Теоретические сведения о построении клиент-серверной архитектуры

Общие сведения

Сокеты (англ. socket — углубление, гнездо, 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.

Интерфейс сокетов впервые появился в BSD Unix. Программный интерфейс сокетов описан в стандарте POSIX.1 и в той или иной мере поддерживается всеми современными операционными системами.

Следует различать клиентские и серверные сокеты. Сокеты подобны почтовым ящикам и телефонным розеткам в том смысле, что они образуют пользовательский интерфейс с сетью, как почтовые ящики формируют интерфейс с почтовой системой, а телефонные розетки позволяют абоненту подключить телефон и соединиться с телефонной системой. Схематично расположение сокетов показано на рис. 1.

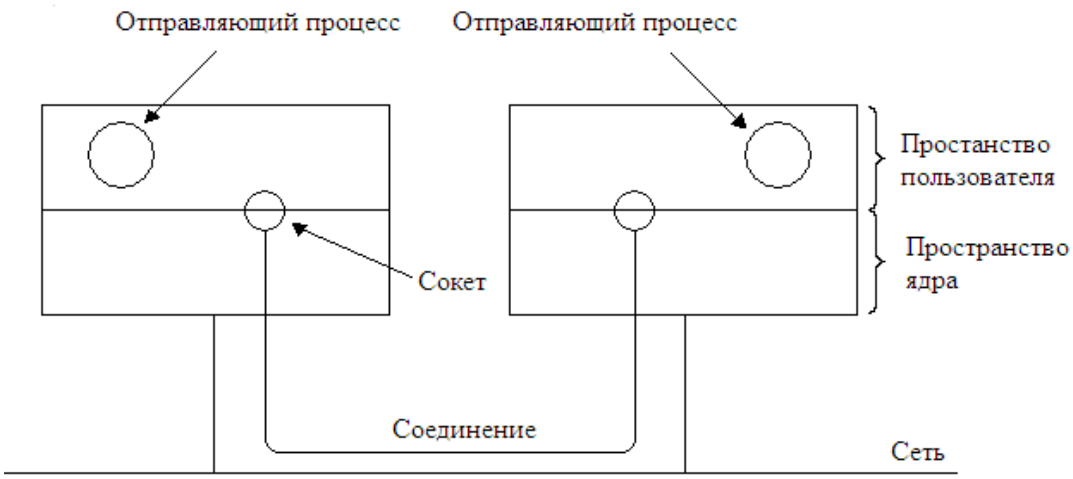

Рис. 1.

Сокеты могут динамически создаваться и уничтожаться. При создании сокета вызывающему процессу возвращается дескриптор файла, который используется для установки или разрыва соединения, а также чтения и записи данных.

Каждый сокет поддерживает определенный тип работы в сети, указываемый при его создании. Наиболее распространенными типами сокетов являются:

- Надежный, ориентированный на соединение байтовый поток.
- Надежный, ориентированный на соединение поток пакетов.
- Ненадежная передача пакетов.

Первый тип сокетов позволяет двум процессам на различных машинах установить между собой эквивалент «трубы» (канала между процессами на одной машине). Байты подаются в канал с одного конца и в том же порядке выходят с другого. Такая система гарантирует, что все посланные байты прибудут на другой конец канала и прибудут именно в том порядке, в котором были отправлены.

Второй тип сокетов отличается от первого тем, что он сохраняет границы между пакетами. Если отправитель пять раз отдельно обращается к системному вызову `write`, каждый раз отправляя по 512 байт, а получатель запрашивает 2560 байт по сокету типа 1, он получит все 2560 байт сразу. При использовании сокета типа 2 ему будут выданы только первые 512 байт. Чтобы получить остальные байты, необходимо повторно прочитать данные из сокета с помощью функции `read` еще четыре раза. Третий тип сокета предоставляет пользователю доступ к «голой» сети. Этот тип сокета особенно полезен для приложений реального времени и ситуаций, в которых пользователь хочет реализовать специальную схему обработки ошибок. Сеть может терять пакеты или доставлять их в неверном порядке. В отличие от сокетов первых двух типов, сокет типа 3 не предоставляет никаких гарантий доставки. Преимущество этого режима заключается в более высокой производительности, которая в некоторых ситуациях оказывается важнее надежности (например, для доставки мультимедиа, при которой скорость ценится существенно выше, нежели сохранность данных по дороге).

При создании сокета один из параметров указывает протокол, используемый для него. Для надежных байтовых потоков, как правило, используется протокол TCP (Transmission Control Protocol – протокол управления передачей). Для ненадежной передачи пакетов обычно применяется протокол UDP (User Data Protocol – пользовательский протокол данных). Для надежного потока пакетов специального протокола нет.

