Направление подготовки: 13.04.02 Электроэнергетика и электротехника

Направленность (профиль): Электродвижение и электроснабжение наземных транспортных средств
Уровень образования: магистратура

Дисциплина: "Информационные и компьютерные технологии в электротехнике"

Комаров В.Г. Занятие 5 10.03.2026 г.

Использование сокетов для информационной связи элементов системы

Цель: *Освоить прикладное программирование в ОС Linux на примере передачи данных с помощью программного интерфейса Socket.*

Задание: *разработать и отладить программу передачи данных для управления транспортным средством с использованием сокета в операционной системе Linux.*

Введение

Socket API был впервые реализован в операционной системе Berkley UNIX. Сейчас этот программный интерфейс доступен практически в любой модификации Unix, в том числе в Linux. Хотя все реализации чем-то отличаются друг от друга, основной набор функций в них совпадает. Изначально сокеты использовались в программах на C/C++, но в настоящее время средства для работы с ними предоставляют многие языки (Java, Python, Scilang и др.).

Сокеты предоставляют весьма мощный и гибкий механизм межпроцессного взаимодействия (IPC). Они могут использоваться для организации взаимодействия программ на одном компьютере, по локальной сети или через Internet, что позволяет создавать распределённые приложения различной сложности. Кроме того, с их помощью можно организовать взаимодействие с программами, работающими под управлением других операционных систем.

Сокеты поддерживают многие стандартные сетевые протоколы (конкретный их список зависит от реализации) и предоставляют унифицированный интерфейс для работы с ними. Наиболее часто сокеты используются для работы в IP-сетях. В этом случае их можно использовать для взаимодействия приложений не только по специально разработанным, но и по стандартным протоколам - HTTP, FTP, Telnet и т. д. Например, можно написать собственный Web-броузер или Web-сервер, способный обслуживать одновременно множество клиентов.

Понятие сокета

Сокет (англ. socket — разъём) — название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и

на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.

Приложение просто пишет данные в сокет; их дальнейшая буферизация, отправка и транспортировка осуществляется используемым стеком протоколов и сетевой аппаратурой. Чтение данных из сокета происходит аналогичным образом.

В программе сокет идентифицируется *дескриптором* - это просто переменная типа **int**. Программа получает дескриптор от операционной системы при создании сокета, а затем передаёт его сервисам socket API для указания сокета, над которым необходимо выполнить то или иное действие.

Атрибуты сокета

С каждым сокетом связываются три атрибута: *домен*, *тип* и *протокол*. Эти атрибуты задаются при создании сокета и остаются неизменными на протяжении всего времени его существования. Для создания сокета используется функция **socket**, имеющая следующий прототип.

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

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

Домен определяет пространство адресов, в котором располагается сокет, и множество протоколов, которые используются для передачи данных. Чаще других используются домены Unix и Internet, задаваемые константами **AF_UNIX** и **AF_INET** соответственно (префикс AF означает "address family" - "семейство адресов"). При задании **AF_UNIX** для передачи данных используется файловая система ввода/вывода Unix. В этом случае сокеты используются для межпроцессного взаимодействия на одном компьютере и не годятся для работы по сети. Константа **AF_INET** соответствует Internet-домену. Сокеты, размещённые в этом домене, могут использоваться для работы в любой IP-сети. Существуют и другие домены (**AF_IPX** для протоколов Novell, **AF_INET6** для новой модификации протокола IP - IPv6 и т. д.), но здесь мы не будем их рассматривать.

