "udp and src host %s and src port %d"

3 void
4 open_pcap(void)
5 {
```

```

6  uint32_t localnet, netmask;
7  char cmd[MAXLINE], errbuf[PCAP_ERRBUF_SIZE], str1[INET_ADDRSTRLEN],
8  str2[INET_ADDRSTRLEN];
9  struct bpf_program fcode;

10 if (device == NULL) {
11     if ((device = pcap_lookupdev(errbuf)) == NULL)
12         err_quit("pcap_lookup: %s", errbuf);
13 }
14 printf("device = %s\n", device);

15 /* жестко задано; promisc=0, to_ms=500 */
16 if ((pd = pcap_open_live(device, snaplen, 0, 500, errbuf)) == NULL)
17     err_quit("pcap_open_live: %s", errbuf);

18 if (pcap_lookupnet(device, &localnet, &netmask, errbuf) < 0)
19     err_quit("pcap_lookupnet %s", errbuf);
20 if (verbose)
21     printf("localnet = %s, netmask = %s\n",
22           Inet_ntop(AF_INET, &localnet, str1, sizeof(str1)),
23           Inet_ntop(AF_INET, &netmask, str2, sizeof(str2)));

24 snprintf(cmd, sizeof(cmd), CMD,
25          Sock_ntop_host(dest, destlen),
26          ntohs(sock_get_port(dest, destlen)));
27 if (verbose)
28     printf("cmd = %s\n", cmd);
29 if (pcap_compile(pd, &fcode, cmd, 0, netmask) < 0)
30     err_quit("pcap_compile: %s", pcap_geterr(pd));
31 if (pcap_setfilter(pd, &fcode) < 0)
32     err_quit("pcap_setfilter: %s", pcap_geterr(pd));

33 if ((datalink = pcap_datalink(pd)) < 0)
34     err_quit("pcap_datalink: %s", pcap_geterr(pd));
35 if (verbose)
36     printf("datalink = %d\n", datalink);
37 }

```

Выбор устройства для захвата пакетов

10-14 Если устройство для захвата пакетов не было задано (с помощью параметра командной строки `-i`), то выбор этого устройства осуществляется с помощью функции `pcap_lookupdev`. С помощью запроса `SIOCGIFCONF` функции `ioctl` выбирается включенное устройство с минимальным порядковым номером, но только не устройство обратной связи. Многие из библиотечных функций `pcap` возвращают сообщения об ошибках в виде строк. Единственным аргументом функции `pcap_lookupdev` является массив, в который записывается строка с сообщением об ошибке.

Открываем устройство

15-17 Функция `pcap_open_live` открывает устройство. Слово `live` присутствует в названии функции потому, что здесь имеется в виду фактическое устройство для захвата пакетов, а не файл, содержащий предыдущие сохраненные пакеты. Первым аргументом функции является имя устройства, вторым — количество байтов, которое нужно сохранять для каждого пакета (значение `snaplen`, которое мы инициализировали числом 200 в листинге 29.2), а третий аргумент — это флаг, указывающий на

смешанный режим. Четвертый аргумент — это значение времени ожидания в миллисекундах, а пятый — указатель на массив, содержащий сообщения об ошибках.

Если установлен флаг смешанного режима, интерфейс переходит в этот режим, в результате чего он принимает все пакеты, проходящие по кабелю. Это обычное состояние программы `tcpdump`. Тем не менее в нашем примере ответы сервера DNS будут посланы непосредственно на наш узел (то есть можно обойтись без смешанного режима).

Четвертый аргумент — время ожидания при считывании. Вместо того чтобы возвращать пакет процессу каждый раз, когда приходит очередной пакет (что может быть весьма неэффективно, так как в этом случае потребуется выполнять множество операций копирования отдельных пакетов из ядра в процесс), это делается, когдачитывающий буфер устройства оказывается заполненным либо когда истекает время ожидания. Если время ожидания при считывании равно нулю, то каждый пакет будет переправляться процессу, как только будет получен.

Получение сетевого адреса и маски подсети

18-23 Функция `pcap_lookupnet` возвращает сетевой адрес и маску подсети для устройства захвата пакетов. При вызове функции `pcap_compile`, которая будет вызвана следующей, нужно задать маску подсети, поскольку с помощью маски фильтр пакетов определяет, является ли IP-адрес адресом широковещательной передачи для данной подсети.

Компиляция фильтра пакетов

24-30 Функция `pcap_compile` получает строку, построенную нами как массив `cmd`, и компилирует ее, создавая тем самым программу для фильтрации (записывая ее в `fcode`). Эта программа будет отбирать те пакеты, которые мы хотим получить.

Загрузка программы фильтрации

31-32 Функция `pcap_setfilter` получает только что скомпилированную программу фильтрации и загружает ее в устройство для захвата пакетов. Таким образом инициируется захват пакетов, выбранных нами путем настройки фильтра.

Определение типа канального уровня

33-36 Функция `pcap_datalink` возвращает тип канального уровня для устройства захвата пакетов. Эта информация нужна нам при захвате пакетов для того, чтобы определить размер заголовка канального уровня, который будет добавлен в начало каждого считываемого нами пакета (см. листинг 29.10).

После вызова функции `open_pcap` функция `main` вызывает функцию `test_udp`, показанную в листинге 29.6. Эта функция посылает запрос DNS и считывает ответ сервера.

Листинг 29.6. Функция `test_udp`: отправка запросов и считывание ответов

```
//udpcksum/udpcksum.c
12 void
13 test_udp(void)
14 {
15 volatile int nsent = 0, timeout = 3;
16 struct udphiphdr *ui;

17 Signal(SIGALRM, sig_alrm);

18 if (sigsetjmp(jmpbuf, 1)) {
19   if (nsent >= 3)
20     err_quit("no response");
```

```

21  printf("timeout\n");
22  timeout *= 2; /* геометрическая прогрессия: 3, 6, 12 */
23 }
24 canjump = 1; /* siglongjmp разрешен */

25 send_dns_query();
26 nsent++;

27 alarm(timeout);
28 ui = udp_read();
29 canjump = 0;
30 alarm(0);

31 if (ui->ui_sum == 0)
32 printf("UDP checksums off\n");
33 else
34 printf("UDP checksums on\n");
35 if (verbose)
36 printf("received UDP checksum = %x\n", ntohs(ui->ui_sum));
37 }

```

Переменные volatile

15 Нам нужно, чтобы две динамические локальные переменные `nsent` и `timeout` сохраняли свои значения после возвращения `siglongjmp` из обработчика сигнала в нашу функцию. Реализация допускает восстановление значений динамических локальных переменных, предшествовавших вызову функции `sigsetjmp` [110, с. 178], но добавление спецификатора `volatile` предотвращает это восстановление.

Установление обработчика сигналов и буфера перехода

15-16 Для сигнала `SIGALRM` устанавливается обработчик сигнала, а функция `sigsetjmp` устанавливает буфер перехода для функции `siglongjmp`. (Эти две функции подробно описаны в разделе 10.15 [110].) Значение 1 во втором аргументе функции `sigsetjmp` указывает, что требуется сохранить текущую маску сигнала, так как мы будем вызывать функцию `siglongjmp` из нашего обработчика сигнала.

Функция `siglongjmp`

19-23 Этот фрагмент кода выполняется, только когда функция `siglongjmp` вызывается из нашего обработчика сигнала. Вызов указывает на возникновение условий, при которых мы входим в состояние ожидания: мы отправили запрос, на который не пришло никакого ответа. Если после того, как мы отправим три запроса, ответа не будет, мы прекращаем выполнение кода. По истечении времени ожидания, отведенного на получение ответа, мы выводим соответствующее сообщение и увеличиваем значение времени ожидания в два раза, то есть задаем экспоненциальное смещение (*exponential backoff*), которое также описано в разделе 20.5. Первое значение времени ожидания равно 3 с, затем — 6 с и 12 с.

Причина, по которой в этом примере мы используем функции `sigsetjmp` и `siglongjmp`, вместо того чтобы просто перехватывать ошибку `EINTR` (как мы поступили в листинге 14.1), заключается в том, что библиотечные функции захвата пакетов (которые вызываются из нашей функции `udp_read`) заново запускают операцию чтения в случае возвращения ошибки `EINTR`. Поскольку мы не хотим модифицировать библиотечные функции, единственным решением для нас является перехватывание сигнала `SIGALRM` и выполнение нелокального перехода (оператора `goto`), который возвращает управление в наш код, а не в библиотечную функцию.

Отправка запроса DNS и считывание ответа

25-26 Функция `send_dns_query` (см. листинг 29.8) отправляет запрос DNS на сервер имен. Функция `dns_read` считывает ответ. Мы вызываем функцию `alarm` для предотвращения «вечной» блокировки функции `read`. Если истекает заданное (в секундах) время ожидания, генерируется сигнал `SIGALRM`, и наш обработчик сигнала вызывает функцию `siglongjmp`.

Анализ полученной контрольной суммы UDP

27-32 Если значение полученной контрольной суммы UDP равно нулю, это значит, что сервер не вычислил и не отправил контрольную сумму.

В листинге 29.7 показана наша функция `sig_alarm` — обработчик сигнала `SIGALRM`.

Листинг 29.7. Функция `sig_alarm`: обработка сигнала `SIGALRM`

```
//udpcsum/udpcsum.c
1 #include "udpcsum.h"
2 #include <setjmp.h>

3 static sigjmp_buf jmpbuf;
4 static int canjump;

5 void
6 sig_alarm(int signo)
7 {
8     if (canjump == 0)
9         return;
10    siglongjmp(jmpbuf, 1);
11 }
```

8-10 Флаг `canjump` был установлен в листинге 29.6 после инициализации буфера перехода функцией `sigsetjmp`. Если флаг был установлен, в результате вызова функции `siglongjmp` управление осуществляется таким образом, как если бы функция `sigsetjmp` из листинга 29.6 возвратила бы значение 1.

В листинге 29.8 показана функция `send_dns_query`, посылающая запрос UDP на сервер DNS. Эта функция формирует запрос DNS.

Листинг 29.8. Функция `send_dns_query`: отправка запроса UDP на сервер DNS

```
//udpcsum/senddnsquery-raw.c
6 void
7 send_dns_query(void)
8 {
9     size_t nbytes;
10    char *buf, *ptr;

11    buf = Malloc(sizeof(struct udiphdr) + 100);
12    ptr = buf + sizeof(struct udiphdr); /* место для заголовков IP и UDP */

13    *((uint16_t*)ptr) = htons(1234); /* идентификатор */
14    ptr += 2;
15    *((uint16_t*)ptr) = htons(0x0100); /* флаги */
16    ptr += 2;
17    *((uint16_t*)ptr) = htons(1); /* количество запросов */
18    ptr += 2;
19    *((uint16_t*)ptr) = 0; /* количество записей в ответе */
20    ptr += 2;
21    *((uint16_t*)ptr) = 0; /* количество авторитетных записей */
22    ptr += 2;
23    *((uint16_t*)ptr) = 0; /* количество дополнительных записей */
```

```

24 ptr += 2;

25 memcpy(ptr, "\001a\014root-servers\003net\000", 20);
26 ptr += 20;
27 *((uint16_t*)ptr) = htons(1); /* тип запроса = A */
28 ptr += 2;
29 *((uint16_t*)ptr) = htons(1); /* класс запроса = 1 (IP-адрес) */
30 ptr += 2;

31 nbytes = (ptr - buf) - sizeof(struct udphdr);
32 udp_write(buf, nbytes),
33 if (verbose)
35 printf("sent: %d bytes of data\n", nbytes);
36 }

```

Инициализация указателя на буфер

11-12 В буфере `buf` имеется место для 20-байтового заголовка IP, 8-байтового заголовка UDP и еще 100 байт для пользовательских данных. Указатель `ptr` установлен на первый байт пользовательских данных.

Формирование запроса DNS

13-24 Для понимания деталей устройства дейтаграммы UDP требуется понимание формата сообщения DNS. Эту информацию можно найти в разделе 14.3 [111]. Мы присваиваем полю идентификации значение 1234, сбрасываем флаги, задаем количество запросов — 1, а затем обнуляем количество записей ресурсов (RR, resource records), получаемых в ответ, количество RR, определяющих полномочия, и количество дополнительных RR.

25-30 Затем мы формируем простой запрос, который располагается после заголовка: запрос типа A IP-адреса узла `a.root-servers.net`. Это доменное имя занимает 20 байт и состоит из 4 фрагментов: однобайтовая часть `a`, 12-байтовая часть `root-servers`, 3-байтовая часть `net` и корневая часть, длина которой занимает 0 байт. Тип запроса 1 (так называемый запрос типа A), и класс запроса также 1.

Запись дейтаграммы UDP

31-32 Это сообщение состоит из 36 байт пользовательских данных (восемь 2-байтовых полей и 20-байтовое доменное имя). Мы вызываем нашу функцию `udp_write` для формирования заголовков UDP и IP и последующей записи дейтаграммы UDP в наш символьный сокет.

В листинге 29.9 показана функция `open_output`, работающая с символьными сокетами.

Листинг 29.9. Функция `open_output`: подготовка символьного сокета

```

2 int rawfd; /* символьный сокет */

3 void
4 open_output(void)
5 {
6     int on=1;
7     /*
8      * Для отправки IP-дейтаграмм нужен символьный сокет
9      * Для его создания нужны права привилегированного пользователя.
10     * Кроме того, необходимо указать параметр сокета IP_HDRINCL.
11    */
12    rawfd = Socket(dest->sa_family, SOCK_RAW, 0);

```

```
13 Setsockopt(rawfd, IPPROTO_IP, IP_HDRINCL, &on., sizeof(on));
14 }
```

Объявление дескриптора символьного сокета

2 Мы объявляем глобальную переменную, в которой будет храниться дескриптор символьного сокета.

Создание сокета и установка IP_HDRINCL

7-13 Мы создаем символьный сокет и включаем параметр сокета IP_HDRINCL. Это позволяет нам формировать IP-дейтаграммы целиком, включая заголовок IP.

В листинге 29.10 показана наша функция `udp_write`, которая формирует заголовки IP и UDP, а затем записывает дейтаграмму в символьный сокет.

Листинг 29.10. Функция `udp_write`: формирование заголовков UDP и IP и запись дейтаграммы IP в символьный сокет

```
//udpcksum/udpwrite.c
19 void
20 udp_write(char *buf, int userlen)
21 {
22     struct udphdr *ui;
23     struct ip *ip;

24     /* заполнение заголовка и вычисление контрольной суммы */
25     ip = (struct ip*)buf;
26     ui = (struct udphdr*)buf;
27     bzero(ui, sizeof(*ui));
28     /* добавляем 8 к длине псевдозаголовка */
29     ui->ui_len = htons((uint16_t)(sizeof(struct udphdr) + userlen));
30     /* добавление 28 к длине IP-дейтаграммы */
31     userlen += sizeof(struct udphdr);

32     ui->ui_pr = IPPROTO_UDP;
33     ui->ui_src.s_addr = ((struct sockaddr_in*)local)->sin_addr.s_addr;
34     ui->ui_dst.s_addr = ((struct sockaddr_in*)dest)->sin_addr.s_addr;
35     ui->ui_sport = ((struct sockaddr_in*)local)->sin_port;
36     ui->ui_dport = ((struct sockaddr_in*)dest)->sin_port;
37     ui->ui_ulen = ui->ui_len;
38     if (zerosum == 0) {
39 #if 1 /* заменить на if 0 для Solaris 2.x. x < 6 */
40         if ((ui->ui_sum = m_cksum((u_int16_t*)in, userlen)) == 0)
41             ui->ui_sum = 0xffff;
42     #else
43         ui->ui_sum = ui->ui_len;
44     #endif
45     }

46     /* заполнение оставшейся части IP-заголовка */
47     /* функция p_output() вычисляет и сохраняет контрольную сумму IP */
48     ip->ip_v = IPVERSION;
49     ip->ip_hl = sizeof(struct ip) >> 2;
50     ip->ip_tos = 0;
51 #if defined(linux) || defined(__OpenBSD__)
52     ip->ip_len = htons(userlen); /* сетевой порядок байтов */
53 #else
```

```
54 ip->ip_len = userlen; /* порядок байтов узла */
55 #endif
56 ip->ip_id = 0; /* это пусть устанавливает уровень IP */
57 ip->ip_off = 0; /* смещение флагов, флаги MF и DF */
58 ip->ip_ttl = TTL_OUT;
59 Sendto(rawfd, buf, userlen, 0, dest, destlen);
60 }
```

Инициализация указателей на заголовки пакетов

24-26 Указатель `ip` указывает на начало заголовка IP (структуре `ip`), а указатель `ui` указывает на то же место, но структура `udphdr` является объединением заголовков IP и UDP.

Обнуление заголовка

27 Мы явным образом записываем в заголовок нули, чтобы предотвратить учет случайного мусора, который мог остаться в буфере, при вычислении контрольной суммы.

Обновление значений длины

28-31 Переменная `ui_len` — это длина дейтаграммы UDP: количество байтов пользовательских данных плюс размер заголовка UDP (8 байт). Переменная `userlen` (количество байтов пользовательских данных, которые следуют за заголовком UDP) увеличивается на 28 (20 байт на заголовок IP и 8 байт на заголовок UDP), для того чтобы соответствовать настоящему размеру дейтаграммы IP.

Заполнение заголовка UDP и вычисление контрольной суммы UDP

32-45 При вычислении контрольной суммы UDP учитывается не только заголовок и данные UDP, но и поля заголовка IP. Эти дополнительные поля заголовка IP образуют то, что называется *псевдозаголовком* (*pseudoheader*). Включение псевдозаголовка обеспечивает дополнительную проверку на то, что если значение контрольной суммы верно, то дейтаграмма была доставлена на правильный узел и с правильным кодом протокола. В указанных строках располагаются операторы инициализации полей в IP-заголовке, формирующих псевдозаголовок. Данный фрагмент кода несколько запутан, но его объяснение приводится в разделе 23.6 [128]. Конечным результатом является запись контрольной суммы UDP в поле `ui_sum`, если не установлен флаг `zerosum` (что соответствует наличию аргумента командной строки `-0`).

Если при вычислении контрольной суммы получается 0, вместо него записывается значение `0xffff`. В обратном коде эти числа совпадают, но протокол UDP устанавливает контрольную сумму в нуль, чтобы обозначить, что она вовсе не была вычислена. Обратите внимание, что в листинге 28.10 мы не проверяем, равно ли значение контрольной суммы нулю: дело в том, что в случае ICMPv4 нулевое значение контрольной суммы не означает ее отсутствия.

ПРИМЕЧАНИЕ

Следует отметить, что в Solaris 2.x, где $x < 6$, в случаях, когда дейтаграммы UDP или сегменты TCP отправляются с символьного сокета при установленном параметре `IP_HDRINCL`, возникает ошибка. Контрольную сумму вычисляет ядро, а мы должны установить поле `ui_sum` равным длине дейтаграммы UDP.

Заполнение заголовка IP

36-49 Поскольку мы установили параметр сокета IP_HDRINCL, нам следует заполнить большую часть полей в заголовке IP. (В разделе 28.3 обсуждается запись в символьный сокет при включенном параметре IP_HDRINCL.) Мы присваиваем полю идентификации нуль (ip_id), что указывает IP на необходимость задания значения этого поля. IP также вычисляет контрольную сумму IP, а функция sendto записывает дейтаграмму IP.

ПРИМЕЧАНИЕ

Обратите внимание, что поле ip_len может иметь либо сетевой порядок байтов, либо порядок байтов узла. Это типичная проблема с совместимостью, возникающая при использовании символьных сокетов.

Следующая функция — это udp_read, показанная в листинге 29.11. Она вызывается из кода, представленного в листинге 29.6.

Листинг 29.11. Функция udp_read: чтение очередного пакета из устройства захвата пакетов

```
//udpcsum/udpread.c
7 struct udpiphdr*
8 udp_read(void)
9 {
10 int len;
11 char *ptr;
12 struct ether_header *eptr;

13 for (;;) {
14     ptr = next_pcap(&len);

15     switch (datalink) {
16     case DLT_NULL: /* заголовок обратной петли = 4 байта */
17         return (udp_check(ptr + 4, len - 4));

18     case DLT_EN10MB:
19         eptr = (struct ether_header*)ptr;
20         if (ntohs(eptr->ether_type) != ETHERTYPE_IP)
21             err_quit("Ethernet type not IP", ntohs(eptr->ether_type));
22         return (udp_check(ptr + 14, len - 14));

23     case DLT_SLIP: /* заголовок SLIP = 24 байта */
24         return (udp_check(ptr + 24, len - 24));

25     case DLT_PPP: /* заголовок PPP = 24 байта */
26         return (udp_check(ptr + 24, len - 24));

27     default:
28         err_quit("unsupported datalink (%d)", datalink);
29     }
30 }
31 }
```

14-29 Наша функция next_pcap (см. листинг 29.12) возвращает следующий пакет из устройства захвата пакетов. Поскольку заголовки канального уровня различаются в зависимости от фактического типа устройства, мы применяем ветвление в зависимости от значения, возвращаемого функцией pcap_datalink.

ПРИМЕЧАНИЕ

Сдвиги на 4, 14 и 24 байта объясняются на рис. 31.9 [128]. Сдвиг, равный 24 байтам, показанный для заголовков SLIP и PPP, применяется в BSD/OS 2.1.

Несмотря на то, что в названии DLT_EN10MB фигурирует обозначение «10МВ», этот тип канального уровня используется для сетей Ethernet, в которых скорость передачи данных равна 100 Мбит/с.

Наша функция `udp_check` (см. листинг 29.13) исследует пакет и проверяет поля в заголовках IP и UDP.

В листинге 29.12 показана функция `next_pcap`, возвращающая следующий пакет из устройства захвата пакетов.

Листинг 29.12. Функция `next_pcap`: возвращает следующий пакет

```
//udpcsum/pcap.c
38 char*
39 next_pcap(int *len)
40 {
41     char *ptr;
42     struct pcap_pkthdr hdr;

43     /* продолжаем следить, пока пакет не будет готов */
44     while ((ptr = (char*)pcap_next(pd, &hdr)) == NULL);

45     *len = hdr.caplen; /* длина захваченного пакета */
46     return (ptr);
47 }
```

43-44 Мы вызываем библиотечную функцию `pcap_next`, возвращающую следующий пакет. Указатель на пакет является возвращаемым значением данной функции, а второй аргумент указывает на структуру `pcap_pkthdr`, которая тоже возвращается заполненной:

```
struct pcap_pkthdr {
    struct timeval ts;      /* временная метка */
    bpf_u_int32 caplen;    /* длина захваченного фрагмента */
    bpf_u_int32 len;       /* полная длина пакета, находящегося в канале */
};
```

Временная отметка относится к тому моменту, когда пакет был считан устройством захвата пакетов, в противоположность моменту фактической передачи пакета процессу, которая может произойти чуть позже. Переменная `caplen` содержит длину захваченных данных (вспомним, что в листинге 29.2 нашей переменной `shaplen` было присвоено значение 200 и она являлась вторым аргументом функции `pcap_open_live` в листинге 29.5). Назначение устройства захвата пакетов состоит в захвате заголовков, а не всего содержимого каждого пакета. Переменная `len` — это полная длина пакета, находящегося в канале. Значение `caplen` будет всегда меньше или равно значению `len`.

45-46 Перехваченная часть пакета возвращается через указатель (аргумент функции), и возвращаемым значением функции является указатель на пакет. Следует помнить, что указатель на пакет указывает фактически на заголовок канального уровня, который представляет собой 14-байтовый заголовок Ethernet в случае кадра Ethernet или 4-байтовый псевдоканальный (pseudo-link) заголовок в случае закольцовки на себя.

Если мы посмотрим на библиотечную реализацию функции `pcap_next`, мы увидим, что между различными функциями существует некоторое «разделение труда», схематически изображенное на рис. 29.5. Наше приложение вызывает функции `pcap_`, среди которых есть как зависящие, так и не зависящие от устройства захвата пакетов. Например, мы показываем, что реализация BPF вызывает функцию `read`, в то время как реализация DLPI вызывает функцию `getmsg`, а реализация Linux вызывает `recvfrom`.

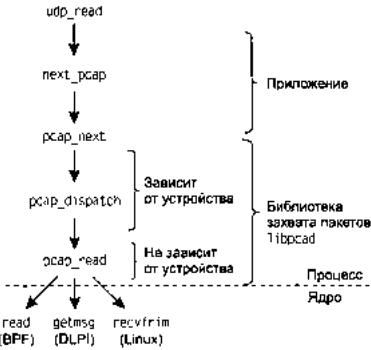


Рис. 29.5. Организация вызовов функций для чтения из библиотеки захвата пакетов

Наша функция `udp_check` проверяет различные поля в заголовках IP и UDP. Она показана в листинге 29.13. Эту проверку необходимо выполнить, так как при получении пакета от устройства захвата пакетов уровень IP не замечает этого пакета. Для символьного сокета это не так.

44-61 Длина пакета должна включать хотя бы заголовки IP и UDP. Версия IP проверяется вместе с длиной и контрольной суммой заголовка IP. Если поле протокола указывает на дейтаграмму UDP, функция возвращает указатель на объединенный заголовок IP/UDP. В противном случае программа завершается, так как фильтр захвата пакетов, заданный при вызове функции `pcap_setfilter` в листинге 29.5, не должен возвращать пакеты никакого другого типа.

Листинг 29.13. Функция `udp_check`: проверка полей в заголовках IP и UDP

```

//udpcksum/udpread.c
38 struct udiphdr*
39 udp_check(char *ptr, int len)
40 {
41     int hlen;
42     struct ip *ip;
43     struct udiphdr *ui;

44     if (len < sizeof(struct ip) + sizeof(struct udphdr))
45         err_quit("len = %d", len);

46     /* минимальная проверка заголовка IP */
47     ip = (struct ip*)ptr;
48     if (ip->ip_v != IPVERSION)
49         err_quit("ip_v = %d", ip->ip_v);
50     hlen = ip->ip_hl << 2;
51     if (hlen < sizeof(struct ip))
52         err_quit("ip_hl = %d", ip->ip_hl);
53     if (len < hlen + sizeof(struct udphdr))
54         err_quit("len = %d, hlen = %d", len, hlen);

55     if ((ip->ip_sum = in_cksum((u_short *)ip, hlen)) != 0)
56         err_quit("ip checksum error");

57     if (ip->ip_p == IPPROTO_UDP) {
58         ui = (struct udiphdr*)ip;
59         return (ui);
60     } else
61         err_quit("not a UDP packet");
62 }
```

Функция `cleanup`, показанная в листинге 29.14, вызывается из функции `main` непосредственно перед тем, как программа завершается, а также вызывается в качестве обработчика сигнала в случае, если пользователь прерывает выполнение программы (см. листинг 29.4).

Листинг 29.14. Функция `cleanup`

```

//udpcksum/cleanup.c
1 void
2 cleanup(int signo)
3 {
4     struct pcap_stat stat;
5
6     fflush(stdout);
7     putc('\n', stdout);
8
9     if (verbose) {
10         if (pcap_stats(pd, &stat) < 0)
11             err_quit("pcap_stats: %s\n", pcap_geterr(pd));
12         printf("%d packets received by filter\n", stat.ps_recv);
13         printf("%d packets dropped by kernel\n", stat.ps_drop);
14     }
15     exit(0);
16 }
```

Получение и вывод статистики по захвату пакетов

8-13 Функция `pcap_stats` получает статистику захвата пакетов: общее количество полученных фильтром пакетов и количество пакетов, переданных ядру.

Пример

Сначала мы запустим нашу программу с аргументом командной строки `-0` и убедимся, что сервер имен отвечает на приходящие дейтаграммы, не содержащие контрольной суммы. Мы также задаем флаг `-v`.

```
macosx # udpcksum -i en1 -0 -v bridget.rudoff.com domain
device = en1
local net = 172.24.37.64. netmask = 255.255.255.224
cmd = udp and src host 206.168.112.96 and src port 53
datalink = 1
sent: 36 bytes of data
UDP checksums on
received UDP checksum = 9d15
```

```
3 packets received by filter
0 packets dropped by kernel
```

Затем мы запускаем нашу программу, обращаясь к локальному серверу имен, в котором отключен подсчет контрольных сумм. Чем дальше, тем сложнее становится найти сервер имен с отключенным подсчетом контрольных сумм.

```
macosx # udpcksum -i en1 -v freebsd4.unpbook.com domain
device = en1
localnet = 172.24.37.64, netmask = 255.255.255.224
cmd = udp and src host 172.24.37.94 and src port 53
datalink = 1
sent: 36 bytes of data
UDP checksums off
received UDP checksum = 0

3 packets received by filter
0 packets dropped by kernel
```

Функции libnet

В этом разделе приводятся альтернативные версии функций `open_output` и `send_dns_query`, в которых вместо символьных сокетов используются функции библиотеки `libnet`. Библиотека `libnet` берет на себя заботу о множестве деталей, в частности, устраняет проблемы с переносимостью, связанные с вычислением контрольных сумм и порядком байтов в заголовке, о которых мы говорили выше. Функция `open_output` представлена в листинге 29.15.

Листинг 29.15. Функция `open_output`, использующая `libnet`

```
//udpcsum/senddnsquery-libnet.c
7 static libnet_t *l; /* дескриптор libnet */

8 void
9 open_output(void)
10 {
11     char errbuf[LIBNET_ERRBUF_SIZE];
12     /* инициализация libnet с символьным сокетом IPv4 */
13     l = libnet_init(LIBNET_RAW4, NULL, errbuf);
14     if (l == NULL) {
15         err_quit("Can't initialize libnet: %s", errbuf);
16     }
17 }
```

Объявление дескриптора `libnet`

7 В библиотеке `libnet` используется непрозрачный тип `libnet_t`. Функция `libnet_init` возвращает указатель на этот тип, который затем передается другим функциям `libnet` для обращения к конкретному сокету. В этом смысле данный тип аналогичен дескрипторам сокетов и устройств `psap`.

Инициализация `libnet`

12-16 Мы вызываем функцию `libnet_init`, запрашивая открытие символьного сокета IPv4. Для этого в качестве первого аргумента указывается константа `LIBNET_RAW4`. В случае возникновения ошибки функция возвращает текст сообщения в аргументе `errbuf`, который мы распечатываем.

Функция `send_dns_query` для `libnet` представлена в листинге 29.16. Сравните ее с функциями `send_dns_query` и `udp_write` для символьных сокетов.

Листинг 29.16. Функция `send_dns_query`, использующая `libnet`

```
//udpcsum/senddnsquery-libnet.c
18 void
19 send_dns_query(void)
20 {
21     char qbuf[24], *ptr;
22     u_int16_t one;
23     int packet_size = LIBNET_UDP_H + LIBNET_DNSV4_H + 24;
24     static libnet_ptag_t ip_tag, udp_tag, dns_tag;

25     /* построение запроса внутри UDP-пакета */
26     ptr = qbuf;
27     memcpy(ptr, "\001a\014root-servers\003net\000", 20);
28     ptr += 20;
29     one = htons(1);
30     memcpy(ptr, &one, 2); /* тип запроса = A */
31     ptr += 2;
32     memcpy(ptr, &one, 2); /* класс запроса = 1 (IP-адрес) */

33     /* формирование пакета DNS */
34     dns_tag = libnet_build_dnsv4(
```

```

35  1234 /* идентификатор */,
36  0x0100 /* флаги: рекурсия разрешена */,
37  1 /* кол-во запросов */, 0 /* кол-во записей в ответе */,
38  0 /* кол-во авторитетных записей */, 0 /* кол-во дополнительных */,
39  qbuf /* запрос */,
40  24 /* длина запроса */, 1, dns_tag);
41 /* формирование заголовка UDP */
42 udp_tag = libnet_build_udp(
43  ((struct sockaddr_in*)local)->
44  sin_port /* порт отправителя */,
45  ((struct sockaddr_in*)dest)->
46  sin_port /* порт получателя */,
47  packet_size /* длина */, 0 /* контрольная сумма */,
48  NULL /* полезные данные */, 0 /* длина полезн. данных */, 1, udp_tag);
49 /* Так как мы установили контр. сумму равной нулю, libnet автоматически */
50 /* рассчитает контр. сумму UDP. Эту функцию можно отключить. */
51 if (zerosum)
52  if (libnet_toggle_checksum(1, udp_tag, LIBNET_OFF) < 0)
53   err_quit("turning off checksums: %s\n", libnet_geterror(1));
54 /* формирование IP-заголовка */
55 ip_tag = libnet_build_ipv4(packet_size + LIBNET_IPV4_H /* длина */,
56  0 /* tos */, 0 /* IP ID */, 0 /* фрагмент */,
57  TTL_OUT /* ttl */, IPPROTO_UDP /* протокол */,
58  0 /* контр. сумма */,
59  ((struct sockaddr_in*)local)->sin_addr.s_addr /* отправитель */,
60  ((struct sockaddr_in*)dest)->sin_addr.s_addr /* получатель */,
61  NULL /* полезные данные */, 0 /* длина полезн. данных */, 1, ip_tag);

62 if (libnet_write(l) < 0) {
63  err_quit("libnet_write: %s\n", libnet_geterror(1));
64 }
65 if (verbose)
66  printf("sent: %d bytes of data\n", packet_size);
67 }

```

Формирование запроса DNS

25-32 Мы начинаем с формирования запроса DNS, которое выполняется так же, как в строках 25–30 листинга 29.8.

34-40 Затем мы вызываем функцию `libnet_build_dnsv4`, которая принимает поля пакета DNS в виде отдельных аргументов. Нам достаточно знать содержимое запроса, а упорядочением этого содержимого в заголовке пакета DNS занимается функция.

Заполнение заголовка UDP и подготовка к вычислению контрольной суммы UDP

42-48 Мы формируем заголовок UDP, вызывая функцию `libnet_build_udp`. Поля заголовка UDP принимаются этой функцией также в виде отдельных аргументов. Если значение переданной контрольной суммы равно 0, `libnet` автоматически рассчитывает контрольную сумму.

49-52 Если пользователь запретил вычисление контрольной суммы, мы должны отключить эту функцию `libnet` явным образом.

Заполнение заголовка IP

53-65 Окончательное формирование пакета требует построения заголовка IPv4 вызовом `libnet_build_ipv4`.

ПРИМЕЧАНИЕ

Библиотека `libnet` автоматически записывает поле `ip_len` в нужном порядке байтов. Это пример повышения переносимости программы благодаря использованию библиотек.

Отправка UDP-дейтаграммы

66-70 Мы вызываем функцию `libnet_write` для отправки подготовленной дейтаграммы в сеть.

Функция `send_dns_query`, использующая `libnet`, состоит всего из 67 строк, тогда как в версии, работавшей с символьными сокетами, общая длина кода составила 96 строк, в которых было по крайней мере 2 трюка, связанных с переносимостью.

29.8. Резюме

Символьные сокеты предоставляют возможность записывать и считывать IP-дейтаграммы, которые могут быть не поняты ядром, а доступ к канальному уровню позволяет считывать и записывать кадры канального уровня *любых* типов (не только дейтаграммы IP). Программа `tcpdump` — это, вероятно, наиболее широко используемая программа, имеющая непосредственный доступ к канальному уровню.

В различных операционных системах применяются различные способы доступа к канальному уровню. Мы рассмотрели пакетный фильтр Беркли, DLPI SVR4 и пакетные сокеты Linux (`SOCK_PACKET`). Но у нас имеется возможность, не вникая в различия перечисленных способов, использовать находящуюся в свободном доступе переносимую библиотеку захвата пакетов `libcap`.

Отправка символьных дейтаграмм осуществляется в разных системах по-разному. Свободно распространяемая библиотека `libnet` скрывает различия между системами и предоставляет интерфейс для вывода через символьные сокеты и непосредственно на канальном уровне.

Упражнения

1. Каково назначение флага `sञjumр` в листинге 29.7?
2. При работе программы `udprcksum` наиболее распространенным сообщением об ошибке является сообщение о недоступности порта ICMP (в пункте назначения не работает сервер имен) или недоступности узла ICMP. В обоих случаях нам не нужно ждать истечения времени ожидания, заданного функцией `udp_read` в листинге 29.6, так как сообщение о подобной ошибке фактически является ответом на наш запрос DNS. Модифицируйте программу таким образом, чтобы она перехватывала эти ошибки ICMP.

Глава 30

Альтернативное устройство клиента и сервера

30.1. Введение

При написании сервера под Unix мы можем выбирать из следующих вариантов управления процессом:

- Наш первый сервер, показанный в листинге 1.5, был *последовательным* (*iterative*), но количество сценариев, для которых этот вариант является предпочтительным, весьма ограничено, поскольку последовательный сервер не может начать обработку очередного клиентского запроса, не закончив полностью обработку текущего запроса.
- В листинге 5.1 показан первый в данной книге *параллельный* (*concurrent*) сервер, который для обработки каждого клиентского запроса порождал дочерний процесс с помощью функции `fork`. Традиционно большинство серверов, работающих под Unix, попадают в эту категорию.
- В разделе 6.8 мы разработали другую версию сервера TCP, в котором имеется только один процесс, обрабатывающий любое количество клиентских запросов с помощью функции `select`.
- В листинге 26.2 мы модифицировали параллельный сервер, создав для каждого клиента по одному потоку вместо одного процесса.

В этой главе мы рассмотрим два других способа модификации устройства параллельного сервера.

- *Предварительное создание дочерних процессов* (*preforking*). В этом случае при запуске сервера выполняется функция `fork`, которая создает определенное количество (пул) дочерних процессов. Обработкой очередного клиентского запроса занимается процесс, взятый из этого набора.

- *Предварительное создание потоков* (*prethreading*). При запуске сервера создается некоторое количество (пул) потоков, и для обработки каждого клиента используется поток из данного набора.

В данной главе мы будем рассматривать множество вопросов, связанных с предварительным созданием потоков и процессов. Например, что произойдет, если в пуле окажется недостаточное количество процессов или потоков? А если их будет слишком много? Как родительский и дочерние процессы (или потоки) синхронизируют свои действия?

Обычно написать клиент легче, чем сервер, за счет простоты управления процессом клиента. Тем не менее мы уже исследовали различные способы написания простого эхо-клиента, которые вкратце изложены в разделе 30.2.

В этой главе мы рассматриваем девять различных способов устройства сервера и взаимодействие каждого из этих серверов с одним и тем же клиентом. Клиент-серверный сценарий типичен для WWW: клиент посыпает небольшой по объему запрос, а сервер отвечает ему, отсылая соответствующие запросу данные. Некоторые из этих серверов мы уже достаточно подробно обсуждали (например, параллельный сервер, вызывающий функцию `fork` для обработки каждого клиентского запроса), в то время как предварительное создание процессов и потоков являются новыми для нас концепциями, которые и будут подробно рассмотрены в этой главе.

Мы запускали различные экземпляры клиента с каждым сервером, измеряя время, которое процессор тратит на обслуживание определенного количества клиентских запросов. Чтобы информация об этом не оказалась рассеянной по всей главе, мы свели все полученные результаты в табл. 30.1, на которую в этой главе будем неоднократно ссылаться. Следует отметить, что значения времени, указанные в этой таблице, соответствуют процессорному времени, затраченному *только на управление процессом*, так как из фактического значения времени процессора мы вычитаем время, которое тратит на выполнение того же задания последовательный сервер, не имеющий накладных расходов, связанных с управлением процессом. Иными словами, нулевой точкой отсчета в данной таблице для нас является время, затраченное последовательным сервером. Для большей наглядности мы включили в таблицу строку для последовательного сервера с нулевыми значениями времени. В этой главе термином *время центрального процессора на управление процессом* (*process control CPU time*) мы обозначаем разность между фактическим значением времени центрального процессора и временем, затраченным последовательным сервером, для каждой конкретной системы.

Таблица 30.1. Сравнительные значения времени, затраченного каждым из обсуждаемых в данной главе сервером

Описание сервера	Время центрального процессора на управление процессом
0 Последовательный (точка отсчета; затраты на управление процессом отсутствуют)	0,0
1 Параллельный сервер, один вызов функции fork для обработки одного клиента	20,90
2 Предварительное создание дочерних процессов, каждый из которых вызывает функцию accept	1,80
3 Предварительное создание дочерних процессов с блокировкой для защиты accept	2,07
4 Предварительное создание дочерних процессов с использованием взаимного исключения для защиты accept	1,75
5 Предварительное создание дочерних процессов, родительский процесс передает дочернему дескриптор сокета	2,58
6 Параллельный сервер, создание одного потока на каждый клиентский запрос	0,99
7 Предварительное создание потоков с использованием взаимного исключения для защиты accept	1,93
8 Предварительное создание потоков, главный поток вызывает accept	2,05

Все приведенные выше значения времени были получены путем запуска клиента, показанного в листинге 30.1, на двух различных узлах в той же подсети, что и сервер. Во всех тестах оба клиента порождали пять дочерних процессов для создания пяти одновременных соединений с сервером, таким образом максимальное количество одновременных соединений с сервером было равно 10. Каждый клиент запрашивал 4000 байт данных от сервера по каждому соединению. В случае, когда тест подразумевает предварительное создание дочерних процессов или потоков при запуске сервера, их количество равно 15.

Некоторые версии нашего сервера работали с предварительно созданным пулом потоков или процессов. Интересным моментом является распределение клиентских запросов по потокам или дочерним процессам, находящимся в накопителе. В табл. 30.2 показаны варианты этого распределения, которые также будут обсуждаться в соответствующих разделах.

Таблица 30.2. Количество клиентов, обслуженных каждым из 15 дочерних процессов или потоков

№	Предварительное процесса создание процессов или без защиты accept потока (строка 2)	Предварительное создание процессов с защитой accept (строка 3)	Предварительное создание процессов, передача дескриптора (строка 5)	Предварительное порождение потоков, защита accept (строка 7)
0	333	347	1006	333
1	340	328	950	323
2	335	332	720	333
3	335	335	583	328
4	332	338	485	329
5	331	340	457	322
6	333	335	385	324
7	333	343	250	360
8	332	324	105	341
9	331	315	32	348
10	334	326	14	358
11	333	340	9	331
12	334	330	4	321
13	332	331	1	329
14	332	336	0	320

5000 5000 5000 5000

30.2. Альтернативы для клиента TCP

Мы уже обсуждали различные способы устройства клиентов, но стоит тем не менее еще раз обратить внимание на относительные достоинства и недостатки этих способов.

1. В листинге 5.4 показан основной способ устройства клиента TCP. С этой программой были связаны две проблемы. Во-первых, когда она блокируется в ожидании ввода пользователя, она не замечает происходящих в сети событий, например отключения собеседника от соединения. Во-вторых, она действует в режиме остановки и ожидания, что неэффективно в случае пакетной обработки.

2. Листинг 6.1 содержит следующую, модифицированную версию клиента. С помощью функции select клиент получает информацию о событиях в сети во время ожидания ввода пользователя. Однако проблема этой версии заключается в том, что программа не способна корректно работать в пакетном режиме. В листинге 6.2 эта проблема решается путем применения функции shutdown.

3. С листинга 16.1 начинается рассмотрение клиентов, использующих неблокируемый ввод-вывод.

4. Первым из рассмотренных нами клиентов, вышедшим за пределы ограничений, связанных с наличием единственного процесса или потока для обслуживания всех запросов, является клиент, изображенный в листинге 16.6. В этом случае использовалась функция fork, и один процесс обрабатывал передачу данных от клиента к серверу, а другой — в обратном направлении.

5. В листинге 26.1 используются два потока вместо двух процессов.

В конце раздела 16.2 мы резюмируем различия между перечисленными версиями. Как мы отметили, хотя версия с неблокируемым вводом-выводом является самой быстродействующей, ее код слишком сложен, а применение двух потоков или двух процессов упрощает код.

30.3. Тестовый клиент TCP

В листинге 30.1^[1] показан клиент, который будет использоваться для тестирования всех вариаций нашего сервера.

Листинг 30.1. Код клиента TCP для проверки различных версий сервера

```
//server/client.c
1 #include "unp.h"

