Московский Государственный Технический Университет

имени Н.Э.Баумана

Кафедра САПР

Федорук В.Г.

Сетевое программирование в ОС UNIX

Данное учебное пособие посвящено изучению средств программирования сетевых приложений в среде операционной системы UNIX.

Основным средством построения информационно-вычислительных сетей из ЭВМ, работающих под управлением ОС UNIX, является комплекс протоколов TCP/IP. Программное обеспечение, реализующее данные протоколы информационного обмена, как правило, входит в состав базового варианта любой современной версии этой ОС.

Для создания распределенных приложений в среде ОС UNIX наибольшее распространение получили следующие три средства:

1. socket-интерфейс прикладной программы с модулем из состава ОС, реализующим сетевое взаимодействие;
2. интерфейс транспортного уровня (TLI - Transport Level Interface);
3. средства удаленного вызова процедур (RPC - Remote Procedure Call).

Причем socket-интерфейс и TLI обеспечивают возможность прикладным программам взаимодействовать, используя стандартные (или очень похожие на них) средства ввода-вывода ОС UNIX. RPC позволяет одной прикладной программе обращаться к другой (но функционирующей на другом узле сети) фактически так же, как одна функция традиционной программы на языке СИ обращается к другой.

Примечание. *Перечисленные средства программирования универсальны и могут быть использованы для работы с различными протоколами сетевого взаимодействия, однако в данном учебном пособии они рассматриваются применительно только к протоколам TCP/IP.*

Рассматриваемые средства предназначены для создания распределенных приложений, функционирующих, в первую очередь, согласно модели взаимодействия "клиент-сервер". Эта модель подразумевает, что из двух (в простейшем случае) параллельно выполняющихся на разных (а, возможно, на одном и том же) узлах сети программ одна является сервером (поставщиком услуг), способным решать некоторую задачу вычислительного или информационного характера, а другая - клиентом (потребителем услуг), который в ходе решения собственной проблемы сталкивается с необходимостью получения ответа на задачи, решаемые сервером. Программа-сервер пассивна - непосредственно к решению своей задачи она приступает только при поступлении к ней запроса от клиента, остальное же время она находится в состоянии ожидания такого запроса. После решения поставленной задачи сервер возвращает клиенту найденный ответ. В типичной ситуации программа-клиент приостанавливает свою работу после выдачи запроса серверу в ожидании получения ответа.

Одна программа-сервер может обслуживать (последовательно) несколько клиентов, организуя очередь запросов от них. Клиент и сервер могут меняться местами ролями (если того, конечно, требует логика решаемой проблемы) в разные моменты времени. Сервер, обрабатывая запрос какого-либо клиента, в свою очередь может стать клиентом, обращаясь за некоторой услугой к другому серверу.

Двумя наиболее общими режимами взаимодействия прикладных программ в вычислительной сети являются:

1. режим с установлением соединения;
2. режим без установления соединения.

Первый режим подразумевает, что взаимодействие осуществляется в три этапа:

- установление логической связи между прикладными программами (по инициативе клиента);
- двусторонний, последовательный (потоковый, без учета каких-либо границ записей), надежный (без потери и дублирования) обмен данными;
- закрытие связи (экстренное или с доставкой буферизованных на передающей стороне данных).

В режиме без установления соединения обмен информацией ведется отдельными блоками данных, часто называемых дейтаграммами и содержащими в себе помимо собственно данных еще и адрес их получателя (соответственно, клиента или сервера). В этом режиме, как правило, не гарантируется надежность доставки данных (они могут быть потеряны или продублированы), может быть нарушена правильная последовательность дейтаграмм на принимающей стороне, но, очевидно, явно присутствуют границы в передаваемых данных.

Программист имеет возможность выбрать режим взаимодействия, отвечающий специфике создаваемого приложения.

Одним из аспектов сетевого программирования является манипулирование адресами, идентифицирующими узлы в вычислительной сети и прикладные программы на этих узлах. К сожалению, способы адресации в сетях, построенных на базе различных протоколов, также различны. Ниже рассматриваются механизмы адресации, принятые в стеке протоколов TCP/IP.

В сетях на основе TCP/IP для идентификации отдельного узла используется уникальное четырехбайтовое число. Только для удобства пользователей сети этим адресам (числам) в соответствие могут быть поставлены символические имена узлов сети. Информация о таком соответствии хранится в одной или нескольких специальных базах данных. Система программирования ОС UNIX предоставляет библиотеку вспомогательных функций, позволяющих работать с этими база- ми (транслируя, например, имена узлов в адреса и обратно).

Каналом выхода в коммуникационную среду, образуемую вычислительной сетью TCP/IP, для любой прикладной программы является так называемый "порт" - чисто абстрактное ("программное") понятие, не имеющее какого-либо соответствия в аппаратуре ЭВМ. Прикладная программа может использовать для общения с другими программами в сети любое количество портов. Каждый порт должен иметь уникальный номер. Номера портов от 1 до 1024 зарезервированы для "широко известных" приложений. Любой номер,

больший 1024, может быть использован программистом для идентификации порта его приложения, необходимо лишь следить за его уникальностью в рамках отдельного узла сети.

В сетях TCP/IP для организации режима взаимодействия с логическим соединением используется протокол транспортного уровня TCP, а режима без установления соединения - протокол UDP. Причем два порта с одинаковым номером, но открытые для взаимодействия по разным протоколам транспортного уровня, считаются различными.

Таким образом: для идентификации партнера по взаимодействию любая сетевая прикладная программа должна специфицировать:

- адрес узла сети, на котором функционирует партнер;
- используемый для взаимодействия протокол транспортного уровня (TCP или UDP);
- номер порта, открытого партнером на его узле сети.

1. Socket-интерфейс

Данное средство было первоначально разработано для обеспечения прикладным программистам в среде ОС UNIX доступа к транспортному уровню стека протоколов TCP/IP. Позже оно было адаптировано для использования и иных протоколов (например, DECnet), а также реализовано в других операционных системах.

Socket (гнездо, разъем) - абстрактное программное понятие, используемое для обозначения в прикладной программе конечной точки канала связи с коммуникационной средой, образованной вычислительной сетью. При использовании протоколов TCP/IP можно говорить, что socket является средством подключения прикладной программы к порту (см. выше) локального узла сети.

Socket-интерфейс представляет собой просто набор системных вызовов и/или библиотечных функций языка программирования СИ, разделенных на четыре группы:

1. локального управления;
2. установления связи;
3. обмена данными (ввода/вывода);
4. закрытия связи.

Ниже рассматривается подмножество функций socket-интерфейса, достаточное для написания сетевых приложений, реализующих модель "клиент-сервер" в режиме с установлением соединения.

1.1. Функции локального управления

Функции локального управления используются, главным образом, для выполнения подготовительных действий, необходимых для организации взаимодействия двух программ-партнеров. Функции носят такое название, поскольку их выполнение носит локальный для программы характер.

1.1.1. Создание socket'a

Создание socket'a осуществляется следующим системным вызовом

```
#include <sys/socket.h>
int socket (domain, type, protocol)
    int domain;
    int type;
    int protocol;
```

Аргумент *domain* задает используемый для взаимодействия набор протоколов (вид коммуникационной области), для стека протоколов TCP/IP он должен иметь символьное значение AF_INET (определено в sys/socket.h).

Аргумент *type* задает режим взаимодействия:

- SOCK_STREAM - с установлением соединения;
- SOCK_DGRAM - без установления соединения.

Аргумент *protocol*задает конкретный протокол транспортного уровня (из нескольких возможных в стеке протоколов). Если этот аргумент задан равным 0, то будет использован протокол "по умолчанию" (TCP для SOCK_STREAM и UDP для SOCK_DGRAM при использовании комплекта протоколов TCP/IP).

При удачном завершении своей работы данная функция возвращает дескриптор socket'a - целое неотрицательное число, однозначно его идентифицирующее. Дескриптор socket'a аналогичен дескриптору файла ОС UNIX.

При обнаружении ошибки в ходе своей работы функция возвращает число "-1".