Тип сокета определяет способ передачи данных по сети. Чаще других применяются:

- **SOCK_STREAM**. Передача потока данных с предварительной установкой соединения. Обеспечивается надёжный канал передачи данных, при котором фрагменты отправленного блока не теряются, не переупорядочиваются и не дублируются. Поскольку этот тип сокетов является самым распространённым, мы будем говорить только о нём.
- **SOCK_DGRAM**. Передача данных в виде отдельных сообщений (датаграмм). Предварительная установка соединения не требуется. Обмен данными происходит быстрее, но является ненадёжным: сообщения могут теряться в пути, дублироваться и переупорядочиваться. Допускается передача сообщения нескольким получателям (multicasting) и широковещательная передача (broadcasting).
- **SOCK_RAW**. Этот тип присваивается низкоуровневым (т. н. "сырым") сокетам. Их отличие от обычных сокетов состоит в том, что с их помощью программа может взять на себя формирование некоторых заголовков, добавляемых к сообщению.

Обратите внимание, что не все домены допускают задание произвольного типа сокета. Например, совместно с доменом Unix используется только тип **SOCK_STREAM**. С другой стороны, для Internet-домена можно задавать любой из перечисленных типов. В этом случае для реализации

SOCK_STREAM используется протокол TCP, для реализации **SOCK_DGRAM** - протокол UDP, а тип **SOCK_RAW** используется для низкоуровневой работы с протоколами IP, ICMP и т. д.

Наконец, последний атрибут определяет протокол, используемый для передачи данных. Как мы только что видели, часто протокол однозначно определяется по домену и типу сокета. В этом случае в качестве третьего параметра функции **socket** можно передать 0, что соответствует протоколу по умолчанию. Тем не менее, иногда (например, при работе с низкоуровневыми сокетами) требуется задать протокол явно. Числовые идентификаторы протоколов зависят от выбранного домена; их можно найти в документации.

Адреса

Прежде чем передавать данные через сокет, его необходимо связать с адресом в выбранном домене (эту процедуру называют именованием сокета). Иногда связывание осуществляется неявно (внутри функций **connect** и **accept**), но выполнять его необходимо во всех случаях. Вид адреса зависит от выбранного вами домена. В Unix-домене это текстовая строка - имя файла, через который происходит обмен данными. В Internet-домене адрес задаётся комбинацией IP-адреса и 16-битного номера порта. IP-адрес определяет хост в сети, а порт - конкретный сокет на этом хосте. Протоколы TCP и UDP используют различные пространства портов.

Для явного связывания сокета с некоторым адресом используется функция **bind**. Её прототип имеет вид:

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

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

В качестве первого параметра передаётся дескриптор сокета, который мы хотим привязать к заданному адресу. Второй параметр, **addr**, содержит указатель на структуру с адресом, а третий - длину этой структуры. Посмотрим, что она собой представляет.

```
struct sockaddr {
    unsigned short    sa_family;    // Семейство адресов, AF_xxx
    char              sa_data[14];  // 14 байтов для хранения адреса
};
```

Поле **sa_family** содержит идентификатор домена, тот же, что и первый параметр функции **socket**. В зависимости от значения этого поля по-разному интерпретируется содержимое массива **sa_data**. Разумеется, работать с этим массивом напрямую не очень удобно, поэтому вы можете использовать вместо **sockaddr** одну из альтернативных структур вида **sockaddr_XX** (XX - суффикс, обозначающий домен: "un" - Unix, "in" - Internet и т. д.). При передаче в функцию **bind** указатель на эту структуру приводится к указателю на **sockaddr**. Рассмотрим для примера структуру **sockaddr_in**.

```
struct sockaddr_in {
    short int          sin_family;  // Семейство адресов
    unsigned short int sin_port;    // Номер порта
    struct in_addr     sin_addr;    // IP-адрес
    unsigned char      sin_zero[8]; // "Дополнение" до размера структуры sockaddr
};
```

Здесь поле **sin_family** соответствует полю **sa_family** в **sockaddr**, в **sin_port** записывается номер порта, а в **sin_addr** - IP-адрес хоста. Поле **sin_addr** само является структурой, которая имеет вид:

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

Зачем понадобилось заключать всего одно поле в структуру? Дело в том, что раньше **in_addr** представляла собой объединение (union), содержащее гораздо большее число полей. Сейчас, когда в ней осталось всего одно поле, она продолжает использоваться для обратной совместимости.