2 #define MAXN 16384 /* максимальное количество байтов, которые могут быть
                     запрошены клиентом от сервера */

3 int
4 main(int argc, char **argv)
5 {
6     int i, j, fd, nchildren, nloops, nbytes;
7     pid_t pid;
8     ssize_t n,
9     char request[MAXN], reply[MAXN];

10    if (argc != 6)
11        err_quit("usage: client <hostname or IPAddr> <port> <#children> "
12                "<#loops/child> <#bytes/request>");

13    nchildren = atoi(argv[3]);
14    nloops = atoi(argv[4]);
15    nbytes = atoi(argv[5]);
16    sprintf(request, sizeof(request), "%d\n", nbytes); /* в конце
                                                               символ новой строки */

17    for (i = 0; i < nchildren; i++) {
18        if ((pid = Fork()) == 0) { /* дочерний процесс */
```

```

19     for (j = 0; j < nloops; j++) {
20         fd = Tcp_connect(argv[1], argv[2]);
21
22         Write(fd, request, strlen(request));
23
24         if ((n = Readn(fd, reply, nbytes)) != nbytes)
25             err_quit("server returned %d bytes", n);
26
27         Close(fd); /* состояние TIME_WAIT на стороне клиента,
28                     а не сервера */
29     }
30 }

31 while (wait(NULL) > 0) /* теперь родитель ждет завершения всех
32                         дочерних процессов */
33 ;
34 if (errno != ECHILD)
35     err_sys("wait error");
36 }
```

10-12 Каждый раз при запуске клиента мы задаем имя узла или IP-адрес сервера, порт сервера, количество дочерних процессов, порождаемых функцией `fork` (что позволяет нам инициировать несколько одновременных соединений с сервером), количество запросов, которое каждый дочерний процесс должен посыпать серверу, и количество байтов, отправляемых сервером в ответ на каждый запрос.

17-30 Родительский процесс вызывает функцию `fork` для порождения каждого дочернего процесса, и каждый дочерний процесс устанавливает указанное количество соединений с сервером. По каждому соединению дочерний процесс посыпает запрос, задавая количество байтов, которое должен вернуть сервер, а затем дочерний процесс считывает это количество данных с сервера. Родительский процесс просто ждет завершения выполнения всех дочерних процессов. Обратите внимание, что клиент закрывает каждое соединение TCP, таким образом состояние TCP TIME_WAIT имеет место на стороне клиента, а не на стороне сервера. Это отличает наше клиент-серверное соединение от обычного соединения HTTP.

При тестировании различных серверов из этой главы мы запускали клиент следующим образом:

```
% client 192.168.1.20 8888 5 500 4000
```

Таким образом создается 2500 соединений TCP с сервером: по 500 соединений от каждого из 5 дочерних процессов. По каждому соединению от клиента к серверу посыпается 5 байт ("4000\n"), а от сервера клиенту передается 4000 байт. Мы запускаем клиент на двух различных узлах, соединяясь с одним и тем же сервером, что дает в сумме 5000 соединений TCP, причем максимальное количество одновременных соединений с сервером в любой момент времени равно 10.

ПРИМЕЧАНИЕ

Для проверки различных веб-серверов существуют изощренные контрольные тесты. Один из них называется WebStone. Информация о нем находится в свободном доступе по адресу <http://www.mindcraft.com/webstone>. Для общего сравнения различных альтернативных устройств сервера, которые мы рассматриваем в этой главе, нам не нужны столь сложные тесты.

Теперь мы представим девять различных вариантов устройства сервера.

30.4. Последовательный сервер TCP

Последовательный сервер TCP полностью обрабатывает запрос каждого клиента, прежде чем перейти к следующему клиенту. Последовательные серверы редко используются, но один из них, простой сервер времени и даты, мы показали в листинге 1.5.

Тем не менее у нас имеется область, в которой желательно применение именно последовательного сервера — это сравнение характеристик других серверов. Если мы запустим клиент следующим образом:

```
% client 192.168.1.20 8888 1 5000 4000
```

и соединимся с последовательным сервером, то получим такое же количество соединений TCP (5000) и такое же количество данных, передаваемых по одному соединению. Но поскольку сервер является последовательным, на нем *не осуществляется никакого управления процессами*. Это дает нам возможность получить базовое значение времени, затрачиваемого центральным процессором на обработку указанного количества запросов, которое потом мы можем вычесть из результатов измерений для других серверов. С точки зрения управления процессами последовательный сервер является самым быстрым, поскольку он вовсе не занимается этим управлением. Взяв последовательный сервер за точку отсчета, мы можем сравнивать результаты измерений быстродействия других серверов, показанные в табл. 30.1.

Мы не приводим код для последовательного сервера, так как он представляет собой тривиальную модификацию параллельного сервера, показанного в следующем разделе.

30.5. Параллельный сервер TCP: один дочерний процесс для каждого клиента

Традиционно параллельный сервер TCP вызывает функцию `fork` для порождения нового дочернего процесса, который будет выполнять обработку очередного клиентского запроса. Это позволяет серверу обрабатывать несколько запросов одновременно, выделяя по одному дочернему процессу для каждого клиента. Единственным ограничением на количество одновременно обрабатываемых клиентских запросов является ограничение операционной системы на количество дочерних процессов, допустимое для пользователя, в сеансе которого работает сервер. Листинг 5.9 содержит пример параллельного сервера, и большинство серверов TCP написаны в том же стиле.

Проблема с параллельными серверами заключается в количестве времени, которое тратит центральный процессор на выполнение функции `fork` для порождения нового дочернего процесса для каждого клиента. Давным-давно, в конце 80-х годов XX века, когда наиболее загруженные серверы обрабатывали сотни или тысячи клиентов за день, это было приемлемо. Но расширение Сети изменило требования. Теперь загруженными считаются серверы, обрабатывающие миллионы соединений TCP в день. Сказанное относится лишь к одиночным узлам, но наиболее загруженные сайты используют несколько узлов, распределяя нагрузку между ними (в разделе 14.2 [112] рассказывается об общепринятым способе распределения этой нагрузки, называемом *циклическим обслуживанием DNS* — *DNS round robin*). В последующих разделах описаны различные способы, позволяющие избежать вызова функции `fork` для каждого клиентского запроса, но тем не менее параллельные серверы остаются широко распространенными.

В листинге 30.2 показана функция `main` для нашего параллельного сервера TCP.

Листинг 30.2. Функция `main` для параллельного сервера TCP

```
//server/serv01.c
1 include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int listenfd, connfd;
6     pid_t childpid;
7     void sig_chld(int), sig_int(int), web_child(int);
8     socklen_t clilen, addrlen;
9     struct sockaddr *cliaddr;

10    if (argc == 2)
11        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
12    else if (argc == 3)
13        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
14    else
```

```

15  err_quit("usage: serv01 [ <host> ] <port#>");
16  cliaddr = Malloc(addrhlen);

17  Signal(SIGCHLD, sig_chld);
18  Signal(SIGINT, sig_int);
19  for (;;) {
20    clilen = addrlen;
21    if ((connfd = accept(listenfd, cliaddr, &clilen)) < 0) {
22      if (errno == EINTR)
23        continue; /* назад к for() */
24      else
25        err_sys("accept error");
26    }
27    if ((childpid = Fork()) == 0) { /* дочерний процесс */
28      Close(listenfd); /* закрываем прослушиваемый сокет */
29      web_child(connfd); /* обрабатываем запрос */
30      exit(0);
31    }
32    Close(connfd); /* родительский процесс закрывает
                       присоединенный сокет */
33  }
34 }
```

Эта функция аналогична функции, показанной в листинге 5.9: она вызывает функцию `fork` для каждого клиентского соединения и обрабатывает сигналы `SIGCHLD`, приходящие от закончивших свое выполнение дочерних процессов. Тем не менее мы сделали эту функцию не зависящей от протокола за счет вызова функции `tcp_listen`. Мы не показываем обработчик сигнала `sig_chld`: он совпадает с показанным в листинге 5.8, но только без функции `printf`.

Мы также перехватываем сигнал `SIGINT`, который генерируется при вводе символа прерывания. Мы вводим этот символ после завершения работы клиента, чтобы было выведено время, потраченное центральным процессором на выполнение данной программы. В листинге 30.3 показан обработчик сигнала. Это пример обработчика сигнала, который никогда не возвращает управление.

Листинг 30.3. Обработчик сигнала SIGINT

```

//server/serv01.c
35 void
36 sig_int(int signo)
37 {
38  void pr_cpu_time(void);
39  pr_cpu_time();
40  exit(0);
41 }
```

В листинге 30.4 показана функция `pr_cpu_time`, вызываемая из обработчика сигнала.

Листинг 30.4. Функция `pr_cpu_time`: вывод полного времени центрального процессора

```

//server/pr_cpu_time.c
1 #include "unp.h"
2 #include <sys/resource.h>

3 #ifndef HAVE_GETRUSAGE_PROTO
4 int getrusage(int, struct rusage*);
5 #endif

6 void
7 pr_cpu_time(void)
8 {
9  double user, sys;
10 struct rusage myusage, childusage;
```

```

11 if (getrusage(RUSAGE_SELF, &myusage) < 0)
12 err_sys("getrusage error");
13 if (getrusage(RUSAGE_CHILDREN, &childusage) < 0)
14 err_sys("getrusage error");

15 user = (double)myusage.ru_utime.tv_sec +
16 myusage.ru_utime.tv_usec / 1000000.0;
17 user += (double)childusage.ru_utime.tv_sec +
18 childusage.ru_utime.tv_usec / 1000000.0;
19 sys = (double)myusage.ru_stime.tv_sec +
20 myusage.ru_stime.tv_usec / 1000000.0;
21 sys += (double)childusage.ru_stime.tv_sec +
22 childusage.ru_stime.tv_usec / 1000000.0;

21 printf("\nuser time = %g, sys time = %g\n", user, sys);
22 }

```

Функция `getrusage` вызывается дважды: она позволяет получить данные об использовании ресурсов вызывающим процессом (`RUSAGE_SELF`) и всеми его дочерними процессами, которые завершили свое выполнение (`RUSAGE_CHILDREN`). Выводится время, затраченное центральным процессором на выполнение пользовательского процесса (общее пользовательское время, `total user time`), и время, которое центральный процессор затратил внутри ядра на выполнение задач, заданных вызывающим процессом (общее системное время, `total system time`).

Возвращаясь к листингу 30.2, мы видим, что для обработки каждого клиентского запроса вызывается функция `web_child`. Эта функция показана в листинге 30.5.

Листинг 30.5. Функция `web_child`: обработка каждого клиентского запроса

```

//server/web_child.c
1 #include "unp.h"

2 #define MAXN 16384 /* максимальное количество байтов, которое клиент
может запросить */

3 void
4 web_child(int sockfd)
5 {
6     int ntwrite;
7     ssize_t nread;
8     char line[MAXLINE], result[MAXN];

9     for (;;) {
10         if ((nread = Readline(sockfd, line, MAXLINE)) == 0)
11             return; /* соединение закрыто другим концом */

12         /* line задает, сколько байтов следует отправлять обратно */
13         ntwrite = atol(line);
14         if ((ntwrite <= 0) || (ntwrite > MAXN))
15             err_quit("client request for bytes", ntwrite);
16         Writen(sockfd, result, ntwrite);
17     }
18 }

```

Установив соединение с сервером, клиент записывает одну строку, задающую количество байтов, которое сервер должен вернуть. Это отчасти похоже на HTTP: клиент отправляет небольшой запрос, а сервер в ответ отправляет требуемую информацию (часто это файл HTML или изображение GIF). В случае HTTP сервер обычно закрывает соединение после отправки клиенту затребованных данных, хотя более новые версии используют *постоянные соединения* (*persistent connection*), оставляя соединения TCP открытыми для дополнительных клиентских запросов. В нашей функции `web_child` сервер допускает

дополнительные запросы от клиента, но, как мы видели в листинге 24.1, клиент посыпает серверу только по одному запросу на каждое соединение, а по получении ответа от сервера это соединение закрывается.

В строке 1 табл. 30.1 показаны результаты измерения времени, затраченного параллельным сервером. При сравнении со следующими строками этой таблицы видно, что параллельный сервер тратит больше процессорного времени, чем все другие типы серверов — то, что мы и ожидали при вызове функции `fork`.

ПРИМЕЧАНИЕ

Один из способов устройства сервера, который мы не рассматриваем в этой главе, — это сервер, инициируемый демоном `inetd` (см. раздел 13.5). С точки зрения управления процессами такой сервер подразумевает использование функций `fork` и `exec`, так что затраты времени центрального процессора будут еще больше, чем показанные в строке 1 для параллельного сервера.

30.6. Сервер TCP с предварительным порождением процессов без блокировки для вызова `accept`

В первом из рассматриваемых нами «усовершенствованных» серверов используется технология, называемая *предварительным созданием процессов* (*preforking*). Вместо того чтобы вызывать функцию `fork` каждый раз при поступлении очередного запроса от клиента, сервер создает при запуске некоторое количество дочерних процессов, и впоследствии они обслуживают клиентские запросы по мере установления соединений с клиентами. На рис. 30.1 показан сценарий, при котором родитель предварительно создал N дочерних процессов, и в настоящий момент имеется два соединения с клиентами.

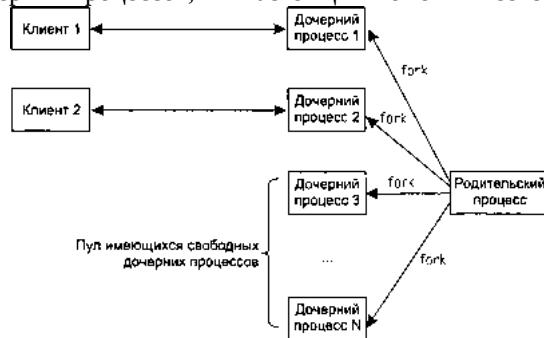


Рис. 30.1. Предварительное создание дочерних процессов сервером

Преимущество этой технологии заключается в том, что обслуживание нового клиента не требует вызова функции `fork` родительским процессом, тем самым стоимость этого обслуживания понижается. Недостатком же является необходимость угадать, сколько дочерних процессов нужно создать при запуске. Если в некоторый момент времени количество имеющихся дочерних процессов будет равно количеству обслуживаемых клиентов, то дополнительные клиентские запросы будут игнорироваться до того момента, когда освободится какой-либо дочерний процесс. Но, как сказано в разделе 4.5, клиентские запросы в такой ситуации игнорируются не полностью. Для каждого из этих дополнительных клиентов ядро выполнит трехэтапное рукопожатие (при этом общее количество соединений не может превышать значения аргумента `backlog` функции `listen`), и при вызове функции `accept` установленные соединения будут переданы серверу. При этом, однако, приложение-клиент может заметить некоторое ухудшение в скорости получения ответа, так как, хотя функция `connect` может быть выполнена сразу же, запрос может не поступать на обработку еще некоторое время.

За счет некоторого дополнительного усложнения кода всегда можно добиться того, что сервер справится со всеми клиентскими запросами. От родительского процесса требуется постоянно отслеживать количество свободных дочерних процессов, и если это количество падает ниже некоторого минимального предела, родитель должен вызвать функцию `fork` и создать недостающее количество дочерних процессов. Аналогично, если количество свободных дочерних процессов превосходит некоторую максимальную величину, некоторые из этих процессов могут быть завершены родителем, так как излишнее количество

свободных дочерних процессов тоже отрицательно влияет на производительность (об этом мы поговорим чуть позже).

Но прежде чем углубляться в детали, исследуем основную структуру этого типа сервера. В листинге 30.6 показана функция `main` для первой версии нашего сервера с предварительным порождением дочерних процессов.

Листинг 30.6. Функция `main` сервера с предварительным порождением дочерних процессов

```
//server/serv02.c
1 #include "unp.h"

2 static int nchildren;
3 static pid_t *pids;

4 int
5 main(int argc, char **argv)
6 {
7     int listenfd, i;
8     socklen_t addrlen;
9     void sig_int(int);
10    pid_t child_make(int, int, int);

11    if (argc == 3)
12        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
13    else if (argc == 4)
14        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
15    else
16        err_quit("usage: serv02 [ <host> ] <port#> <#children>");
17    nchildren = atoi(argv[argc - 1]);
18    pids = Calloc(nchildren, sizeof(pid_t));

19    for (i = 0; i < nchildren; i++)
20        pids[i] = child_make(i, listenfd, addrlen); /* возвращение родительского процесса */
21    Signal (SIGINT, sig_int);

22    for (;;)
23        pause(); /* дочерние процессы завершились */
24 }
```

11-18 Дополнительный аргумент командной строки указывает, сколько требуется создать дочерних процессов. В памяти выделяется место для размещения массива, в который записываются идентификаторы дочерних процессов, используемые функцией `main` при окончании работы программы для завершения этих процессов.

19-20 Каждый дочерний процесс создается функцией `child_make`, которую мы показываем в листинге 30.8.

Код обработчика сигнала `SIGINT`, представленный в листинге 30.7, отличается от кода, приведенного в листинге 30.3.

Листинг 30.7. Обработчик сигнала `SIGINT`

```
//server/serv02.c
25 void
26 sig_int(int signo)
27 {
28     int i;
29     void pr_cpu_time(void);

30     /* завершаем все дочерние процессы */
31     for (i = 0; i < nchildren; i++)
32         kill(pids[i], SIGTERM);
33     while (wait(NULL) > 0) /* ждем завершения всех дочерних процессов */
```

```

34    ;
35  if (errno != ECHILD)
36  err_sys("wait error");

37  pr_cpu_time();
38  exit(0);
39 }

30-34 Функция getrusage сообщает об использовании ресурсов всеми дочерними процессами, завершившими свое выполнение, поэтому мы должны завершить все дочерние процессы к моменту вызова функции pr_cpu_time. Для этого дочерним процессам посыпается сигнал SIGTERM, после чего мы вызываем функцию wait и ждем завершения выполнения дочерних процессов.

```

В листинге 30.8 показана функция child_make, вызываемая из функции main для порождения очередного дочернего процесса.

Листинг 30.8. Функция child_make: создание очередного дочернего процесса

```
//server/child02.c
1 #include "unp.h"
```

```

2 pid_t
3 child_make(int i, int listenfd, int addrlen)
4 {
5 pid_t pid;
6 void child_main(int, int, int);

7 if ( (pid = Fork()) > 0)
8 return (pid); /* родительский процесс */

9 child_main(i, listenfd, addrlen); /* никогда не завершается */
10 }
```

7-9 Функция fork создает очередной дочерний процесс и возвращает родителю идентификатор дочернего процесса. Дочерний процесс вызывает функцию child_main, показанную в листинге 30.9, которая представляет собой бесконечный цикл.

Листинг 30.9. Функция child_main: бесконечный цикл, выполняемый каждым дочерним процессом

```
//server/child02.c
11 void
12 child_main(int i, int listenfd, int addrlen)
13 {
14 int connfd;
15 void web_child(int);
16 socklen_t clilen;
17 struct sockaddr *cliaddr;

18 cliaddr = Malloc(addrlen);

19 printf("child %ld starting\n", (long)getpid());
20 for (;;) {
21 clilen = addrlen;
22 connfd = Accept(listenfd, cliaddr, &clilen);

23 web_child(connfd); /* обработка запроса */
24 Close(connfd);
25 }
26 }
```

20-25 Каждый дочерний процесс вызывает функцию accept, и когда она завершается, функция web_child (см. листинг 30.5) обрабатывает клиентский запрос. Дочерний процесс продолжает выполнение цикла, пока родительский процесс не завершит его.

Реализация 4.4BSD

Если вы никогда ранее не сталкивались с таким типом устройства сервера (несколько процессов, вызывающих функцию `accept` на одном и том же прослушиваемом сокете), вас, вероятно, удивляет, что это вообще может работать. Пожалуй, здесь уместен краткий экскурс, описывающий реализацию этого механизма в Беркли-ядрах (более подробную информацию вы найдете в [128]).

Родитель сначала создает прослушиваемый сокет, а затем — дочерние процессы. Напомним, что каждый раз при вызове функции `fork` происходит копирование всех дескрипторов в каждый дочерний процесс. На рис. 30.2 показана организация структур `proc` (по одной структуре на процесс), одна структура `file` для прослушиваемого дескриптора и одна структура `socket`.

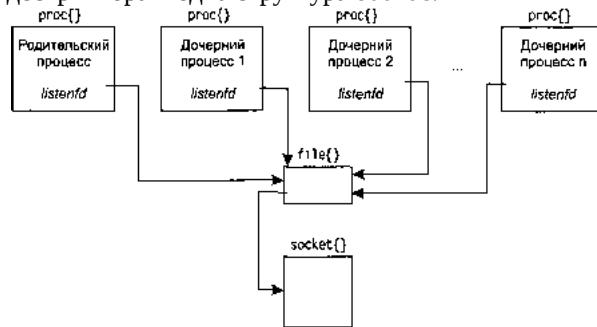


Рис. 30.2. Организация структур `proc`, `file` и `socket`

Дескрипторы — это просто индексы массива, содержащегося в структуре `proc`, который ссылается на структуру `file`. Одна из целей дублирования дескрипторов в дочерних процессах, осуществляемого функцией `fork`, заключается в том, чтобы данный дескриптор в дочернем процессе ссылался на ту же структуру `file`, на которую этот дескриптор ссылается в родительском процессе. Каждая структура `file` содержит счетчик ссылок, который начинается с единицы, когда открывается первый файл или сокет, и увеличивается на единицу при каждом вызове функции `fork` и при каждом дублировании дескриптора (с помощью функции `dup`). В нашем примере с N дочерними процессами счетчик ссылок в структуре `file` будет содержать значение $N+1$ (учитывая родительский процесс, у которого по-прежнему открыт прослушиваемый дескриптор, хотя родительский процесс никогда не вызывает функцию `accept`).

При запуске программы создается N дочерних процессов, каждый из которых может вызывать функцию `accept`, и все они переводятся родительским процессом в состояние ожидания [128, с. 458]. Когда от клиента прибывает первый запрос на соединение, все N дочерних процессов «просыпаются», так как все они были переведены в состояние ожидания по одному и тому же «каналу ожидания» — полю `so_timeo` структуры `socket`, как совместно использующие один и тот же прослушиваемый дескриптор, указывающий на одну и ту же структуру `socket`. Хотя «проснулись» все N дочерних процессов, только один из них будет связан с клиентом. Остальные $N - 1$ снова перейдут в состояние ожидания, так как длина очереди клиентских запросов снова станет равна нулю, после того как первый из дочерних процессов займется обработкой поступившего запроса.

Такая ситуация иногда называется *thundering herd* — более или менее дословный перевод будет звучать как «общая побудка», так как все N процессов должны быть выведены из спящего состояния, хотя нужен всего один процесс, и остальные потом снова «засыпают». Тем не менее этот код работает, хотя и имеет побочный эффект — необходимость «будить» слишком много дочерних процессов каждый раз, когда требуется принять (`accept`) очередное клиентское соединение. В следующем разделе мы исследуем, как это влияет на производительность в целом.

Эффект наличия слишком большого количества дочерних процессов

В табл. 30.1 (строка 2) указано время (1,8 с), затрачиваемое центральным процессором в случае наличия 15 дочерних процессов, обслуживающих не более 10 клиентов. Мы можем оценить эффект «общей побудки», увеличивая количество дочерних процессов и оставляя то же максимальное значение количества обслуживаемых клиентов (10). Мы не показываем результаты, получаемые при увеличении количества дочерних потоков, потому что они не настолько интересны. Поскольку любое количество дочерних потоков свыше 10 может считаться избыточным, проблема «общей побудки» усугубляется, а затрачиваемое на управление процессами время увеличивается.

ПРИМЕЧАНИЕ

Некоторые ядра Unix снабжены функцией, которая выводит из состояния ожидания только один процесс для обработки одного клиентского запроса [107]. Чаще всего она называется `wakeup_one`.

Распределение клиентских соединений между дочерними процессами

Следующей темой обсуждения является распределение клиентских соединений между свободными дочерними процессами, блокированными в вызове функции `accept`. Для получения этой информации мы модифицируем функцию `main`, размещая в совместно используемой области памяти массив счетчиков, которые представляют собой длинные целые числа (один счетчик на каждый дочерний процесс). Это делается следующим образом:

```
long *cptr, *meter(int); /* для подсчета количества клиентов на один
дочерний процесс */
cptr = meter(ncchildren); /* перед порождением дочернего процесса */
```

В листинге 30.10 показана функция `meter`.

Листинг 30.10. Функция `meter`, которая размещает массив в совместно используемой памяти

```
//server/meter.c
1 #include "unp.h"
2 #include <sys/mman.h>
```

```
3 /* Размещаем массив "ncchildren" длинных целых чисел
4  * в совместно используемой области памяти.
5  * Эти числа используются как счетчики количества
6  * клиентов, обслуженных данным дочерним процессом,
7  * см. с. 467-470 книги [110]*/
8 */
```

```
8 long*
9 meter(int ncchildren)
10 {
11     int fd;
12     long *ptr;

13 #ifdef MAP_ANON
14     ptr = Mmap(0, ncchildren * sizeof(long), PROT_READ | PROT_WRITE,
15                MAP_ANON | MAP_SHARED, -1, 0);
16 #else
17     fd = Open("/dev/zero", O_RDWR, 0);

18     ptr = Mmap(0, ncchildren * sizeof(long), PROT_READ | PROT_WRITE,
19                MAP_SHARED, fd, 0);
20     Close(fd);
21 #endif

22     return (ptr);
23 }
```

Мы используем неименованное отображение в память, если оно поддерживается (например, в 4.4BSD), или отображение файла `/dev/zero` (например, SVR4). Поскольку массив создается функцией `mmap` до того, как родительский процесс порождает дочерние, этот массив затем используется совместно родительским и всеми дочерними процессами, созданными функцией `fork`.

Затем мы модифицируем нашу функцию `child_main` (см. листинг 30.9) таким образом, чтобы каждый дочерний процесс увеличивал значение соответствующего счетчика на единицу при завершении функции

accept, а после завершения выполнения всех дочерних процессов обработчик сигнала SIGINT выводил бы упомянутый массив счетчиков.

В табл. 30.2 показано распределение нагрузки по дочерним процессам. Когда свободные дочерние процессы блокированы вызовом функции accept, имеющейся в ядре алгоритм планирования равномерно распределяет нагрузку, так что в результате все дочерние процессы обслуживают примерно одинаковое количество клиентских запросов.

Коллизии при вызове функции select

Рассматривая данный пример в 4.4BSD, мы можем исследовать еще одну проблему, которая встречается довольно редко и поэтому часто остается непонятой до конца. В разделе 16.13 [128] говорится о *коллизиях (collisions)*, возникающих при вызове функции select несколькими процессами на одном и том же дескрипторе, и о том, каким образом ядро решает эту проблему. Суть проблемы в том, что в структуре socket предусмотрено место только для одного идентификатора процесса, который выводится из состояния ожидания по готовности дескриптора. Если же имеется несколько процессов, ожидающих, когда будет готов данный дескриптор, то ядро должно вывести из состояния ожидания все процессы, блокированные в вызове функции select, так как ядро не знает, какие именно процессы ожидают готовности данного дескриптора.

Коллизии при вызове функции select в нашем примере можно форсировать, предваряя вызов функции accept из листинга 30.9 вызовом функции select в ожидании готовности к чтению на прослушиваемом сокете. Дочерние процессы будут теперь блокированы в вызове функции select, а не в вызове функции accept. В листинге 30.11 показана изменяемая часть функции child_main, при этом измененные по отношению к листингу 30.9 строки отмечены знаками +.

Листинг 30.11. Модификация листинга 30.9: блокирование в вызове select вместо блокирования в вызове accept

```
printf("child %ld starting\n", (long)getpid());
+ FD_ZERO(&rset);
for (;;) {
+ FD_SET(listenfd, &rset);
+ Select(listenfd+1, &rset, NULL, NULL, NULL);
+ if (FD_ISSET(listenfd, &rset) == 0)
+ err_quit("listenfd readable");
+
clilen = addrlen;
connfd = Accept(listenfd, cliaddr, &clilen);

web_child(connfd); /* обработка запроса */
Close(connfd);
}
```

Если, проделав это изменение, мы проверим значение счетчика ядра BSD/OS nselcoll, мы увидим, что в первом случае при запуске сервера произошло 1814 коллизий, а во втором случае — 2045. Так как при каждом запуске сервера два клиента создают в сумме 5000 соединений, приведенные выше значения указывают, что примерно в 35-40% случаев вызовы функции select приводят к коллизиям.

Если сравнить значения времени, затраченного центральным процессором в этом примере, то получится, что при добавлении вызова функции select это значение увеличивается с 1,8 до 2,9 с. Частично это объясняется, вероятно, добавлением системного вызова (так как теперь мы вызываем не только accept, но еще и select), а частично — накладными расходами, связанными с коллизиями.

Из этого примера следует вывод, что когда несколько процессов блокируются на одном и том же дескрипторе, лучше, чтобы эта блокировка была связана с функцией accept, а не с функцией select.

30.7. Сервер TCP с предварительным порождением процессов и защитой вызова accept блокировкой файла

Описанная выше реализация, позволяющая нескольким процессам вызывать функцию accept на одном и том же прослушиваемом дескрипторе, возможна только для систем 4.4BSD, в которых функция

accept реализована внутри ядра. Ядра системы SVR4, в которых accept реализована как библиотечная функция, не допускают этого. В самом деле, если мы запустим сервер из предыдущего раздела, в котором имеется несколько дочерних процессов, в Solaris 2.5 (система SVR4), то вскоре после того, как клиенты начнут соединяться с сервером, вызов функции accept в одном из дочерних процессов вызовет ошибку EPROTO, что свидетельствует об ошибке протокола.

ПРИМЕЧАНИЕ

Причины возникновения этой проблемы с библиотечной версией функции accept в SVR4 связаны с реализацией потоков STREAMS и тем фактом, что библиотечная функция accept не является атомарной операцией. В Solaris 2.6 эта проблема решена, но в большинстве реализаций SVR4 она остается.

Решением этой проблемы является защита вызова функции accept при помощи блокировки, так что в данный момент времени только один процесс может быть блокирован в вызове этой функции. Другие процессы также будут блокированы, так как они будут стремиться установить блокировку для вызова функции accept.

Существует несколько способов реализации защиты вызова функции accept, о которых рассказывается во втором томе^[2] данной серии. В этом разделе мы используем блокировку файла функцией fcntl согласно стандарту POSIX.

Единственным изменением в функции main (см. листинг 30.6) будет добавление вызова функции my_lock_init перед началом цикла, в котором создаются дочерние процессы:

```
+ my_lock_init("/tmp/lock.XXXXXX"); /* один файл для всех дочерних
                                         процессов */
for (i = 0; i < nchildren; i++)
    pids[i] = child_make(i, listenfd, addrlen); /* возвращение
                                                 родительского процесса */
```

Функция child_make остается такой же, как в листинге 30.8. Единственным изменением функции child_main (см. листинг 30.9) является блокирование перед вызовом функции accept и снятие блокировки после завершения этой функции:

```
for (;;) {
    clilen = addrlen;
+ my_lock_wait();
    connfd = Accept(listenfd, cliaddr, &clilen);
+ my_lock_release();

    web_child(connfd); /* обработка запроса */
    Close(connfd);
}
```

В листинге 30.12 показана наша функция my_lock_init, в которой используется блокировка файла согласно стандарту POSIX.

Листинг 30.12. Функция my_lock_init: блокировка файла

```
//server/lock_fcntl.c
1 #include "unp.h"

2 static struct flock lock_it, unlock_it;
3 static int lock_fd = -1;
4 /* fcntl() не выполнится, если не будет вызвана функция my_lock_init() */

5 void
6 my_lock_init(char *pathname)
7 {
8     char lock_file[1024];

9     /* копируем строку вызывающего процесса на случай, если это константа */
```

```

10 strcpy(lock_file, pathname, sizeof(lock_file));
11 lock_fd = Mkstemp(lock_file);

12 Unlink(lock_file); /* но lock_fd остается открытым */

13 lock_it.l_type = F_WRLCK;
14 lock_it.l_whence = SEEK_SET;
15 lock_it.l_start = 0;
16 lock_it.l_len = 0;

17 unlock_it.l_type = F_UNLCK;
18 unlock_it.l_whence = SEEK_SET;
19 unlock_it.l_start = 0;
20 unlock_it.l_len = 0;
21 }

```

9-12 Вызывающий процесс задает шаблон для имени файла в качестве аргумента функции `my_lock_init`, и функция `mkstemp` на основе этого шаблона создает уникальное имя файла. Затем создается файл с этим именем и сразу же вызывается функция `unlink`, в результате чего имя файла удаляется из каталога. Если в программе впоследствии произойдет сбой, то файл исчезнет безвозвратно. Но пока он остается открытым в одном или нескольких процессах (иными словами, пока счетчик ссылок для этого файла больше нуля), сам файл не будет удален. (Отметим, что между удалением имени файла из каталога и закрытием открытого файла существует фундаментальная разница.)

13-20 Инициализируются две структуры `flock`: одна для блокирования файла, другая для снятия блокировки. Блокируемый диапазон начинается с нуля (`l_whence =SEEK_SET, l_start=0`). Значение `l_len` равно нулю, то есть блокирован весь файл. В этот файл ничего не записывается (его длина всегда равна нулю), но такой тип блокировки в любом случае будет правильно обрабатываться ядром.

ПРИМЕЧАНИЕ

Сначала автор инициализировал эти структуры при объявлении:

```

static struct flock lock_it = { F_WRLCK, 0, 0, 0, 0 };
static struct flock unlock_it = { F_UNLCK, 0, 0, 0, 0 };

```

но тут возникли две проблемы: у нас нет гарантии, что константа `SEEK_SET` равна нулю, но, что более важно, стандарт POSIX не регламентирует порядок расположения полей этой структуры. POSIX гарантирует только то, что требуемые поля присутствуют в структуре. POSIX не гарантирует какого-либо порядка следования полей структуры, а также допускает наличие в ней полей, не относящихся к стандарту POSIX. Поэтому когда требуется инициализировать эту структуру (если только не нужно инициализировать все поля нулями), это приходится делать через фактический код C, а не с помощью инициализатора при объявлении структуры.

Исключением из этого правила является ситуация, когда инициализатор структуры обеспечивается реализацией. Например, при инициализации взаимного исключения в POSIX в главе 26 мы писали:

```
pthread_mutex_t mlock = PTHREAD_MUTEX_INITIALIZER;
```

Тип данных `pthread_mutex_t` — это некая структура, но инициализатор предоставляетя реализацией и может быть различным для разных реализаций.

В листинге 30.13 показаны две функции, которые устанавливают и снимают блокировку с файла. Они представляют собой вызовы функции `fcntl`, использующие структуры, инициализированные в листинге 30.12.

Листинг 30.13. Функции `my_lock_wait` (установление блокировки файла) и `my_lock_release` (снятие блокировки файла)

```

//server/lock_fcntl.c
23 void
24 my_lock_wait()
25 {

```

```
26 int rc;
27 while ((rc = fcntl(lock_ld, F_SETLKW, &lock_it)) < 0 {
28     if (errno == EINTR)
29         continue;
30     else
31         errsys("fcntl error for my_lock_wait");
32 }
33 }

34 void
35 my_lock_release()
36 {
37     if (fcntl(lock_fd, F_SETLKW, &unlock_it)) < 0)
38     errsys("fcntl error for my_lock_release");
39 }
```

Новая версия нашего сервера с предварительным порождением процессов работает теперь под SVR4, гарантируя, что в данный момент времени только один дочерний процесс блокирован в вызове функции accept. Сравнивая строки 2 и 3 в табл. 30.1 (результаты для серверов Digital Unix и BSD/OS), мы видим, что такой тип блокировки увеличивает время, затрачиваемое центральным процессором на узле сервера.

ПРИМЕЧАНИЕ

Веб-сервер Apache (<http://www.apache.org>) использует технологию предварительного порождения процессов, причем если позволяет реализация, все дочерние процессы блокируются в вызове функции accept, иначе используется блокировка файла для защиты вызова accept.

Эффект наличия слишком большого количества дочерних процессов

Мы можем проверить, возникает ли в данной версии сервера эффект «общей побудки», рассмотренный в предыдущем разделе. Как и раньше, время работы ухудшается пропорционально числу избыточных дочерних процессов.

Распределение клиентских соединений между дочерними процессами

Используя функцию, показанную в листинге 30.10, мы можем исследовать распределение клиентских запросов между свободными дочерними процессами. Результат показан в табл. 30.2. Операционная система распределяет блокировки файла равномерно между ожидающими процессами, и такое поведение характерно для нескольких протестированных нами систем.

30.8. Сервер TCP с предварительным порождением процессов и защитой вызова accept при помощи взаимного исключения

Как мы уже говорили, существует несколько способов синхронизации процессов путем блокирования. Блокировка файла по стандарту POSIX, рассмотренная в предыдущем разделе, переносится на все POSIX-совместимые системы, но она подразумевает некоторые операции с файловой системой, которые могут потребовать времени. В этом разделе мы будем использовать блокировку при помощи взаимного исключения, обладающую тем преимуществом, что ее можно применять для синхронизации не только потоков внутри одного процесса, но и потоков, относящихся к различным процессам.

Функция main остается такой же, как и в предыдущем разделе, то же относится к функциям child_make и child_main. Меняются только три функции, осуществляющие блокировку. Чтобы использовать взаимное исключение между различными процессами, во-первых, требуется хранить это взаимное исключение в разделяемой процессами области памяти, а во-вторых, библиотека потоков должна получить указание о том, что взаимное исключение совместно используется различными процессами.

ПРИМЕЧАНИЕ

Требуется также, чтобы библиотека потоков поддерживала атрибут PTHREAD_PROCESS_SHARED.

Существует несколько способов разделения памяти между различными процессами, что мы подробно описываем во втором томе^[2] данной серии. В этом примере мы используем функцию `mmap` с устройством `/dev/zero`, которое работает с ядрами Solaris и другими ядрами SVR4. В листинге 30.14 показана только функция `my_lock_init`.

Листинг 30.14. Функция `my_lock_init`: использование взаимного исключения потоками, относящимися к различным процессам (технология Pthread)

```
//server/lock_pthread.c
1 #include "unpthread.h"

2 #include <sys/mman.h>
3 static pthread_mutex_t *mptr; /* фактически взаимное исключение будет
   в совместно используемой памяти */

4 void
5 my_lock_init(char *pathname)
6 {
7     int fd;
8     pthread_mutexattr_t mattr;

9     fd = Open("/dev/zero", O_RDWR, 0);

10    mptr = Mmap(0, sizeof(pthread_mutex_t), PROT_READ | PROT_WRITE,
11        MAP_SHARED, fd, 0);
12    Close(fd);

13    Pthread_mutexattr_init(&mattr);
14    Pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
15    Pthread_mutex_init(mptr, &mattr);
16 }
```

9-12 Мы открываем (`open`) файл `/dev/zero`, а затем вызываем `mmap`. Количество байтов (второй аргумент этой функции) — это размер переменной `pthread_mutex_t`. Затем дескриптор закрывается, но для нас это не имеет значения, так как файл уже отображен в память.

13-15 В приведенных ранее примерах взаимных исключений Pthread мы инициализировали глобальные статические взаимные исключения, используя константу `PTHREAD_MUTEX_INITIALIZER` (см., например, листинг 26.12). Но располагая взаимное исключение в совместно используемой памяти, мы должны вызвать некоторые библиотечные функции Pthreads, чтобы сообщить библиотеке о наличии семафора в совместно используемой памяти и о том, что он будет применяться для синхронизации потоков, относящихся к различным процессам. Мы должны инициализировать структуру `pthread_mutexattr_t` задаваемыми по умолчанию атрибутами взаимного исключения, а затем установить значение атрибута `PTHREAD_PROCESS_SHARED`. (По умолчанию значением этого атрибута должно быть `PTHREAD_PROCESS_PRIVATE`, что подразумевает использование взаимного исключения только в пределах одного процесса.) Затем вызов `pthread_mutex_init` инициализирует взаимное исключение указанными атрибутами.

В листинге 30.15 показаны только функции `my_lock_wait` и `my_lock_release`. Они содержат вызовы функций Pthreads, предназначенных для блокирования и разблокирования взаимного исключения.

Листинг 30.15. Функции `my_lock_wait` и `my_lock_release`: использование блокировок Pthread

```
//server/lock_pthread.c
17 void
18 my_lock_wait()
19 {
```

```

20 Pthread_mutex_lock(mptra),
21 }

22 void
23 my_lock_release()
24 {
25 Pthread_mutex_unlock(mptra);
26 }

```

Сравнивая строки 3 и 4 табл. 30.1, можно заметить, что версия, использующая синхронизацию процессов при помощи взаимного исключения, характеризуется более высоким быстродействием, чем версия с блокировкой файла.

30.9. Сервер TCP с предварительным порождением процессов: передача дескриптора

Последней модификацией нашего сервера с предварительным порождением процессов является версия, в которой только родительский процесс вызывает функцию `accept`, а затем «передает» присоединенный сокет какому-либо одному дочернему процессу. Это помогает обойти необходимость защиты вызова `accept`, но требует некоторого способа передачи дескриптора между родительским и дочерним процессами. Эта техника также несколько усложняет код, поскольку родительскому процессу приходится отслеживать, какие из дочерних процессов заняты, а какие свободны, чтобы передавать дескриптор только свободным дочерним процессам.

В предыдущих примерах сервера с предварительным порождением процессов родительскому процессу не приходилось беспокоиться о том, какой дочерний процесс принимает соединение с клиентом. Этим занималась операционная система, организуя вызов функции `accept` одним из свободных дочерних процессов или блокировку файла или взаимного исключения. Из первых двух столбцов табл. 30.2 видно, что операционная система, в которой мы проводим измерения, осуществляет равномерную циклическую загрузку свободных процессов клиентскими соединениями.

В данном примере для каждого дочернего процесса нам нужна некая структура, содержащая информацию о нем. Заголовочный файл `child.h`, в котором определяется структура `Child`, показан в листинге 30.16.

Листинг 30.16. Структура `Child`

```

//server/child.h
1 typedef struct {
2     pid_t child_pid; /* ID процесса */
3     int child_pipefd; /* программный (неименованный) канал между
4                         родительским и дочерним процессами */
5     int child_status; /* 0 = готово */
6     long child_count; /* количество обрабатываемых соединений */
6 } Child;

```

7 `Child *cptr; /* массив структур Child */`

Мы записываем идентификатор дочернего процесса, дескриптор программного канала (`pipe`) родительского процесса, связанного с дочерним, статус дочернего процесса и количество обрабатываемых дочерним процессом клиентских соединений. Это количество выводится обработчиком сигнала `SIGINT` и позволяет нам отслеживать распределение клиентских запросов между дочерними процессами.

Рассмотрим сначала функцию `child_make`, которая приведена в листинге 30.17. Мы создаем канал и доменный сокет Unix (см. главу 14) перед вызовом функции `fork`. После того, как создан дочерний процесс, родительский процесс закрывает один дескриптор (`sockfd[1]`), а дочерний процесс закрывает другой дескриптор (`sockfd[0]`). Более того, дочерний процесс подключает свой дескриптор канала (`sockfd[1]`) к стандартному потоку сообщений об ошибках, так что каждый дочерний процесс просто использует это устройство для связи с родительским процессом. Этот механизм проиллюстрирован схемой, приведенной на рис. 30.3.

Листинг 30.17. Функция `child_make`: передача дескриптора в сервере с предварительным порождением дочерних процессов

```

//server/child05.c
1 #include "unp.h"

```

```

2 #include "child.h"

3 pid_t
4 child_make(int i, int listenfd, int addrlen)
5 {
6     int sockfd[2];
7     pid_t pid;
8     void child_main(int, int, int);