Прежде чем сокет может быть использован для работы в сети, с ним должен быть связан адрес. Этот адрес может принадлежать к одному из нескольких пространств адресов. Наиболее распространенным пространством является пространство адресов Интернета, использующее 32-разарядные числа для идентификации конечных адресатов в протоколе IPv4 и 128-разрядные числа в протоколе IPv6 (5-я версия протокола IP была экспериментальной системой, так и не выпущенной в свет).

Как только сокеты созданы на компьютере-источнике и компьютере-приемнике, между ними может быть установлено соединение (для ориентированной на соединение связи). Одна сторона обращается к системному вызову `listen`, указывая в качестве параметра локальный сокет. При этом системный вызов создает буфер и блокируется до тех пор, пока не прибудут данные. Другая сторона обращается к системному вызову `connect`, задавая в параметрах дескриптор файла для локального сокета и адрес удаленного сокета. Если удаленный компьютер принимает вызов, то система устанавливает соединение между двумя сокетами.

Функции установленного соединения аналогичны функциям канала. Процесс может читать из канала и писать в него, используя дескриптор файла для локального сокета. Когда соединение более не нужно, оно может быть закрыто обычным способом, при помощи системного вызова `close`.

Работа с сокетами в ОС Windows

О клиентах и серверах

Существует два типа сетевых приложений, использующих сокеты: сервер и клиент. У клиентов и серверов различное поведение, соответственно и процесс создания их также различен.

Модель создания сервера:

- Инициализация WinSock Application
- Создание сокета

- Привязывание сокета
- Прослушивание сокета
- Принятие запроса на установление соединения
- Отправка и получение данных
- Отключение

Модель создания клиента:

- Инициализация WinSock Application
- Создание сокета
- Соединение с сервером
- Отправка и получение данных
- Отключение

При написании приложения необходимо использовать WinSock API – интерфейс прикладного программирования для Internet. Он служит для связи между прикладными программами клиента Windows и стеком протоколов TCP/IP. В примерах ниже также будет использоваться заголовочный файл `stdio.h` для стандартного ввода/вывода. Таким образом, шаблон WinSock Application выглядит так:

```
#include <stdio.h>
#include "winsock2.h"

void main()
{
	return;
}
```

Инициализация WinSock

Чтобы удостовериться, что все сокеты Windows поддерживаются системой, приложения WinSock должны быть проинициализированы. Следующая последовательность действия осуществуляет инициализацию WinSock:

- Создание объекта WSDATA.
- Вызов функции WSAStartup, возврат значения типа int
- Проверка ошибок.

Пример:

```
WSADATA wsaData;
int iResult = WSAStartup( MAKEWORD(2,2), &wsaData );
if ( iResult != NO_ERROR )
    printf("Error at WSAStartup()\n");
```

Структура WSADATA содержит информацию о реализации сокетов Windows. Функция WSAStartup вызывается для того, чтобы использовать библиотеку WS2_32.lib. Параметр MAKEWORD(2,2) данной функции запрашивает версию WinSock системы и устанавливает ее как наивысшую допустимую версию сокетов Windows, которая может использоваться.

Создание сокета

После инициализации создается сам сокет. Происходит это следующим образом:

- Создание объекта типа `SOCKET`.
- Вызов функции socket и запись результата ее выполнения в созданную переменную типа `SOCKET`. В качестве параметров используются семейство интернет-адресов (IP), потоковые сокеты и протокол TCP/IP.
- Проверка ошибок, чтобы удостовериться, что сокет действителен.

Пример:

```
SOCKET m_socket;
m_socket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
if ( m_socket == INVALID_SOCKET )
{
    printf( "Error at socket(): %ld\n", WSAGetLastError() );
    WSACleanup();
    return;
}
```

Параметры функции `socket` могут быть изменены для различных реализаций. Выявление ошибок - ключевая часть сетевого программирования. Если вызов функции socket заканчивается неудачей, то она возвращает `INVALID_SOCKET`. Условный оператор `if` в примере выше используется для отслеживания всех ошибок, которые могут возникнуть в процессе создания сокета. `WSAGetLastError` возвращает номер последней возникнувшей ошибки. Иногда может понадобиться больше проверок ошибок.

Привязывание сокета

Для того чтобы сервер мог принять подключение клиента, сетевой адрес должен быть связан с системой. Клиентский модуль использует IP адрес и порт для подключения к сети. Связывание сокета происходит следующим образом:

- Создание объекта `sockaddr_in`.
- Вызов функции `bind` с параметрами: созданный сокет `socket` и созданная структура типа `sockaddr_in`.
- Проверка ошибок.

