Сокеты

Сокет – универсальный интерфейс для создания каналов для межпроцессного взаимодействия.

Сокеты объединили в едином интерфейсе потоковую передачу данных подобную каналам *pipe* и *FIFO* и передачу сообщений, подобную очередям сообщений в *System V IPC*. Кроме того, сокеты добавили возможность создания клиент-серверного взаимодействия (один со многими).

Интерфейс сокетов скрывает механизм передачи данных между процессами. В качестве нижележащего транспорта могут использоваться как внутренний транспорт в ядре Unix, так и практически любые сетевые протоколы. Для достижения такой гибкости используется перегруженная функция назначения сокету имени – *bind()*. Данная функция принимает в качестве параметров идентификатор пространства имён и указатель на структуру, которая содержит имя в соответствующем формате. Это могут быть имена в файловой системе Unix, IP адрес + порт в TCP/UDP, MAC-адрес сетевой карты в протоколе IPX.

Классификация сокетов

Stream

Поток байтов без разделения на записи, подобный чтению-записи в файл или каналам в *Unix*. Процесс, читающий из сокета, не знает, какими порциями производилась запись в сокет пишущим процессом. Данные никогда не теряются и не перемешиваются.

- Непрерывный поток байтов
- Упорядоченный приём данных
- Надёжная доставка данных

Datagram

Передача записей ограниченной длины. Записи на уровне интерфейса сокетов никак не связанны между собой. Отправка записей описывается фразой: "отправил и забыл". Принимающий процесс получает записи по отдельности в непредсказуемом порядке или не получает вовсе.

- Деление потока данных на отдельные записи
- Неупорядоченный приём записей
- Возможна потеря записей

Sequential packets

Надёжная упорядоченная передача с делением на записи. Использовался в *Sequence Packet Protocol* для *Xerox Network Systems*. Не реализован в *TCP/IP*, но может быть имитирован в *TCP* через Urgent Pointer.

- Деление потока данных на отдельные записи
- Упорядоченная передача данных
- Надёжная доставка данных

Raw

Данный тип сокетов предназначен для управление нижележащим сетевым драйвером. В *Unix* требует администраторских полномочий. Примером использования *Raw*-сокета является программа `ping`, которая отправляет и принимает управляющие пакеты управления сетью – *ICMP*. Файл */usr/bin/ping* в старых версиях *Linux* имел флаг смены полномочий *suid*, а в новых версиях – флаги дополнительных полномочий – *cap_net_admin* и *cap_net_raw*.

Имена сокетов

Имена сокетов на сервере назначаются вызовом *bind()*, а на клиенте, как правило, генерируются ядром.

- Inet – сокеты именуются с помощью IP адресов и номеров портов
- Unix – сокетам даются имена объектов типа *socket* в файловой системе
- IPX – имена на основе MAC-адресов сетевых карт
- ... – возможны и другие варианты

TCP/IP

Для передачи данных с помощью семействе протоколов *TCP/IP* реализованы два вида сокетов *Stream* и *Datagram*. Все остальные манипуляции с сетью *TCP/IP* осуществляются через *Raw*-сокеты.

- TCP = Stream
- UDP = Datagram
- ICMP = RAW
- Sequential packets – были экспериментальные реализации в 1990-х, которые не вышли за рамки научных исследований

API Сокетов

Создание сокета

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

int s = socket(int domain, int type, int protocol);
```

domain – семейство протоколов, которое будет использоваться для передачи данных. Имена макросов, задающих домен, начинаются с *PF – protocol family/*

- PF_UNIX – внутреннее межпроцессное взаимодействие
- PF_INET – стек TCP/IP

type – тип сокета

- SOCK_DGRAM – ненадежная передача данных с сохранением границ сообщений (соответствует протоколу UDP),
- SOCK_STREAM – надежная передача данных без сохранения границ сообщений (соответствует протоколу TCP),
- SOCK_SEQ – надежная передача данных с сохранением границ сообщений (в стеке TCP/IP не поддерживается),
- SOCK_RAW – низкоуровневый доступ к протоколу (уровень IP, ICMP).

protocol Поскольку в семействе протоколов *TCP/IP* протокол однозначно связан с типом сокета, а в домене *Unix* понятие протокола вообще отсутствует, то этот параметр всегда равен нулю, что соответствует автовыбору.

В домене *Unix* возможно создание пары соединённых между собой безымянных сокетов, которые буду вести себя подобно неименованному каналу *pipe*. В отличие от неименованных каналов, оба сокета открыты и на чтение и на запись.