9     Socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd);

10    if ((pid = Fork()) > 0) {
11        Close(sockfd[1]);
12        cptr[i].child_pid = pid;
13        cptr[i].child_pipefd = sockfd[0];
14        cptr[i].child_status = 0;
15        return (pid); /* родительский процесс */
16    }
17    Dup2(sockfd[1], STDERR_FILENO); /* канал от дочернего процесса к
18                                     родительскому */
19    Close(sockfd[0]);
20    Close(listenfd); /* дочернему процессу не требуется, чтобы
21                      он был открыт */
22    child_main(i, listenfd, addrlen); /* никогда не завершается */
}

```

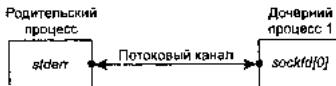


Рис. 30.3. Канал после того, как дочерний и родительский процесс закрыли один конец

После создания всех дочерних процессов мы получаем схему, показанную на рис. 30.4. Мы закрываем прослушиваемый сокет в каждом дочернем процессе, поскольку только родительский процесс вызывает функцию accept. Мы показываем на рисунке, что родительский процесс должен обрабатывать прослушиваемый сокет, а также все доменные сокеты. Как можно догадаться, родительский процесс использует функцию select для мультиплексирования всех дескрипторов.

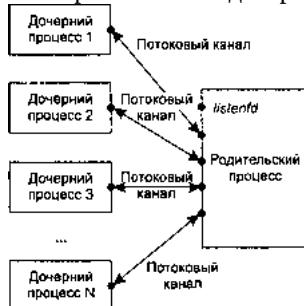


Рис. 30.4. Каналы после создания всех дочерних процессов

В листинге 30.18 показана функция main. В отличие от предыдущих версий этой функции, в данном случае в памяти размещаются все наборы дескрипторов и в каждом наборе включены все биты, соответствующие прослушиваемому сокету и каналу каждого дочернего процесса. Вычисляется также максимальное значение дескриптора и выделяется память для массива структур Child. Основной цикл запускается при вызове функции select.

Листинг 30.18. Функция main, использующая передачу дескриптора

```

//server/serv05.c
1 #include "unp.h"
2 #include "child.h"

```

```

3 static int nchildren;

4 int
5 main(int argc, char **argv)
6 {
7     int listenfd, i, navail, maxfd, nsel, connfd, rc;
8     void sig_int(int);
9     pid_t child_make(int, int, int);
10    ssize_t n;
11    fd_set rset, masterset;
12    socklen_t addrlen, clilen;
13    struct sockaddr *cliaddr;

14    if (argc == 3)
15        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
16    else if (argc == 4)
17        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
18    else
19        err_quit("usage: serv05 [ <host> ] <port#> <#children>");

20    FD_ZERO(&masterset);
21    FD_SET(listenfd, &masterset);
22    maxfd = listenfd;
23    cliaddr = Malloc(addrlen);

24    nchildren = atoi(argv[argc - 1]);
25    navail = nchildren;
26    cptr = Calloc(nchildren, sizeof(Child));

27    /* предварительное создание дочерних процессов */
28    for (i = 0; i < nchildren; i++) {
29        child_make(i, listenfd, addrlen); /* родительский процесс
                                         завершается */
30        FD_SET(cptr[i].child_pipefd, &masterset);
31        maxfd = max(maxfd, cptr[i].child_pipefd);
32    }

33    Signal(SIGINT, sig_int);

34    for (;;) {
35        rset = masterset;
36        if (navail <= 0)
37            FD_CLR(listenfd, &rset); /* выключаем, если нет свободных
                                         дочерних процессов */
38        nsel = Select(maxfd + 1, &rset, NULL, NULL, NULL);

39        /* проверка новых соединений */
40        if (FD_ISSET(listenfd, &rset)) {
41            clilen = addrlen;
42            connfd = Accept(listenfd, cliaddr, &clilen);

43            for (i = 0; i < nchildren; i++)
44                if (cptr[i].child_status == 0)
45                    break; /* свободный */

46            if (i == nchildren)

```

```

47     err_quit("no available children");
48     cptr[i].child_status = 1; /* отмечаем этот дочерний процесс как
49     занятый */
50     cptr[i].child_count++;
51     navail--;
52
53     n = Write_fd(cptr[i].child_pipefd, 1, connfd);
54     Close(connfd);
55     if (--nse1 == 0)
56         continue; /* с результатами select() закончено */
57     }
58     /* поиск освободившихся дочерних процессов */
59     for (i = 0; i < nchildren; i++) {
60         if (FD_ISSET(cptr[i].child_pipefd, &rset)) {
61             if ((n = Read(cptr[i].child_pipefd, &rc, 1)) == 0)
62                 err_quit("child %d terminated unexpectedly", i);
63             cptr[i].child_status = 0;
64             navail++;
65             if (--nse1 == 0)
66                 break; /* с результатами select() закончено */
67         }
68     }

```

Отключение прослушиваемого сокета в случае отсутствия свободных дочерних процессов

36-37 Счетчик `navail` отслеживает количество свободных дочерних процессов. Если его значение становится равным нулю, прослушиваемый сокет в наборе дескрипторов функции `select` выключается. Это предотвращает прием нового соединения в тот момент, когда нет ни одного свободного дочернего процесса. Ядро по-прежнему устанавливает эти соединения в очередь, пока их количество не превысит значения аргумента `backlog` функции `listen`, заданного для прослушиваемого сокета, но мы не хотим их принимать, пока у нас не появится свободный дочерний процесс, готовый обрабатывать клиентский запрос.

Прием нового соединения

39-55 Если прослушиваемый сокет готов для считывания, можно принимать (`accept`) новое соединение. Мы находим первый свободный дочерний процесс и передаем ему присоединенный сокет с помощью функции `write_fd`, приведенной в листинге 15.11. Вместе с дескриптором мы передаем 1 байт, но получатель не интересуется содержимым этого байта. Родитель закрывает присоединенный сокет.

Мы всегда начинаем поиск свободного дочернего процесса с первого элемента массива структур `Child`. Это означает, что новое соединение для обработки поступившего клиентского запроса всегда получает первый элемент этого массива. Этот факт мы проверим при рассмотрении табл. 30.2 и значения счетчика `child_count` после завершения работы сервера. Если мы не хотим оказывать такое предпочтение первому элементу массива, мы можем запомнить, какой дочерний процесс получил последнее клиентское соединение, и каждый раз начинать поиск свободного дочернего процесса со следующего за ним, а по достижении конца массива переходить снова к первому элементу. В этом нет особого смысла (на самом деле все равно, какой дочерний процесс обрабатывает очередное соединение, если имеется несколько свободных дочерних процессов), если только планировочный алгоритм операционной системы не накладывает санкций на процессы, которые требуют относительно больших временных затрат центрального процессора. Более равномерное распределение загрузки между всеми дочерними процессами приведет к выравниванию времен, затраченных на их выполнение.

Обработка вновь освободившихся дочерних процессов

56-66 Когда дочерний процесс заканчивает обработку клиентского запроса, наша функция `child_main` записывает один байт в канал для родительского процесса. Тем самым родительский конец канала становится доступным для чтения. Упомянутый байт считывается (но его значение при этом игнорируется), а дочерний процесс помечается как свободный. Если же дочерний процесс завершит свое выполнение неожиданно, его конец канала будет закрыт, а операция чтения (`read`) возвратит нулевое значение. Это значение перехватывается и дочерний процесс завершается, но более удачным решением было бы записать ошибку и создать новый дочерний процесс для замены завершенного.

Функция `child_main` показана в листинге 30.19.

Листинг 30.19. Функция `child_main`: передача дескриптора в сервере с предварительным порождением дочерних процессов

```
//server/child05.c
23 void
24 child_main(int i, int listenfd, int addrlen)
25 {
26     char c;
27     int connfd;
28     ssize_t n;
29     void web_child(int);

30     printf("child %d starting\n", (long)getpid());
31     for (;;) {
32         if ((n = Read_fd(STDERR_FILENO, &c, 1, &connfd)) == 0)
33             err_quit("read_fd returned 0");
34         if (connfd < 0)
35             err_quit("no descriptor from read_fd");

36         web_child(connfd); /* обработка запроса */
37         Close(connfd);

38         Write(STDERR_FILENO, "", 1); /* сообщаем родительскому процессу
                                         о том, что дочерний освободился */
39     }
40 }
```

Ожидание дескриптора от родительского процесса

32-33 Эта функция отличается от аналогичных функций из двух предыдущих разделов, так как дочерний процесс не вызывает более функцию `accept`. Вместо этого дочерний процесс блокируется в вызове функции `read_fd`, ожидая, когда родительский процесс передаст ему дескриптор присоединенного сокета.

Сообщение родительскому процессу о готовности дочернего к приему новых запросов

38 Закончив обработку очередного клиентского запроса, мы записываем (`write`) 1 байт в канал, чтобы сообщить, что данный дочерний процесс освободился.

В табл. 30.1 при сравнении строк 4 и 5 мы видим, что данный сервер медленнее, чем версия, рассмотренная нами в предыдущем разделе, которая использовала блокировку потоками взаимного исключения. Передача дескриптора по каналу от родительского процесса к дочернему и запись одного байта в канал для сообщения родительскому процессу о завершении обработки клиентского запроса занимает больше времени, чем блокирование и разблокирование взаимного исключения или файла.

В табл. 30.2 показаны значения счетчиков `child_count` из структуры `Child`, которые выводятся обработчиком сигнала `SIGINT` по завершении работы сервера. Дочерние процессы, расположенные ближе к началу массива, обрабатывают большее количество клиентских запросов, как было указано при обсуждении листинга 30.18.

30.10. Параллельный сервер TCP: один поток для каждого клиента

Предыдущие пять разделов были посвящены рассмотрению серверов, в которых для обработки клиентских запросов используются дочерние процессы, либо заранее порождаемые с помощью функции `fork`, либо требующие вызова этой функции для каждого вновь поступившего клиентского запроса. Если же сервер поддерживает потоки, мы можем применить потоки вместо дочерних процессов.

Наша первая версия сервера с использованием потоков показана в листинге 30.20. Это модификация листинга 30.2: в ней создается один поток для каждого клиента вместо одного дочернего процесса для каждого клиента. Эта версия во многом похожа на сервер, представленный в листинге 26.2.

Листинг 30.20. Функция `main` для сервера TCP, использующего потоки

```
//server/serv06.c
1 #include "unpthread.h"

2 int
3 main(int argc, char **argv)
4 {
5     int listenfd, connfd;
6     void sig_int(int);
7     void *doit(void*);
8     pthread_t tid;
9     socklen_t clilen, addrlen;
10    struct sockaddr *cliaddr;

11    if (argc == 2)
12        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
13    else if (argc == 3)
14        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
15    else
16        err_quit("usage: serv06 [ <host> ] <port#>");
17    cliaddr = Malloc(addrlen);
18    Signal (SIGINT, sig_int);
19    for (;;) {
20        clilen = addrlen;
21        connfd = Accept(listenfd, cliaddr, &clilen);

22        Pthread_create(&tid, NULL, &doit, (void*)connfd);
23    }
24 }

25 void*
26 doit(void *arg)
27 {
28     void web_child(int);

29     Pthread_detach(pthread_self());
30     web_child((int)arg);
31     Close((int)arg);
32     return (NULL);
33 }
```

Цикл основного потока

19-23 Основной поток блокируется в вызове функции accept, и каждый раз, когда прибывает новое клиентское соединение, функцией pthread_create создается новый поток. Функция, выполняемая новым потоком, — это функция doit, а ее аргументом является присоединенный сокет.

Функция прочих потоков

25-33 Функция doit выполняется как отсоединенный (detached) поток, потому что основному потоку не требуется ждать ее завершения. Doit вызывает функцию web_child (см. листинг 30.5). Когда эта функция возвращает управление, присоединенный сокет закрывается.

Из табл. 30.1 мы видим, что эта простая версия с использованием потоков является более быстродействующей, чем даже самая быстрая из версий с предварительным порождением процессов. Кроме того, эта версия, в которой каждый клиент обслуживается одним потоком, во много раз быстрее версии, в которой каждый клиент обслуживается специально созданным для него дочерним процессом (первая строка табл. 30.1).

ПРИМЕЧАНИЕ

В разделе 26.5 мы упомянули о трех вариантах преобразования функции, которая не является безопасной в многопоточной среде, в функцию, обеспечивающую требуемую безопасность. Функция web_child вызывает функцию readline, и версия, показанная в листинге 3.12, не является безопасной в многопоточной среде. На примере, приведенном в листинге 30.20, были испробованы вторая и третья альтернативы из раздела 26.5. Увеличение быстродействия при переходе от альтернативы 3 к альтернативе 2 составило менее одного процента, вероятно, потому, что функция readline использовалась лишь для считывания значения счетчика (5 символов) от клиента. Поэтому в данной главе для простоты мы использовали более медленную версию из листинга 3.11 для сервера с предварительным порождением потоков.

30.11. Сервер TCP с предварительным порождением потоков, каждый из которых вызывает accept

Ранее в этой главе мы обнаружили, что версии, в которых заранее создается пул дочерних процессов, работают быстрее, чем те, в которых для каждого клиентского запроса приходится вызывать функцию fork. Для систем, поддерживающих потоки, логично предположить, что имеется та же закономерность: быстрее сразу создать пул потоков при запуске сервера, чем создавать по одному потоку по мере поступления запросов от клиентов. Основная идея такого сервера заключается в том, чтобы создать пул потоков, каждый из которых вызывает затем функцию accept. Вместо того чтобы блокировать потоки в вызове accept, мы используем взаимное исключение, как в разделе 30.8. Это позволяет вызывать функцию accept только одному потоку в каждый момент времени. Использовать блокировку файла для защиты accept в таком случае бессмысленно, так как при наличии нескольких потоков внутри данного процесса можно использовать взаимное исключение.

В листинге 30.21 показан заголовочный файл pthread07.h, определяющий структуру Thread, содержащую определенную информацию о каждом потоке.

Листинг 30.21. Заголовочный файл pthread07.h

```
//server/pthread07.h
1 typedef struct {
2     pthread_t thread_tid; /* идентификатор потока */
3     long thread_count; /* количество обработанных запросов */
4 } Thread;
5 Thread *tptr; /* массив структур Thread */

6 int listenfd, nthreads;
```

```

7 socklen_t addrlen;
8 pthread_mutex_t mlock;
Мы также объявляем несколько глобальных переменных, таких как дескриптор прослушиваемого
сокета и взаимное исключение, которые должны совместно использоваться всеми потоками.
В листинге 30.22 показана функция main.
Листинг 30.22. Функция main для сервера TCP с предварительным порождением потоков
//server/serv07.c
1 #include "unpthread.h"
2 #include "pthread07.h"

3 pthread_mutex_t mlock = PTHREAD_MUTEX_INITIALIZER;

4 int
5 main(int argc, char **argv)
6 {
7 int i;
8 void sig_int(int), thread_make(int);

9 if (argc == 3)
10 listenfd = Tcp_listen(NULL, argv[1], &addrlen);
11 else if (argc == 4)
12 listenfd = Tcp_1listen(argv[1], argv[2], &addrlen);
13 else
14 err_quit("usage: serv07 [ <host> ] <port#> <#threads>");
15 nthreads = atoi(argv[argc - 1]);
16 tptr = Calloc(nthreads, sizeof(Thread));

17 for (i = 0; i < nthreads; i++)
18 thread_make(i); /* завершается только основной поток */

19 Signal(SIGINT, sig_int);

20 for (;;)
21 pause(); /* потоки все выполнили */
22 }

```

Функции `thread_make` и `thread_main` показаны в листинге 30.23.

Листинг 30.23. Функции `thread_make` и `thread_main`

```

//server/pthread07.c
1 #include "unpthread.h"
2 #include "pthread07.h"

3 void
4 thread_make(int i)
5 {
6 void *thread_main(void*);

7 Pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void*)i);
8 return; /* завершается основной поток */
9 }

10 void*
11 thread_main(void *arg)
12 {
13 int connfd;
14 void web_child(int);
15 socklen_t clilen;

```

```

16 struct sockaddr *cliaddr;
17 cliaddr = Malloc(addrhlen);
18 printf("thread %d starting\n", (int)arg);
19 for (;;) {
20     clilen = addrlen;
21     Pthread_mutex_lock(&mlock);
22     connfd = Accept(listenfd, cliaddr, &clilen);
23     Pthread_mutex_unlock(&mlock);
24     tptr[(int)arg].thread_count++;
25     web_child(connfd); /* обработка запроса */
26     Close(connfd);
27 }
28 }
```

Создание потоков

7 Создаются потоки, каждый из которых выполняет функцию `pthread_main`. Единственным аргументом этой функции является порядковый номер потока.

21-23 Функция `thread_main` вызывает функции `pthread_mutex_lock` и `pthread_mutex_unlock` соответственно до и после вызова функции `accept`.

Сравнивая строки 6 и 7 в табл. 30.1, можно заметить, что эта последняя версия нашего сервера быстрее, чем версия с созданием нового потока для каждого клиентского запроса. Этого можно было ожидать, так как в данной версии мы сразу создаем пул потоков и не тратим время на создание новых потоков по мере поступления клиентских запросов. На самом деле эта версия сервера — самая быстродействующая для всех операционных систем, которые мы испытывали.

В табл. 30.2 показано распределение значений счетчика `thread_count` структуры `Thread`, которые мы выводим с помощью обработчика сигнала `SIGINT` по завершении работы сервера. Равномерность этого распределения объясняется тем, что при выборе потока, который будет блокировать взаимное исключение, алгоритм планирования загрузки потоков последовательно перебирает все потоки в цикле.

ПРИМЕЧАНИЕ

В Беркли-ядрах нам не нужна блокировка при вызове функции `accept`, так что мы можем использовать версию, представленную в листинге 30.23, без взаимных исключений. Но в результате этого время, затрачиваемое центральным процессором, увеличится. Если рассмотреть два компонента, из которых складывается время центрального процессора — пользовательское и системное время — то окажется, что первый компонент уменьшается при отсутствии блокировки (поскольку блокирование осуществляется в библиотеке потоков, входящей в пользовательское пространство), но системное время возрастает (за счет эффекта «общей побудки», возникающего, когда все потоки, блокированные в вызове функции `accept`, выходят из состояния ожидания при появлении нового клиентского соединения). Для того чтобы каждое соединение передавалось только одному потоку, необходима некая разновидность взаимного исключения, и оказывается, что быстрее это делают сами потоки, а не ядро.

30.12. Сервер с предварительным порождением потоков: основной поток вызывает функцию `accept`

Последняя рассматриваемая нами версия сервера устроена следующим образом: главный поток создает пул потоков при запуске сервера, после чего он же вызывает функцию `accept` и передает каждое

клиентское соединение какому-либо из свободных на данный момент потоков. Это аналогично передаче дескриптора в версии, рассмотренной нами в разделе 30.9.

При таком устройстве сервера необходимо решить, каким именно образом должна осуществляться передача присоединенного дескриптора одному из потоков в пуле. Существует несколько способов решения этой задачи. Можно, как и прежде, использовать передачу дескриптора, но при этом не требуется передавать дескриптор от одного потокам к другому, так как все они, в том числе и главный поток, принадлежат одному и тому же процессу. Все, что требуется знать потоку, получающему дескриптор, — это номер дескриптора. В листинге 30.24 показан заголовочный файл pthread08.h, определяющий структуру Thread, аналогичный файлу, показанному в листинге 30.21.

Листинг 30.24. Заголовочный файл pthread08.h

```
//server/pthread08.h
1 typedef struct {
2     pthread_t thread_tid; /* идентификатор потока */
3     long thread_count; /* количество обработанных запросов */
4 } Thread;
5 Thread *tptr; /* массив структур Thread */

6 #define MAXNCLI 32
7 int clifd[MAXNCLI], iguret, igureput;
8 pthread_mutex_t clifd_mutex;
9 pthread_cond_t clifd_cond;
```

Определение массива для записи дескрипторов присоединенных сокетов

6-9 Мы определяем массив clifd, в который главный поток записывает дескрипторы присоединенных сокетов. Свободные потоки из пула получают по одному дескриптору из этого массива и обрабатывают соответствующий запрос, igureput — это индекс в данном массиве для очередного элемента, записываемого в него главным потоком, а iguret — это индекс очередного элемента массива, передаваемого свободному потоку для обработки. Разумеется, эта структура данных, совместно используемая всеми потоками, должна быть защищена, и поэтому мы используем условную переменную и взаимное исключение.

В листинге 30.25 показана функция main.

Листинг 30.25. Функция main для сервера с предварительным порождением потоков

```
//server/serv08.c
1 #include "unpthread.h"
2 #include "pthread08.h"

3 static int nthreads;
4 pthread_mutex_t clifd_mutex = PTHREAD_MUTEX_INITIALIZER;
5 pthread_cond_t clifd_cond = PTHREAD_COND_INITIALIZER;

6 int
7 main(int argc, char **argv)
8 {
9     int i, listenfd, connfd;
10    void sig_int(int), thread_make(int);
11    socklen_t addrlen, clilen;
12    struct sockaddr *cliaddr;

13    if (argc == 3)
14        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
15    else if (argc == 4)
16        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
17    else
18        err_quit("usage: serv08 [ <host> ] <port#> <#threads>");
19    cliaddr = Malloc(addrlen);
```

```

20 nthreads = atoi(argv[argc - 1]);
21 tptr = Calloc(nthreads, sizeof(Thread));
22 igure = igure = 0;

23 /* создание всех потоков */
24 for (i = 0; i < nthreads; i++)
25   thread_make(i); /* завершается только основной поток */

26 Signal(SIGINT, sig_int);

27 for (;;) {
28   clilen = addrlen;
29   connfd = Accept(listenfd, cliaddr, &clilen);

30   Pthread_mutex_lock(&clifd_mutex);
31   clifd[igure] = connfd;
32   if (++igure == MAXNCLI)
33     igure = 0;
34   if (igure == igure)
35     err_quit("igure = igure = %d", igure);
36   Pthread_cond_signal(&clifd_cond);
37   Pthread_mutex_unlock(&clifd_mutex);
38 }
39 }
```

Создание пула потоков

23-25 Функция `thread_make` создает все потоки.

Ожидание прихода клиентского соединения

27-38 Основной поток блокируется в вызове функции `accept`, ожидая появления нового соединения. При появлении этого соединения дескриптор присоединенного сокета записывается в следующий элемент массива `clifd` после блокирования взаимного исключения. Мы также следим, чтобы индекс `igure` не совпал со значением индекса `igure`, что укажет на недостаточно большой размер массива. Условная переменная сигнализирует о прибытии нового запроса, и взаимное исключение разблокируется, позволяя одному из потоков пула обслужить прибывший запрос.

Функции `thread_make` и `thread_main` показаны в листинге 30.26. Первая из них идентична функции, приведенной в листинге 30.23.

Листинг 30.26. Функции `thread_make` и `thread_main`

```

//server/pthread08.c
1 #include "unpthread.h"
2 #include "pthread08.h"

3 void
4 thread_make(int i)
5 {
6   void *thread_main(void*);

7   Pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void*)i);
8   return; /* завершается основной поток */
9 }
```

```

10 void*
11 thread_main(void *arg)
12 {
13     int connfd;
14     void web_child(int);

15     printf("thread %d starting\n", (int)arg);
16     for (;;) {
17         Pthread_mutex_lock(&clifd_mutex);
18         while (iget == iput)
19             Pthread_cond_wait(&clifd_cond, &clifd_mutex);
20         connfd = clifd[iget]; /* присоединенный сокет, который требуется
21         обслужить */
22         if (++iget == MAXNCLI)
23             iget = 0;
24         Pthread_mutex_unlock(&clifd_mutex);
25         tptr[(int)arg].thread_count++;

26         web_child(connfd); /* обработка запроса */
27     }
28 }

```

Ожидание присоединенного сокета, который требует обслуживания

17-26 Каждый поток из пула пытается блокировать взаимное исключение, блокирующее доступ к массиву `clifd`. Если после того, как взаимное исключение заблокировано, оказывается, что индексы `iput` и `iget` равны, то вызывается функция `pthread_cond_wait`, и поток переходит в состояние ожидания, так как ему пока нечего делать. После прибытия очередного клиентского запроса основной поток вызывает функцию `pthread_cond_signal`, выводя тем самым из состояния ожидания поток, заблокировавший взаимное исключение. Когда этот поток получает соединение, он вызывает функцию `web_child`.

Значения времени центрального процессора, приведенные в табл. 30.1, показывают, что эта версия сервера медленнее рассмотренной в предыдущем разделе (когда каждый поток из пула сам вызывал функцию `accept`). Причина заключается в том, что рассматриваемая в данном разделе версия использует как взаимное исключение, так и условную переменную, тогда как в предыдущем случае (см. листинг 30.23) применялось только взаимное исключение.

Если мы рассмотрим гистограмму количества клиентов, обслуживаемых каждым потоком из пула, то окажется, что распределение клиентских запросов по потокам будет таким же, как показано в последнем столбце табл. 30.2. Это означает, что если основной поток вызывает функцию `pthread_cond_signal`, то при выборе очередного потока, который будет выведен из состояния ожидания для обслуживания клиентского запроса, осуществляется последовательный перебор всех имеющихся свободных потоков.

30.13. Резюме

В этой главе мы рассмотрели 9 различных версий сервера и их работу с одним и тем же веб-клиентом, чтобы сравнить значения времени центрального процессора, затраченного на управление процессом.

0. Последовательный сервер (точка отсчета — управление процессом отсутствует).
1. Параллельный сервер, по одному вызову функции `fork` для каждого клиента.
2. Предварительное порождение дочерних процессов, каждый из которых вызывает функцию `accept`.
3. Предварительное порождение дочерних процессов с блокировкой файла для защиты функции `accept`.
4. Предварительное порождение дочерних процессов с блокировкой взаимного исключения дочерними процессами для защиты функции `accept`.
5. Предварительное порождение дочерних процессов с передачей дескриптора от родительского процесса дочернему.

6. Параллельный сервер, поочередное создание потоков по мере поступления клиентских запросов.
 7. Предварительное порождение потоков с блокировкой взаимного исключения потоками для защиты функции accept.
 8. Предварительное порождение потоков, основной поток вызывает функцию accept.
- Резюмируя материал этой главы, можно сделать несколько комментариев.
- Если сервер не слишком загружен, хорошо работает традиционная модель параллельного сервера, в которой при поступлении очередного клиентского запроса вызывается функция fork для создания нового дочернего процесса. Этот вариант допускает комбинирование с демоном inetd, принимающим все клиентские запросы. Остальные версии применимы в случае загруженных серверов, таких как веб-серверы.
 - Создание пула дочерних процессов или потоков сокращает временные затраты центрального процессора по сравнению с традиционной моделью (один вызов функции fork для каждого запроса) в 10 и более раз. При этом не слишком усложняется код, но становится необходимо (как говорилось при обсуждении примеров) отслеживать количество свободных дочерних процессов и корректировать его по мере необходимости, так как количество клиентских запросов, которые требуется обслужить, динамически изменяется.
 - Некоторые реализации допускают блокирование нескольких потоков или дочерних процессов в вызове функции accept, в то время как другие реализации требуют использования блокировки того или иного типа для защиты accept. Можно использовать для этого либо блокировку файла, либо блокировку взаимного исключения Pthreads.
 - Как правило, версия, в которой каждый поток или дочерний процесс вызывает функцию accept, проще и быстрее, чем версия, где вызов функции accept осуществляется только основным потоком (или родительским процессом), впоследствии передающим дескриптор присоединенного сокета другому потоку или дочернему процессу.
 - Блокировка всех дочерних процессов или программных потоков в вызове функции accept предпочтительнее, чем блокировка в вызове функции select, что объясняется возможностью появления коллизий при вызове функции select.
 - Использование потоков, как правило, дает больший выигрыш во времени, чем использование процессов. Но выбор между версиями 1 и 6 (один дочерний процесс на каждый запрос и один поток на каждый запрос) зависит от свойств операционной системы и от того, какие еще программы задействованы в обслуживании клиентских запросов. Например, если сервер, принимающий клиентское соединение, вызывает функции fork и exec, то может оказаться быстрее породить с помощью функции fork процесс с одним потоком, чем процесс с несколькими потоками.

Упражнения

1. Почему на рис. 30.2 родительский процесс оставляет присоединенный сокет открытым, вместо того чтобы закрыть его, когда созданы все дочерние процессы?
2. Попробуйте изменить сервер из раздела 30.9 таким образом, чтобы использовать дейтаграммный доменный сокет Unix вместо потокового сокета домена Unix. Что при этом изменяется?
3. Запустите клиент и те серверы из рассмотренных в этой главе, которые позволяет запустить конфигурация вашей системы, и сравните полученные результаты с приведенными в тексте.

Глава 31

Потоки (STREAMS)

31.1. Введение

В этой главе мы приводим обзор потоков STREAMS и функций, используемых приложением для доступа к потоку. Наша цель — понять, как реализованы сетевые протоколы в рамках потоковых систем. Также мы создаем простой клиент TCP с использованием TPI — интерфейса, который обеспечивает доступ к транспортному уровню и обычно применяется сокетами в системах, основанных на потоках. Дополнительную информацию о потоках, в том числе о написании программ для ядер, использующих потоки, можно найти в [98].

ПРИМЕЧАНИЕ

Технология потоков была введена Денисом Ритчи (Dennis Ritchie) [104] и получила широкое распространение с появлением системы SVR3 в 1986 году. Спецификация POSIX определяет STREAMS как «дополнительную группу», то есть система может не поддерживать потоки STREAMS, но если она их поддерживает, то реализация должна соответствовать POSIX. Любая система, производная от System V, должна поддерживать потоки, а различные системы 4x.BSD потоки не поддерживают.

Потоковая система часто обозначается как STREAMS, но поскольку это название не является акронимом, то в данной книге используется слово «потоки».

Не следует смешивать «потоковую систему ввода-вывода» (streams I/O system), которую мы описываем в данной главе, и «стандартные потоки ввода-вывода» (standard I/O streams), а также программные потоки (threads). Второй термин используется применительно к стандартной библиотеке ввода-вывода (например, таким функциям, как fopen, fgets, printf и т.п.).

31.2. Обзор

Потоки обеспечивают двустороннее соединение между процессом и драйвером, как показано на рис. 31.1. Хотя нижний блок на этом рисунке мы и называем драйвером, его не следует ассоциировать с каким-либо аппаратным устройством, поскольку это может быть и драйвер псевдоустройства (например, программный драйвер).

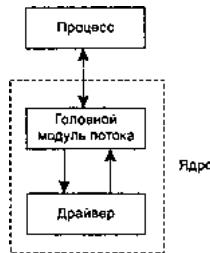


Рис. 31.1. Поток между процессом и драйвером

Головной модуль потока (*stream head*) состоит из программ ядра, которые запускаются при обращении приложения к дескриптору потока (например, при вызове функций read, putmsg, ioctl и т.п.).

Процесс может динамически добавлять и удалять промежуточные модули обработки (*processing modules*) между головным модулем и драйвером. Такой модуль осуществляет некий тип фильтрации сообщений, проходящих в одну или другую сторону по потоку. Этот процесс показан на рис. 31.2.

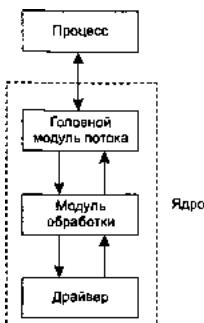


Рис. 31.2. Поток с модулем обработки

В поток может быть помещено любое количество модулей. Под словом «поместить» (push) в данном случае понимается, что каждый новый модуль вставляется сразу после (на рисунке — ниже) головного модуля.

Определенный тип псевдодрайвера называется *мультплексором* (*multiplexor*). Он принимает данные из различных источников. Основанная на потоках реализация набора протоколов TCP/IP, используемая, например, в SVR4, может иметь вид, показанный на рис. 31.3.

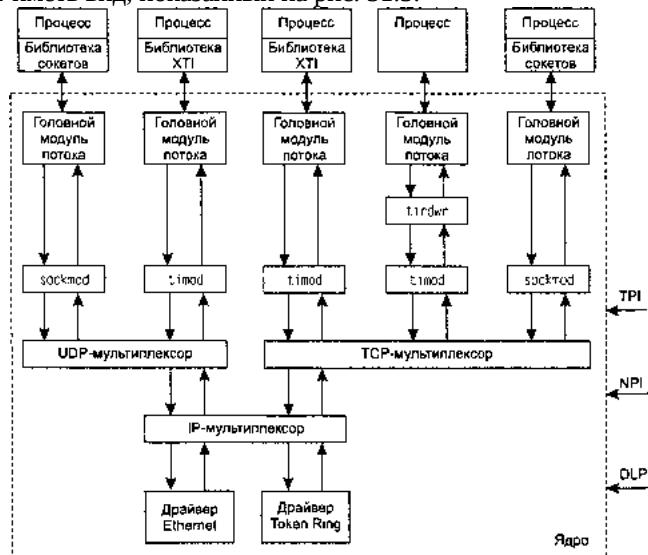


Рис. 31.3. Упрощенный вид реализации набора протоколов TCP/IP, основанной на потоках

- При создании сокета библиотекой сокетов в поток помещается модуль `sockmod`. Именно комбинация библиотеки сокетов и потокового модуля обеспечивает API сокетов для процесса.
- При создании точки доступа XTI библиотекой XTI в поток помещается модуль `timod`. Именно комбинация библиотеки XTI и потокового модуля обеспечивает API XTI для процесса.

ПРИМЕЧАНИЕ

Это одно из немногих мест, где мы говорим об XTI. Предыдущее издание этой книги описывало интерфейс XTI очень подробно, но он уже вышел из широкого употребления, и даже спецификация POSIX больше не включает его, поэтому мы решили исключить ставшие ненужными главы из книги. На рис. 31.3 показано, каким образом обычно реализуется интерфейс XTI. В этой главе мы кратко расскажем о нем, но не будем вдаваться в подробности, потому что причин для использования XTI в настоящее время практически нет.

- Для использования функций `read` или `write` в точке доступа XTI требуется поместить в поток потоковый модуль `t1rdwr`. Это осуществляется процессом, использующим TCP, который на рис. 31.3 изображен четвертым слева. Вероятно, этот процесс тем самым отказался от использования XTI, поэтому мы убрали надпись «библиотека XTI» из соответствующего блока.

■ Формат сетевых сообщений, передаваемых по потокам вверх и вниз, определяют интерфейсы различных сервисов. Мы описываем три наиболее широко распространенных. TPI (*Transport Provider Interface* — *интерфейс поставщика транспортных служб*) [126] определяет интерфейс, предоставляемый поставщиком услуг транспортного уровня (например, TCP или UDP). NPI (*Network Provider Interface* — *интерфейс поставщика сетевого уровня*) [125] определяет интерфейс, предоставляемый поставщиком услуг сетевого уровня (например, IP). DLPI (*Data Link Provider Interface*) — это *интерфейс поставщика канального уровня* [124]. Еще один источник информации по TPI и DLPI, в котором имеются также исходные коды на языке C, — это [98].

Каждый компонент потока — головной модуль, все модули обработки и драйвер — содержат по меньшей мере одну пару очередей: очередь на запись и очередь на чтение. Это показано на рис. 31.4.

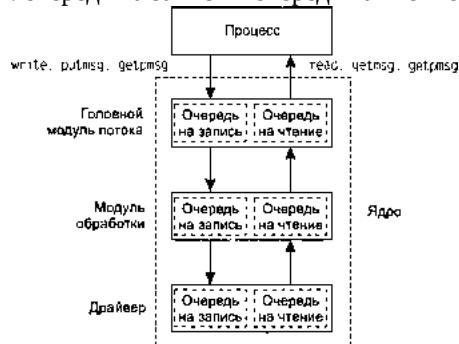


Рис. 31.4. Каждый компонент потока содержит по меньшей мере одну пару очередей

Типы сообщений

Потоковые сообщения могут быть классифицированы как *имеющие высокий приоритет (high priority)*, входящие в полосу приоритета (*priority band*) и обычные (*normal*). Существует 256 полос приоритета со значениями между 0 и 255, причем обычные сообщения соответствуют полосе 0. Приоритет потокового сообщения используется как при постановке сообщения в очередь, так и для управления потоком (*flow control*). По соглашению, на сообщения с высоким приоритетом управление потоком не влияет.

На рис. 31.5 показан порядок следования сообщений в одной конкретной очереди.

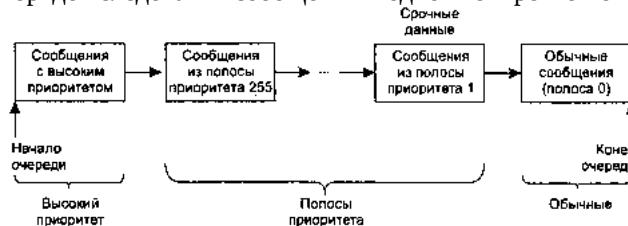


Рис. 31.5. Порядок следования потоковых сообщений в очереди в зависимости от их приоритета

Хотя потоковые системы поддерживают 256 различных полос приоритета, в сетевых протоколах обычно используется полоса 1 для срочных (внеполосных) данных и полоса 0 для обычных данных.

ПРИМЕЧАНИЕ

Внеполосные данные TCP в TPI не рассматриваются как истинные срочные данные. В самом деле, в TCP полоса 0 используется как для обычных, так и для внеполосных данных. Полоса 1 используется для отправки срочных данных в тех протоколах, в которых срочные данные (а не просто срочный указатель, как в TCP) отправляются перед обычными данными. В данном контексте следует внимательно отнести к термину «обычный» (*normal*). В системах SVR, предшествующих SVR4, не было полос приоритета, а сообщения делились на обычные и приоритетные (*priority messages*). В SVR4 были введены полосы приоритета, что потребовало также введения функций *getpmsg* и *putpmsg*, которые мы вскоре опишем. Приоритетные сообщения были переименованы в сообщения с высоким приоритетом, и встал вопрос, как называть сообщения, относящиеся к полосам приоритета от 1 до 255. Наиболее

распространенной является терминология [98], согласно которой все сообщения, которые не являются сообщениями с высоким приоритетом, называются обычными сообщениями и разделяются на подкатегории согласно своим полосам приоритета. Термин «обычное сообщение» в любом случае должен соответствовать сообщению из полосы приоритета 0.

Хотя пока мы говорили только о сообщениях с высоким приоритетом и об обычных сообщениях, существует около 12 типов обычных сообщений и около 18 типов сообщений с высоким приоритетом. С точки зрения приложений и функций `getmsg` и `putmsg`, которые мы опишем в следующем разделе, нам интересны только три различных типа сообщений: `M_DATA`, `M_PROTO` и `M_PCPROTO` (`PC` означает «priority control», то есть приоритетное управление, и подразумевает сообщения с высоким приоритетом). В табл. 31.1 показано, как эти три типа сообщений генерируются функциями `write` и `putmsg`.

Таблица 31.1. Типы потоковых сообщений, генерируемые функциями `write` и `putmsg`

	Функция Управляющая информация?	Данные?	Флаги	Генерируемый тип сообщения
<code>write</code>		Да		<code>M_DATA</code>
<code>putmsg</code>	Нет	Да	0	<code>M_DATA</code>
<code>putmsg</code>	Да	Все равно 0		<code>M_PROTO</code>
<code>putmsg</code>	Да	Все равно <code>MSG_HIPRI</code>		<code>M_PCPROTO</code>

31.3. Функции `getmsg` и `putmsg`

Данные, передаваемые в обоих направлениях по потоку, состоят из сообщений, а каждое сообщение содержит **данные**, **управляющую информацию** или и то и другое. Если мы используем функции `read` или `write`, то мы можем передавать только данные. Для того чтобы процесс мог записывать и считывать как данные, так и управляющую информацию, необходимо добавить две новые функции.

```
#include <stropts.h>

int getmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int *flagsp);
int putmsg(int fd, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flags);

Обе функции возвращают: неотрицательное значение в случае успешного выполнения (см. пояснения в тексте), -1 в случае ошибки

Обе составляющие сообщения — и сами данные, и управляющая информация — описываются структурой strbuf:
struct strbuf {
    int maxlen; /* максимальный размер буфера buf */
    int len;    /* фактическое количество данных в buf */
    char *buf;  /* данные */
};
```

ПРИМЕЧАНИЕ

Обратите внимание на аналогию между структурами `strbuf` и `netbuf`. Имена элементов обеих структур одинаковы.

Однако обе длины в структуре `netbuf` относятся к типу данных `unsigned int` (целое без знака), тогда как обе длины в структуре `strbuf` — к типу `int` (целое со знаком). Причина в том, что некоторые потоковые функции используют значение `-1` элементов `len` и `maxlen` для указания на определенные специальные ситуации.

С помощью функции `putmsg` мы можем отправлять или данные, или управляющую информацию, или и то и другое вместе. Для указания на отсутствие управляющей информации мы можем или задать `ctlptr` как пустой указатель, или установить значение `ctlptr->len` равным `-1`. Этот же способ используется для указания на отсутствие данных.

При отсутствии управляющей информации функцией putmsg генерируется сообщение типа M_DATA (см. табл. 31.1), в противном случае генерируется сообщение типа M_PROTO либо M_PCPROTO в зависимости от значения аргумента flags. Этот аргумент функции putmsg имеет нулевое значение для обычных сообщений, а для сообщений с высоким приоритетом его значение равно RS_HIPRI.

Последний аргумент функции getmsg имеет тип «значение-результат». Если при вызове функции целочисленное значение, на которое указывает аргумент flagsp, — это 0, то возвращается первое сообщение из потока (которое может быть как обычным, так и имеющим высокий приоритет). Если при вызове функции целочисленное значение соответствует RS_HIPRI, то функция будет ждать появления в головном модуле потока сообщения с высоким приоритетом. В обоих случаях в зависимости от типа возвращенного сообщения значение, на которое указывает аргумент flagsp, будет либо 0, либо RS_HIPRI.

Предположим, что мы передаем функции getmsg непустые указатели ct1ptr и dataptr. Тогда указанием на отсутствие управляющей информации (возвращается сообщение типа M_DATA) является значение ct1ptr->len, установленное в -1. Аналогично, если отсутствуют данные, указанием на это является значение -1 элемента dataptr->len.

Если функция putmsg выполнилась успешно, то она возвращает нулевое значение, а в случае ошибки возвращается значение -1. Но функция getmsg возвращает нулевое значение только в том случае, если вызывающему процессу было доставлено все сообщение целиком. Если буфер, предназначенный для приема управляющей информации, слишком мал, то возвращается значение MORECTL (о котором заранее известно, что оно является неотрицательным). Аналогично, если буфер для приема данных оказывается слишком мал, возвращается значение MOREDATA. Если же оба эти буфера оказываются слишком малы, то возвращается логическая сумма этих двух флагов.

31.4. Функции getpmsg и putpmsg

Когда с выпуском SVR4 к потоковым системам была добавлена поддержка различных полос приоритета, появились новые варианты функций getmsg и putmsg.

```
#include <stropts.h>

int getpmsg(int fd, struct strbuf *ct1ptr,
           struct strbuf *dataptr, int *bandp, int *flagsp);
int putpmsg(int fd, const struct strbuf *ct1ptr,
            const struct strbuf *dataptr, int band, int flags);
```

Обе функции возвращают: неотрицательное значение в случае успешного выполнения, -1 в случае ошибки

Аргумент band функции putpmsg должен иметь значение в пределах от 0 до 255 включительно. Если аргумент flags имеет значение MSG_BAND, то генерируется сообщение в соответствующей полосе приоритета. Присваивание аргументу flags значения MSG_BAND и задание полосы 0 эквивалентно вызову функции putmsg. Если значение аргумента flags равно MSG_HIPRI, то аргумент band должен быть равен нулю, и тогда генерируется сообщение с высоким приоритетом. (Обратите внимание на то, что этот флаг имеет название, отличающееся от названия RS_HIPRI, используемого в случае функции putmsg.)

Два целочисленных значения, на которые указывают аргументы bandp и flagsp функции getpmsg, являются аргументами типа «значение-результат». Целочисленное значение, на которое указывает аргумент flagsp функции getpmsg, может соответствовать MSG_HIPRI (для чтения сообщений с высоким приоритетом), MSG_BAND (для чтения сообщений из полосы приоритета, по меньшей мере равной целочисленному значению, на которое указывает аргумент bandp) или MSG_ANY (для чтения любых сообщений). По завершении функции целочисленное значение, на которое указывает аргумент bandp, указывает на полосу приоритета прочитанного сообщения, а целое число, на которое указывает аргумент flagsp, соответствует MSG_HIPRI (если было прочитано сообщение с высоким приоритетом) или MSG_BAND (если было прочитано иное сообщение).

31.5. Функция ioctl

Говоря о потоках, мы снова возвращаемся к функции ioctl, которая уже была описана в главе 17.