Пример создания сокета:

```
sockaddr_in service;
service.sin_family = AF_INET;
service.sin_addr.s_addr = inet_addr( "127.0.0.1" );
service.sin_port = htons( 27015 );
if ( bind( m_socket, (SOCKADDR*) &service, sizeof(service) ) == SOCKET_ERROR
)
{
    printf( "bind() failed.\n" );
    closesocket(m_socket);
    return;
}
```

Структура `sockaddr_in` содержит информация о семействе адресов, IP адресе и номере порта. `sockaddr_in` – это подмножество sockaddr, используется для протокола IP версии 4. Например, при объявлении `sockaddr_in` могут использоваться следующие значения:

- AF_INET – семейство адресов Интернет;
- 127.0.0.1 – локальный IP адрес, с которым сокет будет связан;
- 27015 – номер порта, с которым будет связан сокет.

Прослушивание сокета

После создания и связывания сокета с IP адресом и портом системы, сервер прослушивает эти IP адрес и порт на предмет входящих запросов. Прослушивание осуществляется по следующему алгоритму:

1. Вызов функции `listen` с параметрами созданного сокета и максимальным разрешенным числом подключений
2. Проверка ошибок

Пример:

```
if ( listen( m_socket, 1 ) == SOCKET_ERROR )
    printf( "Error listening on socket.\n");
```

Принятие запроса на установление соединения

Поскольку сокет прослушивается для входящего соединения, то программа должна обрабатывать запросы на установку соединения по этому сокету. Установка соединения может состоять из следующей последовательности действий:

1. Создание временного объекта типа `SOCKET`.
2. Создание бесконечного цикла для ожидания запросов на соединение.
3. При появлении запроса – вызов функции `accept` для обработки запроса.
4. Когда подключение клиента было принято, управление передается от временного сокета оригинальному и завершается проверка новых соединений.

Пример:

```
SOCKET AcceptSocket;
printf( "Waiting for a client to connect...\n" );
while (1)
{
    AcceptSocket = SOCKET_ERROR;
    while ( AcceptSocket == SOCKET_ERROR )
    {
        AcceptSocket = accept( m_socket, NULL, NULL );
    }
    printf( "Client Connected.\n");
    m_socket = AcceptSocket;
    break;
}
```

Отправка и получение данных

Для отправки и получения данных используются функции `send` и `recv`. Рассмотрим пример серверной и клиентской частей приложения.

Пример (серверная часть):

```
int bytesSent;
int bytesRecv = SOCKET_ERROR;
char sendbuf[32] = "Server: Sending Data.";
char recvbuf[32] = "";

bytesRecv = recv( m_socket, recvbuf, 32, 0 );
printf( "Bytes Recv: %ld\n", bytesRecv );

bytesSent = send( m_socket, sendbuf, strlen(sendbuf), 0 );
printf( "Bytes Sent: %ld\n", bytesSent );
```

Пример (клиентская часть):

```
int bytesSent;
int bytesRecv = SOCKET_ERROR;
char sendbuf[32] = "Client: Sending data.";
char recvbuf[32] = "";

bytesSent = send( m_socket, sendbuf, strlen(sendbuf), 0 );
printf( "Bytes Sent: %ld\n", bytesSent );

while( bytesRecv == SOCKET_ERROR )
{
    bytesRecv = recv( m_socket, recvbuf, 32, 0 );
    if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
    {
        printf( "Connection Closed.\n");
        break;
    }
    if (bytesRecv < 0)
        return;
    printf( "Bytes Recv: %ld\n", bytesRecv );
}
```

Фукнции `send` и `recv` возвращают значения типа `int`, отображающих информацию о количестве байтов (отправленных и полученных). В качестве параметров функции используют: активный сокет, буфер (типа `char`), количество байт (отправляемых или полученных), а также различные флаги.