1.1.2. Связывание socket'a

Для подключения socket'a к коммуникационной среде, образованной вычислительной сетью, необходимо выполнить системный вызов **bind**, определяющий в принятом для сети формате локальный адрес канала связи со средой. В сетях TCP/IP socket связывается с локальным портом. Системный вызов bind имеет следующий синтаксис:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
int bind (s, addr, addrlen)
    int s;
    struct sockaddr *addr;
    int addrlen;
```

Аргумент *s* задает дескриптор связываемого socket'a.

Аргумент *addr* в общем случае должен указывать на структуру данных, содержащую локальный адрес, приписываемый socket'у. Для сетей TCP/IP такой структурой является sockaddr_in.

Аргумент *addrlen* задает размер (в байтах) структуры данных, указываемой аргументом addr.

Структура sockaddr_in используется несколькими системными вызовами и функциями socket-интерфейса и определена в include-файле in.h следующим образом:

```
struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
    };
```

Поле sin_family определяет используемый формат адреса (набор протоколов), в нашем случае (для TCP/IP) оно должно иметь значение AF_INET.

Поле sin_addr содержит адрес (номер) узла сети.

Поле sin_port содержит номер порта на узле сети.

Поле sin_zero не используется.

Определение структуры in_addr (из того же include-файла) таково:

```
struct in_addr {
    union {
        u_long S_addr;
        /*
        другие (не интересующие нас)
        члены объединения
        */
        } S_un;
#define s_addr S_un.S_addr
    };
```

Структура sockaddr_in должна быть полностью заполнена перед выдачей системного вызова **bind**. При этом, если поле sin_addr.s_addr имеет значение INADDR_ANY, то системный вызов будет привязывать к socket'у номер (адрес) локального узла сети.

В случае успеха bind возвращает 0, в противном случае - "-1".

1.2. Функции установления связи

Для установления связи "клиент-сервер" используются системные вызовы **listen** и **accept** (на стороне сервера), а также **connect** (на стороне клиента). Для заполнения полей структуры socaddr_in, используемой в вызове **connect**, обычно используется библиотечная функция gethostbyname, транслирующая символическое имя узла сети в его номер (адрес).

1.2.1. Ожидание установления связи

Системный вызов **listen** выражает желание выдавшей его программы-сервера ожидать запросы к ней от программ-клиентов и имеет следующий вид:

```
#include <sys/socket.h>
int listen (s, n)
    int s;
    int n;
```

Аргумент *s* задает дескриптор socket'a, через который программа будет ожидать запросы к ней от клиентов. Socket должен быть предварительно создан системным вызовом **socket** и обеспечен адресом с помощью системного вызова **bind**.

Аргумент *n* определяет максимальную длину очереди входящих запросов на установление связи. Если какой-либо клиент выдаст запрос на установление связи при полной очереди, то этот запрос будет отвергнут.

Признаком удачного завершения системного вызова **listen** служит нулевой код возврата.

1.2.2. Запрос на установление соединения

Для обращения программы-клиента к серверу с запросом на установление логической соединения используется системный вызов **connect**, имеющий следующий вид

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
int connect (s, addr, addrlen)
```

```
    int s;
    struct sockaddr_in *addr;
    int addrlen;
```

Аргумент *s* задает дескриптор socket'a, через который программа обращается к серверу с запросом на соединение. Socket должен быть предварительно создан системным вызовом **socket** и обеспечен адресом с помощью системного вызова **bind**.

Аргумент *addr* должен указывать на структуру данных, содержащую адрес, приписанный socket'у программы-сервера, к которой делается запрос на соединение. Для сетей TCP/IP такой структурой является sockaddr_in. Для формирования значений полей структуры sockaddr_in удобно использовать функцию **gethostbyname**.

Аргумент *addrlen* задает размер (в байтах) структуры данных, указываемой аргументом addr.

Для того, чтобы запрос на соединение был успешным, необходимо, по крайней мере, чтобы программа-сервер выполнила к этому моменту системный вызов **listen** для socket'a с указанным адресом.

При успешном выполнении запроса системный вызов **connect** возвращает 0, в противном случае - "-1" (устанавливая код причины неуспеха в глобальной переменной errno).

Примечание. *Если к моменту выполнения connect используемый им socket не был привязан к адресу посредством* ***bind ,****то такая привязка будет выполнена автоматически.*

Примечание. *В режиме взаимодействия без установления соединения необходимости в выполнении системного вызова* ***connect*** *нет. Однако, его выполнение в таком режиме не является ошибкой - просто меняется смысл выполняемых при этом действий: устанавливается адрес "по умолчанию" для всех последующих посылок дейтаграмм.*

1.2.3. Прием запроса на установление связи

Для приема запросов от программ-клиентов на установление связи в программах-серверах используется системный вызов **accept**, имеющий следующий вид:

```
#include <sys/socket.h>
#include <netinet/in.h>
int accept (s, addr, p_addrlen)
    int s;
    struct sockaddr_in *addr;
    int *p_addrlen;
```

Аргумент *s* задает дескриптор socket'a, через который программа-сервер получила запрос на соединение (посредством системного запроса **listen**).

Аргумент *addr* должен указывать на область памяти, размер которой позволял бы разместить в ней структуру данных, содержащую адрес socket'a программы-клиента, сделавшей запрос на соединение. Никакой инициализации этой области не требуется.

Аргумент *p_addrlen* должен указывать на область памяти в виде целого числа, задающего размер (в байтах) области памяти, указываемой аргументом addr.

Системный вызов **accept** извлекает из очереди, организованной системным вызовом **listen**, первый запрос на соединение и возвращает дескриптор нового (автоматически созданного) socket'a с теми же свойствами, что и socket, задаваемый аргументом s. Этот новый дескриптор необходимо использовать во всех последующих операциях обмена данными.

Кроме того после удачного завершения **accept**:

1. область памяти, указываемая аргументом addr, будет содержать структуру данных (для сетей TCP/IP это sockaddr_in), описывающую адрес socket'a программы-клиента, через который она сделала свой запрос на соединение;
2. целое число, на которое указывает аргумент p_addrlen, будет равно размеру этой структуры данных.

Если очередь запросов на момент выполнения **accept** пуста, то программа переходит в состояние ожидания поступления запросов от клиентов на неопределенное время (хотя такое поведение **accept** можно и изменить).

Признаком неудачного завершения **accept** служит отрицательное возвращенное значение (дескриптор socket'a отрицательным быть не может).

Примечание. *Системный вызов **accept** используется в программах-серверах, функционирующих только в режиме с установлением соединения.*

1.2.4. Формирование адреса узла сети

Для получения адреса узла сети TCP/IP по его символическому имени используется библиотечная функция

```
#include <netinet/in.h>
#include <netdb.h>
struct hostent *gethostbyname (name)
    char *name;
```

Аргумент *name* задает адрес последовательности литер, образующих символическое имя узла сети.

При успешном завершении функция возвращает указатель на структуру hostent, определенную в include-файле netdb.h и имеющую следующий вид

```
struct hostent {
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_lenght;
    char *h_addr;
    };
```

Поле h_name указывает на официальное (основное) имя узла.

Поле h_aliases указывает на список дополнительных имен узла (синонимов), если они есть.

Поле h_addrtype содержит идентификатор используемого набора протоколов, для сетей TCP/IP это поле будет иметь значение AF_INET.

Поле h_lenght содержит длину адреса узла.

Поле h_addr указывает на область памяти, содержащую адрес узла в том виде, в котором его используют системные вызовы и функции socket-интерфейса.

Пример обращения к функции **gethostbyname** для получения адреса удаленного узла в программе-клиенте, использующей системный вызов **connect** для формирования запроса на установления соединения с программой-сервером на этом узле, рассматривается ниже.

1.3. Функции обмена данными

В режиме с установлением логического соединения после удачного выполнения пары взаимосвязанных системных вызовов **connect** (в клиенте) и **accept** (в сервере) становится возможным обмен данными.

Этот обмен может быть реализован обычными системными вызовами **read** и **write**, используемыми для работы с файлами (при этом вместо дескрипторов файлов в них задаются дескрипторы socket'ов).

Кроме того могут быть дополнительно использованы системные вызовы **send** и **recv**, ориентированные специально на работу с socket'ами.

Примечание. *Для обмена данными в режиме без установления логического соединения используются, как правило, системные вызовы* ***sendto****и* ***recvfrom****.* ***Sendto*** *позволяет специфицировать вместе с передаваемыми данными (составляющими дейтаграмму) адрес их получателя.* ***Recvfrom*** *одновременно с доставкой данных получателю информирует его и об адресе отправителя.*

1.3.1. Посылка данных

Для посылки данных партнеру по сетевому взаимодействию используется системный вызов **send**, имеющий следующий вид

```
#include <sys/types.h>
#include <sys/socket.h>
int send (s, buf, len, flags)
    int s;
    char *buf;
    int len;
    int flags;
```

Аргумент *s* задает дескриптор socket'а, через который посылаются данные.

Аргумент *buf* указывает на область памяти, содержащую передаваемые данные.

Аргумент *len* задает длину (в байтах) передаваемых данных.

Аргумент *flags* модифицирует исполнение системного вызова **send**. При нулевом значении этого аргумента вызов **send** полностью аналогичен системному вызову **write**.

При успешном завершении **send** возвращает количество переданных из области, указанной аргументом buf, байт данных. Если канал данных, определяемый дескриптором s, оказывается "переполненным", то **send** переводит программу в состояние ожидания до момента его освобождения.

1.3.2. Получение данных

Для получения данных от партнера по сетевому взаимодействию используется системный вызов **recv**, имеющий следующий вид

```
#include <sys/types.h>
#include <sys/socket.h>
int recv (s, buf, len, flags)
    int s;
    char *buf;
    int len;
    int flags;
```

Аргумент *s* задает дескриптор socket'a, через который принимаются данные.

Аргумент *buf* указывает на область памяти, предназначенную для размещения принимаемых данных.

Аргумент *len* задает длину (в байтах) этой области.

Аргумент *flags* модифицирует исполнение системного вызова **recv**. При нулевом значении этого аргумента вызов **recv** полностью аналогичен системному вызову **read**.

При успешном завершении recv возвращает количество принятых в область, указанную аргументом *buf*, байт данных. Если канал данных, определяемый дескриптором s, оказывается "пустым", то **recv** переводит программу в состояние ожидания до момента появления в нем данных.

1.4. Функции закрытия связи

Для закрытия связи с партнером по сетевому взаимодействию используются системные вызовы **close** и **shutdown**.

