

Сокеты

Технология коммуникации процессов с помощью сокетов позволяет организовать обмен данными, как между процессами, исполняющимися на одном компьютере, так и между процессами, запущенными на разных компьютерах, работающих в сети. Сокет можно рассматривать, как абстрактную структуру данных, используемую процессами для создания точки присоединения канала обмена данными между процессами. При этом, после создания канала обмен данными происходит в стиле общепринятой парадигмы чтения-записи.

По типу организации канала различают сокет, ориентированный на соединение (*stream sockets*), и сокет не подразумевающий наличия постоянного соединения (*datagram sockets*). По умолчанию, первая разновидность сокетов предполагает интерфейс к транспортному уровню, реализованному на TCP (*Transmission Control Protocol*) протоколе, а вторая - на протоколе UDP (*User Datagram Protocol*).

Интерфейс представляет собой совокупность системных вызовов, служащих для подсоединения соответствующего канала обмена, для проведения затем операций типа чтения-записи в канал, а также для отсоединения от канала.

Естественно, сокет определенного типа могут взаимодействовать только с сокетами такого же типа.

Протоколы TCP и UDP (транспортный уровень) опираются на протокол IP, относящийся к сетевому уровню стека протоколов TCP/IP.

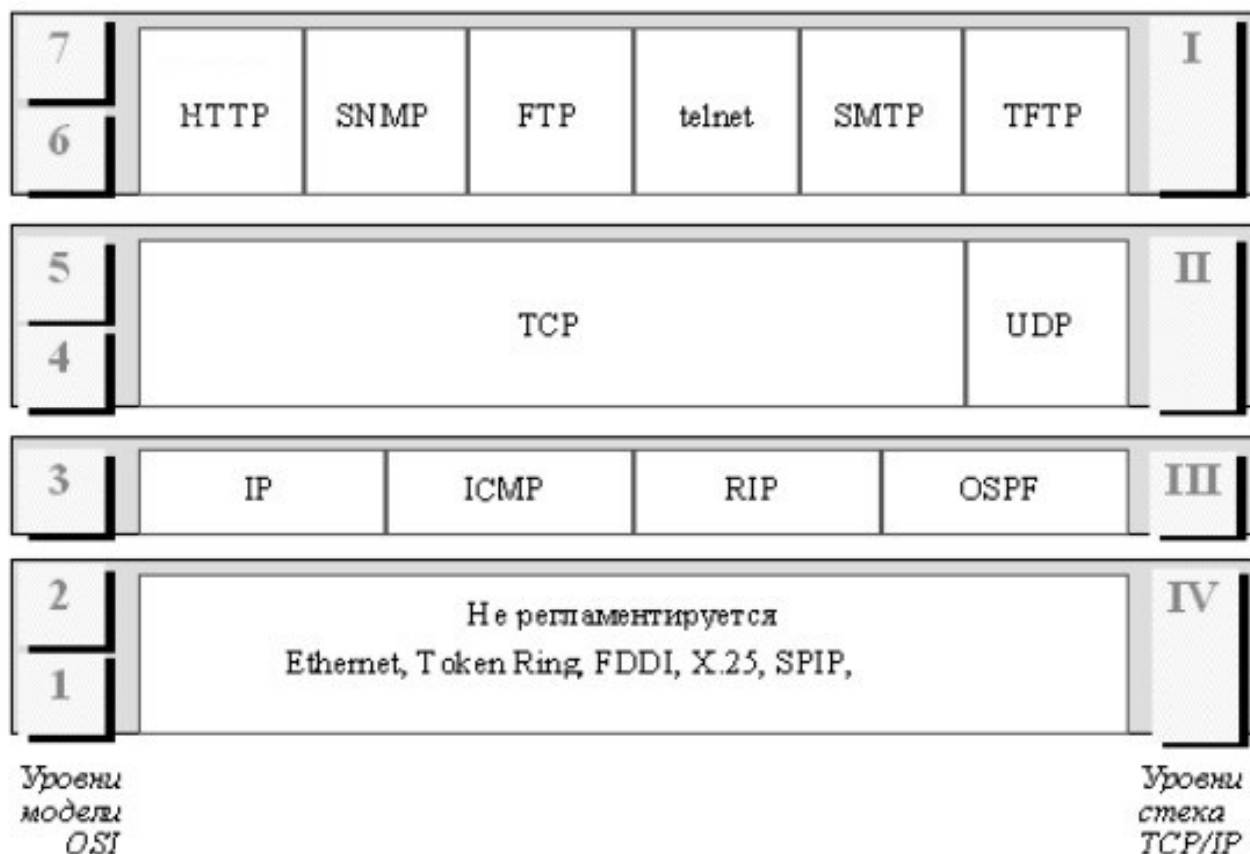
Протокол IP относится к протоколам *без установления соединений*, он поддерживает обработку каждого IP-пакета, как независимой единицы обмена, не связанной с другими IP-пакетами.

