Socket_definition

Сокет

абстрактная структура данных, используемая
для создания точки присоединения канала обмена данными
между процессами.

Сокеты предоставляют собой весьма мощный и гибкий механизм межпроцессного взаимодействия. Они могут использоваться для организации взаимодействия программ на одном компьютере, по локальной сети или через Internet, что позволяет создавать эффективные распределённые приложения.

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

Socket_operating systems interoperability

Сокеты - весьма мощное и удобное средство для **сетевого программирования**.

Наиболее часто используются для работы в IP-сетях.

Поддерживают многие стандартные сетевые протоколы, кроме того, можно использовать для взаимодействия приложений по специально разработанным протоколам. Т.е. сокеты предоставляют унифицированный интерфейс для работы с различными протоколами.

С помощью сокетов можно организовать взаимодействие с приложениями, работающими под **управлением других операционных систем**.
Например, под Windows существует интерфейс Window Sockets, спроектированный на основе socket API. Существующую Linux-программу можно легко адаптировать для работы под Windows.

Socket_attributes

В программе сокет идентифицируется **дескриптором** - это 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.

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 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 (**AF_UNIX** и **AF_INET**). Префикс AF означает "address family" - "семейство адресов". Существуют и другие домены: **AF_IPX** для протоколов Novell, **AF_INET6** для модификации протокола IP - IPv6 и т. д.

Socket_attributes

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. Этот тип сокетов является наиболее распространённым.

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 (так называемым "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.

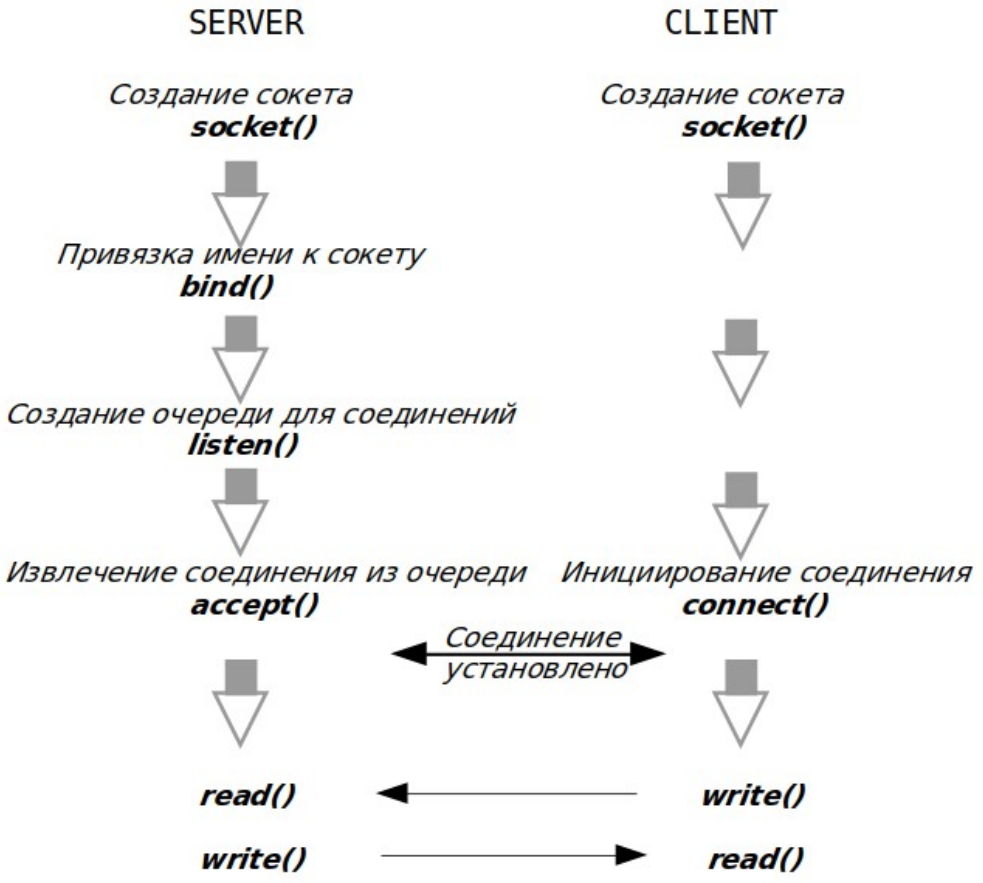
Socket_attributes

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.