1.4.1. Системный вызов close

Для закрытия ранее созданного socket'a используется обычный системный вызов **close**, применяемый в ОС UNIX для закрытия ранее открытых файлов и имеющий следующий вид

```
int close (s)
    int s;
```

Аргумент *s* задает дескриптор ранее созданного socket'a.

Однако в режиме с установлением логического соединения (обеспечивающем, как правило, надежную доставку данных) внутрисистемные механизмы обмена будут пытаться передать/принять данные, оставшиеся в канале передачи на момент закрытия socket'a. На это может потребоваться значительный интервал времени, неприемлемый для

некоторых приложений. В такой ситуации необходимо использовать описываемый далее системный вызов **shutdown**.

1.4.2. Сброс буферизованных данных

Для "экстренного" закрытия связи с партнером (путем "сброса" еще не переданных данных) используется системный вызов **shutdown**, выполняемый перед **close** и имеющий следующий вид

```
int shutdown (s, how)
    int s;
    int how;
```

Аргумент *s* задает дескриптор ранее созданного socket'a.

Аргумент *how* задает действия, выполняемые при очистке системных буферов socket'a:

- 0 - сбросить и далее не принимать данные для чтения из socket'a;
- 1 - сбросить и далее не отправлять данные для посылки через socket;
- 2 - сбросить все данные, передаваемые через socket в любом направлении.

1.5. Пример использования socket-интерфейса

В данном разделе рассматривается использование socket-интерфейса в режиме взаимодействия с установлением логического соединения на очень простом примере взаимодействия двух программ (сервера и клиента), функционирующих на разных узлах сети TCP/IP.

Содержательная часть программ примитивна:

1. сервер, приняв запрос на соединение, передает клиенту вопрос "Who are you?";
2. клиент, получив вопрос, выводит его в стандартный вывод и направляет серверу ответ "I am your client" и завершает на этом свою работу;
3. сервер выводит в стандартный вывод ответ клиента, закрывает с ним связь и переходит в состояние ожидания следующего запроса к нему.

Примечание. *Предлагаемые ниже тексты программ предназначены только для иллюстрации логики взаимодействия программ через сеть, поэтому в них отсутствуют такие атрибуты программ, предназначенных для практического применения, как обработка кодов возврата системных вызовов и функций, анализ кодов ошибок в глобальной переменной errno, реакция на асинхронные события и т.п.*

1.5.1. Программа-сервер

Текст программы-сервера на языке программирования СИ выглядит следующим образом

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  #include <netinet/in.h>
4  #include <netdb.h>
5  #include <memory.h>

6  #define SRV_PORT 1234
7  #define BUF_SIZE 64
```

```
 8  #define TXT_QUEST "Who are you?\n"

 9  main () {
10    int s, s_new;
11    int from_len;
12    char buf[BUF_SIZE];
13    struct sockaddr_in sin, from_sin;

14    s = socket (AF_INET, SOCK_STREAM, 0);
15    memset ((char *)&sin, '\0', sizeof(sin));
16    sin.sin_family = AF_INET;
17    sin.sin_addr.s_addr = INADDR_ANY;
18    sin.sin_port = SRV_PORT;
19    bind (s, (struct sockaddr *)&sin, sizeof(sin));
20    listen (s, 3);
21    while (1) {
22      from_len = sizeof(from_sin);
23      s_new = accept (s, &from_sin, &from_len);
24      write (s_new, TXT_QUEST, sizeof(TXT_QUEST));
25      from_len = read (s_new, buf, BUF_SIZE);
26      write (1, buf, from_len);
27      shutdown (s_new, 0);
28      close (s_new);
29      };
30    }
```

Строки 1...5 описывают включаемые файлы, содержащие определения для всех необходимых структур данных и символических констант.

Строка 6 приписывает целочисленной константе 1234 символическое имя SRV_PORT. В дальнейшем эта константа будет использована в качестве номера порта сервера. Значение этой константы должно быть известно и программе-клиенту.

Строка 7 приписывает целочисленной константе 64 символическое имя BUF_SIZE. Эта константа будет определять размер буфера, используемого для размещения принимаемых от клиента данных.

Строка 8 приписывает последовательности символов, составляющих текст вопроса клиенту, символическое имя TXT_QUEST. Последним символом в последовательности является символ перехода на новую строку '\n'. Сделано это для упрощения вывода текста вопроса на стороне клиента.

В строке 14 создается (открывается) socket для организации режима взаимодействия с установлением логического соединения (SOCK_STREAM) в сети TCP/IP (AF_INET), при выборе протокола транспортного уровня используется протокол "по умолчанию" (0).

В строках 15...18 сначала обнуляется структура данных sin, а затем заполняются ее отдельные поля. Использование константы INADDR_ANY упрощает текст программы, избавляя от необходимости использовать функцию **gethostbyname** для получения адреса локального узла, на котором запускается сервер.

Строка 19 посредством системного вызова **bind** привязывает socket, задаваемый дескриптором s, к порту с номером SRV_PORT на локальном узле. **Bind** завершится успешно при условии, что в момент его выполнения на том же узле уже не функционирует программа, использующая этот номер порта.

Строка 20 посредством системного вызова **listen** организует очередь на три входящих к серверу запроса на соединение.

Строка 21 служит заголовком бесконечного цикла обслуживания запросов от клиентов.

На строке 23, содержащей системный вызов **accept**, выполнение программы приостанавливается на неопределенное время, если очередь запросов к серверу на установление связи оказывается пуста. При появлении такого запроса **accept** успешно завершается, возвращая в переменной s_new дескриптор socket'a для обмена информацией с клиентом.

В строке 24 сервер с помощью системного вызова **write** отправляет клиенту вопрос.

В строке 25 с помощью системного вызова **read** читается ответ клиента.

В строке 26 ответ направляется в стандартный вывод, имеющий дескриптор файла номер 1. Так как строка ответа содержит в себе символ перехода на новую строку, то текст ответа будет размещен на отдельной строке дисплея.

Строка 27 содержит системный вывод **shutdown**, обеспечивающий очистку системных буферов socket'a, содержащих данные для чтения ("лишние" данные могут там оказаться в результате неверной работы клиента).

В строке 28 закрывается (удаляется) socket, использованный для обмена данными с очередным клиентом.

Примечание. *Данная программа (как и большинство реальных программ-серверов) самостоятельно своей работы не завершает, находясь в бесконечном цикле обработки запросов клиентов. Ее выполнение может быть прервано только извне путем посылки ей сигналов (прерываний) завершения. Правильно разработанная программа-сервер должна обрабатывать такие сигналы, корректно завершая работу (закрывая, в частности, посредством close socket с дескриптором s).*

1.5.2. Программа-клиент

Текст программы-клиента на языке программирования СИ выглядит следующим образом

```
 1  #include <sys/types.h>
 2  #include <sys/socket.h>
 3  #include <netinet/in.h>
 4  #include <netdb.h>
 5  #include <memory.h>

 6  #define SRV_HOST "delta"
 7  #define SRV_PORT 1234
 8  #define CLNT_PORT 1235
 9  #define BUF_SIZE 64
10  #define TXT_ANSW "I am your client\n"

11  main () {
12    int s;
13    int from_len;
14    char buf[BUF_SIZE];
15    struct hostent *hp;
16    struct sockaddr_in clnt_sin, srv_sin;
```

```
17    s = socket (AF_INET, SOCK_STREAM, 0);
18    memset ((char *)&clnt_sin, '\0', sizeof(clnt_sin));
19    clnt_sin.sin_family = AF_INET;
20    clnt_sin.sin_addr.s_addr = INADDR_ANY;
21    clnt_sin.sin_port = CLNT_PORT;
22    bind (s, (struct sockaddr *)&clnt_sin, sizeof(clnt_sin));