```
#include <stropts.h>
```

```
int ioctl(int fd, int request, ... /* void *arg */ );  
Возвращает: 0 в случае успешного выполнения, -1 в случае ошибки
```

Единственным изменением относительно прототипа функции, приведенного в разделе 17.2, является включение заголовочного файла, необходимого для работы с потоками.

Существует примерно 30 запросов (*request*), так или иначе влияющих на головной модуль потока. Каждый из запросов начинается с *T_*, и обычно документация на них приводится на странице руководства *streamio*.

31.6. TPI: интерфейс поставщика транспортных служб

На рис. 31.3 мы показали, что TPI — это интерфейс, предоставляющий доступ к транспортному уровню для расположенных выше уровней. Этот интерфейс используется в потоковой среде как сокетами, так и XTI. Из рис. 31.3 видно, что комбинация библиотеки сокетов и *sokmod*, а также комбинация библиотеки XTI и *timod* обмениваются сообщениями TPI с TCP и UDP.

TPI является интерфейсом, *основанным на сообщениях* (*message-based*). Он определяет сообщения, которыми обменивается приложение (например, XTI или библиотека сокетов) и транспортный уровень. Точнее, TPI задает формат этих сообщений и то, какое действие производит каждое из сообщений. Во многих случаях приложение посыпает запрос поставщику (например, «Связать данный локальный адрес»), а поставщик посыпает обратно ответ («Выполнено» или «Ошибка»). Некоторые события, происходящие асинхронно на стороне поставщика (например, прибытие запроса на соединение с сервером), инициируют отправку сигнала или сообщения вверх по потоку.

Мы можем обойти как XTI, так и сокеты, и использовать непосредственно TPI. В этом разделе мы заново перепишем код нашего простого клиента времени и даты с использованием TPI вместо сокетов (сокетная версия представлена в листинге 1.1). Если провести аналогию с языками программирования, то использование XTI или сокетов можно сравнить с программированием на языках высокого уровня, таких как C или Pascal, а непосредственно TPI — с программированием на ассемблере. Мы не являемся сторонниками непосредственного использования TPI в реальной жизни. Но понимание того, как работает TPI, и написание примера с использованием этого протокола позволит нам глубже понять, как работает библиотека сокетов в потоковой среде.

В листинге 31.1^[1] показан наш заголовочный файл *tpi_daytime.h*.

Листинг 31.1. Наш заголовочный файл *tpi_daytime.h*

```
//streams/tpi_daytime.h  
1 #include "unpxti.h"  
2 #include <sys/stream.h>  
3 #include <sys/tihdr.h>  
  
4 void tpi_bind(int, const void*, size_t);  
5 void tpi_connect(int, const void*, size_t);  
6 ssize_t tpi_read(int, void*, size_t);  
7 void tpi_close(int);
```

Нам нужно включить еще один дополнительный заголовочный файл помимо *<sys/tihdr.h>*, содержащего определения структур для всех сообщений TPI.

Листинг 31.2. Функция *main* для нашего клиента времени и даты с использованием TPI

```
//streams/tpi_daytime.c  
1 #include "tpi_daytime.h"  
  
2 int  
3 main(int argc, char **argv)  
4 {  
5     int fd, n;  
6     char recvline[MAXLINE + 1];  
7     struct sockaddr_in myaddr, servaddr;  
  
8     if (argc != 2)  
9         err_quit("usage: tpi_daytime <Ipaddress>");
```

```

10 fd = Open(XTI_TCP, O_RDWR, 0);

11 /* связываем произвольный локальный адрес */
12 bzero(&myaddr, sizeof(myaddr));
13 myaddr.sin_family = AF_INET;
14 myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15 myaddr.sin_port = htons(0);

16 tpi_bind(fd, &myaddr, sizeof(struct sockaddr_in));

17 /* заполняем адрес сервера */
18 bzero(&servaddr, sizeof(servaddr));
19 servaddr.sin_family = AF_INET;
20 servaddr.sin_port = htons(13); /* сервер времени и даты */
21 Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

22 tpi_connect(fd, &servaddr, sizeof(struct sockaddr_in));

23 for (;;) {
24     if ((n = tpi_read(fd, recvline, MAXLINE)) <= 0) {
25         if (n == 0)
26             break;
27         else
28             err_sys("tpi_read error");
29     }
30     recvline[n] = 0; /* завершающий нуль */
31     fputs(recvline, stdout);
32 }
33 tpi_close(fd);
34 exit(0);
35 }

```

Открытие транспортного устройства, связывание локального адреса

10-16 Мы открываем устройство, соответствующее поставщику транспортных служб (обычно /dev/tcp). Мы заполняем структуру адреса сокета Интернета значениями INADDR_ANY и 0 (для порта), указывая тем самым TCP связать произвольный локальный адрес с нашей точкой доступа. Мы вызываем свою собственную функцию tpi_bind (которая будет приведена чуть ниже) для выполнения этого связывания.

Заполнение структуры адреса сервера, установление соединения

17-22 Мы заполняем другую структуру адреса сокета Интернета, внося в нее IP-адрес сервера (из командной строки) и порт (13). Мы вызываем нашу функцию tpi_connect для установления соединения.

Считывание данных с сервера, копирование в стандартный поток вывода

23-33 Как и в случае других клиентов времени и даты, мы просто копируем данные, пришедшие по соединению, в стандартный поток вывода, останавливаясь при получении признака конца файла, присланного сервером (например, сегмент FIN). Мы сделали этот цикл похожим на тот, который использовался в коде сокетного клиента (см. листинг 1.1), поскольку наша функция tpi_read при нормальном завершении соединения на стороне сервера будет возвращать нулевое значение. Затем мы вызываем нашу функцию tpi_close для того, чтобы закрыть эту точку доступа.

Наша функция tpi_bind показана в листинге 31.3.

Листинг 31.3. Функция tpi_bind: связывание локального адреса с точкой доступа

```
//streams/tpi_bind.c
1 #include "tpi_daytime.h"

2 void
3 tpi_bind(int fd, const void *addr, size_t addrlen)
4 {
5     struct {
6         struct T_bind_req msg_hdr;
7         char addr[128];
8     } bind_req;
9     struct {
10    struct T_bind_ack msg_hdr;
11    char addr[128];
12 } bind_ack;
13     struct strbuf ctlbuf;
14     struct T_error_ack *error_ack;
15     int flags;
16     bind_req.msg_hdr.PRIM_type = T_BIND_REQ;
17     bind_req.msg_hdr.ADDR_length = addrlen;
18     bind_req.msg_hdr.ADDR_offset = sizeof(struct T_bind_req);
19     bind_req.msg_hdr.CONIND_number = 0;
20     memcpy(bind_req.addr, addr, addrlen); /* sockaddr_in{} */

21     ctlbuf.len = sizeof(struct T_bind_req) + addrlen;
22     ctlbuf.buf = (char*)&bind_req;
23     Putmsg(fd, &ctlbuf, NULL, 0);

24     ctlbuf maxlen = sizeof(bind_ack);
25     ctlbuf.len = 0;
26     ctlbuf.buf = (char*)&bind_ack;
27     flags = RS_HIPRI;
28     Getmsg(fd, &ctlbuf, NULL, &flags);
29     if (ctlbuf.len < (int)sizeof(long))
30         err_quit("bad length from getmsg");

31     switch (bind_ack.msg_hdr.PRIM_type) {
32     case T_BIND_ACK:
33         return;

34     case T_ERROR_ACK:
35         if (ctlbuf.len < (int)sizeof(struct T_error_ack))
36             err_quit("bad length for T_ERROR_ACK");
37         error_ack = (struct T_error_ack*)&bind_ack.msg_hdr;
38         err_quit("T_ERROR_ACK from bind (%d, %d)",
39                 error_ack->TLI_error, error_ack->UNIX_error);

40     default:
41         err_quit("unexpected message type: %d", bind_ack.msg_hdr.PRIM_type);
42     }
43 }
```

Заполнение структуры T_bind_req

16-20 Заголовочный файл <sys/tihdr.h> определяет структуру `T_bind_req`:

```
struct T_bind_req {  
    long      PRIM_type;      /* T_BIND_REQ */  
    long      ADDR_length;   /* длина адреса */  
    long      ADDR_offset;   /* смещение адреса */  
    unsigned long CONIND_number; /* сообщения о соединении */  
    /* далее следует адрес протокола для связывания */  
};
```

Все запросы TPI определяются как структуры, начинающиеся с поля типа `long`. Мы определяем свою собственную структуру `bind_req`, начинающуюся со структуры `T_bind_req`, после которой располагается буфер, содержащий локальный адрес для связывания. TPI ничего не говорит о содержимом буфера — оно определяется поставщиком. Поставщик TCP предполагает, что этот буфер содержит структуру `sockaddr_in`.

Мы заполняем структуру `T_bind_req`, устанавливая элемент `ADDR_length` равным размеру адреса (16 байт для структуры адреса сокета Интернета), а элемент `ADDR_offset` — равным байтовому сдвигу адреса (он следует непосредственно за структурой `T_bind_req`). У нас нет гарантии, что это местоположение соответствующим образом выровнено для записи структуры `sockaddr_in`, поэтому мы вызываем функцию `memcp`, чтобы скопировать структуру вызывающего процесса в нашу структуру `bind_req`. Мы присваиваем элементу `CONIND_number` нулевое значение, потому что мы находимся на стороне клиента, а не на стороне сервера.

Вызов функции `putmsg`

21-23 TPI требует, чтобы только что созданная нами структура была передана поставщику как одно сообщение `M_PROTO`. Следовательно, мы вызываем функцию `putmsg`, задавая структуру `bind_req` в качестве управляющей информации, без каких-либо данных и с флагом 0.

Вызов функции `getmsg` для чтения сообщений с высоким приоритетом

24-30 Ответом на наш запрос `T_BIND_REQ` будет либо сообщение `T_BIND_ACK`, либо сообщение `T_ERROR_ACK`. Сообщения, содержащие подтверждение, отправляются как сообщения с высоким приоритетом (`M_PCPROTO`), так что мы считываем их при помощи функции `getmsg` с флагом `RS_HIPRI`. Поскольку ответ является сообщением с высоким приоритетом, он получает преимущество перед всеми обычными сообщениями в потоке.

Эти два сообщения выглядят следующим образом:

```
struct T_bind_ack {  
    long      PRIM_type;      /* T_BIND_ACK */  
    long      ADDR_length;   /* длина адреса */  
    long      ADDR_offset;   /* смещение адреса */  
    unsigned long CONIND_number; /* индекс подключения для помещения  
                                в очередь */  
};  
  
/* затем следует связанный адрес */  
struct T_error_ack {  
    long PRIM_type; /* T_ERROR_ACK */  
    long ERROR_prim; /* примитивная ошибка ввода */  
    long TLI_error; /* код ошибки TLI */  
    long UNIX_error; /* код ошибки UNIX */  
};
```

В начале каждого сообщения указан его тип, так что мы можем начать считывать ответ, предполагая, что это сообщение `T_BIND_ACK`, а затем, прочитав его тип, обрабатывать его тем или иным способом. Мы не ждем никаких данных от поставщика, поэтому третий аргумент функции `getmsg` мы задаем как пустой указатель.

ПРИМЕЧАНИЕ

Когда мы проверяем, соответствует ли количество возвращенной управляющей информации по меньшей мере размеру длинного целого, нужно проявить осторожность, преобразуя значение sizeof в целое число. Оператор sizeof возвращает целое число без знака, но существует вероятность того, что значение возвращенного поля len будет -1. Поскольку при выполнении операции сравнения слева располагается значение со знаком, а справа — без знака, компилятор преобразует значение со знаком в значение без знака. Если рассматривать -1 как целое без знака в архитектуре с дополнением до 2, это число получается очень большим, то есть -1 оказывается больше 4 (если предположить, что длинное целое число занимает 4 байта).

Обработка ответа

31-33 Если ответ — это сообщение T_BIND_ACK, то связывание прошло успешно, и мы возвращаемся. Фактический адрес, связанный с точкой доступа, возвращается в элементе addr нашей структуры bind_ack, которую мы игнорируем.

34-39 Если ответ — это сообщение T_ERROR_ACK, мы проверяем, было ли сообщение получено целиком, и выводим три значения, содержащиеся в возвращенной структуре. В этой простой программе при возникновении ошибки мы просто прекращаем выполнение и ничего не возвращаем вызывающему процессу.

Чтобы увидеть ошибки, которые могут возникнуть в результате запроса на связывание, мы слегка изменим нашу функцию main и попробуем связать какой-либо порт, отличный от 0. Например, если мы попробуем связать порт 1 (что требует прав привилегированного пользователя, так как это порт с номером меньше 1024), мы получим следующий результат:

```
solaris % tpi_daytime 127.0.0.1  
T_ERROR_ACK from bind (3, 0)
```

В этой системе значение константы EACCESS равно 3. Если мы поменяем номер порта, задав значение большее 1023, но используемое в настоящий момент другой точкой доступа TCP, мы получим:

```
solaris % tpi_daytime 127.0.0.1  
T_ERROR_ACK from bind (23, 0)
```

В данной системе значение константы EADDRBUSY равно 23.

Следующая функция показана в листинге 31.4. Это функция tpi_connect, устанавливающая соединение с сервером.

Листинг 31.4. Функция tpi_connect: установление соединения с сервером

```
//streams/tpi_connect.c  
1 #include "tpi_daytime.h"  
  
2 void  
3 tpi_connect(int fd, const void *addr, size_t addrlen)  
4 {  
5     struct {  
6         struct T_conn_req msg_hdr;  
7         char addr[128];  
8     } conn_req;  
9     struct {  
10        struct l_conn_con msg_hdr;  
11        char addr[128];  
12    } conn_con;  
13    struct strbuf ctlbuf;  
14    union T_primitives rcvbuf;  
15    struct T_error_ack *error_ack;  
16    struct T_discon_ind *discon_ind;  
17    int flags;
```

```

18 conn_req.msg_hdr.PRIM_type = T_CONN_REQ;
19 conn_req.msg_hdr.DEST_length = addrlen;
20 conn_req.msg_hdr.DEST_offset = sizeof(struct T_conn_req);
21 conn_req.msg_hdr.OPT_length = 0;
22 conn_req.msg_hdr.OPT_offset = 0;
23 memcpy(conn_req.addr, addr, addrlen); /* sockaddr_in{} */
24 ctlbuf.len = sizeof(struct T_conn_req) + addrlen;
25 ctlbuf.buf = (char*)&conn_req;
26 Putmsg(fd, &ctlbuf, NULL, 0);

27 ctlbuf maxlen = sizeof(union T_primitives);
28 ctlbuf.len = 0;
29 ctlbuf.buf = (char*)&rcvbuf;
30 flags = RS_HIPRI;
31 Getmsg(fd, &ctlbuf, NULL, &flags);
32 if (ctlbuf.len < (int)sizeof(long))
33 err_quit("tpi_connect: bad length from getmsg");

34 switch (rcvbuf.type) {
35 case T_OK_ACK:
36 break;

37 case T_ERROR_ACK:
38 if (ctlbuf.len < (int)sizeof(struct T_error_ack))
39 err_quit("tpi_connect: bad length for T_ERROR_ACK");
40 error_ack = (struct T_error_ack*)&rcvbuf;
41 err_quit("tpi_connect: T_ERROR_ACK from conn %d, %d",
42 error_ack->TLI_error, error_ack->UNIX_error);

43 default:
44 err_quit("tpi connect, unexpected message type: &d", rcvbuf.type);
45 }

46 ctlbuf maxlen = sizeof(conn_con);
47 ctlbuf.len = 0;
48 ctlbuf.buf = (char*)&conn_con;
49 flags = 0;
50 Getmsg(fd, &ctlbuf, NULL, &flags);
51 if (ctlbuf.len < (int)sizeof(long))
52 err_quit("tpi_connect2: bad length from getmsg");

53 switch (conn_con.msg_hdr.PRIM_type) {
54 case T_CONN_CON:
55 break;

56 case T_DISCON_IND:
57 if (ctlbuf.len < (int)sizeof(struct T_discon_ind))
58 err_quit("tpi_connect2: bad length for T_DISCON_IND");
59 discon_ind = (struct T_discon_ind*)&conn_con.msg_hdr;
60 err_quit("tpi_connect2: T_DISCON_IND from conn (%d)",
61 discon_ind->DISCON_reason);

62 default:
63 err_quit("tpi_connect2: unexpected message type. %d",
64 conn_con.msg_hdr PRIM_type);
65 }

```

Заполнение структуры запроса и отправка поставщику

18-26 В TPI определена структура `T_conn_req`, содержащая адрес протокола и параметры для соединения:

```
struct T_conn_req {
    long PRIM_type; /* T_CONN_REQ */
    long DEST_length; /* длина адреса получателя */
    long DEST_offset; /* смещение адреса получателя */
    long OPT_length; /* длина параметров */
    long OPT_offset; /* смещение параметров */
    /* затем следуют адреса протокола и параметры соединения */
};
```

Как и в случае функции `tpi_bind`, мы определяем свою собственную структуру с именем `conn_req`, которая включает в себя структуру `T_conn_req`, а также содержит место для адреса протокола. Мы заполняем структуру `conn_req`, обнуляя поля `OPT_length` и `OPT_offset`. Мы вызываем функцию `putmsg` только с управляющей информацией и флагом 0 для отправки сообщения типа `M_PROTO` вниз по потоку.

Чтение ответа

27-45 Мы вызываем функцию `getmsg`, ожидая получить в ответ либо сообщение `T_OK_ACK`, если было начато установление соединения, либо сообщение `T_ERROR_ACK` (которые мы уже показывали выше). В случае ошибки мы завершаем выполнение программы. Поскольку мы не знаем, сообщение какого типа мы получим, то определяем объединение с именем `T_primitives` для приема всех возможных запросов и ответов и размещаем это объединение в памяти как входной буфер для управляющей информации при вызове функции `getmsg`.

```
struct T_ok_ack {
    long PRIM_type; /* T_OK_ACK */
    long CORRECT_prim; /* корректный примитив */
};
```

Ожидание завершения установления соединения

46-65 Сообщение `T_OK_ACK`, полученное нами на предыдущем этапе, указывает лишь на то, что соединение успешно начало устанавливаться. Теперь нам нужно дождаться сообщения `T_CONN_CON`, указывающего на то, что другой конец соединения подтверждает получение запроса на соединение.

```
struct T_conn_con {
    long PRIM_type; /* T_CONN_CON */
    long RES_length; /* длина адреса собеседника */
    long RES_offset; /* смещение адреса собеседника */
    long OPT_length; /* длина параметра */
    long OPT_offset; /* смещение параметра */
    /* далее следуют адрес протокола и параметры собеседника */
};
```

Мы снова вызываем функцию `getmsg`, но ожидаемое нами сообщение посыпается как сообщение типа `M_PROTO`, а не как сообщение `M_PCPROTO`, поэтому мы обнуляем флаги. Если мы получаем сообщение `T_CONN_CON`, значит, соединение установлено, и мы возвращаемся, но если соединение не было установлено (по причине того, что процесс собеседника не запущен, истекло время ожидания или еще по какой-либо причине), то вместо этого вверх по потоку отправляется сообщение `T_DISCON_IND`:

```
struct T_discon_ind {
    long PRIM_type; /* T_DISCON_IND */
    long DISCON_reason; /* причина разрыва соединения */
```

```
    long SEQ_number; /* порядковый номер */
};
```

Мы можем посмотреть, какие ошибки могут быть возвращены поставщиком. Сначала мы задаем IP-адрес узла, на котором не запущен сервер времени и даты:

```
solaris26 % tpi_daytime 192.168.1.10
tpi_connect2: T_DISCON_IND from conn (146)
```

Код 146 соответствует ошибке ECONNREFUSED. Затем мы задаем IP-адрес, который не связан с Интернетом:

```
solaris26 % tpi_daytime 192.3.4.5
tpi_connect2: T_DISCON_IND from conn (145)
```

На этот раз возвращается ошибка ETIMEDOUT. Но если мы снова запустим нашу программу, задавая тот же самый IP-адрес, то получим другую ошибку:

```
solaris26 % tpi_daytime 192.3.4.5
tpi_connect2: T_DISCON_IND from conn (148)
```

На этот раз мы получаем ошибку ENOSTUNREACH. Различие в том, что в первый раз не было возвращено сообщение ICMP о недоступности узла, а во второй раз мы получили это сообщение.

Следующая функция, которую мы рассмотрим, — это tpi_read, показанная в листинге 31.5. Она считывает данные из потока.

Листинг 31.5. Функция tpi_read: считывание данных из потока

```
//streams/tpi_read.c
1 #include "tpi_daytime.h"

2 ssize_t
3 tpi_read(int fd, void *buf, size_t len)
4 {
5     struct strbuf ctlbuf;
6     struct strbuf datbuf;
7     union T_primitives rcvbuf;
8     int flags;

9     ctlbuf maxlen = sizeof(union T_primitives);
10    ctlbuf.buf = (char*)&rcvbuf;

11    datbuf maxlen = len;
12    datbuf.buf = buf;
13    datbuf.len = 0;

14    flags = 0;
15    Getmsg(fd, &ctlbuf, &datbuf, &flags);

16    if (ctlbuf.len >= (int)sizeof(long)) {
17        if (rcvbuf.type == T_DATA_IND)
18            return (datbuf.len);
19        else if (rcvbuf.type == T_ORDREL_IND)
20            return (0);
21        else
22            err_quit("tpi_read: unexpected type %d", rcvbuf.type);
23    } else if (ctlbuf.len == -1)
24        return (datbuf.len);
25    else
26        err_quit("tpi_read: bad length from getmsg");
27 }
```

Считывание управляющей информации и данных, обработка ответа

9-26 На этот раз мы вызываем функцию getmsg для считывания как данных, так и управляющей информации. Структура strbuf, предназначенная для данных, указывает на буфер вызывающего процесса. В потоке события могут развиваться по четырем различным сценариям.

■ Данные могут прибыть в виде сообщения M_DATA, и указанием на это является возвращенное значение длины управляющей информации, равное -1. Данные скопированы в буфер вызывающего процесса функцией getmsg, и функция просто возвращает длину этих данных.

■ Данные могут прибыть как сообщение T_DATA_IND, в этом случае управляющая информация будет содержаться в структуре T_data_ind:

```
struct T_data_ind {  
    long PRIM_type; /* T_DATA_IND */  
    long MORE_flag; /* еще данные */  
};
```

Если возвращено такое сообщение, мы игнорируем поле MORE_flag (оно вообще не задается для таких протоколов, как TCP) и просто возвращаем длину данных, скопированных в буфер вызывающего процесса функцией getmsg.

■ Сообщение T_ORDREL_IND возвращается, если все данные получены и следующим элементом является сегмент FIN:

```
struct T_ordrel_ind {  
    long PRIM_type; /* T_ORDREL_IND */  
};
```

Это нормальное завершение. Мы просто возвращаем нулевое значение, указывая вызывающему процессу, что по соединению получен признак конца файла.

■ Сообщение T_DISCON_IND возвращается, если произошел разрыв соединения. Наша последняя функция — это tpi_close, показанная в листинге 31.6.

Листинг 31.6. Функция tpi_close: отправка запроса о завершении собеседнику

```
//streams/tpi_close.c  
1 #include "tpi_daytime.h"  
  
2 void  
3 tpi_close(int fd)  
4 {  
5     struct T_ordrel_req ordrel_req;  
6     struct strbuf ctlbuf;  
  
7     ordrel_req PRIM_type = T_ORDREL_REQ;  
8     ctlbuf.len = sizeof(struct T_ordrel_req);  
9     ctlbuf.buf = (char*)&ordrel_req;  
10    Putmsg(fd, &ctlbuf, NULL, 0);  
  