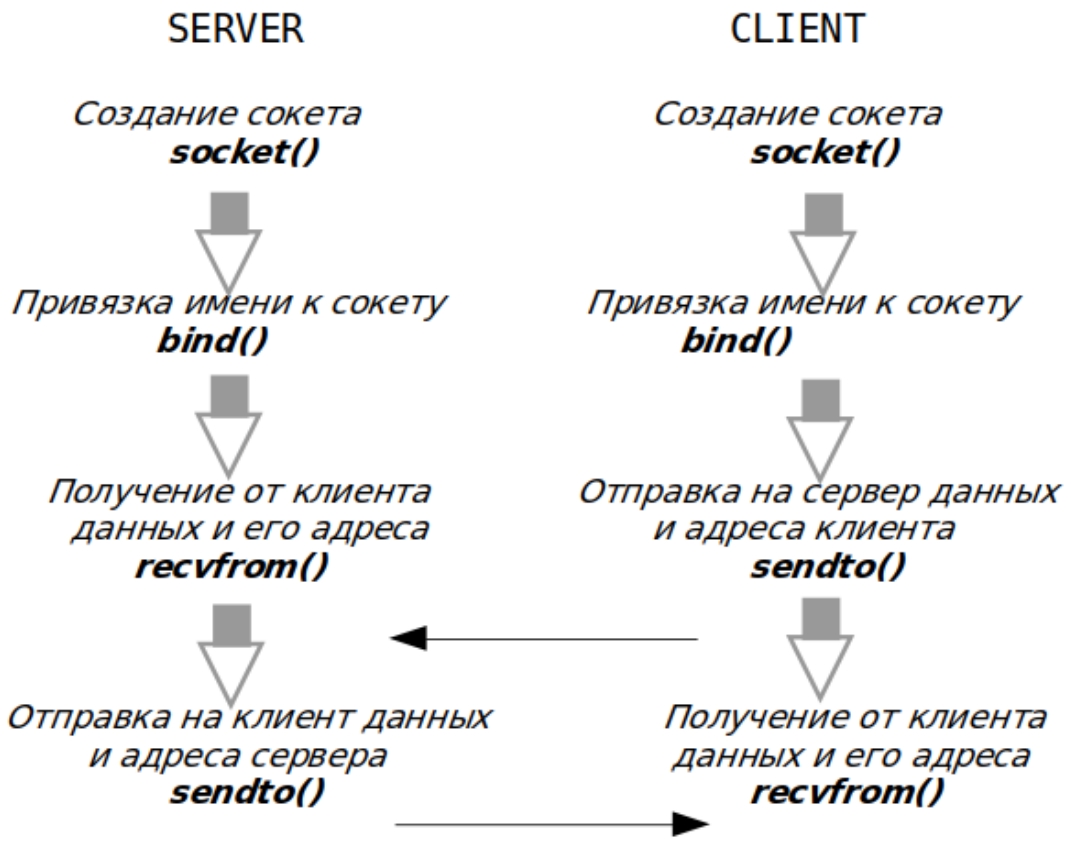
Третий атрибут системного вызова **socket**() определяет **протокол**, используемый для передачи данных. Зачастую протокол однозначно определяется по домену и типу сокета. В этом случае в качестве третьего параметра функции **socket**() можно передавать 0, что соответствует протоколу по умолчанию (on default).

Тем не менее, иногда, например, при работе с низкоуровневыми сокетами SOCK_RAW, требуется задать протокол явно. Числовые идентификаторы протоколов зависят от выбранного домена; их можно найти в справочной документации.

Sockets_stream system calls sequence



Sockets_datagram system calls sequence



Sockets_bind()

Прежде чем передавать данные через сокет, его необходимо связать с адресом в выбранном домене (эту процедуру еще называют именованием сокета).