23    memset ((char *)&srv_sin, '\0', sizeof(srv_sin));
24    hp = gethostbyname (SRV_HOST);
25    srv_sin.sin_family = AF_INET;
26    memcpy ((char *)&srv_sin.sin_addr,hp->h_addr,hp->h_length);
27    srv_sin.sin_port = SRV_PORT;
28    connect (s, &srv_sin, sizeof(srv_sin));
29    from_len = recv (s, buf, BUF_SIZE, 0);
30    write (1, buf, from_len);
31    send (s, TXT_ANSW, sizeof(TXT_ANSW), 0);
32    close (s);
33    exit (0);
34    }
```

В строках 6 и 7 описываются константы SRV_HOST и SRV_PORT, определяющие имя удаленного узла, на котором функционирует программа-сервер, и номер порта, к которому привязан socket сервера.

Строка 8 приписывает целочисленной константе 1235 символическое имя CLNT_PORT. В дальнейшем эта константа будет использована в качестве номера порта клиента.

В строках 17...22 создается привязанный к порту на локальном узле socket.

В строке 24 посредством библиотечной функции **gethostbyname** транслируется символическое имя удаленного узла (в данном случае "delta"), на котором должен функционировать сервер, в адрес этого узла, размещенный в структуре типа hostent.

В строке 26 адрес удаленного узла копируется из структуры типа hostent в соответствующее поле структуры srv_sin, которая позже (в строке 28) используется в системном вызове connect для идентификации программы-сервера.

В строках 29...31 осуществляется обмен данными с сервером и вывод вопроса, поступившего от сервера, в стандартный вывод.

Строка 32 посредством системного вызова **close** закрывает (удаляет) socket.

2. Интерфейс транспортного уровня

Интерфейс транспортного уровня (TLI) был разработан как альтернатива более раннему socket-интерфейсу. Он базируется на средстве ввода-вывода STREAMS, первоначально реализованном в версиях System V операционной системы UNIX. Основное достоинство STREAMS заключается в гибкой, управляемой пользователем многослойности модулей, по конвейерному принципу обрабатывающих информацию, передаваемую от прикладной программы к физической среде хранения/пересылки и обратно. Это делает STREAMS удобным инструментом для реализации стеков протоколов сетевого взаимодействия различной архитектуры (OSI, TCP/IP, DECnet, SNA, XNS и т.п.).

Хотя все современные реализации и версии ОС UNIX поддерживают socket-интерфейс по крайней мере для TCP/IP, для вновь разрабатываемых сетевых приложений настоятельно рекомендуется использовать TLI, что обеспечит их независимость от используемых сетевых протоколов.

С точки зрения прикладного программиста логика TLI очень похожа на логику socket-интерфейса (даже имена функций первого образованы от имен системных вызовов второго добавлением префикса "t_"). TLI реализован в виде библиотеки функций языка программирования СИ, разделенных (как и в случае с socket-интерфейсом) на четыре группы:

1. локального управления;
2. установления связи;
3. обмена данными (ввода/вывода);
4. закрытия связи.

Основу концепции TLI составляют три базовых понятия:

- поставщик транспортных услуг
- пользователь транспорта
- транспортная точка.

Поставщиком транспортных услуг (transport provider) называется набор модулей, реализующих какой-либо конкретный стек протоколов сетевого взаимодействия (в данном учебном пособии - TCP/IP) и обеспечивающий сервис транспортного уровня модели OSI [REF].

Пользователем транспорта (transport user) является любая прикладная программа, использующая сервис, предоставляемый ПТС на локальном узле сети.

Транспортная точка (transport endpoint) - абстрактное понятие (аналогичное socket'у), используемое для обозначения канала связи между пользователем транспорта и поставщиком транспортных услуг на локальном узле сети. Транспортная точка имеем уникальный для всей сети транспортный адрес (для сетей TCP/IP этот адрес образуется триадой: адрес узла сети, номер порта, используемый протокол транспортного уровня). Для ссылки на транспортные точки в функциях TLI используются их дескрипторы, подобные дескрипторам обычных файлов и socket'ов ОС UNIX.

Ниже рассматривается подмножество функций TLI и приводится пример его использования для создания учебного приложения, функционирующего согласно модели "клиент-сервер" в режиме без установления соединения.

2.1. Структуры данных TLI

Функции TLI работают с несколькими универсальными структурами данных. При этом структура одного и того же типа может использоваться как для передачи данных в функции, так и для получения информации из них.

Ниже дается описание некоторых структур данных, используемых TLI.

2.1.1. Структура данных netbuf

Структура netbuf служит составляющей более сложных ("законченных") структур данных TLI. Она используется как для передачи данных в функции TLI, так и для размещения в ней возвращаемой из функций информации. Эта структура имеет следующий вид

```
struct netbuf {
    unsigned int maxlen;
    unsigned int len;
    char *buf;
    };
```

Поле *buf* указывает на область оперативной памяти (буфер), предназначенную для размещения в ней данных, передаваемых в функцию TLI или получаемых от нее. Семантика этих данных зависит от типа "вмещающей" структуры (см. ниже).

Поле *len* в ситуации, когда netbuf используется для передачи информации в функцию TLI, должно содержать длину (в байтах) данных, указываемых полем buf.

Поле *maxlen* в ситуациях, когда netbuf используется для получения информации от функции TLI, должно содержать длину (в байтах) области памяти, отводимой для этой цели и указываемой полем *buf*. *Len* после завершения функции будет содержать действительную длину данных, размещенных в буфере.

2.1.2. Структура данных t_bind

Структура t_bind определена в файле tiuser.h следующим образом

```
struct t_bind {
    struct netbuf addr;
    unsigned int qlen;
    };
```

Поле *addr* типа struct netbuf используется для размещения транспортного адреса транспортной точки.

Назначение поля *qlen* зависит от использующей эту структуру функции TLI.

2.1.3. Структура данных t_call

Структура t_call определена в файле tiuser.h следующим образом

```
struct t_call {
    struct netbuf addr;
    struct netbuf opt;
    struct netbuf udata;
    int sequence;
    };
```

Поле *addr* типа struct netbuf используется для размещения транспортного адреса транспортной точки.

Поле *opt* типа struct netbuf используется для размещения необязательной информации, модифицирующей или описывающей характеристики используемого конкретного поставщика транспортных услуг. Приложения, проектируемые как независимые от поставщика транспортных услуг, этим полем структуры t_call пользоваться не должны.

Поле *udata* типа struct netbuf используется для размещения передаваемых к партнеру или принимаемых от него в ходе взаимодействия через сеть данных.

Назначение поля sequence зависит от использующей эту структуру функции TLI.

2.1.4. Структура данных t_unitdata

Структура t_unitdata определена в файле tiuser.h следующим образом

```
struct t_unitdata {
    struct netbuf addr;
    struct netbuf opt;
    struct netbuf udata;
    };
```

Поле *addr* типа struct netbuf используется для размещения транспортного адреса транспортной точки.

Поле *opt* типа struct netbuf используется для размещения необязательной информации, модифицирующей или описывающей характеристики используемого конкретного поставщика транспортных услуг. Приложения, проектируемые как независимые от поставщика транспортных услуг, этим полем структуры t_unitdata пользоваться не должны.

Поле *udata* типа struct netbuf используется для размещения передаваемых к партнеру или принимаемых от него в ходе взаимодействия через сеть данных.

Структура данных типа struct t_unitdata используется в функциях посылки/приема данных в режиме взаимодействия без установления соединения.

2.2. Функции локального управления

К функциям локального управления относятся функции создания/удаления транспортной точки (t_open/t_close), назначения/снятия транспортного адреса для транспортной точки (t_bind/t_unbind), выделения/освобождения оперативной памяти под структуры данных, используемые TLI (t_alloc/t_free) и другие.

2.2.1. Выделение памяти под TLI-структуры

Динамическое выделение оперативной памяти под различные структуры данных, используемые TLI, удобно осуществлять функцией t_alloc, имеющей следующий вид

```
#include <tiuser.h>
char *t_alloc (fd, structType, fields)
    int fd;
    int structType;
    int fields;
```

Аргумент *fd* задает дескриптор ранее созданной функцией t_open транспортной точки.

Аргумент *structType* задает тип структуры данных, под которую необходимо выделить память, и может принимать следующие значения:

- T_INFO для struct t_info;
- T_BIND для struct t_bind;
- T_CALL для struct t_call;
- T_UNITDATA для struct t_unitdata;
- T_DIS для struct t_discon и др.

Каждая из указанных структур (исключая struct t_info) содержит одно или несколько полей типа struct netbuf. Для каждого из таких полей можно также потребовать динамического выделения памяти. Аргумент fields конкретизирует это требование, допуская задание fields в виде побитового ИЛИ из следующих значений:

- T_ALL для выделения памяти для всех полей типа struct netbuf, имеющихся в структуре;
- T_ADDR для поля addr;
- T_UDATA для поля udata;
- T_OPT для поля opt.

При успешном завершении функция возвращает указатель на размещенную структуру данных, в противном случае - NULL.

2.2.2. Освобождение памяти

Для освобождения оперативной памяти, динамически выделенной под различные структуры данных, используемые TLI, удобно использовать функцию t_free, имеющую следующий вид

```
#include<tiuser.h>
int t_free (ptr, structType)
    char *ptr;
    int structType;
```

Аргумент *ptr* указывает освобождаемую область памяти.

Аргумент *structType* задает тип структуры данных, занимающей память. Этот аргумент может принимать те же значения, что и аналогичный аргумент функции t_alloc.

Функция t_free освобождает оперативную память, занятую собственно структурой и всеми ее буферами типа struct netbuf.

При успешном завершении функция t_free возвращает ноль, иначе - число "-1" и устанавливает код ошибки в глобальной переменной t_errno.

2.2.3. Создание транспортной точки

Создание транспортной точки осуществляется функцией t_open, имеющей следующий вид

```
#include <tiuser.h>
#include <fcntl.h>
int t_open (path, oflags, info)
    char *path;
    int oflags;
    struct t_info *info;
```

Аргумент *path* задает имя файла (располагающегося, как правило, в каталоге /dev), определяющего используемого поставщика транспортных услуг. Для стека протоколов TCP/IP такими файлами могут быть /dev/tcp (режим с установлением логического соединения) и /dev/udp (режим без установления логического соединения).

Аргумент *oflags* задает флаги открытия транспортной точки. Допустимые значения флагов - те же, что и для обычного системного вызова open. Если транспортная точка создается для двустороннего обмена информацией через нее, то значением oflags должно быть O_RDWR.

Аргумент *info* должен указывать на структуру данных типа struct t_info, поля которой заполняются функцией t_open при ее удачном завершении информацией о характеристиках используемого поставщика транспортных услуг. Если info задан как NULL, то информация о протоколе не возвращается. Для выделения памяти под структуру удобно использовать функцию t_alloc [REF].

При успешном завершении функция t_open возвращает дескриптор транспортной точки, используемый для ссылки на нее в большинстве функций TLI. Дескриптор транспортной точки аналогичен дескриптору socket'a. При обнаружении ошибки в ходе своей работы функция возвращает число "-1" и устанавливает код ошибки в глобальной переменной t_errno.

2.2.4. Назначение транспортного адреса

Для назначения транспортного адреса транспортной точке и ее активизации используется функция t_bind, имеющая следующий вид