11    Close(fd);  
12 }
```

Отправка запроса о завершении собеседнику

7-10 Мы формируем структуру T_ordrel_req:

```
struct T_ordrel_req {  
    long PRIM_type; /* T_ORDREL_REQ */  
};
```

и посыпаем ее как сообщение M_PROTO с помощью функции putmsg. Это соответствует функции XTI t_sndrel.

Этот пример позволил нам почувствовать специфику TPI. Приложение посылает сообщения вниз по потоку (запросы), а поставщик посыпает сообщения вверх по потоку (ответы). Некоторые обмены сообщений организованы согласно простому сценарию «запрос-ответ» (связывание локального адреса), в то время как остальные могут занять некоторое время (установление соединения), позволяя нам заняться чем-то другим в процессе ожидания ответа. Для знакомства с TPI мы выбрали этот пример (написание

клиента TCP) из-за его относительной простоты. Если бы мы решили написать с использованием TPI TCP-сервер, обрабатывающий одновременно несколько соединений, это было бы гораздо сложнее.

ПРИМЕЧАНИЕ

Можно сравнить количество системных вызовов, необходимых для осуществления определенных сетевых операций, показанных в этой главе, в случае применения TPI и когда используется ядро, реализующее сокеты. Связывание с локальным адресом в случае TPI требует двух системных вызовов, но в случае сокетного ядра требуется только один вызов [128, с. 454]. Для установления соединения на блокируемом дескрипторе с использованием TPI требуется три системных вызова, а в случае сокетного ядра — только один [128, с. 466].

31.7. Резюме

Иногда сокеты реализуются с использованием потоков STREAMS. Для обеспечения доступа к потоковой подсистеме вводятся четыре новые функции: `getmsg`, `putmsg`, `getpmsg` и `putpmsg`. Также в потоковой подсистеме широко используется уже описанная ранее функция `ioctl`.

TPI представляет собой потоковый интерфейс системы SVR4, предоставляющий доступ из верхних уровней на транспортный уровень. Он используется как сокетами, так и XTI, как показано на рис. 31.3. В этой главе в качестве примера использования основанного на сообщениях интерфейса мы разработали версию клиента времени и даты, в котором непосредственно применяется интерфейс TPI.

Упражнения

1. В листинге 31.6 мы вызываем функцию `putmsg`, чтобы отправить вниз по потоку запрос на нормальное завершение соединения, а затем немедленно вызываем функцию `close` для закрытия потока. Что произойдет, если наш запрос будет потерян потоковой подсистемой, а мы закроем поток?

Приложения

Приложение А

Протоколы IPv4, IPv6, ICMPv4 и ICMPv6

A.1. Введение

В этом приложении приведен обзор протоколов IPv4, IPv6, ICMPv4 и ICMPv6. Данный материал позволяет глубже понять рассмотренные в главе 2 протоколы TCP и UDP. Некоторые возможности IP и ICMP рассматриваются также более подробно и в других главах, например параметры IP (см. главу 27), и программы ping и traceroute (см. главу 28).

A.2. Заголовок IPv4

Уровень IP обеспечивает не ориентированную на установление соединения (connectionless) и ненадежную службу доставки дейтаграмм (RFC 791 [94]). Уровень IP делает все возможное для доставки IP-дейтаграммы определенному адресату, но не гарантирует, что дейтаграмма будет доставлена, прибудет в нужном порядке относительно других пакетов, а также будет доставлена в единственном экземпляре. Если требуется надежная доставка дейтаграммы, она должна быть обеспечена на более высоком уровне. В случае приложений TCP и SCTP надежность обеспечивается транспортным уровнем. Приложению UDP надежность должно обеспечивать само приложение, поскольку уровень UDP также не предоставляет гарантии надежной доставки дейтаграмм, что было показано на примере в разделе 22.5.

Одной из наиболее важных функций уровня IP является *маршрутизация (routing)*. Каждая IP-дейтаграмма содержит адрес отправителя и адрес получателя. На рис. A.1 показан формат заголовка IPv4.

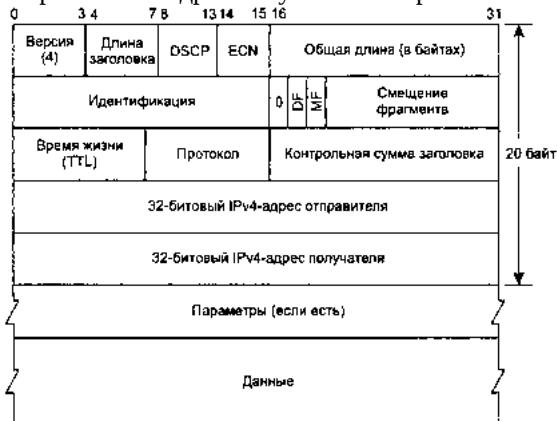


Рис. A.1. Формат заголовка IPv4

■ Значение 4-разрядного поля версия (version) равно 4. Это версия протокола IP, используемая с начала 80-х.

■ В поле длина заголовка (header length) указывается полная длина IP-заголовка, включающая любые параметры, описанные 32-разрядными словами. Максимальное значение этого 4-разрядного поля равно 15, и это значение задает максимальную длину IP-заголовка 60 байт. Таким образом, если заголовок занимает фиксированные 20 байт, то 40 байт остается на различные параметры.

■ 16-разрядное поле кода дифференцированных сервисов (Differentiated Services Code Point, DSCP) (RFC 2474 [82]) и 2-разрядное поле явного уведомления о загруженности сети (Explicit Congestion Notification, ECN) (RFC 3168 [100]) заменили 8-разрядное поле тип службы (сервиса) (type-of-service, TOS), которое описывалось в RFC 1349 [5]. Все 8 разрядов этого поля можно установить с помощью параметра сокета IP_TOS (см. раздел 7.6), хотя ядро может перезаписать любое установленное нами значение при проведении политики Diffserv или реализации ECN.

■ Поле общая длина (total length) имеет размер 16 бит и задает полную длину IP-дейтаграммы в байтах, включая заголовок IPv4. Количество данных в дейтаграмме равно значению этого поля минус длина заголовка, умноженная на 4. Данное поле необходимо, поскольку некоторые каналы передачи

данных заполняют кадр до некоторой минимальной длины (например, Ethernet) и возможна ситуация, когда размер действительной IP-дейтаграммы окажется меньше требуемого минимума.

- 16-разрядное поле *идентификации (identification)* является уникальным для каждой IP-дейтаграммы и используется при фрагментации и последующей сборке в единое целое (см. раздел 2.11). Значение должно быть уникальным для каждого сочетания отправителя, получателя и протокола в течение того времени, пока дейтаграмма может находиться в пути. Если пакет ни при каких условиях не может подвергнуться фрагментации (например, установлен бит DF), нет необходимости устанавливать значение этого поля.

- Бит DF (флаг запрета фрагментации), бит MF (указывающий, что есть еще фрагменты для обработки) и 13-разрядное поле *смещения фрагмента (fragment offset)* также используются при фрагментации и последующей сборке в единое целое. Бит DF полезен при обнаружении транспортной MTU (раздел 2.11).

- 8-разрядное поле *времени жизни (time-to-live, TTL)* устанавливается отправителем и уменьшается на единицу каждым последующим маршрутизатором, через который проходит дейтаграмма. Дейтаграмма отбрасывается маршрутизатором, который уменьшает данное поле до нуля. При этом время жизни любой дейтаграммы ограничивается 255 пересылками. Обычно по умолчанию данное поле имеет значение 64, но можно сделать соответствующий запрос и изменить его с помощью параметров сокета IP_TTL и IP_MULTICAST_TTL (см. раздел 7.6).

- 8-разрядное поле *протокола (protocol)* определяет тип данных, содержащихся в IP-дейтаграмме. Характерные значения этого поля — 1 (ICMPv4), 2 (IGMPv4), 6 (TCP) и 17 (UDP). Эти значения определены в реестре IANA «Номера протоколов».

- 16-разрядная *контрольная сумма заголовка (header checksum)* вычисляется для IP-заголовка (включая параметры). В качестве алгоритма вычисления используется стандартный алгоритм контрольных сумм для Интернета — простое суммирование 16-разрядных обратных кодов, как показано в листинге 28.11.

- Два поля — *IPv4-адрес отправителя (source IPv4 address)* и *IPv4-адрес получателя (destination IPv4 address)* — занимают по 32 бита.

- Поле *параметров (options)* описывается в разделе 27.2, а пример IPv4-параметра маршрута от отправителя приведен в разделе 27.3.

A.3. Заголовок IPv6

На рис. A.2 показан формат заголовка IPv6 (RFC 2460 [27]).

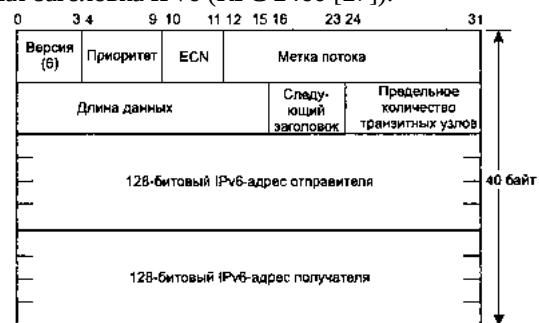


Рис. А.2. Формат заголовка IPv6

- Значение 4-разрядного поля *версии (version)* равно 6. Данное поле занимает первые 4 бита первого байта заголовка (так же как и в версии IPv4, см. рис. А.1), поэтому если получающий стек IP поддерживает обе версии, он имеет возможность определить, какая из версий используется.

Когда в начале 90-х развивался протокол IPv6 и еще не был принят номер версии 6, протокол назывался *IPng* (IP next generation — IP нового поколения). До сих пор можно встретить ссылки на IPng.

- 6-разрядное поле *кода дифференцированных сервисов (Differentiated Services Code Point, DSCP)* (RFC 2474 [82]) и 2-разрядное поле явного уведомления о загруженности сети (*Explicit Congestion Notification, ECN*) (RFC 3168 [100]) заменили 8-разрядное поле класса трафика, которое описывалось RFC 2460. Все 8 бит этого поля можно установить при помощи параметра сокета IPV6_TCLASS (раздел 22.8), но ядро может перезаписать установленное нами значение, выполняя политику Diffserv или реализуя ECN.

■ Поле *метки потока* (*flow label*) занимает 20 разрядов и может заполняться приложением для данного сокета. Поток представляет собой последовательность пакетов от конкретного отправителя определенному получателю, для которых отправитель потребовал специальную обработку промежуточными маршрутизаторами. Если для данного потока отправитель назначил метку, она уже не изменяется. Метка потока, равная нулю (по умолчанию), обозначает пакеты, не принадлежащие потоку. Метка потока не меняется при передаче по сети. Подробное описание использования меток потока приводится в [99]. Интерфейс метки потока еще не определен до конца. Поле `sin6_flowinfo` структуры адреса сокета `sockaddr_in6` (см. листинг 3.3) зарезервировано для будущего использования. Некоторые системы копируют младшие 28 разрядов `sin6_flowinfo` непосредственно в заголовок пакета IPv6, перезаписывая поля DSCP и ECN.

■ Поле *длины данных* (*payload length*) занимает 16 бит и содержит длину данных в байтах, которые следуют за 40 байтами IPv6-заголовка. Нулевое значение этого поля указывает, что длина требует больше 16 бит и содержится в параметре размера увеличенного поля данных (*jumbo payload length option*) (см. рис. 27.5). Данные с увеличенной таким образом длиной называются *джумбограммой* (*jumbogram*).

■ Следующее поле содержит 8 бит и называется *полем следующего заголовка* (*next header*). Оно аналогично полю протокола (*protocol*) IPv4. Действительно, когда верхний уровень в основном не меняется, используются те же значения, например, 6 для TCP и 17 для UDP. Но при переходе от ICMPv4 к ICMPv6 возникло так много изменений, что для последнего было принято новое значение 58. Дейтаграмма IPv6 может иметь множество заголовков, следующих за 40-байтовым заголовком IPv6. Поэтому поле и называется «*полем следующего заголовка*», а не *полем протокола*.

■ Поле *ограничения пересылок или предельного количества транзитных узлов* (*hop limit*) аналогично полю TTL IPv4. Значение этого поля уменьшается на единицу каждым маршрутизатором, через который проходит дейтаграмма, и дейтаграмма отбрасывается тем маршрутизатором, который уменьшает данное поле до нуля. Значение этого поля можно установить и получить с помощью параметров сокета `IPV6_UNICAST_HOPS` и `IPV6_MULTICAST_HOPS` (см. раздел 7.8 и 21.6). Параметр сокета `IPV6_HOPLIMIT` также позволяет установить это поле, а параметр `IPV6_RECVHOPLIMIT` — узнать его значение для полученной дейтаграммы.

ПРИМЕЧАНИЕ

В ранних спецификациях IPv4 говорилось, что маршрутизаторы должны уменьшать значение TTL либо на единицу, либо на количество секунд, в течение которых дейтаграмма находилась на маршрутизаторе, если это количество превышает единицу. Поэтому поле и называлось «*время жизни*». Однако на практике TTL всегда уменьшалось на единицу. IPv6 разрешает уменьшать поле количества транзитных узлов только на единицу, поэтому и название поля было изменено.

■ Два следующих поля *IPv6-адрес отправителя* (*source IPv6 address*) и *IPv6-адрес получателя* (*destination IPv6 address*) занимают по 128 бит.

Наиболее значительным изменением, произошедшим при переходе от IPv4 к IPv6, несомненно, является увеличение поля адресов в IPv6. Другое изменение относится к упрощению заголовка, поскольку чем проще заголовок, тем быстрее он будет обработан маршрутизатором. Кроме того, можно отметить еще несколько различий между заголовками:

■ В IPv6 нет поля длины заголовка, поскольку в заголовке отсутствуют параметры. Существует возможность использовать после фиксированного 40-байтового заголовка дополнительные заголовки, но каждый из них имеет свое поле длины.

■ Два адреса IPv6 выровнены по 64-разрядной границе, если заголовок также является 64-разрядным. Такой подход может увеличить скорость обработки на 64-разрядных архитектурах. Адреса IPv4 имеют 32-разрядное выравнивание в заголовке IPv4, который в целом выровнен по 64 разрядам.

■ В заголовке IPv6 нет поля фрагментации, поскольку для этой цели существует специальный заголовок фрагментации. Такое решение было принято, поскольку фрагментация является исключением, а исключения не должны замедлять нормальную обработку.

■ Заголовок IPv6 не включает в себя свою контрольную сумму. Такое изменение было сделано, поскольку все верхние уровни — TCP, UDP и ICMPv6 — имеют свои контрольные суммы, включающие в себя заголовок верхнего уровня, данные верхнего уровня и такие поля из IPv6-заголовка, как IPv6-адрес

отправителя, IPv6-адрес получателя, длину данных и следующий заголовок. Исключив контрольную сумму из заголовка, мы приходим к тому, что маршрутизатор, перенаправляющий пакет, не должен будет пересчитывать контрольную сумму заголовка после того, как изменит поле ограничения пересылок. Ключевым моментом здесь также является скорость маршрутизации.

Если это ваше первое знакомство с IPv6, также следует отметить главные отличия IPv6 от IPv4:

- В IPv6 отсутствует многоадресная передача (см. главу 20). Групповая адресация (см. главу 21), не являющаяся обязательной для IPv4, требуется для IPv6.
- В IPv6 маршрутизаторы не фрагментируют перенаправляемые пакеты. Если пакет слишком велик, маршрутизатор сбрасывает его и отправляет сообщение об ошибке ICMPv6 (раздел А.6). Фрагментация при использовании IPv6 осуществляется только узлом отправителя.
- IPv6 требует поддержки обнаружения транспортной MTU (раздел 2.11). Технически эта поддержка не является обязательной и может не включаться в реализации, обладающие минимальной функциональностью, такие как сетевые загрузчики, но если узел не обнаруживает транспортную MTU, он не должен отсылать дейтаграммы, размер которых превышает минимальную канальную MTU IPv6 (1280 байт). В разделе 22.9 описываются параметры сокетов, управляющие поведением механизма обнаружения транспортной MTU.
- IPv6 требует поддержки параметра аутентификации (подтверждения прав доступа) и параметра обеспечения безопасности. Эти параметры добавляются после основного заголовка.

A.4. Адресация IPv4

Адреса IPv4 состоят из 32 разрядов и обычно записываются в виде последовательности из четырех чисел в десятичной форме, разделенных точками. Такая запись называется *точечно-десятичной*. Первое из четырех чисел определяет тип адреса (табл. А.1). Исторически IP-адреса делились на пять классов. Три класса направленных адресов эквивалентны друг другу с функциональной точки зрения, поэтому мы показываем их как один диапазон.

Таблица А.1. Диапазоны и классы IP-адресов

Назначение	Класс	Диапазон
Направленная передача	A, B, C	0.0.0.0–223.255.255.255
Многоадресная передача	D	224.0.0.0–239.255.255.255
Экспериментальные	E	240.0.0.0–255.255.255.255

Под сетевым адресом IPv4 подразумевается 32-разрядный адрес и соответствующая ему 32-разрядная маска подсети. Биты маски, равные 1, указывают адрес сети, а нулевые биты — адрес узла. Поскольку биты со значением 1 всегда занимают места в маске непрерывно начиная с крайнего левого бита, а нулевые биты — начиная с крайнего правого бита, то маску адреса можно определить как *префиксную длину* (*prefix length*), указывающую на количество заполненных единицами битов начиная с крайнего левого бита. Например, маска 255.255.255.0 соответствует префиксной длине 24. Такая адресация называется бесклассовой (*classless*), потому что маска указывается явно, а не задается классом адреса. Пример вы можете увидеть на рис. 1.7.

ПРИМЕЧАНИЕ

Маски подсети, не являющиеся непрерывными, не были явно запрещены ни в одном RFC, но такие маски усложняют работу администраторов и не могут быть представлены в префиксной записи. Протокол междоменной маршрутизации Интернета BGP4 может работать только с непрерывными масками. В протоколе IPv6 требование непрерывности маски выдвигается явно.

Использование бесклассовых адресов подразумевает бесклассовую маршрутизацию, которую обычно называют бесклассовой междоменной маршрутизацией (*classless interdomain routing* — CIDR) (RFC 1519 [31]). Бесклассовая междоменная маршрутизация позволяет сократить размер таблиц маршрутизации опорной сети Интернета и снизить скорость расходования адресов IPv4. Все маршруты CIDR характеризуются маской или длиной префикса. Мaska больше не может быть определена по классу адреса. Более подробно CIDR описывается в разделе 10.8 книги [111].

Адреса подсетей

Обычно IPv4-адреса разделяются на подсети (RFC 950 [79]). Такой подход добавляет еще один уровень иерархии адресов:

- идентификатор сети (присваивается предприятию);
- идентификатор подсети (выбирается предприятием);
- идентификатор узла (выбирается предприятием).

Граница между идентификатором сети и идентификатором подсети фиксирована префиксной длиной присвоенного адреса сети. Эта префиксная длина присваивается организациям их интернет-провайдером. Граница же между идентификатором подсети и идентификатором узла выбирается предприятием. Все узлы данной подсети имеют одинаковую *маску подсети*, которая и определяет границу между идентификатором подсети и идентификатором узла. Биты, заполненные единицами в маске подсети, соответствуют идентификатору подсети, а биты, заполненные нулями — идентификатору узла.

В качестве примера рассмотрим предприятие, которому был выделен адрес 192.168.42.0/24. Если это предприятие будет использовать 3-разрядный идентификатор подсети, на идентификатор узла останется 5 разрядов (рис. А.3.).

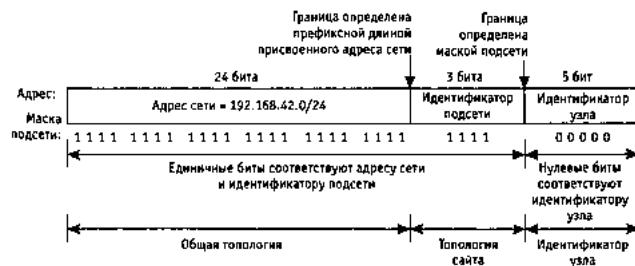


Рис. А.3. 24-разрядный адрес сети с 3-разрядным адресом подсети и 5-разрядным адресом узла
В результате такого деления мы получаем подсети, показанные в табл. А.2.

Таблица А.2. Список подсетей для 3-разрядного адреса подсети и 5-разрядного адреса узла

Подсеть Префикс

0	192.168.42.0/27+
1	192.168.42.32/27+
2	192.168.42.64/27
3	192.168.42.96/27
4	192.168.42.128/27
5	192.168.42.160/27
6	192.168.42.192/27
7	192.168.42.224/27+

В результате мы получаем 6–8 подсетей (идентификаторы 1–6 или 0–7), в каждой из которых может находиться до 30 узлов (идентификаторы 1–30). RFC 950 не рекомендует использовать подсети, идентификаторы которых состоят из одних нулей и одних единиц (знак «+» в табл. А.2). В настоящее время большинство систем поддерживают такие адреса подсетей. Максимальный идентификатор узла (в нашем случае 31) зарезервирован за широковещательным адресом. Идентификатор 0 используется для адресации сети в целом и зарезервирован во избежание конфликтов со старыми системами, в которых нулевой адрес узла использовался в качестве адреса широковещательной передачи. В полностью контролируемых сетях, где такие системы отсутствуют, идентификатор 0 использовать можно. Вообще говоря, сетевые приложения не должны заботиться об идентификаторах подсетей и узлов, рассматривая IP-адреса как непрозрачные объекты.

Адрес закольцовки

По соглашению адрес 127.0.0.1 присвоен *интерфейсу закольцовки на себя (loopback interface)*. Все, что посыпается на этот IP-адрес, получается самим узлом. Обычно этот адрес используется при тестировании клиента и сервера на одном узле. Этот адрес известен под именем INADDR_LOOPBACK.

ПРИМЕЧАНИЕ

Любой адрес из подсети 127/8 можно присвоить интерфейсу закольцовки, но обычно используется именно 127.0.0.1.

Неопределенный адрес

Адрес, состоящий из 32 нулевых битов, является в IPv4 *неопределенным (unspecified)* адресом. В пакете IPv4 он может появиться только как адрес получателя в тех пакетах, которые посланы узлом, находящимся в состоянии загрузки, когда узел еще не знает своего IP-адреса. В API сокетов этот адрес называется *универсальным адресом (wildcard address)* и обычно обозначается `INADDR_ANY`. Указание этого адреса при вызове `bind` для прослушиваемого сокета TCP говорит о том, что сокет будет принимать входящие соединения на любой адрес данного узла.

Частные адреса

RFC 1918 [101] выделяет три диапазона адресов для «частных интрасетей», то есть сетей, не имеющих прямого подключения к Интернету. Эти диапазоны представлены в табл. А.3.

Таблица А.3. Диапазоны частных IP-адресов

Количество адресов	Префикс	Диапазон
16777216	10/8	10.0.0.0–10.255.255.255
1 048 576	172.16/12	172.16.0.0–172.31.255.255
65 536	192.168/16	192.168.0.0–192.168.255.255

Пакеты с этими адресами никогда не должны появляться в Интернете, они зарезервированы для использования в частных сетях. Многие небольшие предприятия используют эти адреса и осуществляют трансляцию сетевых адресов в единственный общий IP-адрес, видимый из Интернета.

Многоинтерфейсность и псевдонимы адресов

Традиционно многоинтерфейсный узел определяется как узел с несколькими интерфейсами, например узел, имеющий два интерфейса Ethernet или интерфейсы Ethernet и PPP. Каждый из интерфейсов должен иметь свой уникальный IPv4-адрес. При подсчете интерфейсов (для определения, является ли узел многоинтерфейсным) интерфейс закольцовки не учитывается.

Маршрутизатор по определению является многоинтерфейсным, поскольку он пересыпает пакеты, поступившие на один интерфейс, через другой интерфейс. Но обратное неверно, то есть многоинтерфейсный узел не является маршрутизатором, если он не передает пакеты. Действительно, многоинтерфейсный узел еще не может рассматриваться как маршрутизатор. Он будет функционировать как маршрутизатор, только если он сконфигурирован для такой работы (обычно администратор должен включить соответствующие параметры конфигурации).

Термин «многоинтерфейсность» является более общим и охватывает два различных сценария (раздел 3.3.4 RFC 1122 [10]).

1. Узел с несколькими интерфейсами является многоинтерфейсным, при этом каждый интерфейс должен иметь свой IP-адрес. Это традиционное определение.

2. Современные узлы имеют возможность присваивать одному физическому интерфейсу несколько IP-адресов. Каждый IP-адрес, созданный в дополнение к первичному, или основному (*primary*), называется *альтернативным именем, псевдонимом (alias)* или *логическим интерфейсом*. Часто альтернативные IP-адреса используют ту же маску подсети, что и основной адрес, но имеют другие идентификаторы узла. Но допустима также ситуация, когда псевдонимы имеют адрес сети или подсети, совершенно отличный от первичного адреса. В разделе 17.6 приведен пример альтернативных адресов.

Таким образом, многоинтерфейсные узлы — это узлы, имеющие несколько интерфейсов IP-уровня, независимо от того, являются ли эти интерфейсы физическими или логическими.

ПРИМЕЧАНИЕ

Довольно часто загруженные серверы имеют несколько соединений с одним коммутатором Ethernet, причем эти соединения настраиваются как одно логическое соединение с повышенной пропускной способностью. Такая система имеет несколько физических интерфейсов, но не считается многоинтерфейсной, поскольку обладает **одним-единственным логическим интерфейсом** с точки зрения уровня IP.

ПРИМЕЧАНИЕ

Многоинтерфейсность также используется в другом контексте. Сеть, имеющая несколько соединений с сетью Интернет, также называется многоинтерфейсной. Например, некоторые сайты имеют два соединения с Интернетом вместо одного, что обеспечивает дублирование на случай неполадок. Транспортный протокол SCTP позволяет передавать информацию о количестве интерфейсов узла его собеседнику.

A.5. Адресация IPv6

Адреса IPv6 содержат 128 бит и обычно записываются как восемь 16-разрядных шестнадцатеричных чисел. Старшие биты 128-разрядного адреса обозначают тип адреса (RFC 3513 [44]). В табл. A.4 приведены различные значения старших битов и соответствующие им типы адресов.

Таблица A.4. Значение старших битов адреса IPv6

Значение	Размер идентификатора	Префикс формата	Документ
Не определен	нет	0000 0000 ... 0000 0000 (128 разрядов)	RFC 3513
Закольцовка	нет	0000 0000 ... 0000 0001 (128 разрядов)	RFC 3513
Глобальный адрес направленной передачи	произвольный	000	RFC 3513
Глобальный адрес NSAP	произвольный	0000001	RFC 1888
Объединяемый глобальный адрес направленной передачи	64 разряда	001	RFC 3587
Глобальный адрес направленной передачи	64 разряда	все остальное	RFC 3513
Локальный в пределах канала адрес направленной передачи	64 разряда	1111 111010	RFC 3513
Локальный в пределах сайта адрес направленной передачи	64 разряда	1111 111011	RFC 3513
Групповой адрес	нет	1111 1111	RFC 3513

Эти старшие биты называются **форматным префиксом**. Например, если 3 старших бита — 001, адрес называется **объединяемым глобальным индивидуальным адресом** (*aggregatable global unicast address*). Если 8 старших битов — 11111111 (0xff), это групповой адрес.

Объединяемые глобальные индивидуальные адреса

Архитектура IPv6 корректировалась в процессе своего развития исходя из результатов внедрения новой версии протокола и из статистики применения старой версии. Согласно изначальному определению объединяемых глобальных индивидуальных адресов, они начинались с префикса 001 и имели фиксированную структуру,строенную в сам адрес. Эта структура, однако, была отменена RFC 3587 [45]. Адреса, начинающиеся с префикса 001, будут и впредь выделяться в первую очередь, однако никаких отличий между ними и другими глобальными адресами больше не будет. Эти адреса будут использоваться в тех областях, где сейчас используются направленные адреса IPv4.

Формат объединяемых индивидуальных адресов определяется в RFC 3513 [44] и RFC 3587 [45] и содержит следующие поля, слева направо:

- глобальный префикс маршрутизации (n разрядов);
- идентификатор подсети (64 - n разрядов);
- идентификатор интерфейса (64 разряда).

На рис. А.4 приведен пример объединяемого глобального индивидуального адреса.



Рис. А.4. Объединяемый глобальный индивидуальный адрес IPv6

Идентификатор интерфейса должен быть построен в модифицированном формате EUI-64 (Extended User Interface — расширенный интерфейс пользователя) [51]. Это расширение множества 48-разрядных адресов IEEE 802 MAC (Media Access Control — уровень управления доступом к среде передачи), которые присвоены большинству карт сетевых интерфейсов локальной сети. Этот идентификатор должен автоматически присваиваться интерфейсу и по возможности основываться на MAC-адресе карты. Более подробное описание построения идентификаторов интерфейса, основанных на EUI-64, описывается в приложении А RFC 3513 [44].

Поскольку модифицированный адрес EUI-64 может быть глобально уникальным идентификатором интерфейса, а сам интерфейс может однозначно идентифицировать пользователя, модифицированный формат EUI-64 создает определенные проблемы, связанные с конфиденциальностью. Может оказаться возможным отслеживать действия и перемещение конкретного пользователя, например путешествующего с портативным компьютером, просто по его IPv6-адресу. RFC 3041 [80] описывает расширения протокола, предназначенные для генерации идентификаторов интерфейса, меняющихся по несколько раз в день и, таким образом, устраниющие описанную проблему.

Тестовые адреса 6bone

6bone — это виртуальная сеть, используемая для тестирования протоколов IPv6 (см. раздел Б.3). Объединяемые глобальные индивидуальные адреса уже назначаются, но сайты, не имеющие права на адресное пространство согласно региональной политике назначения адресов, могут использовать адреса специального формата в сети 6bone RFC 2471 [46] (рис. А.5).

3ffe	Идентификатор сайта 6bone	Идентификатор подсети	Идентификатор интерфейса
16 разрядов	32 разряда	16 разрядов	64 разряда

Рис. А.5. Тестовые адреса IPv6 для сети 6bone

Эти адреса рассматриваются как временные, и узлы, использующие такие адреса, необходимо будет перенумеровать, когда будут назначены объединяемые глобальные индивидуальные адреса.

Старшие три байта имеют значение 0x3ffe. Идентификатор сайта 6bone назначается председателем руководства 6bone. Назначение проводится в том же порядке, в котором оно будет проводиться для реальных адресов IPv6. Активность 6bone постепенно сворачивается по мере того, как начинается внедрение IPv6 (в 2002 году было выделено больше реальных адресов IPv6, чем во всей сети 6bone за 8 лет). Идентификаторы подсети и интерфейса используются, как и раньше, для обозначения подсети и узла.

В разделе 11.2 был показан IPv6-адрес 3ffe:b80:1f8d:1:a00:20ff:fea7:686b для узла freebsd (см. рис. 1.7). Идентификатор 6bone имеет значение 0x0b801f8d, а идентификатор подсети 0x1. Младшие 64 разряда представляют собой модифицированный адрес EUI-64, полученный из MAC-адреса Ethernet-карты узла.

Адреса IPv4, преобразованные к виду IPv6

Адреса IPv4, преобразованные к виду IPv6 (IPv4-mapped IPv6 addresses), позволяют приложениям, запущенным на узлах, поддерживающих как IPv4, так и IPv6, связываться с узлами, поддерживающими только IPv4, в процессе перехода сети Интернет на версию протокола IPv6. Такие адреса автоматически

создаются на серверах DNS (см. табл. 11.3), когда приложением IPv6 запрашивается IPv6-адрес узла, который имеет только адреса IPv4.

Рисунок 12.3 показывает, что использование данного типа адресов с сокетом IPv6 приводит к отправке IPv4-дейтаграммы узлу. Такие адреса не хранятся ни в каких файлах данных DNS — при необходимости они создаются сервером.



Рис. А.6. Адреса IPv4, преобразованные к виду IPv6

На рис. А.6 приведен формат таких адресов. Младшие 32 бита содержат адрес IPv4.

При записи IPv6-адреса последовательная строка из нулей может быть сокращена до двух двоеточий. Вложенный IPv4-адрес представлен в точечно-десятичной записи. Например, преобразованный к виду IPv6 IPv4-адрес 0:0:0:0:FFFF:206.62.226.33 можно сократить до ::FFFF:206.62.226.33.

Адреса IPv6, совместимые с IPv4

Для перехода от версии IPv4 к IPv6 планировалось также использовать адреса IPv6, совместимые с IPv4 (IPv4-compatible IPv6 addresses). Администратор узла, поддерживающего как IPv4, так и IPv6, и не имеющего соседнего IPv6-маршрутизатора, должен создать DNS запись типа AAAA, содержащую адрес IPv6, совместимый с IPv4. Любой другой IPv6-узел, посылающий IPv6-дейтаграмму на адрес IPv6, совместимый с IPv4, должен упаковать (*encapsulate*) IPv6-дейтаграмму в заголовок IPv4 — такой способ называется *автоматическим туннелированием* (*automatic tunnel*). Однако после рассмотрения вопросов, связанных с внедрением IPv6, использование этой возможности заметно сократилось. Более подробно вопросы туннелирования будут рассмотрены в разделе Б.3, а на рис. Б.2 будет приведен пример IPv6-дейтаграмм такого типа, упакованных в заголовок IPv4.

На рис. А.7 показан формат адреса IPv4, совместимого с IPv6.

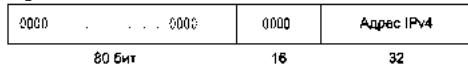


Рис. А.7. Адрес IPv6, совместимый с IPv4

В качестве примера такого адреса можно привести ::206.62.226.33.

Адреса IPv6, совместимые с IPv4 могут появляться и в пакетах IPv6, не передающихся по туннелю, если используется механизм перехода SIIT IPv4/IPv6 (RFC 2765 [83]).

Адрес закольцовки

Адрес IPv6 ::1, состоящий из 127 нулевых битов и единственного единичного бита, является адресом закольцовки IPv6. В API сокетов он называется `in6addr_lopback` или `IN6ADDR_LOOPBACK_INIT`.

Неопределенный адрес

Адрес IPv6, состоящий из 128 нулевых битов, записываемый как 0::0 или просто ::, является *неопределенным адресом IPv6* (*unspecified address*). В пакете IPv6 он может появиться только как адрес получателя в пакетах, посланных узлом, который находится в состоянии загрузки и еще не знает своего IPv6-адреса.

В API сокетов этот адрес называется универсальным адресом, и его использование, например, в функции `bind` для связывания прослушиваемого сокета TCP означает, что сокет будет принимать клиентские соединения, предназначенные любому из адресов узла. Этот адрес имеет имя `in6addr_anу` или `IN6ADDR_ANY_INIT`.

Адрес локальной связи

Адрес локальной связи (*link-local*, локальный в пределах физической подсети) используется для соединения в пределах одной физической подсети, когда известно, что дейтаграмма не будет перенаправляться. Примерами использования таких адресов являются автоматическая конфигурация

адреса во время загрузки и поиска соседних узлов (neighbor discovery) (подобно ARP для IPv4). На рис. A.8 приведен формат такого адреса.

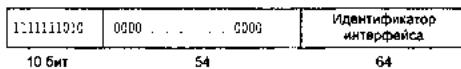


Рис. А.8. IPv6-адрес локальной связи

Такие адреса всегда начинаются с fe80. Маршрутизатор IPv6 не должен перенаправлять дейтаграммы, у которых в поле отправителя или получателя указан адрес локальной связи, по другому соединению. В разделе 11.2 приведен адрес локальной связи, связанный с именем aix-611.

Адрес, локальный на уровне сайта

На момент написания этой книги рабочей группой IETF по IPv6 было принято решение отменить локальные в пределах сайта адреса в их текущей форме. В тех адресах, которые придут им на замену, может использоваться тот же диапазон, который был отведен для локальных на уровне сайта адресов изначально (fec0/10).

Адрес, локальный в пределах сайта, должен был использоваться для адресации внутри предприятия, когда не требуется глобальный префикс. На рис. А.9 показан формат таких адресов.

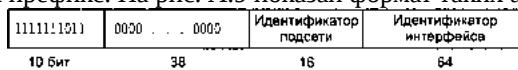


Рис. А.9. IPv6-адрес, локальный в пределах сайта

Маршрутизатор IPv6 не должен перенаправлять дейтаграммы, для которых в поле отправителя или получателя указан такой адрес, за пределы предприятия.

A.6. ICMPv4 и ICMPv6: протоколы управляющих сообщений в сети Интернет

Протокол ICMP (Internet Control Message Protocol) является необходимой и неотъемлемой частью любой реализации IPv4 или IPv6. Протокол ICMP обычно используется для обмена сообщениями об ошибках между узлами, как маршрутизирующими, так и обычными, но иногда этот протокол используется и приложениями. Например, приложения ping и traceroute (см. главу 28) используют протокол ICMP.

Первые 32 бита сообщений совпадают для ICMPv4 и ICMPv6 и приведены на рис. А.10. ICMPv4 документируется в RFC 792 [95], а ICMPv6 — в RFC 2463 [21].

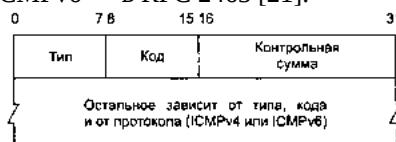


Рис. А.10. Формат сообщений ICMPv4 и ICMPv6

Восьмиразрядное поле *тип* (*type*) указывает тип сообщения ICMPv4 или ICMPv6, а некоторые типы имеют дополнительную 8-разрядную информацию, указанную в поле *кода* (*code*). Поле *контрольной суммы* (*checksum*) является стандартной контрольной суммой, используемой в сети Интернет. Отличия между ICMPv4 и ICMPv6 заключаются в том, какие именно поля используются при подсчете контрольной суммы.

С точки зрения сетевого программирования необходимо понимать, какие сообщения ICMP могут быть возвращены приложению, что именно вызывает ошибку и каким образом эта ошибка возвращается приложению. В табл. А.5 приведены все сообщения ICMPv4 и показано, как они обрабатываются операционной системой 4.4BSD. В последнем столбце приведены значения переменной *errno* — то есть те ошибки, которые возвращаются приложением. В табл. А.6 приведен список сообщений ICMPv6. При использовании TCP ошибка не возвращается приложению немедленно. Если TCP разрывает соединение по тайм-ауту, все накопленные ошибки возвращаются приложению. При использовании UDP ошибка возвращается при очередной операции чтения или записи, но только на присоединенном сокете (раздел 8.9).

Таблица А.5. Обработка различных типов ICMP-сообщений в 4.4BSD

Тип	Код	Описание	Обработчик или errno
0	0	Echo-reply (Эхо-ответ)	Пользовательский процесс (Ping)

	Destination unreachable (Получатель недоступен)	
0	Network unreachable (Сеть недоступна)	EHOSTUNREACH
1	Host unreachable (Узел недоступен)	EHOSTUNREACH
2	Protocol unreachable (Протокол недоступен)	ECONNREFUSED
3	Port unreachable (Порт недоступен)	ECONNREFUSED
4	Fragmentation needed but DF bit set (Необходима фрагментация, но установлен бит DF)	EMSGSIZE
5	Source route failed (Сбой маршрута отправителя)	EHOSTUNREACH
6	Destination network unknown (Неизвестна сеть получателя)	EHOSTUNREACH
7	Destination host unknown (Неизвестен узел получателя)	EHOSTUNREACH
3	8 Source host isolated (Узел отправителя изолирован). Устаревший тип сообщений	EHOSTUNREACH
9	Destination network administratively prohibited (Сеть получателя запрещена администратором)	EHOSTUNREACH
10	Destination host administratively prohibited (Узел получателя запрещен администратором)	EHOSTUNREACH
11	Network unreachable for TOS (Сеть недоступна для TOS)	EHOSTUNREACH
12	Host unreachable for TOS (Узел недоступен для TOS)	EHOSTUNREACH
13	Communication administratively prohibited (Связь запрещена администратором)	(Игнорируется)
14	Host precedence violation (Нарушение порядка старшинства узлов)	(Игнорируется)
15	Precedence cutoff in effect (Действует старшинство узлов)	(Игнорируется)
4	0 Source quench (Отключение отправителя)	Обрабатывается ядром в случае TCP, игнорируется в случае UDP
	Redirect (Перенаправление)	
5	0 Redirect for network (Перенаправление для сети)	Ядро обновляет таблицу маршрутизации
1	Redirect for host (Перенаправление для узла)	Ядро обновляет таблицу маршрутизации
2	Redirect for type-of-service and network (Перенаправление для типа сервиса и сети)	Ядро обновляет таблицу маршрутизации
3	Redirect for type of service and host (Перенаправление для типа сервиса и узла)	Ядро обновляет таблицу маршрутизации
8	0 Echo request (Эхо-запрос)	Ядро генерирует ответ
9	0 Router advertisement (Извещение маршрутизатора)	Пользовательский процесс
10	0 Router solicitation (Запрос маршрутизатору)	Пользовательский процесс
	Time exceeded (Превышено время передачи)	
11	0 TTL equals 0 during transit (Время жизни равно 0 во время передачи)	Пользовательский процесс
1	TTL equals 0 during reassembly (Время жизни равно 0 во время сборки)	Пользовательский процесс
	Parameter problem (Проблема с параметром)	
12	0 IP header bad (Неправильный IP-заголовок). Типичная ошибка	ENOPROTOOPT
1	Required option missed (Пропущен необходимый параметр)	ENOPROTOOPT
13	0 Timestamp request (Запрос отметки времени)	Ядро генерирует ответ

14	0	Timestamp reply (Ответ об отметке времени)	Пользовательский процесс
15	0	Information request (Информационный запрос). Устаревший тип сообщений	(игнорируется)
16	0	Information reply (Информационный ответ). Устаревший тип сообщений	Пользовательский процесс
17	0	Address mask request (Запрос маски адреса)	Ядро генерирует ответ
18	0	Address mask reply (Ответ маски адреса)	Пользовательский процесс

Таблица A.6. Сообщения ICMPv6

Тип	Код	Описание	Обработчик или errno
1	Administratively prohibited, firewall filter (Запрещено администратором, фильтр брандмауэра)		EHOSTUNREACH
2	Not a neighbor, incorrect strict source route (Не сосед, некорректный маршрут отправителя)		EHOSTUNREACH
3	Address unreachable (Адрес недоступен)		EHOSTDOWN
4	Port unreachable (Порт недоступен)		ECONNREFUSED
2	0	Packet too big (Слишком большой пакет)	Ядро выполняет обнаружение транспортной MTU
		Time exceeded (Превышено время передачи)	
3	0	Hop limit exceeded in transit (При передаче превышено значение предельного количества транзитных узлов)	Пользовательский процесс
	1	Fragment reassembly time exceeded (Истекло время сборки из фрагментов)	Пользовательский процесс
		Parameter problem (Проблема с параметром)	
4	0	Erroneous header filed (Ошибка в поле заголовка)	ENOPROTOOPT
	1	Unrecognized next header (Следующий заголовок нераспознаваем)	ENOPROTOOPT
	2	Unrecognized option (Неизвестный параметр)	ENOPROTOOPT
128	0	Echo request (Эхо-запрос (Ping))	Ядро генерирует ответ
129	0	Echo reply (Эхо-ответ (Ping))	Пользовательский процесс (Ping)
130	0	Group membership query (Запрос о членстве в группе)	Пользовательский процесс
131	0	Group membership report (Отчет о членстве в группе)	Пользовательский процесс
132	0	Group membership reduction (Сокращение членства в группе)	Пользовательский процесс
133	0	Router solicitation (Запрос маршрутизатору)	Пользовательский процесс
134	0	Router advertisement (Извещение маршрутизатора)	Пользовательский процесс
135	0	Neighbor solicitation (Запрос соседу)	Пользовательский процесс
136	0	Neighbor advertisement (Извещение соседа)	Пользовательский процесс
137	0	Redirect (Перенаправление)	Ядро обновляет таблицу маршрутизации

Запись «пользовательский процесс» в этой таблице означает, что ядро не обрабатывает сообщение и ждет обработки данного сообщения от пользовательского процесса с символьным сокетом. Также следует отметить, что различные реализации могут обрабатывать одни и те же сообщения по-разному. Например, в Unix сообщения типа Router solicitation (Запрос маршрутизатору) и Router advertisement (Извещение маршрутизатора) обычно обрабатываются как пользовательские процессы, но некоторые реализации могут обрабатывать эти сообщения в ядре.

Версия ICMPv6 сбрасывает старший бит поля тип для сообщения об ошибке (типы 1-4) и устанавливает этот бит для информационного сообщения (типы 128–137).

Приложение Б

Виртуальные сети

Б.1. Введение

Поддержка новых возможностей протокола TCP, например каналов с повышенной пропускной способностью (RFC 1323), требуется только на узле, использующем TCP, тогда как маршрутизаторы в модернизации не нуждаются. Эти изменения, описанные в RFC 1323, постепенно проявляются в реализациях TCP на узлах. Когда устанавливается новое TCP-соединение, каждая сторона может определить, поддерживает ли другая сторона новую возможность, и если для обоих узлов это так, ею можно воспользоваться.

Иная ситуация с изменениями IP-уровня, такими как многоадресная передача, появившаяся в конце 80-х, или новая версия протокола IPv6, возникшая в середине 90-х, поскольку они требуют изменений на всех узлах и на всех маршрутизаторах. Но люди хотят начать использовать новые возможности, не дожидаясь, когда все системы будут модернизированы. Для этого существующий протокол IPv4 был дополнен так называемыми *виртуальными сетями* (*virtual network*), использующими *туннели* (*tunnels*).

Б.2. MBone

Наш первый пример виртуальной сети, построенной с использованием туннелей, — это сеть MBone, которая начала использоваться примерно с 1992 года [29]. Если два или более узла в локальной сети поддерживают многоадресную передачу, то на всех этих узлах могут быть запущены приложения многоадресной передачи, которые могут общаться друг с другом. Для соединения одной локальной сети с другой локальной сетью, также содержащей узлы с возможностью многоадресной передачи, между двумя узлами из этих сетей конфигурируется туннель, как показано на рис. Б.1. На этом рисунке отмечены следующие шаги:

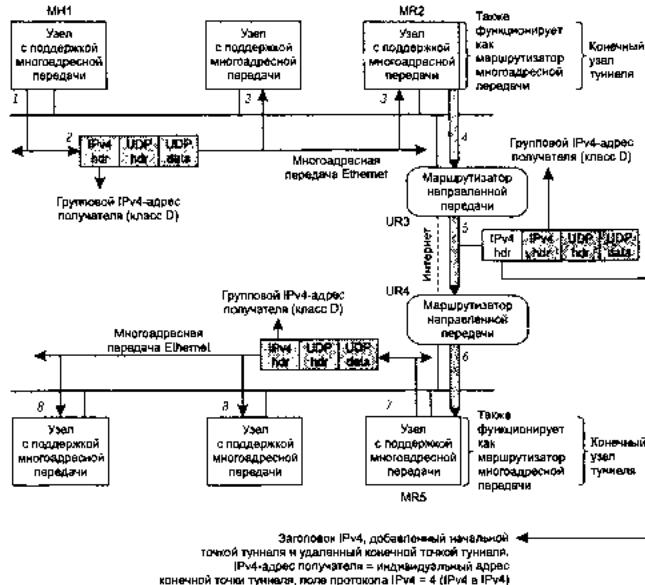


Рис. Б.1. Упаковка IPv4 в IPv4, применяемая в MBone

1. Приложение на узле отправителя MH1 посыпает групповуюдейтаграмму адресам класса D.
2. На рисунке эта дейтаграмма показана как UDP-дейтаграмма, поскольку большинство приложений многоадресной передачи используют протокол UDP. Более подробно о многоадресной передаче и о том, как посыпать и получать многоадресные дейтаграммы, рассказано в главе 21.
3. Дейтаграмма принимается всеми узлами в локальной сети, поддерживающими многоадресную передачу, в том числе и MR2. Отметим, что MR2 также работает как многоадресный маршрутизатор, на котором запущена программа *mrouted*, осуществляющая маршрутизацию многоадресной передаче.

4. MR2 добавляет перед дейтаграммой другой IPv4-заголовок, в котором в поле адреса получателя записан индивидуальный адрес конечного узла туннеля (tunnel endpoint) MR. Этот индивидуальный адрес конфигурируется администратором узла MR2 и считывается программой `mouted` при ее запуске. Аналогичным образом индивидуальный адрес узла MR2 сконфигурирован на узле MR — на другом конце туннеля. В поле протокола нового IPv4-заголовка установлено значение 4, соответствующее упаковке IPv4 в IPv4. Дейтаграмма посыпается следующему маршрутизатору, UR3, который явно указан как маршрутизатор направленной передачи, то есть не поддерживает многоадресную передачу, и поэтому приходится использовать туннель. Выделенная на рисунке серым цветом часть IPv4-дейтаграммы не изменяется по сравнению шагом 1, только значение поля TTL в выделенном цветом IPv4-заголовке уменьшается на 1.

5. UR3 узнает адрес получателя из самого внешнего IPv4-заголовка и перенаправляет дейтаграмму следующему маршрутизатору направленной передачи — UR4.

6. UR4 доставляет дейтаграмму по назначению — узлу MR, который является конечным узлом туннеля.

7. MR получает дейтаграмму, и поскольку в поле протокола указана упаковка IPv4 в IPv4, удаляет внешний IPv4-заголовок и передает оставшуюся часть дейтаграммы (копию той, которая была групповой дейтаграммой в локальной сети, изображенной на рисунке вверху) в качестве многоадресной дейтаграммы в своей локальной сети.

8. Все узлы сети, изображенной на рисунке внизу, поддерживающие многоадресную передачу, получают многоадресную дейтаграмму.

В результате многоадресная дейтаграмма, отправленная в локальной сети, изображенной вверху, передается как многоадресная дейтаграмма в локальной сети, изображенной внизу. Это происходит несмотря на то, что два маршрутизатора, присоединенные к этим двум локальным сетям, а также все маршрутизаторы между ними не поддерживают многоадресную передачу.

В данном примере показана функция маршрутизации многоадресной передачи, осуществляемая программой `mouted`, запущенной на одном из узлов в каждой из локальных сетей. Таким образом начинала свою работу сеть MBone. Но, начиная примерно с 1996 г. большинство основных поставщиков маршрутизаторов стали включать функцию групповой маршрутизации в свои маршрутизаторы. Если бы два маршрутизатора направлена передачи UR3 и UR4 на рис. Б.1 имели возможность маршрутизации многоадресной передачи, то нам не пришлось бы запускать `mouted`, а маршрутизаторы UR3 и UR4 работали бы как маршрутизаторы многоадресной передачи. Но если между UR3 и UR4 существуют другие маршрутизаторы, не поддерживающие многоадресную передачу, туннель все же необходим. Однако конечными пунктами туннеля в этом случае могут стать MR3 (новое имя для UR3, поддерживающего многоадресную передачу) и MR4 (новое имя для UR4, поддерживающего многоадресную передачу), а не MR2 и MR.

ПРИМЕЧАНИЕ

В сценарии, приведенном на рис. Б.1, каждый многоадресный пакет появляется дважды в локальной сети, расположенной вверху рисунка, и дважды в локальной сети, расположенной внизу. Один раз это многоадресный пакет, а второй раз — направленный пакет внутри туннеля, так как пакет идет между узлом, на котором запущена программа `mouted`, и следующим маршрутизатором направленной передачи (то есть между MR2 и UR3, а затем между UR4 и MR). Лишняя копия — это цена туннелирования. Преимущество замены маршрутизаторов направленной передачи UR3 и UR4 на рис. Б.1 на маршрутизаторы многоадресной передачи (те, что мы назвали MR3 и MR4) заключается в том, что мы избежали появления этой дополнительной копии многоадресного пакета в каждой из сетей. Даже если MR3 и MR4 должны установить туннель между собой, поскольку некоторые промежуточные маршрутизаторы между ними (которые на рисунке не показаны) не поддерживают многоадресную передачу, такой вариант предпочтительнее, так как в этом случае не происходит дублирования пакетов в каждой из локальных сетей.

На данный момент сеть MBone практически прекратила свое существование и заменена нормальной многоадресной передачей. Возможно, в многоадресной инфраструктуре Интернета все еще существуют

туннели, но они устанавливаются между маршрутизаторами, поддерживающими многоадресную передачу, и внутри сетей поставщиков услуг Интернета, а потому невидимы для конечного пользователя.

Б.3. 6bone

Виртуальная сеть 6bone была создана в 1996 году по тем же причинам, что и MBone: пользователи в группах узлов, поддерживающих версию протокола IPv6, хотели соединить их вместе с помощью виртуальной сети, не дожидаясь поддержки IPv6 всеми промежуточными маршрутизаторами. На момент написания этой книги сеть 6bone выходит из употребления по мере внедрения IPv6; полное прекращение функционирования 6bone ожидается в июне 2006 года [30]. Мы рассказываем о туннелях только потому, что до сих пор можно встретить настроенные туннели. О динамических туннелях мы расскажем в разделе Б.4.

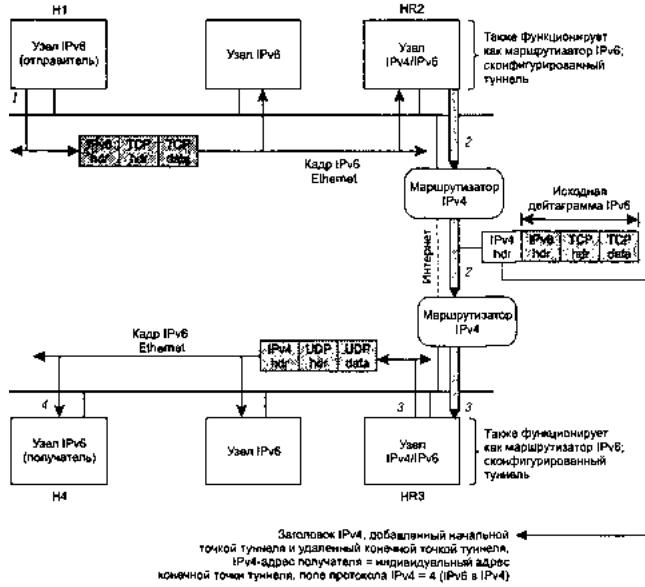


Рис. Б.2. Упаковка IPv6 в IPv4, используемая в сети 6bone

На рис. Б.2 приведен пример двух локальных сетей, поддерживающих IPv6, соединенных с помощью туннеля только через маршрутизаторы IPv4. На рисунке отмечены следующие шаги:

1. Узел H1 локальной сети, показанной на рисунке вверху, посыпает IP-дейтаграмму, содержащую TCP-сегмент, узлу H4 из локальной сети, показанной внизу. Будем называть эти два узла IPv6-узлами, хотя, вероятно, оба они поддерживают и протокол IPv4. В таблице маршрутизации IPv6 на узле H1 записано, что следующим маршрутизатором является узел H2, и IPv6-дейтаграмма отсылается этому маршрутизатору.

2. На узле HR2 имеется сконфигурированный туннель до узла HR3. Этот туннель позволяет посыпать IPv6-дейтаграммы между двумя конечными узлами туннеля через сеть IPv4 путем упаковки IPv6-дейтаграмм в IPv4-дейтаграммы (упаковка IPv6 в IPv4). В поле протокола указано значение 4. Отметим, что оба узла IPv4/IPv6 на концах туннеля — HR2 и HR3 — работают как маршрутизаторы IPv6, поскольку они перенаправляют IPv6-дейтаграммы, получаемые на один интерфейс, через другой интерфейс. Сконфигурированный туннель считается интерфейсом, хотя он является виртуальным, а не физическим интерфейсом.

3. Конечный узел туннеля (HR3) получает упакованную дейтаграмму, отбрасывает IPv4-заголовок и посыпает IPv6-дейтаграмму в свою локальную сеть.

4. Дейтаграмма приходит по назначению на узел H4.

Б.4. Переход на IPv6: 6to4

Механизм перехода 6to4 (бна4) полностью описан в документе «Соединение доменов IPv6 через облака IPv4» (RFC 3056 [17]). Это метод динамического создания туннелей, подобных изображенному на рис. Б.2. В отличие от предыдущих механизмов динамического создания туннелей, которые требовали наличия у всех узлов адресов IPv4, а также явного задания механизма туннелирования, 6to4 реализует

туннелирование исключительно через маршрутизаторы. Это упрощает конфигурацию и позволяет централизованно устанавливать политику безопасности. Кроме того, появляется возможность совмещать функциональность 6to4 с типичной функциональностью трансляции сетевых адресов и межсетевой защиты (например, это может быть сделано на устройстве NAT, расположенному на стороне клиента).

Адреса 6to4 лежат в диапазоне 2002/16. В следующих четырех байтах адреса записывается адрес IPv4 (рис. Б.3). 16-разрядный префикс 2002 и 32-разрядный адрес IPv4 создают общий 48-разрядный идентификатор. Для идентификатора подсети, идущего перед 64-разрядным идентификатором интерфейса, остается 2 байта. Например, нашему узлу freebsd с адресом IPv4 12.106.32.254 соответствует префикс 2002:c6a:20fe/48.



Рис. Б.3. Адреса 6to4

Преимущество 6to4 перед 6bone состоит в том, что тунNELи, формирующие инфраструктуру, образуются автоматически. Для их создания не требуется предварительное конфигурирование. Сайт, использующий 6to4, настраивает основной маршрутизатор на известный адрес передачи наиболее подходящему узлу (anycast) IPv4 192.88.99.1 (RFC 3068 [48]). Он соответствует адресу IPv6 2002::c058:6301:. Маршрутизаторы инфраструктуры IPv6, готовые действовать в качестве шлюзов 6to4, объявляют о маршруте к сети 2002/16 и отправляют упакованный трафик на адрес IPv4, скрытый внутри адреса 6to4. Такие маршрутизаторы могут быть локальными, региональными или глобальными в зависимости от областей действия их маршрутов.

Смысл виртуальных сетей состоит в том, чтобы постепенно исчезнуть с течением времени, когда промежуточные маршрутизаторы обретут требуемую функциональность (в частности, способность работать с IPv6).

Приложение В

Техника отладки

Это приложение содержит некоторые рекомендации и описание методов отладки сетевых приложений. Ни один из приведенных методов не является панацеей от всех возможных проблем, однако существует множество инструментальных средств, с которыми следует ознакомиться, чтобы в дальнейшем использовать подходящие для конкретной среды.

B.1. Трассировка системных вызовов

Многие версии Unix предоставляют возможность трассировки (отслеживания) системных вызовов. Зачастую это может оказаться полезным методом отладки.

Работая на этом уровне, необходимо различать *системный вызов* и *функцию*. Системный вызов является точкой входа в ядро, и именно это можно отследить с помощью инструментальных средств, описанных в данном разделе. Стандарт POSIX и большинство других стандартов используют термин *функция*, вкладывая в это понятие тот же смысл, что и пользователи, хотя на самом деле это может быть системный вызов. Например, в Беркли-ядрах *socket* — это системный вызов, хотя программист приложений может считать, что это обычная функция языка C. В системе SVR4, как будет показано далее, это функция из библиотеки сокетов, которая содержит вызовы *putmsg* и *getmsg*, в действительности являющиеся системными вызовами.

В этом разделе мы рассмотрим системные вызовы, задействованные в работе клиента времени и даты (см. листинг 1.1).

Сокеты ядра BSD

Мы начнем с FreeBSD, операционной системы с Беркли-ядром, в котором все функции сокетов являются системными вызовами. Программа трассировки системных вызовов имеет название *ktrace*. Она выводит информацию о трассировке в файл (по умолчанию имя этого файла *ktrace.out*), который можно вывести на экран с помощью *kdump*. Клиент сокета запускается следующим образом:

```
freebsd % ktrace daytimetcpccli 192.168.42.2
Tue Aug 19 23:35:10 2003
Затем запускаем kdump, чтобы направить трассировочную информацию в стандартный поток вывода.
3211 daytimetcpccli CALL socket(0x2, 0x1, 0)
3211 daytimetcpccli RET socket 3
3211 daytimetcpccli CALL connect(0x3, 0x7fdffffe820, 0x10)
3211 daytimetcpccli RET connect 0
3211 daytimetcpccli CALL read(0x3, 0x7fdffffe830, 0x1000)
3211 daytimetcpccli GIO fd 3 read 26 bytes
    "Tue Aug 19 23:35:10 2003
    "
3211 daytimetcpccli RET read 26/0x1a
...
3211 daytimetcpccli CALL write(0x1, 0x204000, 0x1a)
3211 daytimetcpccli GIO fd 1 wrote 26 bytes
    "Tue Aug 19 23:35:10 2003
    "
3211 daytimetcpccli RET write 26/0x1a
3211 daytimetcpccli CALL read(0x3, 0x7fdffffe830, 0x1000)
3211 daytimetcpccli GIO fd 3 read 0 bytes
    ""
3211 daytimetcpccli RET read 0
3211 daytimetcpccli CALL exit(0)
```

Число 3211 является идентификатором процесса. CALL идентифицирует системный вызов, RET обозначает возвращение управления, G10 подразумевает общую операцию ввода-вывода. Мы видим системные вызовы socket и connect, за которыми следуют вызовы read, возвращающие 26 байт. Наш клиент записывает эти байты в стандартный поток вывода, и при следующем вызове read возвращает нулевое значение (конец файла).

Сокеты ядра Solaris 9

Операционная система Solaris 2.x основывается на SVR4, и во всех версиях ранее 2.6 сокеты реализуются так, как показано на рис. 31.3. Однако во всех версиях SVR4 с подобными реализациями сокетов существует одна проблема: они редко обеспечивают полную совместимость с сокетами Беркли-ядер. Для обеспечения дополнительной совместимости в Solaris 2.6 способ реализации изменен за счет использования файловой системы sockfs. Такой подход обеспечивает поддержку сокетов ядра, что можно проверить с помощью программы truss на нашем клиенте (использующем сокеты).

```
solaris % truss -v connect daytimetcpccli 127.0.0.1  
Mon Sep 8 12:16:42 2003
```

После обычного подключения библиотеки осуществляется первый системный вызов so_socket — системный вызов, инициированный нашим вызовом socket.