И ещё одно важное замечание. Существует два порядка хранения байтов в слове и двойном слове. Один из них называется *порядком хоста* (host byte order), другой - *сетевым порядком* (network byte order) хранения байтов. При указании IP-адреса и номера порта необходимо преобразовать число из порядка хоста в сетевой. Для этого используются функции **htons** (Host TO Network Short) и **htonl** (Host TO Network Long). Обратное преобразование выполняют функции **ntohs** и **ntohl**.

ПРИМЕЧАНИЕ
На некоторых машинах (к PC это не относится) порядок хоста и сетевой порядок хранения байтов совпадают. Тем не менее, функции преобразования лучше применять и там, поскольку это улучшит переносимость программы. Это никак не скажется на производительности, так как препроцессор сам уберёт все "лишние" вызовы этих функций, оставив их только там, где преобразование действительно необходимо.

Установка соединения (сервер)

Установка соединения на стороне сервера состоит из четырёх этапов, ни один из которых не может быть опущен. Сначала сокет создаётся и привязывается к локальному адресу. Если компьютер имеет несколько сетевых интерфейсов с различными IP-адресами, вы можете принимать соединения только с одного из них, передав его адрес функции **bind**. Если же вы готовы соединяться с клиентами через любой интерфейс, задайте в качестве адреса константу **INADDR_ANY**. Что касается номера порта, вы можете задать конкретный номер или 0 (в этом случае система сама выберет произвольный неиспользуемый в данный момент номер порта).

На следующем шаге создаётся очередь запросов на соединение. При этом сокет переводится в режим ожидания запросов со стороны клиентов. Всё это выполняет функция **listen**.

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

Первый параметр - дескриптор сокета, а второй задаёт размер очереди запросов. Каждый раз, когда очередной клиент пытается соединиться с сервером, его запрос ставится в очередь, так как сервер может быть занят обработкой других запросов. Если очередь заполнена, все последующие запросы будут игнорироваться. Когда сервер готов обслужить очередной запрос, он использует функцию **accept**.

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

int accept(int sockfd, void *addr, int *addrlen);
```

Функция **accept** создаёт для общения с клиентом *новый* сокет и возвращает его дескриптор. Параметр **sockfd** задаёт слушающий сокет. После вызова он остаётся в слушающем состоянии и может

принимать другие соединения. В структуру, на которую ссылается **addr**, записывается адрес сокета клиента, который установил соединение с сервером. В переменную, адресуемую указателем **addrlen**, изначально записывается размер структуры; функция **accept** записывает туда длину, которая реально была использована. Если вас не интересует адрес клиента, вы можете просто передать NULL в качестве второго и третьего параметров.

Обратите внимание, что полученный от **accept** новый сокет связан с тем же самым адресом, что и слушающий сокет. Сначала это может показаться странным. Но дело в том, что адрес TCP-сокета не обязан быть уникальным в Internet-домене. Уникальными должны быть только *соединения*, для идентификации которых используются *два* адреса сокетов, между которыми происходит обмен данными.

Установка соединения (клиент)

На стороне клиента для установления соединения используется функция **connect**, которая имеет следующий прототип.

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

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

Здесь **sockfd** - сокет, который будет использоваться для обмена данными с сервером, **serv_addr** содержит указатель на структуру с адресом сервера, а **addrlen** - длину этой структуры. Обычно сокет не требуется предварительно привязывать к локальному адресу, так как функция **connect** сделает это за вас, подобрав подходящий свободный порт. Вы можете принудительно назначить клиентскому сокету некоторый номер порта, используя **bind** перед вызовом **connect**. Делать это следует в случае, когда сервер соединяется только с клиентами, использующими определённый порт (примерами таких серверов являются rlogind и rshd). В остальных случаях проще и надёжнее предоставить системе выбрать порт за вас.

Обмен данными

После того как соединение установлено, можно начинать обмен данными. Для этого используются функции **send** и **recv**. В Unix для работы с сокетами можно использовать также файловые функции **read** и **write**, но они обладают меньшими возможностями, а кроме того не будут работать на других платформах (например, под Windows), поэтому пользоваться ими не рекомендуется.

Функция **send** используется для отправки данных и имеет следующий прототип.