Окончательный клиентский модуль

```
#include <stdio.h>
#include "winsock2.h"

void main()
{
    // Initialize Winsock.
    WSADATA wsaData;
    int iResult = WSAStartup( MAKEWORD(2,2), &wsaData );
    if ( iResult != NO_ERROR )
        printf("Error at WSAStartup()\n");

    // Create a socket.
    SOCKET m_socket;
    m_socket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );

    if ( m_socket == INVALID_SOCKET )
    {
        printf( "Error at socket(): %ld\n", WSAGetLastError() );
        WSACleanup();
        return;
    }

    // Connect to a server.
    sockaddr_in clientService;

    clientService.sin_family = AF_INET;
    clientService.sin_addr.s_addr = inet_addr( "127.0.0.1" );
    clientService.sin_port = htons( 27015 );

    if ( connect( m_socket, (SOCKADDR*) &clientService, sizeof(clientService)
) == SOCKET_ERROR)
    {
        printf( "Failed to connect.\n" );
        WSACleanup();
        return;
    }

    // Send and receive data.
    int bytesSent;
    int bytesRecv = SOCKET_ERROR;
    char sendbuf[32] = "Client: Sending data.";
    char recvbuf[32] = "";

    bytesSent = send( m_socket, sendbuf, strlen(sendbuf), 0 );
    printf( "Bytes Sent: %ld\n", bytesSent );

    while( bytesRecv == SOCKET_ERROR )
    {
        bytesRecv = recv( m_socket, recvbuf, 32, 0 );
        if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
        {
            printf( "Connection Closed.\n");
```

```
            break;
        }
        if (bytesRecv < 0)
            return;
        printf( "Bytes Recv: %ld\n", bytesRecv );
    }
    return;
}
```

Окончательный серверный модуль

```
#include <stdio.h>
#include "winsock2.h"

void main()
{
    // Initialize Winsock.
    WSADATA wsaData;
    int iResult = WSAStartup( MAKEWORD(2,2), &wsaData );
    if ( iResult != NO_ERROR )
        printf("Error at WSAStartup()\n");

    // Create a socket.
    SOCKET m_socket;
    m_socket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );

    if ( m_socket == INVALID_SOCKET )
    {
        printf( "Error at socket(): %ld\n", WSAGetLastError() );
        WSACleanup();
        return;
    }

    // Bind the socket.
    sockaddr_in service;
    service.sin_family = AF_INET;
    service.sin_addr.s_addr = inet_addr( "127.0.0.1" );
    service.sin_port = htons( 27015 );

    if ( bind( m_socket, (SOCKADDR*) &service, sizeof(service) ) ==
SOCKET_ERROR )
    {
        printf( "bind() failed.\n" );
        closesocket(m_socket);
        return;
    }

    // Listen on the socket.
    if ( listen( m_socket, 1 ) == SOCKET_ERROR )
        printf( "Error listening on socket.\n");

    // Accept connections.
    SOCKET AcceptSocket;

    printf( "Waiting for a client to connect...\n" );
```

```
    while (1)
    {
        AcceptSocket = SOCKET_ERROR;
        while ( AcceptSocket == SOCKET_ERROR )
        {
            AcceptSocket = accept( m_socket, NULL, NULL );
        }
        printf( "Client Connected.\n");
        m_socket = AcceptSocket;
        break;
    }

    // Send and receive data.
    int bytesSent;
    int bytesRecv = SOCKET_ERROR;
    char sendbuf[32] = "Server: Sending Data.";
    char recvbuf[32] = "";

    bytesRecv = recv( m_socket, recvbuf, 32, 0 );
    printf( "Bytes Recv: %ld\n", bytesRecv );

    bytesSent = send( m_socket, sendbuf, strlen(sendbuf), 0 );
    printf( "Bytes Sent: %ld\n", bytesSent );

    return;
}
```

Средства POSIX для работы с сокетами

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 в рамках данного курса 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, до конца раздела мы будем говорить только о нём. Остальным типам будут посвящены отдельные разделы.
- 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 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 можно 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, через сокет можно будет принимать соединения только с одного из них, передав его адрес функции `bind`. Если же необходимо соединяться с клиентами через любой интерфейс, в качестве адреса следует указать константу `INADDR_ANY`. Что касается номера порта, можно 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 информация об адресе клиента не нужна, в качестве второго и третьего параметров можно просто передать NULL.

Следует 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 сама, подобрав подходящий свободный порт. Можно 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 подходящий порт самостоятельно.

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).

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 происходит обмен данными, `msg` - указатель на буфер с данными, `len` - длина буфера в байтах, а `flags` - набор битовых флагов, управляющих работой функции (если флаги не используются, в качестве значения этого параметра функции следует передать 0). Вот некоторые из этих флагов (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. Для того чтобы отправить весь буфер целиком необходимо написать свою функцию и многократно вызывать в ней `send` до тех пор, 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` она вернет 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. Точно так же без труда можете быть написана собственная функцию `recvall`, заполняющую буфер целиком. Существует ещё один особый случай, при котором `recv` возвращает 0. Это означает, что соединение было разорвано.