```
so_socket(PF_INET, SOCK_STREAM, IPPROTO_IP, 1) = 3  
connect(3, 0xFFFFBFDEF0, 16, 1) = 0  
AF_INET name = 127.0.0.1 port = 13  
read(3, " M o n S e p 8 1", ... 4096) = 26  
Mon Sep 8 12:48:06 2003  
write(1, " M o n S e p 8 1", ... 26) = 26  
read(3, 0xFFFFBFDF03, 4096) = 0  
_exit(0)
```

Первые три аргумента системного вызова so_socket являются нашими аргументами socket.

Далее мы видим, что connect является системным вызовом, а truss при вызове с флагом -v connect выводит на экран содержимое структуры адреса сокета, на которую указывает второй аргумент (IP-адрес и номер порта). Мы не показываем системные вызовы, относящиеся к стандартным потокам ввода и вывода.

B.2. Стандартные службы Интернета

Рекомендуем ознакомиться со стандартными службами Интернета, приведенными в табл. 2.1. Для тестирования наших клиентов мы много раз использовали службу, позволяющую определить дату и время. Служба, игнорирующая присылаемые данные, является удобным портом, на который можно отправлять данные. Эхо-служба аналогична эхо-серверу, неоднократно упомянутому в этой книге.

ПРИМЕЧАНИЕ

В настоящее время многие сайты перекрывают доступ к этим службам с помощью брандмауэров, так как некоторые атаки типа «отказ в обслуживании» (DoS), имевшие место в 1996 году, были направлены именно на эти службы (см. упражнение 13.3). Тем не менее можно успешно использовать эти службы внутри локальной сети.

B.3. Программа sock

Программа sock, написанная Уильямом Стивенсом, впервые появилась в книге [111], где широко использовалась для генерации специальных условий, большинство которых затем проверялось с помощью программы tcpdump. Удобство этой программы заключается в том, что она генерирует такое множество различных сценариев, что нет необходимости писать специальные тестовые программы.

В этой книге исходный код программы не приведен (более 2000 строк на языке C), но он находится в свободном доступе (см. предисловие).

Программа работает в одном из четырех режимов, и в каждом из них можно использовать либо протокол TCP, либо протокол UDP.

1. Клиент стандартного ввода и стандартного вывода (рис. В.1).



Рис. В.1. Клиент sock: стандартный ввод и стандартный вывод

В клиентском режиме все, что считывается из стандартного потока ввода, передается в сеть, а все, что получается из сети, записывается в стандартный поток вывода. Должны быть указаны IP-адрес сервера и номер порта, и в случае TCP выполняется активное открытие.

2. Сервер стандартного ввода и стандартного вывода. Этот режим аналогичен предыдущему, за исключением того, что программа связывает заранее известный порт со своим сокетом и в случае TCP осуществляется пассивное открытие.

3. Клиент-отправитель (рис. В.2).

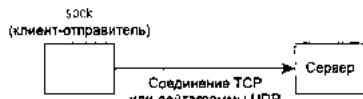


Рис. В.2. Программа sock в качестве клиента-отправителя

Программа осуществляет фиксированное количество передач пакетов некоторого определенного размера в сеть.

4. Сервер-получатель (рис. В.3).

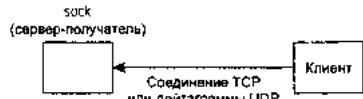


Рис. В.3. Программа sock в качестве сервера-получателя

Программа осуществляет фиксированное количество считываний из сети.

Эти четыре рабочих режима соответствуют следующим четырем командам:

```
sock [параметры] узел служба
sock [параметры] -s [узел] служба
sock [параметры] -i узел служба
sock [параметры] -is [узел] служба
```

где **узел** — это имя или IP-адрес узла, а **служба** — это имя или номер порта. В двух серверных режимах выполняется связывание с универсальным адресом, если не задан необязательный параметр **узел**.

Можно также определить около 40 параметров командной строки, запускающих дополнительные возможности программы. Здесь мы не будем подробно останавливаться на этих параметрах, отметим только, что можно использовать почти все параметры сокетов, упомянутые в главе 7. Запуск программы без аргументов выводит на экран краткое описание всех параметров:

- b n связывает n в качестве клиентского локального номера порта
- c конвертирует символ новой строки в CR/LF и наоборот
- f a.b.c.d.p удаленный IP-адрес = a.b.c.d, удаленный номер порта = p
- g a.b.c.d свободная маршрутизация
- h половинное закрытие TCP при получении EOF из стандартного потока ввода
- i отправка данных на сокет, прием данных с сокета (w/-s)
- j a.b.c.d присоединение к группе многоадресной передачи
- k осуществляет write или writev порциями
- l a.b.c.d.p клиентский локальный IP-адрес = a.b.c.d. локальный номер порта = p
- n n размер буфера для записи клиентом "рассылки" (по умолчанию 1024)
- o НЕ присоединять UDP-клиент
- p n время ожидания (в мс) перед каждым считыванием или записью (рассылка/прием)
- q n размер очереди на прослушиваемом сокете для сервера TCP
(по умолчанию 5)
- r n количество байтов за одну операцию считывания (read) для сервера "приема"
(по умолчанию 1024)

```
-s    работает как сервер, а не как клиент
-u    использовать UDP вместо TCP
-v    подробный вывод
-w n количество байтов для каждой записи (write) клиента "рассылки"
      (по умолчанию 1024)
-x n время (в ms) для SO_RCVTIMEO (получение тайм-аута)
-y n время (в ms) для SO_SNDFTIMEO (отправка тайм-аута)
-A    параметр SO_REUSEADDR
-B    параметр SO_BROADCAST
-D    параметр SO_DEBUG
-E    параметр IP_RECVSTADDR
-F    порождение дочерних процессов (fork) после установления соединения
      (параллельный TCP-сервер)
-G a.b.c.d жесткая маршрутизация
-H n параметр IP_TOS (16=min del, 8=max thru, 4=max rel, 2=min cost)
-I    сигнал SIGIO
-J n параметр IP_TTL
-K    параметр SO_KEEPALIVE
-L n параметр SO_LINGER, n = linger time
-N    параметр TCP_NODELAY
-O n время (в мс) для ожидания после вызова listen, но перед первым приемом (accept)
-P n время (в мс) перед первым считыванием или записью (рассылка/прием)
-Q n время (в мс) ожидания после получения FIN, но перед закрытием
-R n параметр SO_RCVBUF
-S n параметр SO_SNDBUF
-T    параметр SO_REUSEPORT
-U n войти в срочный режим, прежде чем записать число n (только для отправителя)
-V    использовать writev() вместо write(): включает -k
-W    игнорировать ошибки записи для клиента приема
-X n параметр TCP_MAXSEG (устанавливает MSS)
-Y    параметр SO_DONTROUTE
-Z    MSG_PEEK
-2    параметр IP_ONESICAST (255.255.255.255) для широковещательной передачи
```

B.4. Небольшие тестовые программы

Другим полезным методом отладки, которым автор пользовался при написании книги, является создание небольших тестовых программ, позволяющих увидеть, как работает одно конкретное свойство в тщательно выстроенной тестовой ситуации. При написании небольших тестовых программ полезно иметь набор библиотечных функций-оберток и некоторых простых функций вывода сообщений об ошибках, наподобие тех, что использовались на протяжении всей книги. Такой подход уменьшает размер создаваемого кода и в то же время обеспечивает требуемую проверку ошибок.

B.5. Программа tcpdump

Бесценным средством отладки в сетевом программировании является такая программа, как `tcpdump`. Она считывает пакеты из сети и выводит на экран большое количество информации об этих пакетах. Эта программа также позволяет нам задать некоторые критерии отбора пакетов, в результате чего будут выводиться только пакеты, удовлетворяющие этим критериям. Например,

```
% tcpdump '(udp and port daytime) or icmp'
```

выводит только UDP-дейтаграммы с номером порта отправителя или получателя, равным 13 (сервер времени и даты), или ICMP-пакеты. Следующая команда:

```
% tcpdump 'tcp and port 80 and tcp[13:1] & 2 != 0'
```

выводит только TCP-сегменты с номером порта отправителя или получателя, равным 80 (сервер HTTP), у которых установлен флаг SYN. Флаг SYN имеет значение 2 в 13-м байте от начала TCP-заголовка.

Следующая команда:

```
% tcpdump 'tcp and tcp[0:2] > 7000 and tcp[0:2] <= 7005'
```

выводит только те TCP-сегменты, у которых номер порта отправителя лежит в интервале от 7001 до 7005. Номер порта отправителя занимает 2 байта в самом начале TCP-заголовка (нулевое смещение).

В приложении А книги [111] более подробно описано действие данной программы.

ПРИМЕЧАНИЕ

Эта программа доступна по адресу <http://www.tcpdump.org/> и работает под множеством реализаций Unix. Она написана Ван Якобсоном (Van Jacobson), Крэгом Лересом (Craig Leres) и Стивеном МакКанном (Steven McCanne) из LBL, и в настоящее время сопровождается командой `tcpdump.org`.

Некоторые поставщики предлагают свои программы, обладающие теми же возможностями. Например, в Solaris 2.x есть программа `snoop`. Но программа `tcpdump` функционирует под множеством версий Unix, а возможность использования одного и того же средства в неоднородном окружении является большим преимуществом.

B.6. Программа netstat

В тексте книги много раз использовалась программа `netstat`. Эта программа служит для следующих целей.

- Она выводит статус точек доступа сети. Это было показано в разделе 5.6, когда мы прослеживали статус нашей точки доступа при запуске клиента и сервера.
- Она показывает, к какой группе принадлежит каждый из интерфейсов узла. Обычно для этой цели используется флаг `-ia`, а в Solaris 2.x используется флаг `-g`.
- С параметром `-s` эта программа сообщает статистику по каждому протоколу. Подобный пример был приведен в разделе 8.13, когда мы говорили о недостаточном управлении потоками в UDP.
- При использовании параметра `-r` программа выводит таблицу маршрутизации, а с параметром `-i` — информацию об интерфейсе. Эта возможность была использована в разделе 1.9, когда с помощью программы `netstat` мы выясняли топологию сети.

Программа `netstat` обладает и другими возможностями, а многие поставщики добавляют свои собственные. Обратитесь к руководству по вашей системе.

B.7. Программа lsof

Название `lsof` происходит от «list open files» (перечислить открытые файлы). Как и `tcpdump`, эта программа является общедоступной и представляет собой удобное средство для отладки, которое было перенесено на множество версий Unix.

Одним из общих способов применения программы `lsof` при работе в сети является выявление процесса, имеющего открытый сокет, по указанному IP-адресу или порту. Программа `netstat` позволяет выяснить, какой IP-адрес или порт используется, а также узнать состояние TCP-соединения, но она не позволяет идентифицировать процесс. Например, чтобы определить, какой процесс запустил сервер времени и даты, выполним следующую команду:

```
solaris % lsof -i TCP:daytime
COMMAND PID USER FD TYPE DEVICE SIZE/OFF INODE NAME
inetd 222 root 15u inet 0xf5a801f8 0t0      TCP    *:daytime
```

В выводе приводятся следующие данные: команда (данный сервис обеспечивается сервером `inetd`), идентификатор процесса, владелец процесса, дескриптор (15 и `u` означает, что он открыт на чтение и на запись), тип сокета, адрес протокола блока управления, размер смещения файла (не имеет значения для сокета), тип протокола и имя.

Еще один из традиционных случаев применения данной программы имеет место, когда мы запускаем сервер, который связывает свой заранее известный порт и получает ошибку, указывающую, что адрес уже используется. Тогда мы запускаем программу `lsof`, чтобы выяснить, каким процессом используется данный порт.

Поскольку программа lsof сообщает об открытых файлах, она не может сообщать о точках доступа, не ассоциированных с открытым файлом, то есть точках доступа TCP в состоянии TIME_WAIT.

ПРИМЕЧАНИЕ

Программа находится по адресу <ftp://vic.cc.purdue.edu/pub/tools/unix/lsof>. Она написана Виком Абелем (Vic Abell).

Некоторые поставщики предлагают свои программы с похожими возможностями. Например, в BSD/OS предлагается программа fstat. Однако программа lsof работает под множеством версий Unix, а использование одного инструмента в неоднородном окружении вместо подбора различных средств для каждой среды является большим преимуществом.

Приложение Г

Различные исходные коды

Г.1. Заголовочный файл unp.h

Почти каждая программа в этой книге начинается с подключения заголовочного файла unp.h, показанного в листинге Г.1^[1]. Этот файл подключает все стандартные системные заголовочные файлы, необходимые для работы большинства программ, а также некоторые общие системные заголовочные файлы. В нем также определены такие константы, как MAXLINE, прототипы функций ANSI C для тех функций, которые мы определяем в тексте (например, readline), и все используемые нами функции-обёртки. Сами прототипы в приведенном ниже листинге мы не показываем.

Листинг Г.1. Заголовочный файл unp.h

```
//lib/unp.h
1 /* Наш собственный заголовочный файл */

2 #ifndef __unp_h
3 #define __unp_h

4 #include "../config.h" /* параметры конфигурации для данной ОС */
5 /* "../config.h" генерируется сценарием configure */

6 /* изменив список директив #include,
7   нужно также изменить файл acsite.m4 */

8 #include <sys/types.h> /* основные системные типы данных */
9 #include <sys/socket.h> /* основные определения сокетов */
10 #include <sys/time.h> /* структура timeval{} для функции select() */
11 #include <time.h> /* структура timespec{} для функции pselect() */
12 #include <netinet/in.h> /* структура sockaddr_in{} и другие сетевые
                           определения */
13 #include <arpa/inet.h> /* inet(3) функции */
14 #include <errno.h>
15 #include <fcntl.h> /* для неблокируемых сокетов */
16 #include <netdb.h>
17 #include <signal.h>
18 #include <stdio.h>
19 #include <stdlib.h>
20 #include <string.h>
21 #include <sys/stat.h> /* для констант S_XXX */
22 #include <sys/uio.h> /* для структуры iovec{} и ready/writev */
23 #include <unistd.h>
24 #include <sys/wait.h>
25 #include <sys/un.h> /* для доменных сокетов Unix */

26 #ifdef HAVE_SYS_SELECT_H
27 #include <sys/select.h> /* для удобства */
28#endif

29 #ifdef HAVE_SYS_SYSCTL_H
30 #include <sys/sysctl.h>
31#endif

32 #ifdef HAVE_POLL_H
33 #include <poll.h> /* для удобства */
```

```
34 #endif

35 #ifdef HAVE_SYS_EVENT_H
36 #include <sys/event.h> /* для kqueue */
37#endif

38 #ifdef HAVE_STRINGS_H
39 #include <strings.h> /* для удобства */
40#endif

41 /* Три заголовочных файла обычно нужны для вызова ioctl
42   для сокета/файла: <sys/ioctl.h>, <sys/filio.h>,
43   <sys/sockio.h> */
44 #ifdef HAVE_SYS_IOCTL_H
45 #include <sys/ioctl.h>
46#endif
47 #ifdef HAVE_SYS_FILIO_H
48 #include <sys/filio.h>
49#endif
50 #ifdef HAVE_SYS_SOCKIO_H
51 #include <sys/sockio.h>
52#endif

53 #ifdef HAVE_PTHREAD_H
54 #include <pthread.h>
55#endif

56 #ifdef HAVE_NET_IF_DL_H
57 #include <net/if_dl.h>
58#endif

59 #ifdef HAVE_NETINET_SCTP_H
60 #include <netinet/sctp.h>
61#endif

62 /* OSF/1 фактически запрещает recv() и send() в <sys/socket.h> */
63 #ifdef __osf__
64 #undef recv
65 #undef send
66 #define recv(a,b,c,d) recvfrom(a,b,c,d,0,0)
67 #define send(a,b,c,d) sendto(a,b,c,d,0,0)
68#endif

69 #ifndef INADDR_NONE
70 #define INADDR_NONE 0xffffffff /* должно было быть в <netinet/in.h> */
71#endif

72 #ifndef SHUT_RD      /* три новые константы Posix.1g */
73 #define SHUT_RD     0 /* отключение чтения */
74 #define SHUT_WR     1 /* отключение записи */
75 #define SHUT_RDWR   2 /* отключение чтения и записи */
76#endif

77 #ifndef INET_ADDRSTRLEN
78 #define INET_ADDRSTRLEN 16 /* "ddd.ddd.ddd.ddd\0"
79 1234567890123456 */
```

```
80 #endif

81 /* Нужно, даже если нет поддержки IPv6, чтобы мы всегда могли
82   разместить в памяти буфер требуемого размера без директив #ifdef */
83 #ifndef INET6_ADDRSTRLEN
84 #define INET6_ADDRSTRLEN 46 /* максимальная длина строки адреса IPv6:
85 "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx" или
86 "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:ddd.ddd.ddd\0"
87 123456789012345678901234567890123456 */
88#endif

89 /* Определяем bzero() как макрос, если эта функция отсутствует в
  стандартной библиотеке С */
90#ifndef HAVE_BZERO
91#define bzero(ptr,n) memset(ptr, 0, n)
92#endif

93 /* В более старых распознавателях отсутствует gethostbyname2() */
94#ifndef HAVE_GETHOSTBYNAME2
95#define gethostbyname2(host, family) gethostbyname((host))
96#endif

97 /* Структура, возвращаемая функцией recvfrom_flags() */
98 struct in_pktinfo {
99   struct in_addr ipi_addr; /* IPv4-адрес получателя */
100  int     ipi_ifindex; /* полученный индекс интерфейса */
101};

102 /* Нам нужны более новые макросы CMSG_LEN() и CMSG_SPACE(), но в
103   настоящее время их поддерживают далеко не все реализации. Им требуется
104   макрос ALIGN(), но это зависит от реализации */
105#ifndef CMSG_LEN
106#define CMSG_LEN(size) (sizeof(struct cmsghdr) + (size))
107#endif
108#ifndef CMSG_SPACE
109#define CMSG_SPACE(size) (sizeof(struct cmsghdr) + (size))
110#endif

111 /* POSIX требует макрос SUN_LEN(), но он определен
112   не во всех реализациях. Этот макрос 4.4BSD работает
113   независимо от того, имеется ли поле длины */
114#ifndef SUN_LEN
115#define SUN_LEN(su) \
116  (sizeof(*(su)) - sizeof((su)->sun_path) + strlen((su)->sun_path))
117#endif

118 /* В POSIX "домен Unix" называется "локальным IPC".
119   Но пока не во всех системах определены AF_LOCAL и PF_LOCAL */
120#ifndef AF_LOCAL
121#define AF_LOCAL AF_UNIX
122#endif
123#ifndef PF_LOCAL
124#define PF_LOCAL PF_UNIX
125#endif

126 /* POSIX требует определения константы INFTIM в <poll.h>, но во многих
```

```

127     системах она по-прежнему определяется в <sys/stropts.h>. Чтобы
128     не подключать все функции работы с потоками, определяем ее здесь.
129     Это стандартное значение, но нет гарантии, что оно равно -1 */
130 #ifndef INFTIM
131 #define INFTIM (-1) /* бесконечный тайм-аут */
132 #ifdef HAVE_POLL_H
133 #define INFTIM_UNPH /* надо указать в unpxti.h, что эта константа
134 определена здесь */
135 #endif
136 /* Это значение можно было бы извлечь из SOMAXCONN в <sys/socket.h>,
137 но многие ядра по-прежнему определяют его как 5,
138 хотя на самом деле поддерживается гораздо больше */
139 #define LISTENQ 1024 /* второй аргумент функции listen() */
140 /* РАЗЛИЧНЫЕ константы */
141 #define MAXLINE 4096 /* максимальная длина текстовой строки */
142 /* Определение номера порта, который может быть использован для
143 взаимодействия клиент-сервер */
144 #define SERV_PORT 9877 /* клиенты и серверы TCP и UDP */
145 #define SERV_PORT_STR "9877" /* клиенты и серверы TCP и UDP */
146 #define UNIXSTR_PATH "/tmp/unix.str" /* потоковые клиенты и серверы
147 домена Unix */
148 /* Дальнейшие определения сокращают преобразования типов
149 аргументов-указателей */
150 #define SA struct sockaddr
151 /* RFC 3493: протокольно-независимая структура адреса сокета
152 */
153 #define __SS_MAXSIZE 128
154 #define __SS_ALIGNSIZE (sizeof(int64_t))
155 #ifndef HAVE_SOCKADDR_SA_LEN
156 #define __SS_PADS1SIZE (__SS_ALIGNSIZE - sizeof(u_char) -
157 sizeof(sa_family_t))
158 #else
159 #define __SS_PAD1SIZE (__SS_ALIGNSIZE - sizeof(sa_family_t))
160 #endif
161 #define __SS_PAD2SIZE (__SS_MAXSIZE - 2*__SS_ALIGNSIZE)

162 struct sockaddr_storage {
163 #ifdef HAVE_SOCKADDR_SA_LEN
164     u_char ss_len;
165 #endif
166     sa_family_t ss_family;
167     char __ss_pad1[__SS_PAD1SIZE];
168     int64_t ss_align;
169     char __ss_pad2[__SS_PAD2SIZE];
170 };

```

```

171 #endif

172 #define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
173 /* заданные по умолчанию разрешения на доступ для новых файлов */
174 #define DIR_MODE (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)
175 /* разрешения по умолчанию на доступ к файлам для новых каталогов */

176 typedef void Sigfunc(int); /* для обработчиков сигналов */

177 #define min(a, b) ((a) < (b) ? (a) : (b))
178 #define max(a, b) ((a) > (b) ? (a) : (b))

179 #ifndef HAVE_ADDRINFO_STRUCT
180 #include "../lib/addrinfo.h"
181#endif

182 #ifndef HAVE_IF_NAMEINDEX_STRUCT
183 struct if_nameindex {
184     unsigned int if_index; /* 1, 2, ... */
185     char *if_name; /* имя, заканчивающееся нулем: "le0", ... */
186 };
187#endif

188 #ifndef HAVE_TIMESPEC_STRUCT
189 struct timespec {
190     time_t tv_sec; /* секунды */
191     long tv_nsec; /* и наносекунды */
192 };
193#endif

```

Г.2. Заголовочный файл config.h

Для обеспечения переносимости всего исходного кода, используемого в тексте книги, применялась утилита GNU autoconf. Ее можно загрузить по адресу <http://ftp.gnu.org/gnu/autoconf>. Эта программа генерирует сценарий интерпретатора с названием `configure`, который надо запустить после загрузки программного обеспечения в свою систему. Этот сценарий определяет, какие свойства обеспечивает ваша система Unix: имеется ли в структуре адреса сокета поле длины, поддерживается ли многоадресная передача, поддерживаются ли структуры адреса сокета канального уровня, и т.д. В результате получается файл с названием `config.h`. Этот файл — первый заголовочный файл, включенный в `unpr.h` (см. предыдущий раздел). В листинге Г.2 показан заголовочный файл `config.h` для BSD/OS 3.0.

Строки, начинающиеся с `#define`, относятся к тем свойствам, которые обеспечены данной системой. Закомментированные строки и строки, начинающиеся с `#undef`, относятся к свойствам, данной системой не поддерживаемым.

Листинг Г.2. Заголовочный файл config.h для BSD/OS

```

i386-pc-bsdi3.0/config.h
1 /* config.h. Автоматически генерируется сценарием configure. */
2 /* Определяем константы, если имеется соответствующий заголовочный файл */
3 #define CPU_VENDOR_OS "i386-pc-bsdi3.0"
4 /* #undef HAVE_NETCONFIG_H */ /* <netconfig.h> */
5 /* #undef HAVE_NETDIR_H */ /* <netdir.h> */
6 #define HAVE_PTHREAD_H 1 /* <pthread.h> */
7 #define HAVE_STRINGS_H 1 /* <strings.h> */
8 /* #undef HAVE_XTI_INET_H */ /* <xti_inet.h> */
9 #define HAVE_SYS_FILIO_H 1 /* <sys/filio.h> */
10 #define HAVE_SYS_IOCTL_H 1 /* <sys/ioctl.h> */
11 #define HAVE_SYS_SELECT_H 1 /* <sys/select.h> */

```

```

12 #define HAVE_SYS_SOCKIO_H 1    /* <sys/sockio.h> */
13 #define HAVE_SYS_SYSCTL_H 1    /* <sys/sysctl.h> */
14 #define HAVE_SYS_TIME_H 1       /* <sys/time.h> */

15 /* Определена, если можно подключить <time.h> и <sys/time.h> */
16 #define TIME_WITH_SYS_TIME 1

17 /* Определены, если имеются соответствующие функции */
18 #define HAVE_BZERO 1
19 #define HAVE_GETHOSTBYNAME2 1
20 /* #undef HAVE_PSELECT */
21 #define HAVE_VSNPRINTF 1

22 /* Определены, если прототипы функций есть в заголовочном файле */
23 /* #undef HAVE_GETADDRINFO_PROTO */ /* <netdb.h> */
24 /* #undef HAVE_GETNAMEINFO_PROTO */ /* <netdb.h> */
25 #define HAVE_GETHOSTNAME_PROTO 1      /* <unistd.h> */
26 #define HAVE_GETRUSAGE_PROTO 1        /* <sys/resource.h> */
27 #define HAVE_HSTRERROR_PROTO 1        /* <netdb.h> */
28 /* #undef HAVE_IF_NAMEINDEX_PROTO */ /* <net/if.h> */
29 #define HAVE_INET_ATON_PROTO 1        /* <arpa/inet.h> */
30 #define HAVE_INET_PTON_PROTO 1        /* <arpa/inet.h> */
31 /* #undef HAVE_ISFDTYPE_PROTO */ /* <sys/stat.h> */
32 /* #undef HAVE_PSELECT_PROTO */ /* <sys/select.h> */
33 #define HAVE_SNPRINTF_PROTO 1         /* <stdio.h> */
34 /* #undef HAVE_SOCKETMARK_PROTO */ /* <sys/socket.h> */

35 /* Определены, если определены соответствующие структуры */
36 /* #undef HAVE_ADDRINFO_STRUCT */ /* <netdb.h> */
37 /* #undef HAVE_IF_NAMEINDEX_STRUCT */ /* <net/if.h> */
38 #define HAVE_SOCKADDR_DL_STRUCT 1     /* <net/if_dl.h> */
39 #define HAVE_TIMESPEC_STRUCT 1        /* <time.h> */

40 /* Определены, если имеется указанное свойство */
41 #define HAVE_SOCKADDR_SA_LEN 1        /* в sockaddr{} есть поле sa_len */
42 #define HAVE_MSGHDR_MSG_CONTROL 1 /* в msghdr{} есть поле msg_control */

43 /* Имена устройств XTI для TCP и UDP */
44 /* #undef HAVE_DEV_TCP */           /* большинство здесь */
45 /* #undef HAVE_DEV_XTI_TCP */        /* для AIX */
46 /* #undef HAVE_DEV_STREAMS_XTISO_TCP */ /* для OSF 3.2 */

47 /* При необходимости определяем типы данных */
48 /* #undef int8_t */                /* <sys/types.h> */
49 /* #undef int16_t */               /* <sys/types.h> */
50 /* #undef int32_t */               /* <sys/types.h> */
51 #define uint8_t unsigned char      /* <sys/types.h> */
52 #define uint16_t unsigned short    /* <sys/types.h> */
53 #define uint32_t unsigned int      /* <sys/types.h> */
54 /* #undef size_t */                /* <sys/types.h> */
55 /* #undef ssize_t */               /* <sys/types.h> */

56 /* socklen_t должен иметь тип uint32_t, но configure определяет его
57 как unsigned int. т. к. это значение используется в начале компиляции.
58 иногда до того, как в данной реализации определяется тип uint32_t */
59 #define socklen_t unsigned int /* <sys/socket.h> */
60 #define sa_family_t SA_FAMILY_T /* <sys/socket.h> */

```

```

61 #define SA_FAMILY_T uint8_t

62 #define t_scalar_t int32_t /* <xti.h> */
63 #define t_uscalar_t uint32_t /* <xti.h> */

64 /* Определены, если система поддерживает указанное свойство */
65 #define IPV4 1      /* IPv4, V в верхнем регистре */
66 #define IPv4 1      /* IPv4, v в нижнем регистре, на всякий случай */
67 /* #undef IPV6 */ /* IPv6, V в верхнем регистре */
68 /* #undef IPv6 */ /* IPv6, v в нижнем регистре, на всякий случай */
69 #define UNIXDOMAIN 1 /* доменные сокеты Unix */
70 #define UNIXdomain 1 /* доменные сокеты Unix */
71 #define MCAST 1     /* поддержка многоадресной передачи */

```

Г.3. Стандартные функции обработки ошибок

В этой книге мы определяем набор своих собственных функций для обработки ошибок. Причина, по которой мы создаем эти функции, заключается в том, что они позволяют нам обрабатывать ошибки с помощью одной строки кода, как, например, показано ниже:

```

if (условие ошибки)
    err_sys(формат printf с любым количеством аргументов);
вместо
if (условие ошибки) {
    char buff[200];
    sprintf(buff, sizeof(buff), формат printf с любым количеством аргументов);
    perror(buff);
    exit(1);
}

```

Наши функции обработки ошибок используют следующую возможность ANSI C: список аргументов может иметь переменную длину. Более подробную информацию об этом вы найдете в разделе 7.3 книги [68].

В табл. Г.1 показано, в чем заключаются различия между функциями обработки ошибок. Если глобальная целочисленная переменная `daemon_proc` отлична от нуля, то сообщение об ошибке передается функции `syslog` с указанным уровнем, в противном случае оно отправляется в стандартный поток вывода сообщений об ошибках.

Таблица Г.1. Стандартные функции обработки ошибок

Функция strerror (errno ?) Завершение ? Уровень syslog

err_dump	Да	abort();	LOG_ERR
err_msg	Нет	return;	LOG_INFO
err_quit	Нет	exit(1);	LOG_ERR
err_ret	Да	return;	LOG_INFO
err_sys	Да	exit(1);	LOG_ERR

В листинге Г.3 показаны первые пять функций из табл. Г.1.

Листинг Г.3. Стандартные функции обработки ошибок

```

//lib/error.c
1 #include "unp.h"

2 #include <stdarg.h> /* заголовочный файл ANSI C */
3 #include <syslog.h> /* для syslog() */

4 int daemon_proc; /* устанавливается в ненулевое значение с
                     помощью daemon_init() */

5 static void err_doit(int, int, const char*, va_list);

```

```
6 /* Нефатальная ошибка, связанная с системным вызовом.
7     Выводим сообщение и возвращаем управление */

8 void
9 err_ret(const char *fmt , ...)
10 {
11 va_list ap;

12 va_start(ap, fmt);
13 err_doit(1, LOG_INFO, fmt, ap);
14 va_end(ap);
15 return;
16 }

17 /* Фатальная ошибка, связанная с системным вызовом.
18     Выводим сообщение и завершаем работу */

19 void
20 err_sys(const char *fmt)
21 {
22 va_list ap;
23 va_start(ap, fmt);
24 err_doit(1, LOG_ERR, fmt, ap);
25 va_end(ap);
26 exit(1);
27 }

28 /* Фатальная ошибка, связанная с системным вызовом.
29     Выводим сообщение, сохраняем дамп памяти процесса и заканчиваем работу */

30 void
31 err_dump(const char *fmt, ... )
32 {
33 va_list ap;

34 va_start(ap, fmt);
35 err_doit(1, LOG_ERR, fmt, ap);
36 va_end(ap);
37 abort(); /* сохраняем дамп памяти и заканчиваем работу */
38 exit(1);
39 }

40 /* Нефатальная ошибка, не относящаяся к системному вызову.
41     Выводим сообщение и возвращаем управление */

42 void
43 err_msg(const char *fmt , ...)
44 {
45 va_list ap;

46 va_start(ap, fmt);
47 err_doit(0, LOG_INFO, fmt, ap);
48 va_end(ap);
49 return;
50 }
```

```
51 /* Фатальная ошибка, не относящаяся к системному вызову.  
52     Выводим сообщение и заканчиваем работу. */  
  
53 void  
54 err_quit(const char *fmt, ...)  
55 {  
56     va_list ap;  
  
57     va_start(ap, fmt);  
58     err_doit(0, LOG_ERR, fmt, ap);  
59     va_end(ap);  
60     exit(1);  
61 }  
  
62 /* Выводим сообщение и возвращаем управление.  
63     Вызывающий процесс задает "errnoflag" и "level" */  
  
64 static void  
65 err_doit(int errnoflag, int level, const char *fmt, va_list ap)  
66 {  
67     int errno_save, n;  
68     char buf[MAXLINE + 1];  
  
69     errno_save = errno; /* значение может понадобиться вызвавшему  
                         процессу */  
70 #ifdef HAVE_VSNPRINTF  
71     vsnprintf(buf, MAXLINE, fmt, ap); /* защищенный вариант */  
72 #else  
73     vsprintf(buf, fmt, ap); /* незащищенный вариант */  
74 #endif  
75     n = strlen(buf);  
76     if (errnoflag)  
77         sprintf(buf + n, MAXLINE - n, ": %s", strerror(errno_save));  
78     strcat(buf, "\n");  
  
79     if (daemon_proc) {  
80         syslog(level, buf);  
81     } else {  
82         fflush(stdout); /* если stdout и stderr совпадают */  
83         fputs(buf, stderr);  
84         fflush(stderr);  
85     }  
86     return;  
87 }
```

Приложение Д

Решения некоторых упражнений

Глава 1

1.3. В операционной системе Solaris получаем:

```
solaris % daytimetcpccli 127.0.0.1  
socket error: Protocol not supported
```

Для получения дополнительной информации об этой ошибке сначала используем программу grep, чтобы найти строку Protocol not supported в заголовочном файле <sys/errno.h>.

```
solaris % grep 'Protocol not supported' /usr/include/sys/errno.h  
#define EPROTONOSUPPORT 120 /* Protocol not supported */
```

Это значение errno возвращается функцией socket. Далее смотрим в руководство пользователя:

```
solaris % man socket
```

В большинстве руководств пользователя в конце под заголовком «Errors» приводится дополнительная, хотя и лаконичная информация об ошибках.

1.4. Заменяем первое описание на следующее:

```
int sockfd, n, counter = 0;
```

Добавляем оператор

```
counter++;
```

в качестве первого оператора цикла while. Наконец, прежде чем прервать программу, выполняем printf("counter = %d\n", counter);

На экран всегда выводится значение 1.

1.5. Объявим переменную i типа int и заменим вызов функции write на следующий:

```
for (i = 0; i < strlen(buff); i++)  
    Write(connfd, &buff[i], 1);
```

Результат зависит от расположения клиентского узла и узла сервера. Если клиент и сервер находятся на одном узле, счетчик обычно равен 1. Это значит, что даже если сервер выполнит функцию write 26 раз, данные будут возвращены за одну операцию считывания (read). Но если клиент запущен в Solaris 2.5.1, а сервер в BSD/OS 3.0, счетчик обычно равен 2. Просмотрев пакеты Ethernet, мы увидим, что первый символ отправляется в первом пакете сам по себе, а следующий пакет содержит остальные 25 символов. (Обсуждение алгоритма Нагла в разделе 7.9 объясняет причину такого поведения.)

Цель этого примера — продемонстрировать, что разные реализации TCP по-разному поступают с данными, поэтому наше приложение должно быть готово считывать данные как поток байтов, пока не будет достигнут конец потока.

Глава 2

2.1 Зайдите на веб-страницу <http://www.iana.org/numbers.htm> и найдите журнал под названием «IP Version Number». Номер версии 0 зарезервирован, версии 1-3 не использовались, а версия 5 представляет собой потоковый протокол Интернета (Internet Stream Protocol).

2.2. Все RFC бесплатно доступны по электронной почте, через FTP или Web. Стартовая страница для поиска находится по адресу <http://www.ietf.org>. Одним из мест расположения RFC является каталог <ftp://ftp.isi.edu/in-notes>. Для начала следует получить файл с текущим каталогом RFC, обычно это файл rfc-index.txt. HTML-версия хранится в файле <http://www.rfc-editor.org/rfc-index.html>. Если с помощью какого-либо редактора осуществить поиск термина «stream» (поток) в указателе RFC, мы выясним, что RFC 1819 определяет версию 2 потокового протокола Интернета. Какую бы информацию, которая может содержаться в RFC, мы ни искали, для поиска следует использовать указатель (каталог) RFC.

2.3. В версии IPv4 при таком значении MSS генерируется 576-байтовая дейтаграмма (20 байт для заголовка IPv4 и 20 байт для заголовка TCP). Это минимальный размер буфера для сборки фрагментов вIpv4.

2.4. В данном примере сервер (а не клиент) осуществляет активное закрытие.

2.5. Узел в сети Token Ring не может посыпать пакет, содержащий больше, чем 1460 байт данных, поскольку полученное им значение MSS равно 1460. Узел в сети Ethernet может посыпать пакет размером до 4096 байт данных, но не превышающий величину MTU исходящего интерфейса (Ethernet) во избежание фрагментации. Протокол TCP не может превысить величину MSS, объявленную другой стороной, но он всегда может посыпать пакеты меньшего размера.

2.6. В разделе «Protocol Numbers» (номера протоколов) RFC «Assigned Numbers» («Присвоенные номера») указано значение 89 для протокола OSPF.

2.7. Выборочное уведомление указывает лишь на получение пакетов с конкретными последовательными номерами. Кумулятивное уведомление сообщает о получении данных вплоть до конкретного порядкового номера (включительно). При освобождении буфера отправки в соответствии с выборочным уведомлением система может удалять только те данные, доставка которых была подтверждена явно, но не те, номера которых меньше или больше подтвержденных.

Глава 3

3.1. В языке С функция не может изменить значение аргумента, передаваемого по значению. Чтобы вызванная функция изменила значение, передаваемое вызывающим процессом, требуется, чтобы вызывающий процесс передал указатель на значение, подлежащее изменению.

3.2. Указатель должен увеличиваться на количество считанных или записанных байтов, но в языке С нет возможности увеличивать указатели типа `void` (поскольку компилятору не известно, на какой тип данных указывает указатель).

Глава 4

4.1. Посмотрите на определение констант, начинающихся с `INADDR_`, кроме `INADDR_ANY` (состоит из нулевых битов) и `INADDR_NONE` (состоит из единичных битов). Например, адрес многоадресной передачи класса D `INADDR_MAX_LOCAL_GROUP` определяется как `0xe00000ff` с комментарием «`224.0.0.255`», что явно указывает на порядок байтов узла.

4.2. Приведем новые строки, добавленные после вызова `connect`:

```
len = sizeof(cliaddr);
Getsockname(sockfd, (SA*)&cliaddr, &len);
printf("local addr: %s\n",
Sock_ntop((SA*)&cliaddr, len));
```

Это требует описания переменной `len` как `socklen_t`, а `cliaddr` как структуры `struct sockaddr_in`. Обратите внимание, что аргумент типа «значение-результат» для функции `getsockname(len)` должен быть до вызова функции инициализирован размером переменной, на которую указывает второй аргумент. Наиболее частая ошибка программирования при использовании аргументов типа «значение-результат» заключается в том, что про эту инициализацию забывают.

4.3. Когда дочерний процесс вызывает функцию `close`, счетчик ссылок уменьшается с 2 до 1, так что клиенту не посыпается сегмент FIN. Позже, когда родительский процесс вызывает функцию `close`, счетчик ссылок уменьшается до нуля, и тогда сегмент FIN посыпается.

4.4. Функция `accept` возвращает значение `EINVAL`, так как первый аргумент не является прослушиваемым сокетом.

4.5. Вызов функции `listen` без вызова функции `bind` присваивает прослушиваемому сокету динамически назначаемый порт.

Глава 5

5.1. Длительность состояния `TIME_WAIT` должна находиться в интервале между 1 и 4 минутами, что дает величину MSL от 30 с до 2 мин.

5.2. Наши клиент-серверные программы не работают с двоичными файлами. Допустим, что первые 3 байта в файле являются двоичной единицей (1), двоичным нулем (0) и символом новой строки. При вызове функции `fgets` в листинге 5.4 либо считывается `MAXLINE - 1` символов, либо считаются символы до символа новой строки или до конца файла. В данном примере функция считает три символа, а затем прервет строку нулевым байтом. Но вызов функции `strlen` в листинге 5.4 возвращает значение 1, так как

она остановится на первом нулевом байте. Один байт посыпается серверу, но сервер блокируется в своем вызове функции `readline`, ожидая символа новой строки. Клиент блокируется, ожидая ответа от сервера. Такое состояние называется *зависанием*, или *взаимной блокировкой*: оба процесса блокированы и при этом каждый ждет от другого некоторого действия, которое никогда не произойдет. Проблема заключается в том, что функция `fgets` обозначает нулевым байтом конец возвращаемых ею данных, поэтому данные, которые она считывает, не должны содержать нулевой байт.

5.3. Программа Telnet преобразует входные строки в NVT ASCII (см. раздел 26.4 книги [111]), что означает прерывание каждой строки 2-символьной последовательностью CR (carriage return — возврат каретки) и LF (linefeed — новая строка). Наш клиент добавляет только разделитель строк (newline), который в действительности является символом новой строки (linefeed, LF). Тем не менее можно использовать клиент Telnet для связи с нашим сервером, поскольку наш сервер отражает каждый символ, включая CR, предшествующий каждому разделителю строк.

5.4. Нет, последние два сегмента из последовательности завершения соединения не посылаются. Когда клиент посыпает серверу данные после уничтожения дочернего процесса сервера (ввод строки `another line`, см. раздел 5.12), сервер TCP отвечает сегментом RST. Сегмент RST прекращает соединение, а также предотвращает переход в состояние TIME_WAIT на стороне сервера (конец соединения, осуществивший активное закрытие).

5.5. Ничего не меняется, потому что процесс, запущенный на узле сервера, создает прослушиваемый сокет и ждет прибытия запросов на соединение. На третьем шаге мы посыпаем сегмент данных, предназначенный для установленного соединения TCP (состояние ESTABLISHED). Наш сервер с прослушиваемым сокетом не увидит этот сегмент данных, и TCP сервера по-прежнему будет посыпать клиенту сегмент RST.

5.6. В листинге Д.1^[1] приведена программа. Запуск этой программы в Solaris генерирует следующий вывод:

```
solaris % tsigpipe 192.168.1.10
SIGPIPE received
write error: Broken pipe
```

Начальный вызов функции `sleep` и переход в режим ожидания на 2 с нужен, чтобы сервер времени и даты отправил ответ и закрыл свой конец соединения. Первая функция `write` отправляет сегмент данных серверу, который отвечает сегментом RST (поскольку сервер времени и даты полностью закрыл свой сокет). Обратите внимание, что наш TCP позволяет писать в сокет, получивший сегмент FIN. Второй вызов функции `sleep` позволяет получить от сервера сегмент RST, а во втором вызове функции `write` генерируется сигнал SIGPIPE. Поскольку наш обработчик сигналов возвращает управление, функция `write` возвращает ошибку EPIPE.

Листинг Д.1. Генерация SIGPIPE

```
//tcpcliserv/tsigpipe.c
1 #include "unp.h"

2 void
3 sig_pipe(int signo)
4 {
5     printf("SIGPIPE received\n");
6     return;
7 }

8 int
9 main(int argc, char **argv)
10 {
11     int sockfd;
12     struct sockaddr_in servaddr;

13     if (argc != 2)
14         err_quit("usage: tcpcli <Ipaddress>");

15     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
```

```

16 bzero(&servaddr, sizeof(servaddr));
17 servaddr.sin_family = AF_INET;
18 servaddr.sin_port = htons(13); /* сервер времени и даты */
19 Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

20 Signal(SIGPIPE, sig_pipe);

21 Connect(sockfd, (SA*)&servaddr, sizeof(servaddr));

22 sleep(2);
23 Write(sockfd, "hello", 5);
24 sleep(2);
25 Write(sockfd, "world", 5);