```
int send(int sockfd, const void *msg, int len, int flags);
```

Здесь **sockfd** - это, как всегда, дескриптор сокета, через который мы отправляем данные, **msg** - указатель на буфер с данными, **len** - длина буфера в байтах, а **flags** - набор битовых флагов, управляющих работой функции (если флаги не используются, передайте функции 0). Вот некоторые из них (полный список можно найти в документации):

- **MSG_OOB**. Предписывает отправить данные как *срочные* (out of band data, OOB). Концепция срочных данных позволяет иметь два параллельных канала данных в одном соединении. Иногда это бывает удобно. Например, Telnet использует срочные данные для передачи

команд типа Ctrl+C. В настоящее время использовать их не рекомендуется из-за проблем с совместимостью (существует два разных стандарта их использования, описанные в RFC793 и RFC1122). Безопаснее просто создать для срочных данных отдельное соединение.
•**MSG_DONTROUTE**. Запрещает маршрутизацию пакетов. Нижележащие транспортные слои могут проигнорировать этот флаг.

Функция **send** возвращает число байтов, которое на самом деле было отправлено (или -1 в случае ошибки). Это число может быть меньше указанного размера буфера. Если вы хотите отправить весь буфер целиком, вам придётся написать свою функцию и вызывать в ней **send**, пока все данные не будут отправлены. Она может выглядеть примерно так.

```
int sendall(int s, char *buf, int len, int flags)
{
    int total = 0;
    int n;

    while(total < len)
    {
        n = send(s, buf+total, len-total, flags);
        if(n == -1) { break; }
        total += n;
    }

    return (n==-1 ? -1 : total);
}
```

Использование **sendall** ничем не отличается от использования **send**, но она отправляет весь буфер с данными целиком.

Для чтения данных из сокета используется функция **recv**.

```
int recv(int sockfd, void *buf, int len, int flags);
```

В целом её использование аналогично **send**. Она точно так же принимает дескриптор сокета, указатель на буфер и набор флагов. Флаг **MSG_OOB** используется для приёма срочных данных, а **MSG_PEEK** позволяет "подсмотреть" данные, полученные от удалённого хоста, не удаляя их из системного буфера (это означает, что при следующем обращении к **recv** вы получите те же самые данные). Полный список флагов можно найти в документации. По аналогии с **send** функция **recv** возвращает количество прочитанных байтов, которое может быть меньше размера буфера. Вы без труда сможете написать собственную функцию **recvall**, заполняющую буфер целиком. Существует ещё один особый случай, при котором **recv** возвращает 0. Это означает, что соединение было разорвано.

Закрытие сокета

Закончив обмен данными, закройте сокет с помощью функции **close**. Это приведёт к разрыву соединения.