```
#include <tiuser.h>
int t_bind (fd, req, ret)
    int fd;
    struct t_bind *req;
    struct t_bind *ret;
```

Аргумент *fd* задает дескриптор транспортной точки, созданной ранее с помощью функции t_open.

Аргумент *req* указывает на структуру t_bind, которая должна определять требуемый транспортный адрес для точки (поле req->addr) и максимальное количество запросов на соединение (поле req->qlen), одновременно обрабатываемых программой. Для программы-клиента поле req->qlen должно быть нулевым, а для программ-серверов,

работающих в режиме с установлением логического соединения, оно, как правило, содержит 1 (необходимо учитывать, что не все поставщики транспортных услуг могут обеспечивать одновременную обработку сразу нескольких соединений к одной транспортной точке). Для программ-серверов, функционирующих в режиме без установления логического соединения, поле req->qlen смысла не имеет.

Если аргумент *req* имеет значение NULL, то функция t_bind сама назначит произвольный транспортный адрес для точки.

Аргумент *ret* должен указывать на область памяти под структуру t_bind, в которой после успешного выполнения функции будет размещена информация о транспортном адресе, назначенном транспортной точке. Если этот аргумент равен NULL, то информация о назначенном транспортном адресе возвращена не будет.

При успешном завершении функция t_bind возвращает ноль, иначе - число "-1" и устанавливает код ошибки в глобальной переменной t_errno.

2.2.5. Снятие транспортного адреса

Для снятия транспортного адреса у транспортной точки используется функция t_unbind, имеющая следующий вид

```
#include <tiuser.h>
int t_unbind (fd)
    int fd;
```

Аргумент *fd* задает дескриптор транспортной точки, которой ранее с помощью функции t_bind был назначен транспортный адрес.

После выполнения функции t_unbind для транспортной точки обмен данными через нее становится невозможным.

При успешном завершении функция t_unbind возвращает ноль, иначе - число "-1" и устанавливает код ошибки в глобальной переменной t_errno.

2.2.6. Удаление транспортной точки

Удаление транспортной точки осуществляется функцией t_close, имеющей следующий вид

```
#include <tiuser.h>
int t_close (fd)
    int fd;
```

Аргумент *fd* задает дескриптор транспортной точки, созданной ранее с помощью функции t_open.

При успешном завершении функция t_close возвращает ноль, иначе - число "-1" и устанавливает код ошибки в глобальной переменной t_errno.

2.3. Функции установления связи

Для установления логического соединения "клиент-сервер" в TLI используются функции t_listen, t_accept (на стороне сервера), t_connect (на стороне клиента), а также ряд других.

2.3.1. Ожидание запроса на соединение

Ожидание в программе-сервере запроса от клиента на соединение реализуется функцией t_listen, имеющей следующий вид

```
#include <tiuser.h>
int t_listen (fd, call)
    int fd;
    struct t_call *call;
```

Аргумент *call* должен указывать на область памяти под структуру t_call, в которой после успешного выполнения функции будет размещена следующая информация: транспортный адрес (call->addr) транспортной точки програм- мы-клиента, через которую она делает запрос на установление соединения; необязательные характеристики соединения (call->opt); необязательные данные (call->udata), передаваемые клиентом серверу вместе с запросом на соединение (однако, не любой поставщик транспортных услуг обеспечивает возможность передачи данных вместе с запросом на соединение); уникальный идентификатор соединения (call->sequence), имеющий смысл для программы-сервера только, если она допускает обслуживание одновременно нескольких соединений с нею.

Функция t_listen извлекает из очереди запросов на установление соединения первый запрос и возвращает в области памяти, указываемой аргументом call, описанную выше информацию клиента. Если очередь запросов на момент выполнения t_listen пуста, то программа переходит в состояние ожидания поступления запросов от клиентов на неопределенное время (хотя такое поведение t_listen можно и изменить).

При успешном завершении функция t_listen возвращает ноль, иначе - число "-1" и устанавливает код ошибки в глобальной переменной t_errno.

Примечание. *Обратите внимание: схожие по названию функция t_listen и системный вызов listen из socket-интерфейса имеют различный смысл.*

2.3.2. Прием запроса на соединение

Прием в программе-сервере запроса от клиента на соединение, "услышанного" функцией t_listen, реализуется функцией t_accept, имеющей следующий вид

```
#include <tiuser.h>
int t_accept (fd, resfd, call)
    int fd;
    int resfd;
    struct t_call *call;
```

Аргумент *fd* задает дескриптор транспортной точки, через которую ранее выполненная функция t_listen получила запрос на соединение.

Аргумент *resfd* задает дескриптор еще одной транспортной точки, созданной с теми же свойствами, что и точка, задаваемая аргументом fd, но имеющей другой транспортный адрес.

Аргумент *call* указывает на структуру данных типа t_call, поля которой должны содержать следующую информацию: транспортный адрес (call->addr) транспортной точки программы-клиента, через которую она сделала запрос на установление соединения; необязательные характеристики соединения (call->opt); необязательные данные (call->udata), возвращаемые сервером клиенту вместе с подтверждением установления соединения (однако, не любой поставщик транспортных услуг обеспечивает возможность такой передачи данных); уникальный идентификатор (call->sequence), присвоенный соединению функцией t_listen.

После успешного выполнения в программе-сервере функции t_accept устанавливается логическое соединение с клиентом и становится возможным обмен данными с ним через дескриптор resfd.

В типичной программе сервере транспортная точка с дескриптором resfd создается и активизируется после успешного завершения функции t_listen с помощью функций t_open и t_bind. Допустимой является ситуация, когда resfd = fd, но тогда программа-сервер до момента закрытия соединения с клиентом теряет возможность получать и ставить в очередь запросы на соединение от других клиентов.

При успешном завершении функция t_accept возвращает ноль, иначе - число "-1" и устанавливает код ошибки в глобальной переменной t_errno.

Программа-сервер может отказаться от установления соединения с клиентом, используя функцию t_snddis.

Примечание. *Обратите внимание: схожие по названию функция t_accept и системный вызов accept из socket-интерфейса имеют различный смысл.*

2.3.4. Отвергнуть запрос на соединение

Программа-сервер может отвергнуть запрос клиента на соединение, "услышанный" функцией t_listen, используя функцию t_snddis, имеющую следующий вид

```
#include <tiuser.h>
int t_snddis (fd, call)
    int fd;
    struct t_call *call;
```

Аргумент *fd* задает дескриптор транспортной точки, через которую ранее выполненная функция t_listen получила запрос на соединение.

Аргумент *call* указывает на структуру данных типа t_call, поля которой должны содержать следующую информацию: уникальный идентификатор (call->sequence), присвоенный соединению функцией t_listen; необязательные данные (call->udata), возвращаемые сервером клиенту вместе с информацией об отклонении запроса на соединения (однако, не любой поставщик транспортных услуг обеспечивает возможность такой передачи данных); поля call->addr и call->opt не используются.

При успешном завершении функция t_snddis возвращает ноль, иначе - число "-1" и устанавливает код ошибки в глобальной переменной t_errno.

Примечание. *Функция t_snddis используется также для "экстренного" закрытия ранее установленного соединения, при этом аргумент call формируется несколько иначе.*

2.3.5. Запрос на установление соединения

Для обращения программы-клиента к серверу с запросом на установление логической соединения используется функция t_connect, имеющая следующий вид

```
#include <tiuser.h>
int t_connect (fd, sndcall, rcvcall)
    int fd;
    struct t_call *sndcall;
    struct t_call *rcvcall;
```

Аргумент *fd* задает дескриптор транспортной точки, созданной ранее с помощью функции t_open и активизированной функцией t_bind.

Аргумент *sndcall* указывает на структуру данных типа t_call, в которой функции передается следующая информация: транспортный адрес (sndcall->addr) транспортной точки программы-сервера, к которой клиент делает запрос на установление соединения; необязательные характеристики соединения (sndcall->opt); необязательные данные (sndcall->udata), передаваемые клиентом серверу вместе с запросом на соединение (однако, не любой постав- щик транспортных услуг обеспечивает возможность передачи данных вместе с запросом на соединение).

Поле sndcall->sequence не используется и может принимать произвольное значение.

Аргумент *rcvcall* должен указывать на область памяти под структуру t_call, в которой после успешного выполнения функции будет размещена следующая информация: транспортный адрес (rcvcall->addr) транспортной точки в программе-сервере, с которой установлено соединение; необязательные характеристики соединения (rcvcall->opt); необязательные данные (rcvcall->udata), передаваемые клиенту сервером (посредством функции t_accept) вместе с подтверждением соединения (однако, не любой поставщик транспортных услуг обеспечивает возможность такой передачи данных); поле rcvcall->sequence не используется.

При успешном установлении соединения функция t_connect возвращает ноль. Если же сервер отверг запрос на соединение, то t_connect возвращает "-1" и устанавливает код ошибки TLOOK в глобальной переменной t_errno.

2.4. Функции обмена данными

В режиме с установлением логического соединения для обмена данными используются функции t_snd и t_rcv.

В режиме без установления логического соединения для обмена данными используются функции t_sndudata и t_rcvudata.

2.4.1. Посылка данных в режиме с установлением соединения

Для посылки данных партнеру по сетевому взаимодействию в режиме с установлением логического соединения используется функция t_snd, имеющая следующий вид