26 exit(0);
27 }

```

5.7. В предположении, что узел сервера поддерживает модель системы с гибкой привязкой (см. раздел 8.8), все будет работать. Узел сервера примет IP-дейтаграмму (которая в данном случае содержит TCP-сегмент), прибывшую на самый левый канал, даже если IP-адрес получателя является адресом самого правого канала. Это можно проверить, если запустить наш сервер на узле *linux* (см. рис. 1.7), а затем запустить клиент на узле *solaris*, но на стороне клиента задать другой IP-адрес сервера (206.168.112.96). После установления соединения, запустив на стороне сервера программу *netstat*, мы увидим, что локальный IP-адрес является IP-адресом получателя из клиентского сегмента SYN, а не IP-адресом канала, на который прибыл сегмент SYN (как отмечалось в разделе 4.4).

5.8. Наш клиент был запущен в системе Intel с прямым порядком байтов, где 32-разрядное целое со значением 1 хранится так, как показано на рис. Д.1.

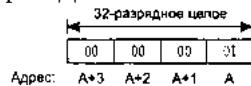


Рис. Д.1. Представление 32-разрядного целого числа 1 в формате прямого порядка байтов

Четыре байта посыпаются на сокет в следующем порядке: A, A + 1, A + 2 и A + 3, и там хранятся в формате обратного порядка байтов, как показано на рис. Д.2.

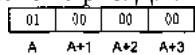


Рис. Д.2. Представление 32-разрядного целого числа с рис. Д.1 в формате обратного порядка байтов

Значение 0x01000000 интерпретируется как 16 777 216. Аналогично, целое число 2, отправленное клиентом, интерпретируется сервером как 0x02000000, или 33 554 432. Сумма этих двух целых чисел равна 50 331 648, или 0x03000000. Когда это значение, записанное в обратном порядке байтов, отправляется клиенту, оно интерпретируется клиентом как целое число 3.

Но 32-разрядное целое число -22 представляется в системе с прямым порядком байтов так, как показано на рис. Д.3 (мы предполагаем, что используется поразрядное дополнение до двух для отрицательных чисел).

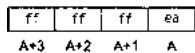


Рис. Д.3. Представление 32-разрядного целого числа -22 в формате прямого порядка байтов

В системе с обратным порядком байтов это значение интерпретируется как 0xeaffffff, или -352 521 537. Аналогично, представление числа -77 в прямом порядке байтов выглядит как 0xfffffffffb3, но в системах с обратным порядком оно представляется как 0xb3fffffff, или -1 275 068 417. Сложение, выполняемое сервером, приводит к результату 0x9effffff, или -1 627 389 954. Полученное значение в обратном порядке байтов посыпается через сокет клиенту, где в прямом порядке байтов оно интерпретируется как 0xfffffe9e, или -16 777 314 — это то значение, которое выводится в нашем примере.

5.9. Метод правильный (преобразование двоичных значений в сетевой порядок байтов), но нельзя использовать функции *htonl* и *ntohl*. Хотя символ l в названиях данных функций обозначает «long», эти функции работают с 32-разрядными целыми (раздел 3.4). В 64-разрядных системах *long* занимает 64 бита, и эти две функции работают некорректно. Для решения этой проблемы следует определить две новые функции *hton64* и *ntoh64*, но они не будут работать в системах, представляющих значения типа *long* 32 битами.

5.10. В первом сценарии сервер будет навсегда блокирован при вызове функции `readn` в листинге 5.14, поскольку клиент посыпает два 32-разрядных значения, а сервер ждет два 64-разрядных значения. В случае, если клиент и сервер поменяются узлами, клиент будет посыпать два 64-разрядных значения, а сервер считает только первые 64 бита, интерпретируя их как два 32-разрядных значения. Второе 64-разрядное значение останется в приемном буфере сокета сервера. Сервер отправит обратно 32-разрядное значение, и клиент навсегда заблокируется в вызове функции `readn` в листинге 5.13, поскольку будет ждать для считывания 64-разрядное значение.

5.11. Функция IP-маршрутизации просматривает IP-адрес получателя (IP-адрес сервера) и пытается по таблице маршрутизации определить исходящий интерфейс и следующий маршрутизатор (см. главу 9 [111]). В качестве адреса отправителя используется первичный IP-адрес исходящего интерфейса, если сокет еще не связан с локальным IP-адресом.

Глава 6

6.1. Массив целых чисел содержится внутри структуры, а язык С позволяет использовать со структурами оператор присваивания.

6.2. Если функция `select` сообщает, что сокет готов к записи, причем буфер отправки сокета вмещает 8192 байта, а мы вызываем для этого блокируемого сокета функцию `write` с буфером размером 8193 байта, то функция `write` может заблокироваться, ожидая места для последнего байта. Операции считывания на блокируемом сокете будут возвращать сообщение о неполном считывании, если доступны какие-либо данные, но операции записи на блокируемом сокете заблокированы до принятия всех данных ядром. Поэтому, чтобы избежать блокирования при использовании функции `select` для проверки на возможность записи, следует переводить сокет в неблокируемый режим.

6.3. Если оба дескриптора готовы для чтения, выполняется только первый тест — тест сокета. Но это не нарушает работоспособность клиента, а только лишь уменьшает его эффективность. Поэтому если при завершении функции `select` оба дескриптора готовы для чтения, первое условие `if` оказывается истинным, в результате чего сначала вызывается функция `readline` для считывания из сокета, а затем функция `fputs` для записи в стандартный поток вывода. Следующее условие `if` пропускается (поскольку мы добавили `else`), но функция `select` вызывается снова, сразу находит стандартное устройство ввода, готовое к чтению, и завершается. Суть в том, что условие готовности стандартного потока ввода для чтения сбрасывается считыванием из сокета, а не возвратом функции `select`.

6.4. Воспользуйтесь функцией `getrlimit` для получения значений константы `RLIMIT_NOFILE`, а затем вызовите функцию `setrlimit` для установки текущего гибкого предела (`rlim_cur`) равным жесткому пределу (`rlim_max`). Например, в Solaris 2.5 гибкий предел равен 64, но любой процесс может увеличить это значение до используемого по умолчанию значения жесткого предела (1024).

6.5. Серверное приложение непрерывно посыпает данные клиенту, клиент TCP подтверждает их прием и сбрасывает.

6.6. Функция `shutdown` с аргументами `SHUT_WR` и `SHUT_RDWR` всегда посыпает сегмент FIN, в то время как функция `close` посыпает сегмент FIN только если в момент вызова функции `close` счетчик ссылок дескриптора равен 1.

6.7. Функция `readline` возвращает ошибку, и наша функция-обертка `Readline` завершает работу сервера. Но серверы должны справляться с такими ситуациями. Обратите внимание на то, как мы обрабатываем эти условия в листинге 6.6, хотя даже этот код не является удовлетворительным. Рассмотрим, что произойдет, если соединение между клиентом и сервером прервется и время ожидания одного из ответов сервера будет превышено. Возвращаемой ошибкой может быть ошибка `ETIMEDOUT`.

Обычно сервер не должен прекращать свою работу из-за подобных ошибок. Он должен записать ее в файл журнала, закрыть сокет и продолжать обслуживание других клиентов. Следует понимать, что обработка таких ошибок путем прекращения работы сервера недопустима для серверов, у которых один процесс выполняет обработку всех клиентов. Но если сервер был дочерним процессом, обрабатывающим только один клиент, то прекращение работы одного дочернего процесса не отразится ни на родительском процессе (который, по нашему предположению, обрабатывает все новые соединения и порождает новые дочерние процессы), ни на одном из других дочерних процессов, обрабатывающих другие клиенты.

Глава 7

7.2. Решение упражнения приведено в листинге Д.2. Вывод строки данных, возвращаемых сервером, был удален, поскольку это значение нам не нужно.

Листинг Д.2. Вывод размера приемного буфера сокета и MSS до и после установления соединения

```
//sockopt/rcvbuf.c
1 #include "urp.h"
2 #include <netinet/tcp.h> /* для TCP_MAXSEG */

3 int
4 main(int argc, char **argv)
5 {
6     int sockfd, rcvbuf, mss;
7     socklen_t len;
8     struct sockaddr_in servaddr;

9     if (argc != 2)
10        err_quit("usage: rcvbuf <Ipaddress>");

11    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

12    len = sizeof(rcvbuf);
13    Getsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &rcvbuf, &len);
14    len = sizeof(mss);
15    Getsockopt(sockfd, IPPROTO_TCP, TCP_MAXSEG, &mss, &len);
16    printf("defaults: SO_RCVBUF = %d. MSS = %d\n", rcvbuf, mss);

17    bzero(&servaddr, sizeof(servaddr));
18    servaddr.sin_family = AF_INET;
19    servaddr.sin_port = htons(13); /* сервер времени и даты */
20    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

21    Connect(sockfd, (SA*)&servaddr, sizeof(servaddr));

22    len = sizeof(rcvbuf);
23    Getsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &rcvbuf, &len);
24    len = sizeof(mss);
25    Getsockopt(sockfd, IPPROTO_TCP, TCP_MAXSEG, &mss, &len);
26    printf("after connect: SO_RCVBUF = %d, MSS = %d\n", rcvbuf, mss);

27    exit(0);
28 }
```

Не существует какого-то одного «правильного» вывода для данной программы. Результаты зависят от системы. Некоторые системы (в особенности Solaris 2.5.1 и более ранние версии) всегда возвращают нулевой размер буфера сокета, не давая нам возможности увидеть, что происходит с этим значением в процессе соединения.

До вызова функции connect выводится значение MSS по умолчанию (часто 536 или 512), а значение, выводимое после вызова функции connect, зависит от возможных параметров MSS, полученных от собеседника. Например, в локальной сети Ethernet после выполнения функции connect MSS может иметь значение 1460. Однако после соединения (connect) с сервером в удаленной сети значение MSS может быть равно значению по умолчанию, если только ваша система не поддерживает обнаружение транспортной MTU. Если это возможно, запустите во время работы вашей программы программу tcpdump или подобную ей (см. раздел В.5), чтобы увидеть фактическое значение параметра MSS в сегменте SYN, полученном от собеседника.

Многие реализации после установления соединения округляют размер приемного буфера сокета в большую сторону, чтобы он было кратным MSS. Чтобы узнать размер приемного буфера сокета после установления соединения, можно исследовать пакеты с помощью программы типа tcpdump и посмотреть, каков размер объявленного окна TCP.

7.3. Разместите в памяти структуру linger по имени ling и проинициализируйте ее следующим образом:

```
str_cli(stdin, sockfd);

ling.l_onoff = 1;
ling.l_linger = 0;
Setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &ling, sizeof(ling));

exit(0);
```

Это заставит TCP на стороне клиента прекратить работу путем отправки сегмента RST вместо нормального обмена четырьмя сегментами. Дочерний процесс сервера вызывает функцию readline, возвращает ошибку ECONNRESET и выводит следующее сообщение:

```
readline error: Connection reset by peer
```

Клиентский сокет не должен проходить через состояние ожидания TIME_WAIT, даже если клиент выполняет активное закрытие.

7.4. Первый клиент вызывает функции setsockopt, bind и connect. Но если второй клиент вызовет функцию bind между вызовами функций bind и connect первого клиента, возвращается ошибка EADDRINUSE. Но как только первый клиент установит соединение с собеседником, вызов функции bind второго клиента будет работать, поскольку сокет первого клиента уже присоединен. В случае возвращения ошибки EADDRINUSE второму клиенту следует вызывать bind несколько раз, а не останавливаться при появлении первой ошибки — это единственный способ справиться с данной ситуацией.

7.5. Запускаем программу на узле без поддержки многоадресной передачи (MacOS X 10.2.6).

```
macosx % sock -s 9999 & запускаем первый сервер с универсальным адресом
[1] 29697
macosx % sock -s 172.24.37.78 9999 пробуем второй сервер, но без -A
can't bind local address: Address already in use
macosx % sock -s -A 172.24.37.78 9999 & пробуем опять с -A: работает
[2] 29699
macosx % sock -s -A 127.0.0.1 9999 & третий сервер с -A: работает
[3] 29700
macosx % netstat -na | grep 9999
tcp4 0 0 127.0.0.1.9999    *.* LISTEN
tcp4 0 0 206.62.226.37.9999 *.* LISTEN
tcp4 0 0 *.*.9999          *.* LISTEN
```

7.6. Теперь попробуем проделать то же на узле с поддержкой многоадресной передачи, но без поддержки параметра SO_REUSEADDR (Solaris 9).

```
solaris % sock -s -u 8888 & запускаем первый
[1] 24051
solaris % sock -s -u 8888
can't bind local address: Address already in use
solaris % sock -s -u -A 8888 & снова пробуем запустить второй с -A:
работает
solaris % netstat -na | grep 8888 мы видим дублированное связывание
*.8888 Idle
* 8888 Idle
```

В этой системе задавать параметр SO_REUSEADDR было необходимо только для второго связывания. Наконец, запускаем сценарий в MacOS X 10.2.6, где поддерживается как многоадресная передача, так и параметр SO_REUSEPORT. Сначала пробуем использовать SO_REUSEADDR для обоих серверов, но это не работает.

```
macosx % sock -u -s -A 7777 &
[1] 17610
macosx % sock -u -s -A 7777
can't bind local address: Address already in use
```

Тогда пробуем использовать параметр SO_REUSEPORT только для второго сервера. Это также не работает, так как полностью дублированное связывание требует включения данного параметра для всех сокетов, совместно использующих соединение.

```

macosx % sock -u -s 8888 &
[1] 17612
macosx % sock -u -s -T 8888
can't bind local address: Address already in use
Наконец, задаем параметр SO_REUSEPORT для обоих серверов, и этот вариант работает.
macosx % sock -u -s -T 9999 &
[1] 17614
macosx % sock -u -s -T 9999 &
[2] 17615
macosx % netstat -na | grep 9999
udp4 0 0 *.9999 *.*
```

7.7. Этот параметр (-d) не делает ничего, поскольку программа ping использует ICMP-сокет, а параметр сокета SO_DEBUG влияет только на TCP-сокеты. Описание параметра сокета SO_DEBUG всегда было довольно расплывчатым, наподобие «этот параметр допускает отладку на соответствующем уровне протокола», и единственный уровень протокола, где реализуется данный параметр — это TCP.

7.8. Временная диаграмма приведена на рис. Д.4.

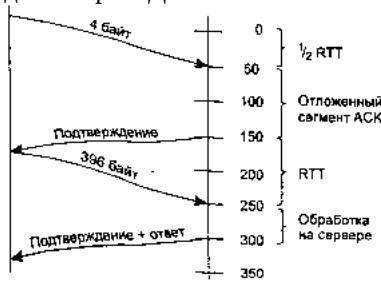


Рис. Д.4. Взаимодействие алгоритма Нагла с задержанными сегментами ACK

7.9. Установка параметра сокета TCP_NODELAY приводит к немедленной отправке данных из второй функции write, даже если имеется еще один небольшой пакет, ожидающий отправки. Это показано на рис. Д.5. Полное время в данном примере превышает 150 мс.

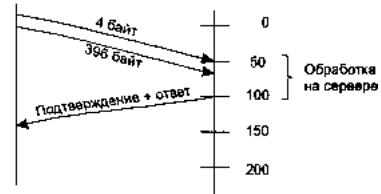


Рис Д.5. Предотвращение алгоритма Нагла путем установки параметра TCP_NODELAY

7.10. Как показано на рис. Д.6, преимущество данного решения состоит в уменьшении числа пакетов.

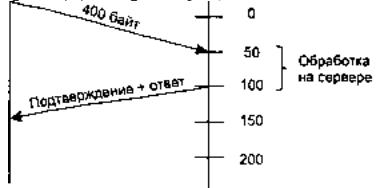


Рис. Д.6. Использование функции writev вместо параметра сокета TCP_NODELAY

7.11. В разделе 4.2.3.2 говорится: «задержка ДОЛЖНА быть меньше 0,5 с, а в потоке полноразмерных сегментов СЛЕДУЕТ использовать сегмент ACK по крайней мере для каждого второго сегмента». Беркли-реализации задерживают сегмент ACK не более, чем на 200 мс [128, с. 821].

7.12. Родительский процесс сервера в листинге 5.1 большую часть времени блокирован в вызове функции accept, а дочерний процесс в листинге 5.2 большую часть времени блокирован в вызове функции read, который содержится в функции readline. Проверка работоспособности с помощью параметра SO_KEEPALIVE не влияет на прослушиваемый сокет, поэтому в случае, если клиентский узел выйдет из строя, родительский процесс не пострадает. Функция read дочернего процесса возвратит ошибку ETIMEDOUT примерно через 2 ч после последнего обмена данными через соединение.

7.13. Клиент, приведенный в листинге 5.4, большую часть времени блокирован вызовом функции fgets, который, в свою очередь, блокирован операцией чтения из стандартной библиотеки ввода-вывода на стандартном устройстве ввода. Когда примерно через 2 ч после последнего обмена данными через соединение истечет время таймера проверки работоспособности и проверочные сообщения не выявят работоспособности сервера, ошибка сокета, ожидающая обработки, примет значение ETIMEDOUT. Но клиент блокирован вызовом функции fgets, поэтому он не увидит этой ошибки, пока не осуществит чтение или запись на сокете. Это одна из причин, по которой в главе 6 листинг 5.4 был изменен таким образом, чтобы использовать функцию select.

7.14. Этот клиент большую часть времени блокирован вызовом функции select, которая сообщает, что сокет готов для чтения, как только ожидающая обработка ошибки будет установлена в ETIMEDOUT (как показано в предыдущем решении).

7.15. Происходит обмен только двумя сегментами, а не четырьмя. Вероятность того, что таймеры двух систем будут строго синхронизированы, очень мала, следовательно, на одном конце соединения таймер проверки работоспособности сработает немного раньше, чем на другом. Первый из сработавших таймеров посыпает проверочное сообщение, заставляя другой конец послать в ответ сегмент ACK. Но получение проверочного сообщения приводит к тому, что таймеру проверки работоспособности с более медленными часами будет присвоено новое значение — он сдвинется на 2 ч вперед.

7.16 Изначально в API сокетов не было функции listen. Вместо этого четвертый аргумент функции socket содержал параметр сокета, а параметр SO_ACCEPTCONN использовался для задания прослушиваемого сокета. Когда добавилась функция listen, флаг остался, но теперь его может устанавливать только ядро [128, с. 456].

Глава 8

8.1. Да. Функция read возвращает 4096 байт данных, а функция recvfrom возвращает 2048 байт (первую из двух дейтаграмм). Функция recvfrom на сокете дейтаграмм никогда не возвращает больше одной дейтаграммы, независимо от того, сколько приложение запрашивает.

8.2. Если протокол использует структуры адреса сокета переменной длины, clilen может быть слишком длинным. В главе 15 будет показано, что это не вызывает проблем со структурами адреса доменного сокета Unix, но корректным решением будет использовать для функции sendto фактическую длину, возвращаемую функцией recvfrom.

8.4. Запуск программы ping с такими параметрами позволяет просмотреть ICMP-сообщения, получаемые узлом, на котором она запущена. Мы используем уменьшенное количество отправляемых пакетов вместо обычного значения 1 пакет в секунду, только чтобы уменьшить объем выводимой на экран информации. Если запустить наш UDP-клиент на узле solaris, указав IP-адрес сервера 192.168.42.1, а затем запустить программу ping, получим следующий вывод:

```
aix % ping -v -I 60 127.0.0.1
PING 127.0.0.1: {127.0.0.1}: 56 data bytes
 64 bytes from 127.0.0.1: icmp_seq=0 ttl=255 time=0 ms
 36 bytes from 192.168.42.1: Destination Port Unreachable
Vr HL TOS Len Fig Off TTL Pro cks Src Dst Data
 4 5 00 0022 0007 0 0000 1e 11 c770 192 168 42.2 192.168.42.1
UDP: from port 40645. to port 9877 (decimal)
```

ПРИМЕЧАНИЕ

Не все версии ping выводят сообщения об ICMP-ошибках, даже если задан параметр -v.

8.5. Прослушиваемый сокет может иметь приемный буфер определенного размера, но прослушиваемым TCP-сокетом данные никогда не принимаются. Большинство реализаций не выделяют заранее память под буферы отправки и приема. Размеры буферов сокета, определяемые параметрами SO_SNDBUF и SO_RCVBUF, являются предельными значениями для соответствующего сокета.

8.6. Запустим программу sock с параметром -u (использовать UDP) и параметром -l (определяет локальный адрес и порт) на многоинтерфейсном узле freebsd.

```
freebsd % sock -u -l 12.106.32.254.4444 192.168.42.2 8888
```

hello

Локальный IP-адрес подключен к Интернету (см. рис. 1.7), но чтобы достичь получателя, дейтаграмма должна выйти через другой интерфейс. Наблюдая за сетью с помощью программы `tcpdump`, мы увидим, что IP-адрес отправителя, связанный с клиентом, не является адресом исходящего интерфейса.

```
14:28:29.614846 12.106.32.254.444 > 192.168.42.2.8888. udp 6
14:28:29.615255 192.168.42.2 > 12 106.32.254: icmp: 192.168 42.2
    udp port 8888 unreachable
```

8.7. Использование функции `printf` на стороне клиента приведет к возникновению задержки между отправками дейтаграмм, что позволит серверу получать большее количество дейтаграмм. Использование функции `printf` на стороне сервера приведет к тому, что сервер будет терять большее количество дейтаграмм.

8.8. Наибольший размер IPv4-дейтаграммы составляет 65 535 байт и ограничивается 16-разрядным полем полной длины, показанным на рис. А.1. IP-заголовок требует 20 байт, UDP-заголовок — 8 байт, и для пользовательских данных остается не более 65 507 байт. В IPv6 (без поддержки джумбограмм) размер IP-заголовка составляет 40 байт, и под пользовательские данные отводится 65 487 байт.

В листинге Д.3 приведена новая версия `dg_cli`. Если забыть установить размер буфера отправки, Беркли-ядра возвратят из функции `sendto` ошибку `EMSGSIZE`, поскольку размер буфера отправки сокета обычно меньше, чем максимальный размер UDP-дейтаграммы (чтобы убедиться в этом, выполните упражнение 7.1).

Листинг Д.3. Запись дейтаграммы UDP/IPv4 максимального размера

```
//udpcliserv/dgclibig.c
1 #include "unp.h"

2 #undef MAXLINE
3 #define MAXLINE 65507

4 void
5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
6 {
7     int size;
8     char sendline[MAXLINE], recvline[MAXLINE + 1];
9     ssize_t n;

10    size = 70000;
11    Setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &size, sizeof(size));
12    Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));

13    Sendto(sockfd, sendline, MAXLINE, 0, pservaddr, servlen);

14    n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

15    printf("received %d bytes\n", n);
16 }
```

Но если установить размеры буферов сокета клиента, как показано в листинге Д.3, и запустить программу, сервер ничего не возвратит. С помощью программы `tcpdump` можно убедиться, что клиентская дейтаграмма отправляется серверу, но если в сервер поместить функцию `printf`, вызов функции `recvfrom` не возвратит дейтаграмму. Проблема заключается в том, что приемный буфер UDP-сокета сервера меньше, чем посланная нами дейтаграмма, поэтому дейтаграмма отбрасывается и не доставляется на сокет. В системах BSD/OS это можно проверить, запустив программу `netstat -s` и проверив счетчик, указывающий количество дейтаграмм, отброшенных из-за переполнения буферов сокета (`dropped due to full socket buffers`), до и после получения нашей длинной дейтаграммы. Решением является модификация сервера путем задания размеров буферов приема и отправки сокета.

В большинстве сетей дейтаграмма длиной 65 535 байт фрагментируется. Как отмечалось в разделе 2.9, IP-уровнем должен поддерживаться размер буфера для сборки фрагментов, равный всего лишь 576 байт. Поэтому некоторые узлы не получат дейтаграмму максимального размера, посыпанную в данном упражнении. Кроме того, во многих Беркли-реализациях, включая 4.4BSD-Lite2, имеется ошибка,

связанная со знаковыми типами данных, которая не позволяет UDP принимать дейтаграммы больше, чем 32 767 байт (см. строка 95, с. 770 [128]).

Глава 9

9.1. В некоторых ситуациях функция `sctp_peeloff` может оказаться очень полезной. Примером приложения, которому может понадобиться эта функция, является традиционный сервер дейтаграмм, обрабатывающий небольшие транзакции, которому периодически приходится устанавливать долговременные соединения. Чаще всего сервер передает одно-два коротких сообщения, но время от времени поступает запрос на проверку базы сервера, и тогда ему приходится передавать большие объемы данных. В такой ситуации имеет смысл отделить ассоциацию, по которой передаются проверочные данные, для обработки ее отдельным процессом или потоком.

9.2. На стороне сервера выполняется автоматическое закрытие после закрытия ассоциации клиентом. SCTP не поддерживает состояние неполного закрытия, поэтому когда клиент вызывает `close`, все подготовленные сервером данные сбрасываются и ассоциация закрывается.

9.3. Сокет типа «один-к-одному» требует вызова `connect`, поэтому когда собеседнику отсылается сегмент COOKIE, никаких данных в буфере отправки быть еще не может. Сокет типа «один-ко-многим» допускает отправку данных с одновременной установкой соединения. Поэтому сегмент COOKIE в этом случае может быть совмещен с сегментом DATA.

9.4. Собеседник, с которым устанавливается ассоциация, может прислать данные только в том случае, если у него будет готов сегмент DATA до того, как соединение будет установлено, то есть если на обеих сторонах используются сокеты типа «один-ко-многим» и каждая сторона выполняет операцию `send` с неявной установкой соединения. Такой процесс установки ассоциации называется коллизией пакетов INIT и подробно описывается в главе 4 [117].

9.5. В некоторых случаях не все связанные адреса могут быть переданы собеседнику. В частности, если приложение связало с сокетом как частные, так и общие IP-адреса, собеседник получит информацию только об общих IP-адресах. Еще одним примером являются локальные в рамках канала адреса IPv6, которые не обязательно сообщаются собеседнику.

Глава 10

10.1 Если функция `sctp_sendmsg` возвращает ошибку, сообщение не будет отправлено, а приложение вызовет функцию `sctp_recvmsg` и заблокируется в ней навсегда, ожидая ответного сообщения, которое никогда не придет.

Чтобы избежать этой неприятности, нужно проверять коды возврата. Если при отправке возникла ошибка, клиент не должен пытаться получить ответ. Ему следует просто сообщить о возникшей ошибке.

Если функция `sctp_recvmsg` вернет ошибку, никаких сообщений получено не будет, но сервер все равно попытается отправить сообщение, что может привести к установлению ассоциации. Для предотвращения этого следует проверять код ошибки и, в зависимости от его значения, сообщать об ошибке и закрывать сокет (при этой операции также может быть возвращена ошибка) или повторно вызывать `sctp_recvmsg`.

10.2. Если сервер получает запрос и завершает работу, клиент (в его нынешней форме) зависает навечно в ожидании ответа сервера. Клиенту следует включить доставку уведомлений о событиях для данной ассоциации. Когда сервер завершит работу, клиент получит соответствующее сообщение и сможет принять какие-либо меры, например связаться с другим сервером. Альтернативным решением может быть установка таймера и завершение работы по истечении времени ожидания.

10.3. Чтобы каждая порция данных была помещена в свой пакет, мы установили размер сообщения 800 байт. Более правильным решением будет получение значения параметра сокета `SCTP_MAXSEG` для определения размера данных, помещающихся в один пакет.

10.4. Алгоритм Нагла (управляемый параметром сокета `SCTP_NODELAY`, см. раздел 7.10) вызывает проблемы только при передаче данных небольших объемов. Если данные передаются порциями такого размера, что SCTP вынужден передавать их немедленно, никакого замедления быть не может. Установка небольшого размера `out_sz` исказит результаты, потому что в некоторых случаях передача будет задерживаться до получения выборочных уведомлений от собеседника. Поэтому при передаче данных небольшого размера алгоритм Нагла следует отключать.

10.5. Если приложение устанавливает ассоциацию и изменяет количество потоков, количество потоков в данной ассоциации не меняется. Количество потоков может быть задано только для новых ассоциаций, но не для существующих.

Сокет типа «один-ко-многим» позволяет устанавливать ассоциации неявно. Для изменения параметров ассоциации необходимо вызывать sendmsg со вспомогательными данными. Фактически при этом обязательно использовать неявное установление ассоциации.

Глава 11

11.1. В листинге Д.4 приведена программа, вызывающая функцию gethostbyaddr.

Листинг Д.4. Изменение листинга 11.1 для вызова функции gethostbyaddr

```
//names/hostent2.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     char *ptr, **pptr;
6     char str[INET6_ADDRSTRLEN];
7     struct hostent *hptr;

8     while (--argc > 0) {
9         ptr = *++argv;
10        if ( (hptr = gethostbyname(ptr)) == NULL) {
11            err_msg("gethostbyname error for host: %s: %s",
12                  ptr, hstrerror(h_errno));
13            continue;
14        }
15        printf("official hostname: %s\n", hptr->h_name);
16        for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
17            printf(" alias: %s\n", *pptr);

18        switch (hptr->h_addrtype) {
19        case AF_INET:
20 #ifdef AF_INET6
21        case AF_INET6:
22 #endif
23            pptr = hptr->h_addr_list;
24            for (; *pptr != NULL; pptr++) {
25                printf("\taddress: %s\n",
26                      Inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));

27                if ((hptr = gethostbyaddr(*pptr, hptr->h_length,
28                    ptr->h_addrtype)) == NULL)
29                    printf("\t(gethostbyaddr failed)\n");
30                else if (hptr->h_name != NULL)
31                    printf("\tname = %s\n", hptr->h_name);
32                else
33                    printf("\t(no hostname returned by gethostbyaddr)\n");
34            }
35            break;

36        default:
37            err_ret("unknown address type");
38            break;
39    }
```

```
40 }
41 exit(0);
42 }
```

Эта программа корректно работает на узле с единственным IP-адресом. Если запустить программу из листинга 11.1 на узле с четырьмя IP-адресами, то получим:

```
freebsd % hostent cnn.com
official hostname: cnn.com
address: 64.236.16.20
address: 64.236.16.52
address: 64.236.16.84
address: 64.236.16.116
address: 64.236.24.4
address: 64.236.24.12
address: 64.236.24.20
address: 64.236.24.28
```

Но если запустить программу из листинга Д.4 на том же узле, в выводе будет только первый IP-адрес:

```
freebsd % hostent2 cnn.com
official hostname: cnn.com
address: 64.236.24.4
name = www1.cnn.com
```

Проблема заключается в том, что две функции, `gethostbyname` и `gethostbyaddr`, совместно используют одну и ту же структуру `hostent`, как было показано в разделе 11.18. Когда наша новая программа вызывает функцию `gethostbyaddr`, она повторно использует данную структуру вместе с областью памяти, на которую структура указывает (массив указателей `h_addr_list`), стирая три оставшиеся IP-адреса, возвращаемые функцией `gethostbyname`.

11.2. Если ваша система не поддерживает повторно входимую версию функции `gethostbyaddr` (см. раздел 11.19), то прежде чем вызывать функцию `gethostbyaddr`, вам следует создать копию массива указателей, возвращаемых функцией `gethostbyname`, и данных, на которые указывает этот массив.

11.3. Сервер `chargen` отправляет клиенту данные до тех пор, пока клиент не закрывает соединение (то есть пока вы не завершите выполнение клиента).

11.4. Эта возможность поддерживается некоторыми распознавателями, но переносимая программа не может использовать ее, потому что POSIX никак ее не оговаривает. В листинге Д.5 приведена измененная версия. Порядок тестирования строки с именем узла имеет значение. Сначала мы вызываем функцию `inet_pton`, поскольку она обеспечивает быстрый тест «внутри памяти» (*in-memory*) для проверки, является ли строка допустимым IP-адресом в точечно-десятичной записи. Только если тест заканчивается неудачно, мы запускаем функцию `gethostbyname`, которая обычно требует некоторых сетевых ресурсов и времени.

Если строка является допустимым IP-адресом в точечно-десятичной записи, мы создаем свой массив указателей (`addrs`) на один IP-адрес, оставив без изменений цикл, использующий `pptr`.

Поскольку адрес уже был переведен в двоичное представление в структуре адреса сокета, мы заменяем вызов функции `memscpy` в листинге 11.2 на вызов функции `memmove`, так как при вводе IP-адреса в точечно-десятичной записи исходное и конечное поля в данном вызове одинаковые.

Листинг Д.5. Допускаем как использование IP-адреса в точечно-десятичной записи, так и задание имени узла, номера порта или имени службы

```
//names/daytimetccli2.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd, n;
6     char recvline[MAXLINE + 1];
7     struct sockaddr_in servaddr;
8     struct in_addr **pptr, *addrs[2];
9     struct hostent *hp;
10    struct servent *sp;
```

```

11 if (argc != 3)
12 err_quit("usage: daytimetcpccli2 <hostname> <service>");

13 bzero(&servaddr, sizeof(servaddr));
14 servaddr.sin_family = AF_INET;

15 if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) == 1) {
16     addrs[0] = &servaddr.sin_addr;
17     addrs[1] = NULL;
18     pptr = &addrs[0];
19 } else if ((hp = gethostbyname(argv[1])) != NULL) {
20     pptr = (struct in_addr**)hp->h_addr_list;
21 } else
22     err_quit("hostname error for %s: %s", argv[1], hstrerror(h_errno));

23 if ((n = atoi(argv[2])) > 0)
24     servaddr.sin_port = htons(n);
25 else if ((sp = getservbyname(argv[2], "tcp")) != NULL)
26     servaddr.sin_port = sp->s_port;
27 else
28     err_quit("getservbyname error for %s", argv[2]);

29 for (; *pptr != NULL; pptr++) {
30     sockfd = Socket(AF_INET, SOCK_STREAM, 0);

31     memmove(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
32     printf("trying %s\n",
33           Sock_ntop((SA*)&servaddr, sizeof(servaddr)));

34     if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) == 0)
35         break; /* успех */
36     err_ret("connect error");
37     close(sockfd);
38 }
39 if (*pptr == NULL)
40     err_quit("unable to connect");

41 while ((n = Read(sockfd, recvline, MAXLINE)) > 0) {
42     recvline[n] = 0; /* завершающий нуль */
43     Fputs(recvline, stdout);
44 }
45 exit(0);
46 }

```

11.5. Программа приведена в листинге Д.6.

Листинг Д.6. Модификация листинга 11.2 для работы с IPv4 и IPv6

```

//names/daytimetcpccli3.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd, n;
6     char recvline[MAXLINE + 1];
7     struct sockaddr_in servaddr;
8     struct sockaddr_in6 servaddr6;
9     struct sockaddr *sa;

```

```

10  socklen_t sal_en;
11  struct in_addr **pptr;
12  struct hostent *hp;
13  struct servent *sp;

14  if (argc != 3)
15    err_quit("usage: daytimetcpccli3 <hostname> <service>");

16  if ((hp = gethostbyname(argv[1])) == NULL)
17    err_quit("hostname error for %s: %s", argv[1], hstrerror(h_errno));

18  if ((sp = getservbyname(argv[2], "tcp")) == NULL)
19    err_quit("getservbyname error for %s", argv[2]);

20  pptr = (struct in_addr**)hp->h_addr_list;
21  for (; *pptr != NULL; pptr++) {
22    sockfd = Socket(hp->h_addrtype, SOCK_STREAM, 0);

23    if (hp->h_addrtype == AF_INET) {
24      sa = (SA*)&servaddr;
25      salen = sizeof(servaddr);
26    } else if (hp->h_addrtype == AF_INET6) {
27      sa = (SA*)&servaddr6;
28      salen = sizeof(servaddr6);
29    } else
30      err_quit("unknown addrtype %d", hp->h_addrtype);

31    bzero(sa, salen);
32    sa->sa_family = hp->h_addrtype;
33    sock_set_port(sa, salen, sp->s_port);
34    sock_set_addr(sa, salen, *pptr);

35    printf("trying %s\n", Sock_ntop(sa, salen));

36    if (connect(sockfd, sa, salen) == 0)
37      break; /* успех */
38    err_ret("connect error");
39    close(sockfd);
40  }
41  if (*pptr == NULL)
42    err_quit("unable to connect");

43  while ((n = Read(sockfd, recvline, MAXLINE)) > 0) {
44    recvline[n] = 0; /* завершающий нуль */
45    Fputs(recvline, stdout);
46  }
47  exit(0);
48 }

```

Используем значение `h_addrtype`, возвращаемое функцией `gethostbyname`, для определения типа адреса. Также используем функции `sock_set_port` и `sock_set_addr` (см. раздел 3.8), чтобы установить два соответствующих поля в структуре адреса сокета.

Эта программа работает, однако имеется два ограничения. Во-первых, мы должны обрабатывать все различия, следя за `h_addrtype` и задавая соответствующим образом `sa` или `salen`. Более удачным решением было бы иметь библиотечную функцию, которая не только просматривает имя узла и имя службы, но и заполняет всю структуру адреса сокета (например, `getaddrinfo`, см. раздел 11.6). Во-вторых, эта программа компилируется только на узлах с поддержкой IPv6. Чтобы ее можно было откомпилировать на узле,

поддерживающим только IPv4, следует добавить в код огромное количество директив `#ifdef`, что, несомненно, усложнит программу.

11.7. Разместите в памяти большой буфер (превышающий по размеру любую структуру адреса сокета) и вызовите функцию `getsockname`. Третий аргумент является аргументом типа «значение-результат», возвращающим фактический размер адресов протоколов. К сожалению, это допускают только структуры адреса сокета с фиксированной длиной (IPv4 и IPv6). Нет гарантии, что этот буфер будет работать с протоколами, которые могут вернуть структуру адреса сокета переменной длины (доменные сокеты Unix, см. главу 15).

11.8. Сначала размещаем в памяти массивы, содержащие имя узла и имя службы:

```
char host[NI_MAXHOST], serv[NI_MAXSERV];
```

После того как функция `accept` возвращает управление, вызываем вместо функции `sock_ntop` функцию `getnameinfo`:

```
if (getnameinfo(cliaddr, len, host, NI_MAXHOST, serv, NI_MAXSERV,
    NI_NUMERICHOST | NI_NUMERICSERV) == 0)
    printf("connection from %s.%s\n", host, serv);
```

Поскольку мы имеем дело с сервером, определяем флаги `NI_NUMERICHOST` и `NI_NUMERICSERV`, чтобы избежать поиска в DNS и `/etc/services`.

11.9. Первая проблема состоит в том, что второй сервер не может связаться (`bind`) с тем же портом, что и первый сервер, поскольку не установлен параметр сокета `SO_REUSEADDR`. Простейший способ справиться с такой ситуацией — создать копию функции `udp_server`, переименовать ее в `udp_server_reuseaddr`, сделать так, чтобы она установила параметр сокета, и вызывать ее в сервере.

11.10. Когда клиент выводит `Trying 206.62.226.35...`, функция `gethostname` возвращает IP-адрес. Пауза перед этим выводом означает, что распознаватель ищет имя узла. Вывод `Connected to bsdi.unpbook.com.` значит, что функция `connect` возвратила управление. Пауза между этими двумя выводами говорит о том, что функция `connect` пытается установить соединение.

Глава 12

12.1. Далее приведен сокращенный листинг. Обратите внимание, что клиент FTP в системе freebsd всегда пытается использовать команду EPRT (независимо от версии IP), но если это не срабатывает, то он пробует команду PORT.