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.

Прототип функции **bind**() для явного связывания сокета с некоторым адресом:

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

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

Первый параметр - дескриптор сокета, который мы хотим привязать к заданному адресу.
Второй параметр, addr, содержит указатель на структуру с адресом сокета.
Третий параметр - длина этой структуры.

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

Sockets_addressing

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_in (Internet домен), sockaddr_un (Unix домен и тд.). При передаче в функцию **bind**() указатель на эту структуру приводится к указателю на sockaddr. Например, структура **sockaddr_in** имеет вид:

```
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:

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

Sockets_host/network byte order

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**.

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**.

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 сказывается на производительности, так как при компиляции препроцессор сам уберёт все "лишние" вызовы этих функций, оставив их только там, где преобразование, действительно, необходимо.

Sockets_listen()

Установка соединения на стороне сервера состоит из четырёх этапов.

Сначала сокет создаётся (вызов **socket**()) и привязывается к локальному адресу (вызов **bind**()).

Если компьютер имеет несколько сетевых интерфейсов с различными IP-адресами, то можно принимать соединения только с одного из них, передав его адрес функции **bind**(). Если же приложение готово соединяться с клиентами через любой интерфейс, то в качестве адреса задается константа INADDR_ANY. В качестве номера порта, можно задать конкретный номер либо 0 (в этом случае ОС сама выберет 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.

Sockets_accept()

Когда сервер готов обслужить очередной поступивший 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 (если адрес клиента, не интересует можно просто передать NULL в качестве второго и третьего параметров). В переменную, адресуемую указателем addrlen, изначально записывается размер структуры; функция **accept**() записывает туда длину, которая реально была использована. 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.

Sockets_client side connect()

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);
```

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 на клиенте не требуется предварительно привязывать к локальному адресу, так как функция **connect**() это сделает сама, подобрав подходящий свободный порт. Можно принудительно назначить клиентскому сокету некоторый номер порта, используя **bind**() перед вызовом **connect**(). Это нужно в случае, когда сервер соединяется только с клиентами, использующими определённый порт (например, rlogind и rshd). В остальных случаях проще и надёжнее предоставить выбор порта системе.

После того как соединение установлено, можно начинать обмен данными. Для этого используются функции **send**() и **recv**(). В Linux 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 не рекомендуется ими пользоваться.

Sockets_send()

Функция **send**() используется для отправки данных и имеет следующий прототип:

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

sockfd - дескриптор сокета, через который отправляются данные,
msg - указатель на буфер с данными,
len - длина буфера в байтах,
flags - набор битовых флагов, управляющих работой функции
(если флаги не используются, функции передается 0).
Некоторые из 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**. 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 необходимо отправить сразу весь **буфер целиком**, составляется **собственная функция**.

Sockets_sendall()

```
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);
}
```

Sockets_recv()

Для чтения данных из сокета используется функция:

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

В целом её синтаксис и использование аналогично рассмотренной **send**().
Функция **recv**( ) точно так же принимает дескриптор сокета,
указатель на буфер и набор флагов.

Флаг **MSG_OOB** используется для приёма срочных данных,
Флаг **MSG_PEEK** позволяет "подсмотреть" данные, полученные от удалённого хоста, не извлекая их из системного буфера (это означает, что при следующем обращении к **recv**() будут получены те же самые данные).
Полный список флагов можно найти в справочной документации.

По аналогии с **send**() функция **recv**() возвращает количество прочитанных байтов, которое может быть меньше размера буфера. Можно также написать собственную функцию **recvall**(), заполняющую буфер целиком.
Существует ещё один **особый случай**, при котором **recv**() возвращает 0.
Это означает, что соединение было **разорвано**.

Sockets_close()

Закончив обмен данными, сокет закрывается с помощью функции **close**().

Это приводит к разрыву соединения.

```
#include <unistd.h>

int close(int fd);
```

Если требуется из двунаправленного канала (full-duplex) сделать однонаправленный (half-duplex) используется функция **shutdown**().