В протоколе IP, как известно, нет механизмов обычно применяемых для обеспечения достоверности данных на узле получателя.

Если во время продвижения пакета происходит какая-либо ошибка, то протокол IP по своей инициативе ничего не предпринимает для ее исправления.

Например, если пакет отбрасывается на каком-либо роутере, то, максимум, что происходит — это оповещение узла-отправителя этим роутером о произошедшем событии с помощью протокола ICMP (*Internet Control Messaging Protocol*).

На уровне протокола IP не существует механизма проверки правильности очередности доставки пакетов (дейтаграмм) на узел-получатель и отсутствия дублирования пакетов. То есть, протокол IP реализует политику доставки «по возможности»



Вопросы обеспечения надежной доставки данных по составной сети в стеке TCP/IP решает протокол TCP. Полный стек протоколов реализован только на конечных узлах.

То есть протокол TCP обеспечивает надежную доставку сообщений, используя в качестве основы ненадежный дейтаграммный протокол IP. Для решения этой задачи протокол TCP использует метод продвижения данных с *установлением логического соединения*.

Именно *логическое соединение* дает возможность участникам обмена следить за тем, чтобы данные не были потеряны, искажены или продублированы, а также чтобы они пришли к получателю в том порядке, в котором были отправлены.

Информация, поступающая к протоколу TCP от протоколов более высокого уровня (от прикладных процессов), рассматривается протоколом TCP как *неструктурированный поток байтов*.

Поступающие данные буферизуются средствами TCP.