```
freebsd % ftp aix-4
Connected to aix-4.unpbook.com.
220 aix FTP server ...
...
230 Guest login ok. access restrictions apply.
ftp> debug
Debugging on (debug=1).
ftp> passive
Passive mode: off; fallback to active mode= off
ftp> dir
---> EPRT |1|192.168.42.1|50484|
500 'EPRT |1|192.168.42.1|50484|' command not understood.
disabling epsv4 for this connection
---> PORT 192.168.42.1.197.52
200 PORT command successful.
---> LIST
150 Opening ASCII mode data connection for /bin/ls
...
freebsd % ftp ftp.kame.net
Trying 2001.200.0.4819:203:47ff:fea5:3085...
Connected to orange.kame.net.
220 orange.kame.net FTP server ...
...
230 Guest login ok. access restrictions apply.
ftp> debug
```

```

Debugging on (debug=1).
ftp> passive
Passive mode: off; fallback to active mode: off.
ftp> dir
---> EPRT |2|3ffe:b80:3:9ad1::2|50480|
200 EPRT command successful
---> LIST
150 Opening ASCII mode data connection for '/bin/ls'.

```

Глава 13

13.1. Все сообщения об ошибках, даже ошибка загрузки, такая как неправильный аргумент командной строки, должны сохраняться в файлах журнала с помощью функции `syslog`.

13.2. TCP-версии серверов `echo`, `discard` и `chargen` запускаются как дочерние процессы, после того как демон `inetd` вызовет функцию `fork`, поскольку эти три сервера работают, пока клиент не прервёт соединение. Два других TCP-сервера, `time` и `daytime`, не требуют использования функции `fork`, поскольку эти службы легко реализовать (получить текущую дату, преобразовать ее, записать и закрыть соединение). Эти два сервера обрабатываются непосредственно демоном `inetd`. Все пять UDP-служб обрабатываются без использования функции `fork`, поскольку каждая из них генерирует единственную дейтаграмму в ответ на клиентскую дейтаграмму, которая запускает эту службу. Эти пять служб обрабатываются напрямую демоном `inetd`.

13.3. Это известная атака типа «отказ в обслуживании» [18]. Первая дейтаграмма с порта 7 заставляет сервер `chargen` отправить дейтаграмму обратно на порт 7. На эту дейтаграмму приходит эхо-ответ, и серверу `chargen` посыпается другая дейтаграмма. Происходит зацикливание. Одним из решений, реализованным в системе BSD/OS, является игнорирование дейтаграмм, направленных любому внутреннему серверу, если номер порта отправителя пришёлшей дейтаграммы принадлежит одному из внутренних серверов. Другим решением может быть запрещение этих внутренних служб — либо с помощью демона `inetd` на каждом узле, либо на маршрутизаторе, связывающем внутреннюю сеть организации с Интернетом.

13.4. IP-адрес и номер порта клиента могут быть получены из структуры адреса сокета, заполняемой функцией `accept`.

Причина, по которой демон `inetd` не делает этого для UDP-сокета, состоит в том, что чтение дейтаграмм (`recvfrom`) осуществляется с помощью функции `exec` сервером, а не самим демоном `inetd`.

Демон `inetd` может считывать дейтаграмму с флагом `MSG_PEEK` (см. раздел 14.7), только чтобы получить IP-адрес и номер порта клиента, но оставляет саму дейтаграмму для чтения серверу.

Глава 14

14.1. Если не установлен обработчик, первый вызов функции `signal` будет возвращать значение `SIG_DFL`, а вызов функции `signal` для восстановления обработчика просто вернет его в исходное состояние.

14.3. Приведем цикл `for`:

```

for (;;) {
    if ((n = Recv(sockfd, recvline, MAXLINE, MSG_PEEK)) == 0)
        break; /* сервер закрыл соединение */

    ioctl(sockfd, FIONREAD, &npend);
    printf("%d bytes from PEEK, %d bytes pending\n", n, npend);

    n = Read(sockfd, recvline, MAXLINE);
    recvline[n] = 0; /* завершающий нуль */
    Fputs(recvline, stdout);
}

```

14.4. Данные продолжают выводиться, поскольку выход из функции `main` — это то же самое, что и возврат из этой функции. Функция `main` вызывается программой запуска на языке С следующим образом:

```
exit(main(argc, argv));
```

Следовательно, вызывается функция `exit`, а затем и программа очистки стандартного ввода-вывода.

Глава 15

15.1. Функция `unlink` удаляет имя файла из файловой системы, и когда клиент позже вызовет функцию `connect`, она не выполнится. Это не влияет на прослушиваемый сокет сервера, но клиенты не смогут выполнить функции `connect` после вызова функции `unlink`.

15.2. Клиент не сможет соединиться с сервером с помощью функции `connect`, даже если полное имя существует, поскольку для успешного соединения с помощью функции `connect` доменный сокет Unix должен быть открыт и связан с этим полным именем (см. раздел 15.4).

15.3. При выводе адреса протокола клиента путем вызова функции `sock_ntop` мы получим сообщение `datagram from (no pathname bound)` (дайтаграмма от (имя не задано)), поскольку по умолчанию с сокетом клиента не связывается никакое имя.

Одним из решений является проверить доменный сокет Unix в функциях `udp_client` и `udp_connect` и связать с сокетом при помощи функции `bind` временное полное имя. Это приведет к зависимости от протокола в библиотечной функции, но не в нашем приложении.

15.4. Даже если мы заставим сервер вернуть в функции `write` 1 байт на его 26- байтовый ответ, использование функции `sleep` на стороне клиента гарантирует, что все 26 сегментов будут получены до вызова функции `read`, в результате чего функция `read` вернет полный ответ. Это еще одно подтверждение того, что TCP является потоком байтов с отсутствием границ записей.

Чтобы использовать доменные протоколы Unix, запускаем клиент и сервер с двумя аргументами командной строки `/local` (или `/unix`) и `/tmp/daytime` (или любое другое временное имя, которое вы хотите использовать). Ничего не изменится: 26 байт будут возвращаться функцией `read` каждый раз, когда будет запускаться клиент.

Поскольку для каждой функции `send` сервер определяет флаг `MSG_EOR`, каждый байт рассматривается как логическая запись, и функция `read` при каждом вызове возвращает 1 байт. Причина в том, что Беркли-реализации поддерживают флаг `MSG_EOR` по умолчанию. Однако этот факт не документирован и не может использоваться в серийном коде. В данном примере мы используем эту особенность, чтобы показать разницу между потоком байтов и ориентированным на записи протоколом. С точки зрения реализации, каждая операция вывода идет в `mbuf` (буфер памяти) и флаг `MSG_EOR` сохраняется ядром вместе с `mbuf`, когда `mbuf` переходит из отправляющего сокета в приемный буфер принимающего сокета. Когда вызывается функция `read`, флаг `MSG_EOR` все еще присоединен к каждому `mbuf`, так что основная подпрограмма ядра `read` (поддерживающая флаг `MSG_EOR`, поскольку некоторые протоколы используют этот флаг) сама возвращает каждый байт. Если бы вместо `read` мы использовали `recvmsg`, флаг `MSG_EOR` возвращался бы в поле `msg_flags` каждый раз, когда `recvmsg` возвращала бы 1 байт. Такой подход в TCP не срабатывает, поскольку отправляющий TCP не анализирует флаг `MSG_EOR` в отсылаемом `mbuf` и любом случае у нас нет возможности передать этот флаг принимающему TCP в TCP-заголовке. (Выражаем благодарность Мату Томасу (Matt Thomas) за то, что он указал нам это недокументированное «средство».)

15.5. В листинге Д.7 приведена реализация данной программы.

Листинг Д.7. Определение фактического количества собранных в очередь соединений для различных значений аргумента `backlog`

```
//debug//backlog.c
1 #include "unp.h"

2 #define PORT 9999
3 #define ADDR "127 0.0.1"
4 #define MAXBACKLOG 100

5 /* глобальные переменные */
6 struct sockaddr_in serv;
7 pid_t pid; /* дочерний процесс */

8 int pipefd[2];
9 #define pfd pipefd[1] /* сокет родительского процесса */
10 #define cfd pipefd[0] /* сокет дочернего процесса */
```

```
11 /* прототипы функций */
12 void do_parent(void);
13 void do_child(void);

14 int
15 main(int argc, char **argv)
16 {
17     if (argc != 1)
18         err_quit("usage: backlog");

19     Socketpair(AF_UNIX, SOCK_STREAM, 0, pipefd);

20     bzero(&serv, sizeof(serv));
21     serv.sin_family = AF_INET;
22     serv.sin_port = htons(PORT);
23     Inet_nton(AF_INET, ADDR, &serv.sin_addr);

24     if ((pid = Fork()) == 0)
25         do_child();
26     else
27         do_parent();

28     exit(0);
29 }

30 void
31 parent_alrm(int signo)
32 {
33     return; /* прерывание блокированной функции connect() */
34 }

35 void
36 do_parent(void)
37 {
38     int backlog, j, k, junk, fd[MAXBACKLOG + 1];

39     Close(cfd);
40     Signal(SIGALRM, parent_alrm);

41     for (backlog = 0; backlog <= 14; backlogs) {
42         printf("backlog = %d. ", backlog);
43         Write(pfd, &backlog, sizeof(int)); /* сообщение значения дочернему процессу */
44         Read(pfd, &junk, sizeof(int)); /* ожидание дочернего процесса */

45         for (j = 1; j <= MAXBACKLOG; j++) {
46             fd[j] = Socket(AF_INET, SOCK_STREAM, 0);
47             alarm(2);
48             if (connect(fd[j], (SA*)&serv, sizeof(serv)) < 0) {
49                 if (errno != EINTR)
50                     err_sys("connect error, j = %d", j);
51                 printf("timeout, %d connections completed\n", j - 1);
52                 for (k = 1; k <= j; k++)
53                     Close(fd[k]);
54                 break; /* следующее значение backlog */
55             }
56         }
57         alarm(0);
58     }
59 }
```

```

57    }
58    if (j > MAXBACKLOG)
59      printf("Id connections?\n", MAXBACKLOG);
60  }
61 backlog = -1; /* сообщаем дочернему процессу, что все сделано */
62 Write(pfd, &backlog, sizeof(int));
63 }

64 void
65 do_child(void)
66 {
67   int listenfd, backlog, junk;
68   const int on = 1;

69 Close(pfd);

70 Read(cfd, &backlog, sizeof(int)); /* ожидание родительского процесса */
71 while (backlog >= 0) {
72   listenfd = Socket(AF_NET, SOCK_STREAM, 0);
73   Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
74   Bind(listenfd, (SA*)&serv, sizeof(serv));
75   Listen(listenfd, backlog); /* начало прослушивания */

76   Write(cfd, &junk, sizeof(int)); /* сообщение родительскому процессу */

77   Read(cfd, &backlog, sizeof(int)); /* ожидание родительского процесса */
78   Close(listenfd); /* также закрывает все соединения в очереди */
79 }
80 }

```

Глава 16

16.1. Дескриптор используется совместно родительским и дочерним процессами, поэтому его счетчик ссылок равен 2. Если родительский процесс вызывает функцию `close`, счетчик ссылок уменьшается с 2 до 1, и пока он больше нуля, сегмент FIN не посыпается. Еще одна цель вызова функции `shutdown` — послать сегмент FIN, даже если дескриптор больше нуля.

16.2. Родительский процесс продолжит запись в сокет, получивший сегмент FIN, а первый сегмент, посланный серверу, вызовет получение сегмента RST в ответ. После этого функция `write` пошлет родительскому процессу сигнал `SIGPIPE`, как показано в разделе 5.12.

16.3. Когда дочерний процесс вызывает функцию `getppid` для отправки сигнала `SIGTERM`, возвращаемый идентификатор процесса будет равен 1. Это указывает на процесс `init`, наследующий все продолжающие работать дочерние процессы, родительские процессы которых завершились. Дочерний процесс будет пытаться послать сигнал процессу `init`, не имея необходимых прав доступа. Но если не исключается, что данный клиент будет запущен с правами привилегированного пользователя, позволяющими посыпать сигналы процессу `init`, то возвращенное функцией `getppid` значение должно быть проверено перед отправкой сигнала.

16.4. Если удалить эти две строки, вызывается функция `select`. Но функция `select` немедленно завершится, поскольку соединение установлено и сокет открыт для записи. Эта проверка и оператор `goto` предотвращают ненужный вызов функции `select`.

16.5. Это может случиться, если сервер отправляет данные сразу, как только завершается его функция `accept`, и если узел клиента занят, когда приходит второй пакет трехэтапного рукопожатия для завершения соединения со стороны клиента (см. рис. 2.5). SMTP-серверы, например, немедленно отсыпают клиенту сообщение по новому соединению, прежде чем произвести из него считывание.

Глава 17

17.1. Нет, это не имеет значения, поскольку первые три элемента объединения в листинге 17.1 являются структурами адреса сокета.

Глава 18

18.1. Элемент `sdl_nlen` будет равен 5, а элемент `sdl_alen` будет равен 8. Для этого требуется 21 байт, поэтому размер округляется до 24 байт [128, с. 89] в предположении, что используется 32-разрядная архитектура.

18.2. На этот сокет никогда не посыпается ответ от ядра. Данный параметр сокета (`SO_USELOOPBACK`) определяет, посыпает ли ядро ответ отправляющему процессу, как показано на с. 649-650 [128]. По умолчанию этот параметр включен, поскольку большинство процессов ожидают ответа. Но отключение данного параметра препятствует отправке ответов отправителю.

Глава 20

20.1. Если вы получаете большое количество ответов, они могут следовать каждый раз в разном порядке. Правда, отправляющий узел обычно выводится первым, поскольку дейтаграммы, направленные к нему или от него, не появляются в реальной сети.

20.2. Когда в FreeBSD обработчик сигналов записывает байт в канал, а затем завершается, функция `select` возвращает ошибку `EINTR`. Она вызывается заново и при завершении сообщает о возможности чтения из канала.

Глава 21

21.1. Если запустить программу, то она не выведет ничего. Для предотвращения получения многоадресных дейтаграмм сервером, не ожидающим их, ядро не доставляет дейтаграммы на сокет, не выполнявший никаких многоадресных операций (в частности, не присоединявшийся к группам). Происходит следующее. В адресе получателя UDP-дейтаграммы стоит 224.0.0.1 — это группа всех узлов, в которой должны состоять узлы, поддерживающие многоадресную передачу. UDP-дейтаграмма посыпается как многоадресный кадр Ethernet, и все узлы с поддержкой многоадресной передачи должны получить ее, поскольку все они входят в указанную группу. Все отвечающие узлы передают полученную UDP-дейтаграмму серверу времени и даты (обычно он является частью демона `inetd`), даже если этот сокет не находится в группе. Однако ядро сбрасывает полученную дейтаграмму, потому что процесс, связанный с портом сервера времени и даты, не установил параметры многоадресной передачи.

21.2. В листинге Д.8 показаны простые изменения функции `main` для связывания (`bind`) с адресом многоадресной передачи и портом 0.

Листинг Д.8. Функция `main` UDP-клиента, осуществляющая связывание с адресом многоадресной передачи

```
//mcast/udpcli06.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int sockfd;
6     socklen_t salen;
7     struct sockaddr *cli, *serv;

8     if (argc != 2)
9         err_quit("usage: udpcli06 <Ipaddress>");

10    sockfd = Udp_client(argv[1], "daytime", (void**)&serv, &salen);

11    cli = Malloc(salen);
12    memcpy(cli, serv, salen); /* копируем структуру адреса сокета */
```

```
13 sock_set_port(cli, salen, 0); /* и устанавливаем порт в 0 */
14 Bind(sockfd, cli, salen);

15 dg_cli(stdin, sockfd, serv, salen);

16 exit(0);
17 }
```

К сожалению, все три системы, на которых проводилась проверка — FreeBSD 4.8, MacOS X и Linux 2.4.7, — позволяют использовать функцию `bind`, а затем посылают UDP-дейтаграммы с IP-адресом многоадресной передачи отправителя.

21.3. Если мы запустим программу `ping` для группы узлов 224.0.0.1 на нашем узле `aix`, получим следующий вывод:

```
solaris % ping 224.0.0.1
PING 224.0.0.1: 56 data bytes
64 bytes from 192.168.42.2: icmp_seq=0 ttl=255 time=0 ms
64 bytes from 192.168.42.1: icmp_seq=0 ttl=64 time=1 ms (DUP!)
^C
---224.0.0.1 PING Statistics---
1 packets transmitted. 1 packets received. +1 duplicates. 0% packet loss
round-trip min/avg/max = 0/0/0 ms
```

Ответили оба узла в правой сети Ethernet на рис. 1.7.

ПРИМЕЧАНИЕ

Для предотвращения определенных типов атак некоторые системы не отвечают на широковещательные и многоадресные ICMP-запросы. Чтобы получить ответ от freebsd, нам пришлось специально настроить эту систему:

```
freebsd % sysctl net.inet.icmp.bmcastecho=1
```

21.5. Величина 1 073 741 824 преобразуется в значение с плавающей точкой и делится на 4 294 967 296, что дает значение 0,250. В результате умножения на 1 000 000 получаем значение 250 000 в микросекундах, а это одна четверть секунды. Наибольшая дробная часть получается при делении 4 294 967 295 на 429 4967 296 и составляет 0,99 999 999 976 716 935 634. Умножая это число на 1 000 000 и отбрасывая дробную часть, получаем 999 999 — наибольшее значение количества микросекунд.

Глава 22

22.1. Вспомните, что функция `sock_ntop` использует свой собственный статический буфер для хранения результата. Если мы вызовем ее дважды в качестве аргумента в вызове `printf`, второй вызов приведет к перезаписи результата первого вызова.

22.2. Да, если ответ содержит 0 байт пользовательских данных (например, структура `hdr`).

22.3. Поскольку функция `select` не изменяет структуру `timeval`, которая определяет ее ограничение по времени, нам следует заметить время отправки первого пакета (оно возвращается в миллисекундах функцией `rtt_ts`). Если функция `select` сообщает, что сокет готов к чтению, заметьте текущее время, а если функция `recvmsg` вызывается повторно, вычислите новый тайм-аут для функции `select`.

22.4. Обычным решением будет создать по одному сокету на каждый адрес интерфейса, как было сделано в разделе 22.6, и отправлять ответ с того же сокета, на который пришел запрос.

22.5. Вызов функции `getaddrinfo` без аргумента имени узла и без флага `AI_PASSIVE` заставляет эту функцию считать, что используется локальный адрес 0::1 (для IPv6) или 127.0.0.1 (для IPv4). Напомним, что структура адреса сокета IPv6 возвращается функцией `getaddrinfo` перед структурой адреса сокета IPv4 при условии, что поддерживается протокол IPv6. Если узел поддерживает оба протокола, вызов функции `socket` в `udp_client` закончится успешно при указании семейства протоколов `AF_INET6`.

В листинге Д.9 приведена не зависящая от протокола версия программы.

Листинг Д.9. Не зависящая от протокола версия программы из раздела 22.6

```
//advio/udpserv04.c
```

```

1 #include "unpifi.h"

2 void mydg_echo(int, SA*, socklen_t);

3 int
4 main(int argc, char **argv)
5 {
6     int sockfd, family, port;
7     const int on = 1;
8     pid_t pid;
9     socklen_t salen;
10    struct sockaddr *sa, *wild;
11    struct ifi_info *ifi, *ifihead;

12    if (argc == 2)
13        sockfd = Udp_client(NULL, argv[1], (void**)&sa, &salen);
14    else if (argc == 3)
15        sockfd = Udp_client(argv[1], argv[2], (void**)&sa, &salen);
16    else
17        err_quit("usage: udpserv04 [ <host> ] <service or port>");
18    family = sa->sa_family;
19    port = sock_get_port(sa, salen);
20    Close(sockfd); /* хотим узнать семейство, порт salen */

21    for (ifihead = ifi = Get_ifi_info(family, 1),
22         ifi != NULL; ifi = ifi->ifi_next) {

23        /* связывание с многоадресными адресами */
24        sockfd = Socket(family, SOCK_DGRAM, 0);
25        Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

26        sock_set_port(ifi->ifi_addr, salen, port);
27        Bind(sockfd, ifi->ifi_addr, salen);
28        printf("bound %s\n", Sock_ntop(ifi->ifi_addr, salen));

29        if ((pid = Fork()) == 0) { /* дочерний процесс */
30            mydg_echo(sockfd, ifi->ifi_addr, salen);
31            exit(0); /* никогда не выполняется */
32        }
33        if (ifi->ifi_flags & IFF_BROADCAST) {
34            /* попытка связывания с широковещательным адресом */
35            sockfd = Socket(family, SOCK_DGRAM, 0);
36            Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

37            sock_set_port(ifi->ifi_brdaddr, salen, port);
38            if (bind(sockfd, ifi->ifi_brdaddr, salen) < 0) {
39                if (errno == EADDRINUSE) {
40                    printf("EADDRINUSE: %s\n",
41                           Sock_ntop(ifi->ifi_brdaddr, salen));
42                    Close(sockfd);
43                    continue;
44                } else
45                    err_sys("bind error for %s",
46                           Sock_ntop(ifi->ifi_brdaddr, salen));
47            }
48            printf ("bound %s\n", Sock_ntop(ifi->ifi_brdaddr, salen));

```

```

49     if ((pid = Fork()) == 0) { /* дочерний процесс */
50         mydg_echo(sockfd, ifi->ifi_brdaddr, salen);
51         exit(0); /* никогда не выполняется */
52     }
53 }
54 }

55 /* связывание с универсальным адресом */
56 sockfd = Socket(family, SOCK_DGRAM, 0);
57 Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

58 wild = Malloc(salen);
59 memcpy(wild, sa, salen); /* копирует семейство и порт */
60 sock_set_wild(wild, salen);

61 Bind(sockfd, wild, salen);
62 printf("bound %s\n", Sock_ntop(wild, salen));

63 if ((pid = Fork()) == 0) { /* дочерний процесс */
64     mydg_echo(sockfd, wild, salen);
65     exit(0); /* никогда не выполняется */
66 }
67 exit(0);
68 }

69 void
70 mydg_echo(int sockfd, SA *myaddr, socklen_t salen)
71 {
72     int n;
73     char mesg[MAXLINE];
74     socklen_t len;
75     struct sockaddr *cli;

76     cli = Malloc(salen);

77     for (;;) {
78         len = salen;
79         n = Recvfrom(sockfd, mesg, MAXLINE, 0, cli, &len);
80         printf("child %d, datagram from %s",
81             getpid(), Sock_ntop(cli, len));
82         printf(", to %s\n", Sock_ntop(myaddr, salen));

83         Sendto(sockfd, mesg, n, 0, cli, len),
84     }
85 }

```

Глава 24

24.1. Да, разница есть. В первом примере два байта отсылаются с единственным срочным указателем, который указывает на байт, следующий за *b*. Во втором же примере (вызываются две функции) сначала отсылается символ *a* с указателем срочности, который указывает на следующий за ним байт, а за этим сегментом следует еще один TCP-сегмент, содержащий символ *b* с другим указателем срочности, указывающим на следующий за ним байт.

24.2. В листинге Д.10 приведена версия программы с использованием функции *poll*.

Листинг Д.10. Версия программы из листинга 24.4, использующая функцию poll вместо функции select

```
//oob/tcprecv03p.c
1 #include "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int listenfd, connfd, n, justreadoob = 0;
6     char buff[100];
7     struct pollfd pollfd[1];

8     if (argc == 2)
9         listenfd = Tcp_listen(NULL, argv[1], NULL);
10    else if (argc == 3)
11        listenfd = Tcp_listen(argv[1], argv[2], NULL);
12    else
13        err_quit("usage: tcprecv03p [ <host> ] <port#>");

14    connfd = Accept(listenfd, NULL, NULL);

15    pollfd[0].fd = connfd;
16    pollfd[0].events = POLLRDNORM;
17    for (;;) {
18        if (justreadoob == 0)
19            pollfd[0].events |= POLLRDBAND;

20        Poll(pollfd, 1, INFTIM);

21        if (pollfd[0].revents & POLLRDBAND) {
22            n = Recv(connfd, buff, sizeof(buff) - 1, MSG_OOB);
23            buff[n] = 0; /* завершающий нуль */
24            printf("read %d OOB byte: %s\n", n, buff);
25            justreadoob = 1;
26            pollfd[0].events &= ~POLLRDBAND; /* отключение бита */
27        }
28        if (pollfd[0].revents & POLLRDNORM) {
29            if ((n = Read(connfd, buff, sizeof(buff) - 1)) == 0) {
30                printf("received EOF\n");
31                exit(0);
32            }
33            buff[n] = 0; /* завершающий нуль */
34            printf("read %d bytes %s\n", n, buff);
35            justreadoob = 0;
36        }
37    }
38 }
```

Глава 25

25.1. Нет, такая модификация приведет к ошибке. Проблема состоит в том, что queue уменьшается до того, как завершается обработка элемента массива dg[iget], что позволяет обработчику сигналов считывать новую дейтаграмму в данный элемент массива.

Глава 26

26.1. В примере с функцией `fork` будет использоваться 101 дескриптор, один прослушиваемый сокет и 100 присоединенных сокетов. Но каждый из 101 процесса (один родительский и 100 дочерних) имеет только один открытый дескриптор (игнорируем все остальные, такие как стандартный поток ввода, если сервер не является демоном). В случае сервера с потоками используется 101 дескриптор для одного процесса. Каждым потоком (включая основной) обрабатывается один дескриптор.

26.2. Обмена двумя последними сегментами завершения TCP-соединения (сегмент FIN сервера и сегмент ACK клиента в ответ на сегмент FIN сервера) не произойдет. Это переведет клиентский конец соединения в состояние `FIN_WAIT_2` (см. рис. 2.4). Беркли-реализации прервут работу клиентского конца, если он остался в этом состоянии, по тайм-ауту через 11 минут [128, с. 825–827]. У сервера же в конце концов закончатся дескрипторы.

26.3. Это сообщение будет выводиться основным программным потоком в том случае, когда он считывает из сокета признак конца файла и *при этом* другой поток продолжает работать. Простейший способ выполнить это — объявить другую внешнюю переменную по имени `done`, инициализируемую нулем. Прежде чем функция `copyto` программного потока вернет управление, она установит эту переменную в 1. Основной программный поток проверит эту переменную, и если она равна нулю, выведет сообщение об ошибке. Поскольку значение переменной устанавливает только один программный поток, нет необходимости в синхронизации.

Глава 27

27.1. Ничего не изменится. Все системы являются соседями, поэтому гибкая маршрутизация идентична жесткой.

27.2. Мы бы поместили EOL (нулевой байт) в конец буфера.

27.3. Поскольку программа `ping` создает символьный (неструктурированный) сокет (см. главу 28), она получает полный IP-заголовок, включая все IP-параметры, для каждой дейтаграммы, которую она считывает с помощью функции `recvfrom`.

27.4. Потому что сервер `rlogind` запускается демоном `inetd` (см. раздел 13.5).

27.5. Проблема заключается в том, что пятый аргумент функции `setsockopt` является указателем на длину, а не самой длиной. Эта ошибка, вероятно, была выявлена, когда впервые использовались прототипы ANSI C.

Ошибка оказалась безвредной, поскольку, как отмечалось, для отключения параметра сокета `IP_OPTIONS` можно либо задать пустой указатель в качестве четвертого аргумента, либо установить нулевое значение в пятом аргументе (длине) [128, с. 269].

Глава 28

28.1. Недоступными являются поле номера версии и поле следующего заголовка в IPv6. Поле полезной длины доступно либо как аргумент одной из функций вывода, либо как возвращаемое значение одной из функций ввода, но если требуется параметр увеличенного поля данных (*jumbo payload option*), сам параметр приложению недоступен. Заголовок фрагментации также недоступен приложению.

28.2. В конце концов приемный буфер клиентского сокета заполнится, и при этом функция демона `write` будет заблокирована. Мы не хотим, чтобы это произошло, поскольку демон тогда перестанет обрабатывать данные на всех своих сокетах. Простейшим решением является следующее: демон должен сделать свой конец соединения домена Unix с клиентом неблокируемым. Для этого демон должен вызывать функцию `write` вместо функции-обертки `Write` и игнорировать ошибку `EWOULDBLOCK`.

28.3. По умолчанию Беркли-ядра допускают широковещательную передачу через символьный сокет [128, с. 1057]. Поэтому параметр сокета `SO_BROADCAST` необходимо определять только для UDP-сокетов.

28.4. Наша программа не проверяет адреса многоадресной передачи и не устанавливает параметр сокета `IP_MULTICAST_IF`. Следовательно, ядро выбирает исходящий интерфейс, вероятно, просматривая таблицу маршрутизации для 224.0.0.1. Мы также не устанавливаем значение поля `IP_MULTICAST_TTL`, поэтому по умолчанию оно равно 1, и это правильное значение.

Глава 29

29.1. Этот флаг означает, что буфер перехода устанавливается функцией `sigsetjmp` (см. листинг 29.6). Хотя этот флаг может казаться лишним, существует вероятность, что сигнал может быть доставлен после того, как устанавливается обработчик ошибок, но перед тем как вызывается функция `sigsetjmp`. Даже если программа не вызывает генерацию сигнала, сигнал всё равно может быть сгенерирован другим путем (например, как в случае с командой `kill`).

Глава 30

30.1. Родительский процесс оставляет прослушиваемый сокет открытым в том случае, если ему позже будет необходимо создать дополнительный дочерний процесс с помощью функции `fork` (это будет расширением нашего кода).

30.2. Для передачи дескриптора действительно можно вместо потокового сокета использовать сокет дейтаграмм. В случае сокета дейтаграмм родительский процесс не получает признака конца файла на своем конце канала, когда дочерний процесс прерывается преждевременно, но для этих целей родительский процесс может использовать сигнал `SIGCHLD`. Следует иметь в виду, что эта ситуация отличается от случая с применением нашего демона `icmprd` (см. раздел 28.7): тогда между клиентом и сервером не было иерархических отношений (родительский процесс — дочерний процесс), поэтому использование признака конца файла было единственным способом для сервера обнаружить исчезновение клиента.

Глава 31

31.1. Здесь предполагается, что по умолчанию для протокола осуществляется нормальное завершение при закрытии потока, и для TCP это правильно.

Литература

Все документы RFC находятся в свободном доступе и могут быть получены по электронной почте, через анонимные FTP-серверы или WWW. Стартовая точка для поиска — <http://www.ietf.org>. Документы RFC расположены по адресу <ftp://ftp.rfc-editor.org/in-notes>. Отдельные документы RFC не снабжены адресами URL.

Пункты, помеченные как «интернет-проект», — это еще не законченные разработки IETF (Internet Engineering Task Force — целевая группа инженерной поддержки Интернета). После выхода этой книги в свет эти проекты, возможно, изменятся или будут опубликованы как RFC. Они находятся в свободном доступе, как и документы RFC. Основное хранилище интернет-проектов — <http://www.ietf.org>. Часть URL, содержащая имя файла, приведена рядом с названием каждого проекта, так как в ней содержится номер версии.

Для книг, статей и других источников, имеющих электронные версии, указаны адреса сайтов. Они могут меняться, поэтому следите за списком обновлений на сайте этой книги <http://www.unpbook.com>.

1. Albitz, P. and Liu, C. 2001. *DNS and Bind, Fourth Edition*. O'Reilly & Associates, Sebastopol, CA.
2. Allman, M., Floyd, S., and Partridge, C. 2002. "Increasing TCP's Initial Window," RFC 3390.
3. Allman, M., Ostermann, S., and Metz, C. W. 1998. "FTP Extensions for IPv6 and NATs," RFC 2428.
4. Allman, M., Paxson, V., and Stevens, W. R. 1999. "TCP Congestion Control," RFC 2581.
5. Almquist, P. 1992. "Type of Service in the Internet Protocol Suite," RFC 1349 (obsoleted by RFC 2474). Обсуждается использование поля *type* сервиса в заголовке IPv4.
6. Baker, F. 1995. "Requirements for IP Version 4 Routers," RFC 1812.
7. Borman, D. A. 1997a. "Re: Frequency of RST Terminated Connections," end2end-interest mailing list (<http://www.unpbook.com/borman.97jan30.txt>).
8. Borman, D. A. 1997b. "Re: SYN/RST cookies," tcp-impl mailing list (<http://www.unpbook.com/borman.97jun06.txt>).
9. Borman, D. A., Deering, S. E., and Hinden, R. 1999. "IPv6 Jumbograms," RFC 2675.
10. Braden, R. T. 1989. "Requirements for Internet Hosts —Communication Layers," RFC 1122. Первая часть Host Requirements RFC: канальный уровень, IPv4, ICMPv4, IGMPv4, ARP, TCP и UDP.
11. Braden, R. T. 1992. "TIME-WAIT Assassination Hazards in TCP," RFC 1337.
12. Braden, R. T., Borman, D. A., and Partridge, C. 1988. "Computing the Internet checksum," RFC 1071.
13. Bradner, S. 1996. "The Internet Standards Process - Revision 3," RFC 2026.
14. Bush, R. 2001. "Delegation of IP6.ARPA," RFC 3152.
15. Butenhof, D. R. 1997. *Programming with POSIX Threads*. Addison-Wesley, Reading, MA.
16. Cain, B., Deering, S. E., Kouvelas, I., Fenner, B., and Thyagarajan, A. 2002. "Internet Group Management Protocol, Version 3," RFC 3376.
17. Carpenter, B. and Moore, K. 2001. "Connection of IPv6 Domains via IPv4 Clouds," RFC 3056.
18. CERT, 1996a. "UDP Port Denial-of-Service Attack," Advisory CA-96.01, Computer Emergency Response Team, Pittsburgh, PA.
19. CERT, 1996b. "TCP SYN Flooding and IP Spoofing Attacks," Advisory CA-96.21, Computer Emergency Response Team, Pittsburgh, PA.
20. Cheswick, W. R., Bellovin, S. M., and Rubin, A. D. 2003. Firewalls and Internet Security: Repelling the Wily Hacker, Second Edition. Addison-Wesley, Reading, MA.
21. Conta, A. and Deering, S. E. 1998. "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," RFC 2463.
22. Conta, A. and Deering, S. E. 2001. "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," draft-ietf-ipngwg-icmp-v3-02.txt (Internet Draft).
23. Crawford, M. 1998a. "Transmission of IPv6 Packets over Ethernet Networks," RFC 2464.
24. Crawford, M. 1998b. "Transmission of IPv6 Packets over FDDI Networks," RFC 2467.
25. Crawford, M., Narten, T., and Thomas, S. 1998. "Transmission of IPv6 Packets over Token Ring Networks," RFC 2470.
26. Deering, S. E. 1989. "Host extensions for IP multicasting," RFC 1112.
27. Deering, S. E. and Hinden, R. 1998. "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460.
28. Draves, R. 2003. "Default Address Selection for Internet Protocol version 6 (IPv6)," RFC 3484.

29. Eriksson, H. 1994. "MBONE: The Multicast Backbone," *Communications of the ACM*, vol. 37, no. 8, pp. 54–60.
30. Fink, R. and Hinden, R. 2003. "6bone (IPv6 Testing Address Allocation) Phase-out," draft-fink-6bone-phaseout-04.txt (Internet Draft).
31. Fuller, V., Li, T., Yu, J. Y., and Varadhan, K. 1993. "Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy," RFC 1519.
32. Garfinkel, S. L., Schwartz, A., and Spafford, E. H. 2003. *Practical UNIX & Internet Security, 3rd Edition*. O'Reilly & Associates, Sebastopol, CA.
33. Gettys, J. and Nielsen, H. F. 1998. *SMUX Protocol Specification* (<http://www.w3.org/TR/WD?mux>).
34. Gierth, A. 1996. *Private communication*.
35. Gilligan, R. E. and Nordmark, E. 2000. "Transition Mechanisms for IPv6 Hosts and Routers," RFC 2893.
36. Gilligan, R. E., Thomson, S., Bound, J., McCann, J., and Stevens, W. R. 2003. "Basic Socket Interface Extensions for IPv6," RFC 3493.
37. Gilligan, R. E., Thomson, S., Bound, J., and Stevens, W. R. 1997. "Basic Socket Interface Extensions for IPv6," RFC 2133 (устарело после выхода RFC 2553).
38. Gilligan, R. E., Thomson, S., Bound, J., and Stevens, W. R. 1999. "Basic Socket Interface Extensions for IPv6," RFC 2553 (устарело после выхода RFC 3493).
39. Haberman, B. 2002. "Allocation Guidelines for IPv6 Multicast Addresses," RFC 3307.
40. Haberman, B. and Thaler, D. 2002. "Unicast-Prefix-based IPv6 Multicast Addresses," RFC 3306.
41. Handley, M. and Jacobson, V. 1998. "SDP: Session Description Protocol," RFC 2327.
42. Handley, M., Perkins, C., and Whelan, E. 2000. "Session Announcement Protocol," RFC 2974.
43. Harkins, D. and Carrel, D. 1998. "The Internet Key Exchange (IKE)," RFC 2409.
44. Hinden, R. and Deering, S. E. 2003. "Internet Protocol Version 6 (IPv6) Addressing Architecture," RFC 3513.
45. Hinden, R., Deering, S. E., and Nordmark, E. 2003. "IPv6 Global Unicast Address Format," RFC 3587.
46. Hinden, R., Fink, R., and Postel, J. B. 1998. "IPv6 Testing Address Allocation," RFC 2471.
47. Holbrook, H. and Cheriton, D. 1999. "IP multicast channels: EXPRESS support for large-scale single-source applications," *Computer Communication Review*, vol. 29, no. 4, pp. 65–78.
48. Huitema, C. 2001. "An Anycast Prefix for 6to4 Relay Routers," RFC 3068.
49. IANA, 2003. *Protocol/Number Assignments Directory* (<http://www.iana.org/numbers.htm>).
50. IEEE, 1996. "Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]," IEEE Std 1003.1, 1996 Edition, Institute of Electrical and Electronics Engineers, Piscataway, NJ.
- Данная версия POSIX.1 (называемая также ISO/IEC 9945-1:1996) содержит базовый интерфейс API (1990), расширения реального времени 1003.1b (1993), программные потоки Pthreads 1003.1c (1995) и технические поправки 1003.1i (1995). Чтобы сделать заказ, обратитесь на сайт <http://www.ieee.org>. К сожалению, стандарты IEEE не распространяются свободно через Интернет.
51. IEEE, 1997. *Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority*. Institute of Electrical and Electronics Engineers, Piscataway, NJ (<http://standards.ieee.org/regauth/oui/tutorials/EUI64.html>).
52. Jacobson, V. 1988. "Congestion Avoidance and Control," *Computer Communication Review*, vol. 18, no. 4, pp. 314–329 (<ftp://ftp.ee.lbl.gov/papers/congav0id.ps.z>).
- Классическая статья, описывающая алгоритмы медленного старта и предотвращения перегрузки сети для TCP.
53. Jacobson, V., Braden, R. T., and Borman, D. A. 1992. "TCP Extensions for High Performance," RFC 1323.
- Описывается параметр масштабирования окна, параметр отметки времени, алгоритм PAWS, а также приводятся причины необходимости этих модификаций.
54. Jacobson, V., Braden, R. T., and Zhang, L. 1990. "TCP Extension for High-Speed Paths," RFC 1185 (устарело после выхода RFC 1323).
55. Josey, A., ed. 1997. *Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification*. Prentice Hall, Uppser Saddle River, NJ.
56. Josey, A., ed. 2002. *The Single UNIX Specification —The Authorized Guide to Version 3*. The Open Group, Berkshire, UK.
57. Joy, W. N. 1994. *Private communication*.
58. Karn, P. and Partridge, C. 1991. "Improving Round-Trip Time Estimates in Reliable Transport Protocols," *ACM Transactions on Computer Systems*, vol. 9, no. 4, pp. 364–373.

59. Katz, D. 1993. "Transmission of IP and ARP over FDDI Networks," RFC 1390.
60. Katz, D. 1997. "IP Router Alert Option," RFC 2113.
61. Kent, S.T. 1991. "U.S. Department of Defense Security Options for the Internet Protocol," RFC 1108.
62. Kent, S. T. 2003a. "IP Authentication Header," draft-ietf-ipsec-rfc2402bis-04.txt (Internet Draft).
63. Kent, S. T. 2003b. "IP Encapsulating Security Payload (ESP)," draft-ietf-ipsec-esp-v3-06.txt (Internet Draft).
64. Kent, S. T. and Atkinson, R.J. 1998a. "Security Architecture for the Internet Protocol," RFC 2401.
65. Kent, S.T. and Atkinson, R.J. 1998b. "IP Authentication Header," RFC 2402.
66. Kent, S. T. and Atkinson, R. J. 1998c. "IP Encapsulating Security Payload (ESP)," RFC 2406.
67. Kernighan, B. W. and Pike, R. 1984. *The UNIX Programming Environment* Prentice Hall, Englewood Cliffs, NJ.
68. Kernighan, B. W. and Ritchie, D. M. 1988. *The C Programming Language, Second Edition*. Prentice Hall, Englewood Cliffs, NJ.
69. Lanciani, D. 1996. "Re: sockets: AF_INET vs. PF_INET," Message-ID: <3561@news.IPSWITCH.COM>, USENET comp.protocols.tcp-ip Newsgroup (<http://www.unpbook.com/lanciani.96apr10.txt>).
70. Maslen, T. M. 1997. "Re: gethostbyXXXX() and Threads," Message-ID: <maslen.862463630@shellx>, USENET comp.programming.threads Newsgroup (<http://www.unpbook.com/maslen.97may01.txt>).
71. McCann, J., Deering, S.E., and Mogul, J.C. 1996. "Path MTU Discovery for IP version 6," RFC 1981.
72. McCanne, S. and Jacobson, V. 1993. "The BSD Packet Filter: A New Architecture for User-Level Packet Capture," *Proceedings of the 1993 Winter USENIX Conference*, San Diego, CA, pp. 259–269.
73. McDonald, D. L., Metz, C.W., and Phan, B.G. 1998. "PF_KEY Key Management API, Version 2," RFC 2367.
74. McKusick, M.K., Bostic, K., Karels, M.J., and Quarterman, J.S. 1996. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, Reading, MA.
75. Meyer, D. 1998. "Administratively Scoped IP Multicast," RFC 2365.
76. Mills, D. L. 1992. "Network Time Protocol (Version 3) Specification, Implementation," RFC 1305.
77. Mills, D. L. 1996. "Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI," RFC 2030.
78. Mogul, J.C. and Deering, S.E. 1990. "Path MTU discovery," RFC 1191.
79. Mogul, J.C. and Postel, J.B. 1985. "Internet Standard Subnetting Procedure," RFC 950.
80. Narten, T. and Draves, R. 2001. "Privacy Extensions for Stateless Address Auto-configuration in IPv6," RFC 3041.
81. Nemeth, E. 1997. *Private communication*.
82. Nichols, K., Blake, S., Baker, F., and Black, D. 1998. "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," RFC 2474.
83. Nordmark, E. 2000. "Stateless IP/ICMP Translation Algorithm (SIIT)," RFC 2765.
84. Ong, L., Rytina, I., Garcia, M., Schwarzbauer, H., Coene, L., Lin, H., Juhasz, I., Holdrege, M., and Sharp, C. 1999. "Framework Architecture for Signaling Transport," RFC 2719.
85. Ong, L. and Yoakum, J. 2002. "An Introduction to the Stream Control Transmission Protocol (SCTP)," RFC 3286.
86. The Open Group, 1997. *CAE Specification, Networking Services (XNS), Issue 5*. The Open Group, Berkshire, UK.
- Спецификация сокетов и XTI для Unix 98. Это руководство также содержит приложения, в которых описано использование XTI с NetBIOS, протоколов OSI, SNA, а также Netware IPX и SPX. Эти приложения охватывают использование сокетов и XTI с ATM.
87. Partridge, C. and Jackson, A. 1999. "IPv6 Router Alert Option," RFC 2711.
88. Partridge, C., Mendez, T., and Milliken, W. 1993. "Host Anycasting Service," RFC 1546.
89. Partridge, C. and Pink, S. 1993. "A Faster UDP," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 429–440.
90. Paxson, V. 1996. "End-to-End Routing Behavior in the Internet," *Computer Communication Review*, vol. 26, no. 4, pp. 25–38 (<ftp://ftp.ee.1b1.gov/papers/routing.SIGCOMM.ps.Z>).
91. Paxson, V. and Allman, M. 2000. "Computing TCP's Retransmission Timer," RFC 2988.
92. Plauger, P.J. 1992. *The Standard C Library*. Prentice Hall, Englewood Cliffs, NJ.
93. Postel, J.B. 1980. "User Datagram Protocol," RFC 768.
94. Postel, J.B. 1981a. "Internet Protocol," RFC 791.
95. Postel, J.B. 1981b. "Internet Control Message Protocol," RFC 792.

96. Postel, J.B. 1981c. "Transmission Control Protocol," RFC 793.
97. Pusateri, T. 1993. "IP Multicast over Token-Ring Local Area Networks," RFC 1469.
98. Rago, S.A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, MA.
99. Rajahalme, J., Conta, A., Carpenter, B., and Deering, S.E. 2003. "IPv6 Flow Label Specification," draft-ietf-ipv6-flow-label-07.txt (Internet Draft).
100. Ramakrishnan, K., Floyd, S., and Black, D. 2001. "The Addition of Explicit Congestion Notification (ECN) to IP," RFC 3168.
101. Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G.J., and Lear, E. 1996. "Address Allocation for Private Internets," RFC 1918.
102. Reynolds, J.K. 2002. "Assigned Numbers: RFC 1700 is Replaced by an On-line Database," RFC 3232.
103. Reynolds, J.K. and Postel, J.B. 1994. "Assigned Numbers," RFC 1700 (устарело после выхода RFC 3232).
104. Ritchie, D.M. 1984. "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1897–1910.
105. Salus, P.H. 1994. *A Quarter Century of Unix*. Addison-Wesley, Reading, MA.
106. Salus, P.H. 1995. *Casting the Net: From ARPANET to Internet and Beyond*. Addison-Wesley, Reading, MA.
107. Schimmel, C. 1994. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley, Reading, MA.
108. Spero, S. 1996. *Session Control Protocol (SCP)* (<http://www.w3.org/Protocols/HTTP-NG/http-ng-scp.html>).
109. Srinivasan, R. 1995. "XDR: External Data Representation Standard," RFC 1832.
110. Stevens, W.R. 1992. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA. Программирование в Unix — детальное описание.
111. Stevens, W.R. 1994. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, MA. Введение в протоколы Интернета.
112. Stevens, W.R. 1996. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*. Addison-Wesley, Reading, MA.
113. Stevens, W.R. and Thomas, M. 1998. "Advanced Sockets API for IPv6," RFC 2292 (устарело после выхода RFC 3542).
114. Stevens, W.R., Thomas, M., Nordmark, E., and Jinmei, T. 2003. "Advanced Sockets Application Program Interface (API) for IPv6," RFC 3542.
115. Stewart, R. R., Bestler, C., Jim, J., Ganguly, S., Shah, H., and Kashyap, V. 2003a. "Stream Control Transmission Protocol (SCTP) Remote Direct Memory Access (RDMA) Direct Data Placement (DDP) Adaptation," draft-stewart-rddp-sctp-02.txt (Internet Draft).
116. Stewart, R.R., Ramalho, M., Xie, Q., Tuexen, M., Rytina, I., Belinchon, M., and Conrad, P. 2003b. "Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration," draft-ietf-tsvwg-addip-sctp-07.txt (Internet Draft).
117. Stewart, R.R. and Xie, Q. 2001. *Stream Control Transmission Protocol (SCTP): A Reference Guide*. Addison-Wesley, Reading, MA.
118. Stewart, R.R., Xie, Q., Morneau, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., and Paxson, V. 2000. "Stream Control Transmission Protocol," RFC 2960.
119. Stone, J., Stewart, R.R., and Otis, D. 2002. "Stream Control Transmission Protocol (SCTP) Checksum Change," RFC 3309.
120. Tanenbaum, A. S. 1987. *Operating Systems Design and Implementation*. Prentice Hall, Englewood Cliffs, NJ.
121. Thomson, S. and Huitema, C. 1995. "DNS Extensions to support IP version 6," RFC 1886.
122. Torek, C. 1994. "Re: Delay in re-using TCP/IP port," Message-ID: <199501010028.QAA16863@elf.bsd.com>, USENET comp.unix.wizards Newsgroup (<http://www.unpbook.com/torek.94dec31.txt>).
123. Touch, J. 1997. "TCP Control Block Interdependence," RFC 2140.
124. Unix International, 1991. *Data Link Provider Interface Specification*. Unix International, Parsippany, NJ, Revision 2.0.0 (<http://www.unpbook.com/dlpi.2.0.0.ps>). Более новая версия этой спецификации доступна по адресу <http://www.rdg.opengroup.org/pubs/catalog/web.htm>.
125. Unix International, 1992a. *Network Provider Interface Specification*. Unix International, Parsippany, NJ, Revision 2.0.0 (<http://www.unpbook.eom/npi.2.0.0.ps>).

126. Unix International, 1992b. *Transport Provider Interface Specification*. Unix International, Parsippany, NJ, Revision 1.5 (<http://www.unpbook.eom/tpi.1.5.ps>). Более новая версия этой спецификации доступна по адресу <http://www.rdg.opengroup.org/pubs/catalog/web.htm>.

127. Vixie, P. A. 1996. *Private communication*.

128. Wright, G.R. and Stevens, W.R. 1995. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, Reading, MA.

Реализация протоколов Интернета в операционной системе 4.4BSD-Lite.

notes

Примечания

1

Все исходные коды программ, опубликованные в этой книге, вы можете найти по адресу <http://www.piter.com>.

Стивенс У. UNIX: взаимодействие процессов. — СПб.: Питер, 2002.

Имеется в виду адрес, записанный с помощью символов подстановки. — *Примеч. перев.*

4

Иногда переводится как «экстренный режим» или «режим срочности». — *Прим. перев.*

Также (ошибочно) используется термин «указатель срочности». — *Примеч. перев.*

6

Также используется термин «срочное смещение». — *Примеч. перев.*

Используются также термины «расширенные заголовки» и «дополнительные заголовки». — *Примеч.*
перев.