```
int result;
int sockfd[2];
result=socketpair(AF_UNIX, SOCK_STREAM, 0, sockfd);
```

Назначение имени

Для того, чтобы клиенты могли подключаться к серверу, сервер должен иметь заранее известное имя. Вызов *bind()* обеспечивает назначение имени серверному сокету. Сервер получит имя клиентского сокета в момент соединения (stream) или получения сообщения

(datagram), поэтому на клиентской стороне имя сокету, как правило, назначается ядром ОС, хотя и явное присвоение с помощью *bind()* остаётся доступным.

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

int bind(int sockfd, struct sockaddr *localaddr, int addrlen);
```

Второй параметр функции *bind()* – адрес – формально описан как указатель на структуру *sockaddr* с удобным размером 16 байт. *sockaddr* можно рассматривать как суперкласс (без методов) от которого наследуются реально используемые классы *sockaddr_un*, *sockaddr_in* и т.д. Все они наследуют поле *sa_family* – тип адреса, благодаря которому *bind()* корректно интерпретирует переданную ему структуру данных. Для того, чтобы избежать предупреждений компилятора, рекомендуется явно преобразовывать тип второго параметра к *struct sockaddr **.

Макросы, которые присваиваются полю *sa_family* по своему числовому значению совпадают с соответствующими макросами определяющими семейство протоколов, но начинаются с *AF – address family*.

```
struct sockaddr {
  u_short   sa_family;
  char      sa_data[14];
};
```

Имя в домене *Unix* – строка с именем сокета в файловой системе.

```
struct sockaddr_un {
  short sun_family; /* AF_UNIX */
  char  sun_path[108];
};
```

Имя в домене *Internet* – *IP*-адрес и номер порта, которые хранятся в виде целых числе в формате *BIG ENDIAN*. Для заполнения структуры они должны быть преобразованы из локального представления в сетевое функциями *htonl()* и *htons()* для длинных и коротких целых соответственно. Упаковка *IP*-адреса в дополнительную структуру связана, скорее всего, с какими-то историческими причинами.

```
struct sockaddr_in {
  short         sin_family; /* AF_INET */
  u_short       sin_port;       /* Port Number */
  struct in_addr    sin_addr;/* Internet address */
  char          sin_zero[8];    /*Not used*/
}
```

```
struct in_addr {
  unsigned long int s_addr;
}
```

Соединение с сервером (в основном *Stream*)

Для сокета типа *Stream* вызов *connect()* соединяет сокет клиента с сокетом сервера, создавая поток передачи данных. Адрес сервера *servaddr* заполняется по тем же правилам, что и адрес, передаваемый в *bind()*.

Для сокета типа *Datagram* вызов *connect()* запоминает адрес получателя, для отправки сообщений вызовом *send()*. Можно пропустить этот вызов и отправлять сообщения вызовом *sendto()*, явно указывая адрес получателя для каждого сообщения.

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

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

Прослушивание сокетов сервером (только *Stream*)

Вызов *listen()* на стороне сервера превращает сокет в фабрику сокетов, которая будет с помощью вызова *accept()* возвращать новый транспортный сокет на каждый вызов *connect()* со стороны клиентов.

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

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

backlog – количество запросов клиентов *connect()*, которые будут храниться в очереди ожидания, пока сервер не вызовет *accept()*.

Обработка запроса клиента.