```
#include <tiuser.h>
int t_snd (fd, buf, len, flags)
    int fd;
    char *buf;
```

```
    unsigned int len;
    int flags;
```

Аргумент *fd* задает дескриптор транспортной точки, через которую посылаются данные.

Аргумент *buf* указывает на область памяти, содержащую передаваемые данные.

Аргумент *len* задает длину (в байтах) передаваемых данных.

Аргумент *flags* модифицирует исполнение функции t_snd. При нулевом значении этого аргумента функция t_snd полностью аналогична системному вызову write.

При успешном завершении t_snd возвращает количество переданных из области, указанной аргументом buf, байт данных. Если канал данных, определяемый дескриптором fd, оказывается "переполненным", то t_snd переводит программу в состояние ожидания до момента его освобождения.

2.4.2. Прием данных в режиме с установлением соединения

Для получения данных от партнера по сетевому взаимодействию в режиме с установлением логического соединения используется функция t_rcv, имеющая следующий вид

```
#include <tiuser.h>
int t_rcv (fd, buf, len, flags)
    int fd;
    char *buf;
    unsigned int len;
    int flags;
```

Аргумент *fd* задает дескриптор транспортной точки, через которую принимаются данные.

Аргумент *buf* указывает на область памяти, предназначенную для размещения принимаемых данных.

Аргумент *len* задает длину (в байтах) этой области.

Аргумент *flags* модифицирует исполнение системного вызова recv. При нулевом значении этого аргумента вызов t_rcv полностью аналогичен системному вызову read.

При успешном завершении t_rcv возвращает количество принятых в область, указанную аргументом buf, байт данных. Если канал данных, определяемый дескриптором *fd*, оказывается "пустым", то t_rcv переводит программу в состояние ожидания до момента появления в нем данных.

2.4.3. Посылка данных в режиме без установления соединения

Для посылки данных, составляющих дейтаграмму, партнеру по сетевому взаимодействию в режиме без установления логического соединения используется функция t_sndudata, имеющая следующий вид

```
#include <tiuser.h>
int t_sndudata (fd, unitdata)
    int fd;
```

```
    struct t_unitdata *unitdata;
```

Аргумент *fd* задает дескриптор транспортной точки, через которую посылаются данные.

Аргумент *unitdata* указывает на структуру данных типа t_unitdata, в которой функции передается следующая информация: транспортный адрес (unitdata->addr) транспортной точки программы-партнера по взаимодействию, которой посылается дейтаграмма; необязательные характеристики соединения (unitdata->opt); собственно данные (unitdata->udata), составляющие дейтаграмму, передаваемую партнеру по взаимодействию.

Если канал данных, определяемый дескриптором *fd*, оказывается "переполненным", то t_sndudata переводит программу в состояние ожидания до момента его освобождения.

При успешном выполнении функция t_sndudata возвращает ноль, в противном случае - число "-1" и устанавливает код ошибки в глобальной переменной t_errno.

2.4.4. Прием данных в режиме без установления соединения

Для получения данных, составляющих дейтаграмму, от партнера по сетевому взаимодействию в режиме без установления логического соединения используется функция t_rcvudata, имеющая следующий вид

```
#include <tiuser.h>
int t_rcvudata (fd, unitdata, flags)
    int fd;
    struct t_unitdata *unitdata;
    int *flags;
```

Аргумент *fd* задает дескриптор транспортной точки, через которую посылаются данные.

Аргумент *unitdata* указывает на структуру данных типа t_unitdata, в которой функции передается следующая информация: транспортный адрес (unitdata->addr) транспортной точки программы-партнера по взаимодействию, которой посылается дейтаграмма; необязательные характеристики соединения (unitdata->opt); собственно данные (unitdata->udata), составляющие дейтаграмму, передаваемую партнеру по взаимодействию.

Аргумент *unitdata* должен указывать на область памяти под структуру t_unitdata, в которой после успешного выполнения функции будет размещена следующая информация: транспортный адрес (unitdata->addr) транспортной точки в программе-партнере по взыимодействию, отправившей дейтаграмму; необязательные характеристики соединения (unitdata->opt); собственно данные (unitdata->udata), составляющие дейтаграмму, принимаемую от партнера по взаимодействию.

Аргумент *flags* должен указывать область памяти (типа int), в которой функция t_rcvudata может установить флаг T_MORE, сигнализирующий о том, что в канале передачи остались еще данные, составляющие дейтаграмму. Такая ситуация может возникнуть в случае, если размер буфера в unitdata->udata недостаточен для размещения в нем сразу всей дейтаграммы.

Если канал данных, определяемый дескриптором *fd*, оказывается "пустым", то t_rcvudata переводит программу в состояние ожидания до момента появления в нем данных.

При успешном выполнении функция t_rcvudata возвращает ноль, в противном случае - число "-1" и устанавливает код ошибки в глобальной переменной t_errno.

2.5. Функции закрытия соединения

TLI поддерживает две процедуры закрытия связи в режиме с установлением логического соединения: упорядоченную и экстренную.

Упорядоченная процедура реализуется парой функций t_sndrel и t_rcvrel и обеспечивает надежную доставку к партнеру по взаимодействию всех данных, планируемых для передачи. Однако не все поставщики транспортных услуг поддерживают эту процедура закрытия связи.

Экстренная процедура закрытия логического соединения реализуется функциями t_snddis и t_rcvdis.

В данном учебном пособии процедуры закрытия логического соединения не рассматриваются.

2.6. Пример использования TLI

В данном разделе рассматривается использование TLI в режиме взаимодействия без установления логического соединения на очень простом примере взаимодействия двух программ (сервера и клиента), функционирующих на разных узлах сети TCP/IP.

Содержательная часть программ примитивна:

1. клиент направляет серверу вопрос "What must I do?";
2. сервер, получив вопрос, выводит его в стандартный вывод и возвращает клиенту ответ "Continue" или "Cancel";
3. клиент выводит в стандартный вывод ответ сервера и прекращает свою работу, если ответом было "Cancel", или выполняет действия первого пункта.

Примечание. *Предлагаемые ниже тексты программ предназначены только для иллюстрации логики взаимодействия программ через сеть, поэтому в них отсутствуют такие атрибуты программ, предназначенных для практического применения, как обработка кодов возврата системных вызовов и функций, анализ кодов ошибок в глобальных переменных errno и t_errno, реакция на асинхронные события и т.п.*

2.6.1. Программа-сервер

Текст программы-сервера на языке программирования СИ выглядит следующим образом

```
 1  #include <tiuser.h>
 2  #include <fcntl.h>
 3  #include <stdio.h>
 4  #include <sys/socket.h>
 5  #include <netinet/in.h>
 6  #include <netdb.h>
 7  #include <memory.h>
 8  #include <sys/time.h>

 9  #define SRV_PORT 1234
10  #define CONT_TXT "Continue\n"
```

```
11  #define CANC_TXT "Cancel\n"

12  main () {
13    int fd;
14    int flags;
15    time_t secs;
16    struct t_bind *bind;
17    struct t_unitdata *ud;
18    struct sockaddr_in *p_addr;
19    extern int t_errno;

20    fd = t_open("/dev/udp", O_RDWR, NULL);
21    bind = (struct t_bind *) t_alloc (fd, T_BIND, T_ADDR);
22    memset (bind->addr.buf, '\0', bind->addr.maxlen);
23    p_addr = (struct sockaddr_in *) bind->addr.buf;
24    p_addr->sin_family = AF_INET;
25    p_addr->sin_addr.s_addr = INADDR_ANY;
26    p_addr->sin_port = SRV_PORT;
27    bind->addr.len = sizeof(struct sockaddr_in);
28    bind->qlen = 0;
29    t_bind (fd, bind, bind);
30    ud = (struct t_unitdata *)t_alloc(fd,T_UNITDATA,T_ALL);