```
#include <unistd.h>

int close(int fd);
```

Вы также можете запретить передачу данных в каком-то одном направлении, используя **shutdown**.

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

Параметр **how** может принимать одно из следующих значений:

- 0 - запретить чтение из сокета
- 1 - запретить запись в сокет
- 2 - запретить и то и другое

Хотя после вызова **shutdown** с параметром **how**, равным 2, вы больше не сможете использовать сокет для обмена данными, вам всё равно потребуется вызвать **close**, чтобы освободить связанные с ним системные ресурсы.

Обработка ошибок

В процессе работы с сокетами могут происходить (и часто происходят) ошибки. Так вот: если что-то пошло не так, все рассмотренные нами функции возвращают -1, записывая в глобальную переменную **errno** код ошибки. Соответственно, вы можете проанализировать значение этой переменной и предпринять действия по восстановлению нормальной работы программы, не прерывая её выполнения. А можете просто выдать диагностическое сообщение (для этого удобно использовать функцию **perror**), а затем завершить программу с помощью **exit**. Именно так будем поступать в демонстрационных примерах.

Отладка программ

Возникает вопрос, как можно отлаживать сетевую программу, если под рукой нет сети. Оказывается, можно обойтись и без неё. Достаточно запустить клиента и сервер на одной машине, а затем использовать для соединения адрес *интерфейса внутренней петли* (loopback interface). В программе ему соответствует константа INADDR_LOOPBACK (не забудьте применять к ней функцию **htonl**!). Пакеты, направляемые по этому адресу, в сеть не попадают. Вместо этого они передаются стеку протоколов TCP/IP как только что принятые. Таким образом моделируется наличие виртуальной сети, в которой вы можете отлаживать ваши сетевые приложения.

Для простоты вначале будем использовать в демонстрационных примерах интерфейс внутренней петли.

Эхо-клиент и эхо-сервер

Клиент это процесс, инициирующий соединение, а сервер это процесс, ожидающий поступления запросов на подключение. Теперь, когда мы изучили основные функции для работы с сокетами, самое время посмотреть, как они используются на практике. Для этого рассмотрим две небольшие демонстрационные программы. Эхо-клиент (терминал) посылает сообщение "ELTRO start!" серверу и выводит на экран терминала ответ сервера. Его код приведён в листинге 2. Эхо-сервер читает всё, что передаёт ему клиент, а затем просто отправляет полученные данные обратно. Его код содержится в листинге 1.

Последовательность выполнения задания

Разрабатываем программы в режиме *интерфейса внутренней петли* (loopback interface). Для этого запускаем на одном из терминалов программу `"EchoServer"` (файл `ES.cpp`)

Листинг 1. Эхо-сервер.

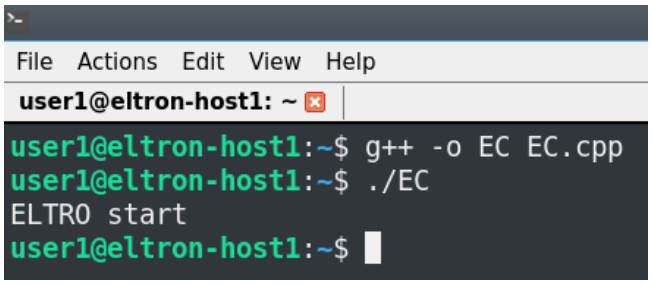
```
//Программа "EchoServer" @ES.cpp

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <cstdlib>
#include <cstddef>
#include <stdio.h>
#include <unistd.h>

int main()
{
    int sock, listener;
    struct sockaddr_in addr;
    char buf[1024];
    int bytes_read;

    listener = socket(AF_INET, SOCK_STREAM, 0);
    if(listener < 0)
    {
        perror("socket");
        exit(1);
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if(bind(listener, (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
        perror("bind");
        exit(2);
    }

    listen(listener, 1);

    while(1)
    {
        sock = accept(listener, NULL, NULL);
        if(sock < 0)
        {
            perror("accept");
            exit(3);
        }

        while(1)
        {
            bytes_read = recv(sock, buf, 1024, 0);
            if(bytes_read <= 0) break;
            send(sock, buf, bytes_read, 0);
        }

        close(sock);
    }

    return 0;
}

user1@eltron-host1:~$ g++ -o ES ES.cpp

user1@eltron-host1:~$ ./ES
```

Затем на другом терминале запускаем и отлаживаем программу "EchoClient" (файл EC.cpp)

Листинг 2. Эхо-клиент.