Клиентский *connect()* будет заблокирован до тех пор, пока сервер не вызовет *accept()*. *accept()* возвращает транспортный сокет, который связан с сокетом для которого клиент вызвал *connect()*. Этот сокет используется как файловый дескриптор для вызовов *read()*, *write()*, *send()* и *recv()*.

В переменную *clntaddr* заносится адрес подключившегося клиента.

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

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

Чтение/запись данных

Для операций чтения-записи данных через сокеты могут применяться стандартные вызовы *read()* и *write()*, однако существуют и более специализированные вызовы:

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

ssize_t write(int fildes, const void *buf, size_t nbyte);
ssize_t send(int sockfd, const char *msg, int len, int flags);
ssize_t sendto(int sockfd, const char *msg, int len, int flags,const struct sockaddr *toaddr, int tolen) ;
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);

ssize_t read(int fildes, const void *buf, size_t nbyte);
ssize_t recv(int sockfd, char *buf, int len, int flags);
ssize_t recvfrom(int sockfd, char *buf, int len, int flags, struct sockaddr *fromaddr, int *fromlen);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

Все вызовы применимы и к потоковым сокетам и к сокетам датаграмм. При попытке прочитать датаграмму в слишком маленький буфер, её хвост будет утерян.

write(fd,buf,size) == send(fd,buf,size,0) == sendto(fd,buf,size,0,NULL,0)

send() может применяться только к тем сокетам, для которых выполнен *connect()*.

При использовании *sendto()* с потоковым сокетом адрес *toaddr* игнорируется если был выполнен *connect()*. Если же *connect()* не был выполнен – в *errno* возвращается ошибка *ENOTCONN*.

sendmsg() и *recvmsg()* близки к вызовам *writev()* и *readv()*, поскольку позволяют одним вызовом отправить/принять несколько буферов данных.

Флаги *send()*:

- *MSG_DONTWAIT* – неблокирующая отправка. В случае невозможности отправить порцию данных возвращается -1, а переменная *errno* выставляется в *EAGAIN*.
- *MSG_OOB* – отправка внеочередных данных (*out-of-band*) если они поддерживаются протоколом

Флаги *recv()*:

- *MSG_DONTWAIT* – неблокирующее чтение
- *MSG_OOB* – приём внеочередных данных
- *MSG_PEEK* – "подглядывание" – чтение данных без удаления их из канала

Управление окончанием соединения (в основном *Stream*)

Вызов *close()* закрывает сокет и освобождает все связанные с ним структуры данных.

Для контроля над закрытием потоковых сокетов используется вызов *shutdown()*, который позволяет управлять окончанием соединения.

```
int shutdown () (int sock, int cntl);
```

Аргумент cntl может принимать следующие значения:

- 0: больше нельзя получать данные из сокета;
- 1: больше нельзя посылать данные в сокет;
- 2: больше нельзя ни посылать, ни принимать данные через этот сокет.

Асинхронные операции – select

Для реализации клиент-серверной архитектуры на основе сокетов необходимо предоставить разработчику сервера инструмент для параллельной работы с несколькими клиентами. Возможные варианты:

- создание нового процесса для каждого клиента. Плохо масштабируется, поскольку требует дополнительных ресурсов на создание и последующее планирование процессов. Нити масштабируются лучше, но в ранних реализациях *Unix* они отсутствовали.
- вызов *callback*ов при поступлении данных от пользователя – в *Unix* не реализовано
- бесконечный цикл с попытками неблокирующего чтения-записи. Занимает процессорное время.
- блокирующая операция, ожидающая появления сокетов, доступных для чтения-записи.

Последний вариант является наиболее часто используемым в *Unix* и реализуется вызовами *select()* и *poll()*.

Вызовы отличаются по формату параметров, но эквивалентны по своему назначению. Они приостанавливают выполнение процесса, до появления данных от клиента, появления возможности отправить данные клиенту, появления ошибки приёма-передачи или до истечения

таймаута. Если точнее, то для операций чтения-записи проверяется, что они не будут заблокированы.