31    while (1) {
32      t_rcvudata (fd, ud, &flags);
33      write (1, ud->udata.buf, ud->udata.len);
34      secs = time (NULL);
35      if (secs % 3) {
36        strcpy (ud->udata.buf, CONT_TXT);
37        ud->udata.len = sizeof(CONT_TXT);
38        }
39       else {
40        strcpy (ud->udata.buf, CANC_TXT);
41        ud->udata.len = sizeof(CANC_TXT);
42        };
43      t_sndudata (fd, ud);
44      };
45    }
```

Строки 1...8 описывают включаемые файлы, содержащие определения для всех необходимых структур данных и символических констант.

Строка 9 приписывает целочисленной константе 1234 символическое имя SRV_PORT. В дальнейшем эта константа будет использована в качестве номера порта сервера. Значение этой константы должно быть известно и программе-клиенту.

Строки 10 и 11 приписывают последовательностям символов, составляющих тексты возможных ответов клиенту, символические имена CONT_TXT и CANC_TXT. Последним символом в последовательностях является символ перехода на новую строку '\n'. Сделано это для упрощения вывода текста ответа на стороне клиента.

В строке 20 создается (открывается) транспортная точка для организации режима взаимодействия без установления логического соединения с помощью протокола транспортного уровня UDP в сети TCP/IP (/dev/udp). Транспортная точка будет использоваться для двустороннего обмена информацией (O_RDWR). Третий аргумент функции t_open задан как NULL, поскольку данную программу особые характеристики поставщика транспортных услуг не интересуют.

В строке 21 выделяется оперативная память под структуру данных типа struct t_bind и под буфер данных, определяемый полем addr этой структуры (размер памяти, выделяемой под этот буфер, функция t_alloc вычисляет самостоятельно на основе информации о конкретном поставщике транспортных услуг). Поле addr этой структуры в своем буфере будет содержать транспортный адрес транспортной точки, который для поставщиков транспортных услуг UDP и TCP имеет тот же формат, что и адрес socket'a.

В строках 22...27 сначала обнуляется структура данных типа struct sockaddr_in, на которую указывает bind->addr.buf а затем заполняются ее отдельные поля. Использование константы INADDR_ANY упрощает текст программы, избавляя от необходимости использовать функцию gethostbyname для получения адреса локального узла, на котором запускается сервер.

В строке 28 переменной bind->qlen присваивается значение 0, поскольку наш сервер предназначен для работы в режиме без установления соединения.

Строка 29 посредством функции t_bind привязывает к транспортной точке транспортный адрес, описанный в структуре, на которую указывает bind. T_bind завершится успешно при условии, что в момент его выполнения на том же узле уже не функционирует программа, использующая этот же транспортный адрес.

Строка 31 служит заголовком бесконечного цикла обслуживания запросов от клиентов.

В строке 32 с помощью функции t_rcvudata читается запрос клиента.

В строке 33 текст запроса направляется в стандартный вывод, имеющий дескриптор файла номер 1. Так как строка ответа содержит в себе символ перехода на новую строку, то текст ответа будет размещен на отдельной строке дисплея.

Строка 34 содержит обращение к функции time, возвращающей количество секунд времени, прошедших с 1 января 1970 г. до текущего момента. Это значение используется в программе-сервере для выбора варианта ответа клиенту.

В строке 43 сервер с помощью функции t_sndudata отправляет клиенту ответ на его запрос, выбранный из двух возможных вариантов псевдослучайным образом.

Примечание. *Данная программа (как и большинство реальных программ-серверов) самостоятельно своей работы не завершает, находясь в бесконечном цикле обработки запросов клиентов. Ее выполнение может быть прервано только извне путем посылки ей сигналов (прерываний) завершения. Правильно разработанная программа-сервер должна обрабатывать такие сигналы, корректно завершая работу.*

2.6.2. Программа-клиент

Текст программы-клиента на языке программирования СИ выглядит следующим образом

```
1  #include <tiuser.h>
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <sys/socket.h>
5  #include <netinet/in.h>
6  #include <netdb.h>
7  #include <memory.h>
```

```
 8  #define SRV_HOST "delta"
 9  #define SRV_PORT 1234
10  #define ASK_TXT "What must I do?\n"
11  #define CONT_TXT "Continue\n"
12  #define CANC_TXT "Cancel\n"

13  main () {
14    int fd;
15    int flags;
16    struct t_unitdata *ud;
17    struct sockaddr_in *p_addr;
18    struct hostent *hp;
19    extern int t_errno;

20    fd = t_open ("/dev/udp", O_RDWR, NULL);
21    t_bind (fd, NULL, NULL);
22    ud = (struct t_unitdata *) t_alloc (fd, T_UNITDATA, T_ALL);
23    memset (ud->addr.buf, '\0', ud->addr.maxlen);
24    p_addr = (struct sockaddr_in *) ud->addr.buf;
25    hp = gethostbyname (SRV_HOST);
26    p_addr->sin_family = AF_INET;
27    memcpy((char *)&(p_addr->sin_addr),hp->h_addr,hp->h_length);
28    p_addr->sin_port = SRV_PORT;
29    ud->addr.len = sizeof(struct sockaddr_in);
30    while (1) {
31      strcpy (ud->udata.buf, ASK_TXT);
32      ud->udata.len = sizeof(ASK_TXT);
33      t_sndudata (fd, ud);
34      t_rcvudata (fd, ud, &flags);
35      write (1, ud->udata.buf, ud->udata.len);
36      if ( strcmp(ud->udata.buf, CONT_TXT) )
37        break;
38      };
39    t_free ((char *) ud, T_UNITDATA);
40    t_close (fd);
41    exit (0);
42    }
```

В строках 8 и 9 описываются константы SRV_HOST и SRV_PORT, определяющие имя удаленного узла, на котором функционирует программа-сервер, и номер порта, к которому привязана транспортная точка сервера.

В строках 20 и 21 создается транспортная точка, имеющая не интересующий нас в этой программе транспортный адрес.

В строке 22 выделяется оперативная память под структуру данных типа struct t_unitdata и под три буфера, определяемых полями addr, opt и udata этой структуры (размер памяти, выделяемой под эти буфера, функция t_alloc вычисляет самостоятельно на основе информации о конкретном поставщике транспортных услуг).

В строке 25 посредством библиотечной функции gethostbyname транслируется символическое имя удаленного узла (в данном случае "delta"), на котором должен функционировать сервер, в адрес этого узла, размещенный в структуре типа hostent.

В строке 27 адрес удаленного узла копируется из структуры типа struct hostent в соответствующее поле структуры типа struct sockaddr_in, которая размещена в буфере ud->addr.

В строках 31 и 32 заполняются поля структуры ud->udata передаваемой серверу информацией и длиной этой информации.

В строке 33 с помощью функции t_sndudata посылается запрос серверу.

В строке 34 с помощью функции t_rcvudata принимается ответ от сервера. При этом транспортный адрес транспортной точки отправителя ответа (сервера) размещается функцией в ud->addr, а сами данные, составляющие ответ, - в ud->udata.

В строке 39 освобождается оперативная память, занимавшаяся структурой типа struct t_unitdata.

Строка 40 посредством функции t_close закрывает (удаляет) транспортную точку.

3. Вызов удаленных процедур

Средства вызова удаленных процедур (RPC) является составной частью более общего средства, называемого Open Network Computing (ONC), разработанного фирмой Sun Microsystems и получившего всеобщее признание в качестве промышленного стандарта.

ONC помимо RPC включает в себя средства внешнего представления данных (XDR), необходимые для организации обмена информацией в гетерогенных сетях, включающих в себя ЭВМ различной архитектуры, и средства монтирования удаленных файловых систем (NFS), обеспечивающее доступ пользователям локального узла сети к файлам, физически расположенным на удаленных узлах, как к файлам локальным.

Средство RPC реализует модель "клиент-сервер", где роль клиента играет прикладная программа, обращающаяся к набору процедур (функций), исполняемых на удаленном узле в качестве сервера. RPC предоставляет прикладным программистам сервис более высокого уровня, чем ранее рассмотренных два, т.к. обращение за услугой к удаленным процедурам выполняется в привычной для программиста манере вызова "локальной" функции языка программирования СИ. RPC реализовано, как правило, на базе socket-интерфейса и/или TLI. При пересылке данных между узлами сети в RPC для их внешнего представления используется стандарт XDR.

Средство RPC предоставляет программистам сервис трех уровней:

1. препроцессор rpcgen, преобразующий исходные тексты "монолитных" программ на языке программирования СИ в исходные тексты программы-клиента и программы-сервера по спецификации программиста;
2. библиотека функций вызова удаленных процедур по их идентификаторам;
3. библиотека низкоуровневых функций доступа к внутренним механизмам RPC и ниже лежащим протоколам транспортного уровня.

В данном учебном учебном пособии рассматриваются только средства RPC среднего уровня.

Согласно идеологии RPC все процедуры (функции) некоторого распределенного приложения, планируемые к исполнению на одном и том же удаленном узле вычислительной сети, объединяются в единый модуль, оформляемый в виде исполняемого файла и характеризующегося уникальным "номером программы". Допустимо иметь несколько вариантов такого модуля, идентифицируемых уникальным "номером версии". Каждая процедура в составе модуля имеет уникальный "номер процедуры". Таким образом, для однозначной идентификации конкретной процедуры-сервера используется четверка:

- имя узла сети;
- номер программы на этом узле;
- номер версии программы;
- номер процедуры в программе.

Каждая процедура-сервер прежде, чем она станет доступной для обращения к ней, должна быть зарегистрирована на соответствующем узле сети. Регистрация процедуры делает ее известной под соответствующими номерами (программы, версии и, собственно, процедуры) сетевому демону **portmapper** на локальном узле сети. Удаленный RPC-клиент, обращаясь скрытно от пользователя к этому демону, может получить точный

сетевой адрес процедуры, который и будет использовать в дальнейшем для прямых обращений к процедуре-серверу.

Процедура-сервер, создаваемая средствами RPC любого уровня, должна иметь единственный аргумент и единственный результат. Это ограничение заставляет прикладного программиста в случае необходимости передачи в процедуру (или возврата из нее) нескольких аргументов (результатов) компоновать их в сложные агрегаты данных (структуры, массивы, списки и т.п.).

Для создания распределенных приложений средствами RPC среднего уровня достаточно использовать три функции: **registerrpc**, **svc_run** (на стороне сервера) и **callrpc** (на стороне клиента).

Примечание. *Столь малое количество функций объясняется тем, что средний уровень средств RPC беден с точки зрения возможностей выбора используемого транспорта данных, управления количеством ретрансляций данных, назначения тайм-аутов, организации асинхронной обработки и т.п.*

Программисты распределенных приложений кроме собственно функций RPC обязаны также использовать функции преобразования данных во внешнее представление согласно стандарту XDR (так называемые XDR-функции).

3.1 Регистрации процедуры-сервера

Регистрация процедуры в качестве сервера на узле сети выполняется функцией registerrpc, имеющей следующий вид