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

Параметр how может принимать одно из следующих значений:

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

После вызова **shutdown**() с параметром how, равным 2, сокет больше не используется для обмена данными, но чтобы освободить связанные с ним системные ресурсы, всё равно потребуется вызвать **close**().

Sockets_debugging

Как можно отлаживать сетевую программу, если под рукой нет сети.
Можно обойтись и без неё.
Достаточно запустить приложение-клиент и приложение-сервер
на одной машине, а затем использовать для соединения адрес
интерфейса внутренней петли (**loopback interface**).
В программе интерфейсу внутренней петли соответствует
константа **INADDR_LOOPBACK** (к ней применяется функция htonl()).
Пакеты, направляемые по этому адресу, в сеть не попадают.
Вместо этого они передаются **стеку протоколов** TCP/IP
как будто они только что принятые.
Таким образом моделируется наличие виртуальной сети, в которой можно
отлаживать сетевые приложения.

Sample AF_INET , SOCK_STREAM Clnt

```c
#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);
    }
```

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

    printf(buf);
    close(sock);

    return 0;
}
```

Sample AF_INET , SOCK_STREAM Srv

```c
#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;
}
```

Sockets_client’s service concurency

Параллельное обслуживание клиентов становится актуальным, когда сервер должен

обслуживать большое количество запросов.

Даже на одной машине с одним процессором можно добиться существенного выигрыша в производительности. Допустим, сервер отправил какие-то данные клиенту и ждёт подтверждения. Пока оно путешествует по сети, сервер вполне мог бы заняться другими клиентами. Для реализации такого алгоритма обслуживания чаще всего применяется способ с **fork**()

Этот способ подразумевает создание дочернего процесса для обслуживания каждого нового клиента. При этом родительский процесс занимается только прослушиванием порта и приёмом соединений. Чтобы добиться такого поведения, сразу после **accept**() сервер вызывает функцию **fork**() для создания дочернего процесса. Далее анализируется значение, которое вернула эта функция. В родительском процессе оно содержит идентификатор дочернего, а в дочернем процессе равно нулю. Используя этот признак, мы переходим к очередному вызову **accept**() в родительском процессе, а дочерний процесс обслуживает клиентский процесс и завершается _exit().

Такой способ неявно подразумевает, что все клиенты обслуживаются независимо друг от друга.
В случае, если клиентов очень много, создание нового процесса для обслуживания каждого из них может оказаться слишком дорогостоящей операцией, поглощающей ресурсы сервера.

Преимущество такого подхода состоит в том, что он позволяет писать весьма компактные, понятные программы, в которых код установки соединения отделён от кода обслуживания клиента.

Sockets_concurency server

```c
#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 = INADDR_ANY;
  if(bind(listener, (struct sockaddr *)&addr, sizeof(addr)) < 0)
  {
    perror("bind");
    exit(2);
  }

  listen(listener, 1);
```

Sockets_concurency server

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

    switch(fork())
    {
    case -1:
      perror("fork");
      break;

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

      close(sock);
      _exit(0);

    default:
      close(sock);
    }
  }
    close(listener);

  return 0;
}
```

Sockets_datagram exchange

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**().

Sockets_sendto() 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 известно, аналогичным образом работает функция **accept**().

Sockets_datagram connected

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. Естественно, *никакого соединения* при этом не устанавливается. Операционная система просто запоминает адрес, который передан функции **connect**(), а затем использует его при отправке данных.

Характерно, что присоединённый сокет может получать данные только от сокета, с которым он был соединён.

Программа s*ender* демонстрирует применение как обычного, так и присоединённого сокета и отправляет два сообщения. Программа *receiver,* используя обычный сокет, получает сообщения и печатает их на экране.

Sockets_datagram sender

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

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

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

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

Sockets_datagram sender

```
    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;
}
```

Sockets_datagram receiver

```
#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);
  }
```

Sockets_datagram receiver

```
    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;
}
```

Thanks for your attention

Спасибо за внимание !

vladimir.shmakov.2012@gmail.com