Закрытие сокета

После завершения обмена данным, сокет должен быть закрыт `close`. Это приведёт к разрыву соединения.

```
#include <unistd.h>

int close(int fd);
```

Также можно 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, использовать сокет для обмена данными будет невозможно, в дальнейшем 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. Если что-то пошло не так, все рассмотренные выше функции возвращают -1, записывая в глобальную переменную `errno` код ошибки. Соответственно, можно проанализировать значение этой переменной и предпринять действия по восстановлению нормальной работы программы, не прерывая её выполнения. А можно просто выдать диагностическое сообщение (для этого удобно использовать функцию `perror`), а затем завершить программу с помощью `exit`. Именно такой подход используется в демонстрационных примерах.

Отладка программ

Часто возникает вопрос, как можно отлаживать сетевую программу, если под рукой нет сети. Оказывается, можно обойтись и без неё. Достаточно запустить клиентское и серверное приложения на одной машине, а затем использовать для соединения адрес *интерфейса внутренней петли* (loopback interface). В программе ему соответствует константа `INADDR_LOOPBACK` (к ней также следует применять 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 можно отлаживать сетевые приложения.

Для простоты в демонстрационных примерах используется интерфейс внутренней петли.

Эхо-клиент и эхо-сервер

Рассмотрим использование основных функций для работы с сокетами на примере двух небольших демонстрационных программ. Эхо-клиент посылает сообщение "Hello there!" и выводит на экран ответ сервера. Его код приведён в листинге 1. Эхо-сервер читает всё, что передаёт ему клиент, а затем просто отправляет полученные данные обратно. Его код содержится в листинге 2.

Листинг 1. Эхо-клиент.

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

char message[] = "Hello there!\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;
}
```

Листинг 2. Эхо-сервер.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.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;
}
```

Обмен датаграммами (UDP)

Как уже говорилось, датаграммы используются в программах довольно редко. В большинстве случаев надёжность передачи критична для приложения, и вместо изобретения собственного надёжного протокола поверх UDP программисты предпочитают использовать TCP. Тем не менее, иногда датаграммы оказываются полезны. Например, их удобно использовать при транслировании звука или видео по сети в реальном времени, особенно при широковещательном транслировании.

Поскольку для обмена датаграммами не нужно устанавливать соединение, использовать их гораздо проще. Создав сокет с помощью `socket` и `bind`, можно тут же использовать его для отправки или получения данных. Для этого понадобятся функции `sendto` и `recvfrom`.

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
               const struct sockaddr *to, int tolen);
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
                 struct sockaddr *from, int *fromlen);
```

Функция `sendto` очень похожа на `send`. Два дополнительных параметра `to` и `tolen` используются для указания адреса получателя. Для задания адреса используется структура `sockaddr`, как и в случае с функцией `connect`. Функция `recvfrom` работает аналогично `recv`. Получив очередное сообщение, она записывает его адрес в структуру, на которую ссылается `from`, а записанное количество байт - в переменную, адресуемую указателем `fromlen`. Как мы знаем, аналогичным образом работает функция `accept`.

Некоторую путаницу вносят *присоединённые датаграммные сокеты* (connected datagram sockets). Дело в том, что для сокета с типом `SOCK_DGRAM` тоже можно вызвать функцию `connect`, а затем использовать `send` и `recv` для обмена данными. Нужно понимать, что никакого соединения при этом не устанавливается. Операционная система просто запоминает адрес, переданный функции `connect`, а затем использует его при отправке данных. Следует обратить внимание, что присоединённый сокет может получать данные только от сокета, с которым он соединён.

Для иллюстрации процесса обмена датаграммами рассмотрим две небольшие программы - sender (листинг 3) и receiver (листинг 4). Первая отправляет сообщения "Hello there!" и "Bye bye!", а вторая получает их и печатает на экране. Программа sender демонстрирует применение как обычного, так и присоединённого сокета, а receiver использует обычный.

Листинг 3. Программа sender.

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

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

char msg1[] = "Hello there!\n";
char msg2[] = "Bye bye!\n";

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

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

    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425);
    addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
    sendto(sock, msg1, sizeof(msg1), 0,
           (struct sockaddr *)&addr, sizeof(addr));

    connect(sock, (struct sockaddr *)&addr, sizeof(addr));
    send(sock, msg2, sizeof(msg2), 0);

    close(sock);

    return 0;
}
```

Листинг 4. Программа receiver.

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

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

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if(sock < 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(sock, (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
        perror("bind");
        exit(2);
    }

    while(1)
    {
        bytes_read = recvfrom(sock, buf, 1024, 0, NULL, NULL);
        buf[bytes_read] = '\0';
        printf(buf);
    }

    return 0;
}
```