```
#include <sys/types.h>
#include <rpc/rpc.h>
int registerrpc (prognum, vernum, procnum, procname,
                 inproc, outproc)
u_long prognum;
u_long vernum;
u_long procnum;
char *(*procname) ();
xdrproc_t inproc;
xdrproc_t outproc;
```

Аргументы *prognum*, *vernum* и *procnum* задают номера программы, версии и процедуры соответственно. Номера версии и процедуры назначаются программистом произвольно. Номер же программы, находящейся в стадии разработки, должен назначаться из диапазона 0x20000000...0x3fffffff.

Аргумент *procname* задает функцию языка программирования СИ, регистрируемую в качестве сервера. Эта функция (процедура) вызывается с указателем на ее аргумент и должна возвращать указатель на свой результат, располагаемый в статической или динамически выделенной (функциями **malloc** или **calloc**) памяти. Для хранения результата нельзя использовать автоматически выделяемую память (напоминаем, что локальные переменные функций располагаются именно в такой памяти).

Аргументы *inproc* и *outproc* задают XDR-функции преобразования, соответственно, аргумента и ее результата.

При успешном выполнении функция **registerrpc** возвращает 0, иначе - число "-1".

3.2. Диспетчеризация запросов к процедурам-серверам

Для приема запросов к процедурам-серверам от клиентов и диспетчеризации их используется функция svc_run, имеющая следующий вид

```
#include <rpc/rpc.h>
void svc_run ();
```

Не имеющая аргументов функция svc_run должна вызываться после регистрации всех диспетчируемых ею процедур-серверов. При успешном выполнении svc_run никогда не возвращает управление в вызвавшую ее программу.

3.3. Запрос к процедуре-серверу

Для запроса к удаленной процедуре-серверу из программы-клиента используется функция **callrpc**, имеющая следующий вид

```
#include <sys/types.h>
#include <rpc/rpc.h>
int callrpc (host, prognum, vernum, procnum,
                    inproc, in, outproc, out)
char *host;
u_long prognum;
u_long vernum;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
```

Аргумент *host* задает имя узла, на котором функционирует вызы- ваемая процедура-сервер.

Аргументы *prognum*, *vernum* и *procnum* задают номера программы, версии и, собственно, вызываемой процедуры-сервера. К моменту вызова процедуры она должна быть зарегистрирована на узле сети, определяемом аргументом *host*.

Аргумент *in* должен указывать на данные, передаваемые процеду- ре-серверу в качестве аргумента.

Аргумент *out* должен указывать на область памяти, предназначенную для размещения в ней результата работы процедуры-сервера.

Аргументы *inproc* и *outproc* задают XDR-функции преобразования, соответственно, аргумента процедуры-сервера и ее результата.

На время обработки процедурой-сервером запроса к ней программа-клиент переходит в состояние ожидания результата.

При успешном выполнении вызова удаленной процедуры-сервера функция **registerrpc** возвращает 0, иначе - число "-1".

3.4. XDR-функции

Для преобразования данных в/из XDR-формат библиотека функций RPC содержит ряд функций, некоторые из них перечислены ниже:

- **xdr_int** - для преобразования целых;
- **xdr_u_int** - для преобразования беззнаковых целых;
- **xdr_short** - для преобразования коротких целых;
- **xdr_u_short** - для преобразования беззнаковых коротких целых;
- **xdr_long** - для преобразования длинных целых;
- **xdr_u_long** - для преобразования беззнаковых длинных целых;
- **xdr_char** - для преобразования символов;
- **xdr_u_char** - для преобразования беззнаковых символов;
- **xdr_wrapstring** - для преобразования строк символов (заканчивающихся символом '\0').

В ситуациях, когда в процедуру-сервер аргумент не передается (или от нее не возвращается результат), используется функция-"заглушка" **xdr_void**.

XDR-функции универсальны: в зависимости от контекста их использования они могут преобразовывать данные из внутреннего представления, специфичного для ЭВМ данной архитектуры, во внешнее согласно стандарту XDR и наоборот, а также динамически выделять/освобождать память под сложные агрегаты данных.

При необходимости прикладной программист может легко расширить имеющийся набор XDR-функций, создав свои собственные на базе уже имеющихся.

3.5. Пример использования средств RPC

В данном разделе рассматривается использование средств RPC на очень простом примере взаимодействия программы-клиента с одной удаленной процедурой-сервером.

Содержательная часть программ примитивна:

1. процедура-сервер, получив в качестве аргумента запроса целое число, возвращает клиенту строку символов "even" ("четное") или "odd" ("нечетное");
2. программа-клиент обращается к процедуре-серверу, передавая ей целое число, являющееся результатом подсчета количества символов в некоторой строке, и выводит в стандартный вывод ответ процедуры-сервера.

3.5.1. Включаемый файл

При разработке прикладных программ средствами RPC целесообразно для номеров программ, версий и процедур определять символические имена в едином файле, включаемом в исходный текст как процедур-серверов, так и клиентов. В нашем примере таким файлом будет файл **common.h**, имеющий следующий вид

```
#define MY_PROG  (u_long) 0x20000001
#define MY_VER   (u_long) 1
#define MY_PROC1 (u_long) 1
```

3.5.2. Программа-сервер

Текст программы-сервера на языке программирования СИ выглядит следующим образом

```
 1  #include <rpc/rpc.h>
 2  #include <stdio.h>
 3  #include "common.h"

 4  char **proc1 ();

 5  main ()
 6    {
 7    registerrpc (MY_PROG, MY_VER, MY_PROC1,
                   proc1, xdr_int, xdr_wrapstring);
 8    svc_run();
 9    fprintf (stderr, "Error: svc_run returned\n");
10    exit (1);
11    }

12  char **proc1 (indata_p)
13    int *indata_p;
14    {
15    static char *res;
16    static char even[] = {"even"};
17    static char odd[] = {"odd"};
18    printf ("Number recieved is %d\n", *indata_p);
19    if (*indata_p % 2) {
20      res = odd; }
21     else {
22      res = even; };
23    return &res;
24    }
```

Строки 1...3 описывают включаемые файлы, содержащие определения для всех необходимых структур данных и символических констант.

Строка 4 объявляет тип функции proc1. Это объявление необходимо, поскольку в нашей программе функция **proc1** используется (в строке 7) раньше по тексту, чем определяется (в строке 12).

В строке 7 функция **proc1** регистрируется в качестве процедуры-сервера, имеющей аргумент в виде целого числа и результат в виде строки символов. Таким образом можно зарегистрировать произвольное количество процедур-серверов с одинаковыми номерами программы и версии.

В строке 8 программа посредством обращения к функции **svc_run** переходит в состояние ожидания запросов клиентов. Нормально функционирующая программа никогда к выполнению следующих за вызовом **svc_run** операторов не переходит.

В строках 9 и 10 выводится сообщение об ошибке и завершается выполнение программы с кодом ошибки 1.

Строки 12...24 содержат описание функции **proc1.** Необходимо еще раз подчеркнуть, что функция работает с указателями на ее аргумент и результат.

Переменные *res*, *even* и *odd* объявлены статическими, чтобы они сохранялись в памяти программы и после выхода из функции **proc1**.

3.5.3. Программа-клиент

Текст программы-клиента на языке программирования СИ выглядит следующим образом

```
 1  #include <rpc/rpc.h>
 2  #include <stdio.h>
 3  #include "common.h"

 4  main (argc, argv)
 5    int argc;
 6    char **argv;
 7    {
 8    int arg;
 9    char *answer;
10    int stat;
11    if (argc < 3) exit (1);
12    arg = strlen (argv[2]);
13    if (stat = callrpc (argv[1], MY_PROG, MY_VER, MY_PROC1,
             xdr_int, &arg, xdr_wrapstring, &answer) != 0) {
14      clnt_perrno (stat);
15      exit (2);
16      };
17    printf ("Number of letters in %s is %s\n",
              argv[2], answer);
18    exit (0);
19    }
```

В строке 11 проверяется количество аргументов командной строки, с помощью которой программа-клиент вызывается на выполнение (аргументов не должно быть меньше двух).

В строке 12 переменной целого типа *arg* присваивается число, равное количеству символов второго аргумента командной строки.

В строке 13 вызывается удаленная процедура-сервер, резидентная на узле сети, имя которого задается первым аргументом командной строки. Процедуре передается указатель на ее аргумент (целое число), сама же она возвращает также указатель на свой результат (при этом память под возвращаемую строку символов нужного размера автоматически выделяется XDR-функцией **xdr_wrapstring**).

Строка 14 содержит обращение к функции **clnt_perrno**, которая выводит в *stderr* текст сообщения, характеризующего ошибку, которая по разным причинам может возникнуть при выполнении вызова удаленной процедуры.

Строка 17 выводит в *stdout* сообщение, содержащее ответ удаленной процедуры-сервера.