```
//Программа "EchoClient" @EC.cpp

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <cstdlib>
#include <cstddef>
#include <stdio.h>
#include <unistd.h>

char message[] = "ELTRO start\n";
char buf[sizeof(message)];

int main()
{
    int sock;
    struct sockaddr_in addr;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if(sock < 0)
    {
        perror("socket");
        exit(1);
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425); // или любой другой порт...
    addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
    if(connect(sock, (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
        perror("connect");
        exit(2);
    }

    send(sock, message, sizeof(message), 0);
    recv(sock, buf, sizeof(message), 0);

    printf(buf);
    close(sock);

    return 0;
}

user1@eltron-host1:~$ g++ -o EC EC.cpp

user1@eltron-host1:~$ ./EC
```

Получаем ответ:



Теперь модифицируем простую программу Эхо-сервера в программу управляемого сервера `"ControlServer"`, а Эхо-клиента в программу управляющего терминала `"ControlTerminal".` Для этого пишем тексты программ с помощью редактора nano и компилируем их. После этого запускаем их в отдельных терминалах.

Листинг 3. Управляемый-сервер.

```
//Программа "ControlServer" @CS.cpp


#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <cstdlib>
#include <cstddef>
#include <stdio.h>
#include <unistd.h>

int main()
{
    int sock, listener;
    struct sockaddr_in addr;
    char RxD[]="s\n"; //Принимаемые данные
      char TxD[] = "Сообщение принято.Процесс запущен!\n"; //Передаваемые данные
    int bytes_RxD;

    listener = socket(AF_INET, SOCK_STREAM, 0);
    if(listener < 0)
    {
        perror("socket");
        exit(1);
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if(bind(listener, (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
        perror("bind");
```

```
        exit(2);
    }

    listen(listener, 1);

    while(1)
    {
        sock = accept(listener, NULL, NULL);
        if(sock < 0)
        {
            perror("accept");
            exit(3);
        }

        while(1)
        {
            bytes_RxD = recv(sock, RxD, sizeof(RxD), 0);
            if(bytes_RxD <= 0) break;
            send(sock, TxD, sizeof(TxD), 0);
        }

      printf("ПРИНЯТЫХ БАЙТ: %d\n", sizeof(RxD));
      printf("КОД_РЕЖИМА: %s\n", RxD);
            close(sock);
    }

    return 0;
}
```

```
user1@eltron-host1:~$ g++ -o CS CS.cpp
user1@eltron-host1:~$ ./CS
```

Затем на другом терминале запускаем программу "ControlTerminal" (файл CT.cpp)

Листинг 4. Управляющий-терминал.

```
//Программа "ControlTerminal" @CT.cpp
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <cstdlib>
#include <cstddef>
#include <stdio.h>
#include <unistd.h>

char TxD[]="s\n"; //Передаваемые данные
char RxD[32];//Принимаемые данные

int main()
{
 printf("Введите сообщение:\n");
 scanf("%s", TxD);
 printf("TxD = %s\n", TxD);
 printf("Количество байт = %d\n", sizeof(TxD));
    int sock;
    struct sockaddr_in addr;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if(sock < 0)
    {
        perror("socket");
        exit(1);
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425); // или любой другой порт...
    addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
    if(connect(sock, (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
        perror("connect");
        exit(2);
    }

    send(sock, TxD, sizeof(TxD), 0);
    recv(sock, RxD, sizeof(RxD), 0);
```

```
      printf("ПРИНЯТЫХ БАЙТ: %d\n", sizeof(RxD));
      printf("КВИТАНЦИЯ: %s\n", RxD);
    close(sock);

    return 0;
}
user1@eltron-host1:~$ g++ -o CT CT.cpp
user1@eltron-host1:~$ ./TT
Введите сообщение:
```

Вводим сообщение

```
Start ELTRO
TxD = s
Количество байт = 3
```

и получаем ответ (квитанцию) от сервера

```
ПРИНЯТЫХ БАЙТ: 32
КВИТАНЦИЯ: Сообщение получено.Процесс запущен!
```

На терминале сервера можем увидеть код установленного режима

```
ПРИНЯТЫХ БАЙТ: 3
КОД_РЕЖИМА: s
```