Реализация этих вызовов позволяет использовать их для отслеживания состояния любых файловых дескрипторов, а не только сокетов.

SELECT

Вызов *select()* получает три битовых набора флагов (чтение, запись, ошибка) размером с максимальное доступное число открытых файловых дескрипторов. Флаг в какой-то позиции означает что мы наблюдаем за соответствующим файловым дескриптором.

Параметр *nfds* задает номер максимального выставленного флага и служит для оптимизации.

```
#include <sys/select.h>
int select(int nfds,
   fd_set *readfds,
   fd_set *writefds,
   fd_set *exceptfds,
   struct timeval *timeout);
```

Для манипуляции флагами используется следующие функции, которые позволяют очистить набор флагов, установить флаг, сбросить флаг, проверить состояние флага:

```
void FD_ZERO(fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
```

При изменении состояния каких-либо интересующего нас файловых дескрипторов *select()* сбрасывает все флаги и выставляет те, которые обозначают, какие события и на каких файловых дескрипторах произошли. Возвращается значение, указывающее сколько флагов возвращено. Если событий не было и возврат из *select()* произошёл по таймауту, все наборы флагов обнуляются и возвращается ноль.

В случае ошибки возвращается -1. Значение флагов не определено.

Таймаут задаётся структурой *timeval*, содержащей секунды и микросекунды

```
struct timeval {
    long    tv_sec;         /* seconds */
    long    tv_usec;        /* microseconds */
};
```

Поскольку вызов *sleep()* работает с точностью до секунды, то для приостановки процесса на более короткие промежутки времени часто используют *select()* с указателями *NULL* вместо указателей на флаги.

POLL

Вызов *poll()* функционально эквивалентен *select*. Его параметры как бы "вывернуты наизнанку" по сравнению с *select()*. Вместо трёх наборов битовых файлов в *poll()* массив интересующих файловых дескрипторов размером *nfds*. С каждым файловым дескриптором связаны две переменные: флаги интересующих событий и флаги случившихся событий. Время таймаута задаётся в миллисекундах.

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Структура *pollfd*

struct pollfd { int fd; /* file descriptor */ *short events;* / requested events */ *short revents;* / returned events */ };

Битовые флаги в *events* определяются макросами:

```
#define POLLIN      0x0001    /* Можно считывать данные */
#define POLLPRI     0x0002    /* Есть срочные данные */
#define POLLOUT     0x0004    /* Запись не будет блокирована */
#define POLLERR     0x0008    /* Произошла ошибка */
#define POLLHUP     0x0010    /* Обрыв связи */
#define POLLNVAL    0x0020    /* Неверный запрос: fd не открыт */
```

Диаграмма взаимодействия сокетов datagram

Ниже представлена временная диаграмма соединения клиента и сервера через сокет типа *Datagram*

Сервер		Клиент
Создание сокета socket()		Создание сокета socket()
Присвоение имени bind()		
Начало цикла работы с клиентами		
Прием сообщения с адресом отправителя recvfrom()	<=	Отправка сообщения по адресу sendto()

Сервер		Клиент
Извлечение адреса клиента из ответа recvfrom()		
Отправка сообщения по адресу sendto()	=>	Приём сообщения recv()
		Закрытие сокета close()
Конец цикла работы с клиентами		
Закрытие сокета close()		

Диаграмма взаимодействия сокетов stream

Ниже представлена временная диаграмма соединения клиента и сервера через сокет типа *Stream*

Сервер		Клиент
Создание сокета socket()		Создание сокета socket()
Присвоение имени bind()		
Создание очереди запросов listen()		
Начало цикла работы с клиентами		
Выбор соединения из очереди accept()	<=	Установка соединения connect()
read()	<=	write()
write()	=>	read()
Закрытие транспортного сокета close()		Закрытие сокета close()
Конец цикла работы с клиентами		