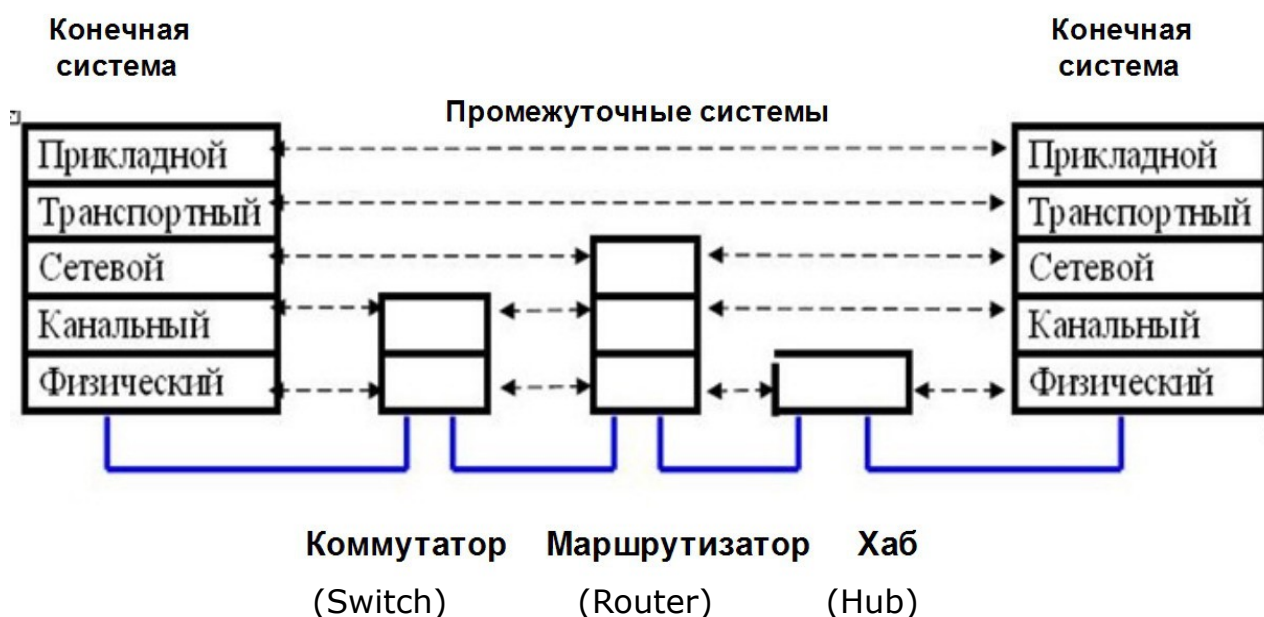
Для передачи на сетевой уровень из буфера «вырезается» некоторая непрерывная часть данных, которая называется *сегментом* и снабжается заголовком.

Процедуры установления, конфигурирования и разрыва логического соединения, естественно, требуют определенных накладных расходов, что является платой за надежность передачи данных протоколом TCP.

Протокол UDP гораздо проще. Его функции сводятся к простой передаче данных между прикладным и сетевым уровнями, а также примитивному контролю искажений в передаваемых данных. При контроле искажений протокол UDP только диагностирует, но не исправляет ошибку.

Если контрольная сумма показывает, что в поле данных UDP-дейтаграммы произошла ошибка, протокол UDP просто отбрасывает поврежденную дейтаграмму.

Естественно, транспортировка данных протоколом UDP менее затратна и более быстра (при прочих равных условиях), передача с помощью TCP .



Коммуникационным устройствам в промежуточных узлах сети для продвижения пакетов достаточно функциональности нижних трех уровней.

Хаб — это устройство, которое работает только с потоком битов поэтому ограничивается поддержкой протокола физического уровня.

Коммутаторы локальных сетей поддерживают протоколы двух нижних уровней, что дает им возможность работать в пределах стандартных топологий.

Маршрутизаторы должны поддерживать протоколы всех трех нижних уровней, так как сетевой уровень необходим маршрутизаторам для объединения сетей различных технологий, а протоколы нижних уровней - для взаимодействия с конкретными сетями, образующими составную сеть (Ethernet, Frame Relay, X25 и др.). Каждый компьютер может выполнять несколько процессов, более того, даже отдельный прикладной процесс может иметь несколько точек входа, выступающих в качестве адресов назначения для пакетов данных.

Поэтому после того, как пакет доставлен по сети средствами протокола IP на сетевой интерфейс компьютера-получателя данные необходимо переправить конкретному процессу-получателю.

Протоколы TCP и UDP ведут для каждого приложения

две системные очереди:

очередь данных, поступающих к приложению из сети,

и очередь данных, отправляемых этим приложением в сеть.

Пакеты, поступающие на транспортный уровень, организуются

операционной системой в виде множества очередей

к точкам входа различных прикладных процессов.

В терминологии TCP IP такие системные очереди называются *портами* (порты приложений не надо путать с сетевыми интерфейсами оборудования и др., тоже называемыми портами).

При этом входная и выходная очереди одного приложения рассматриваются как один порт.

Для однозначной идентификации портов им присваивают номера.

Номера портов используются для адресации приложений.

Номера из диапазона от 0 до 1023 называются назначенными, являются уникальными в пределах Интернета

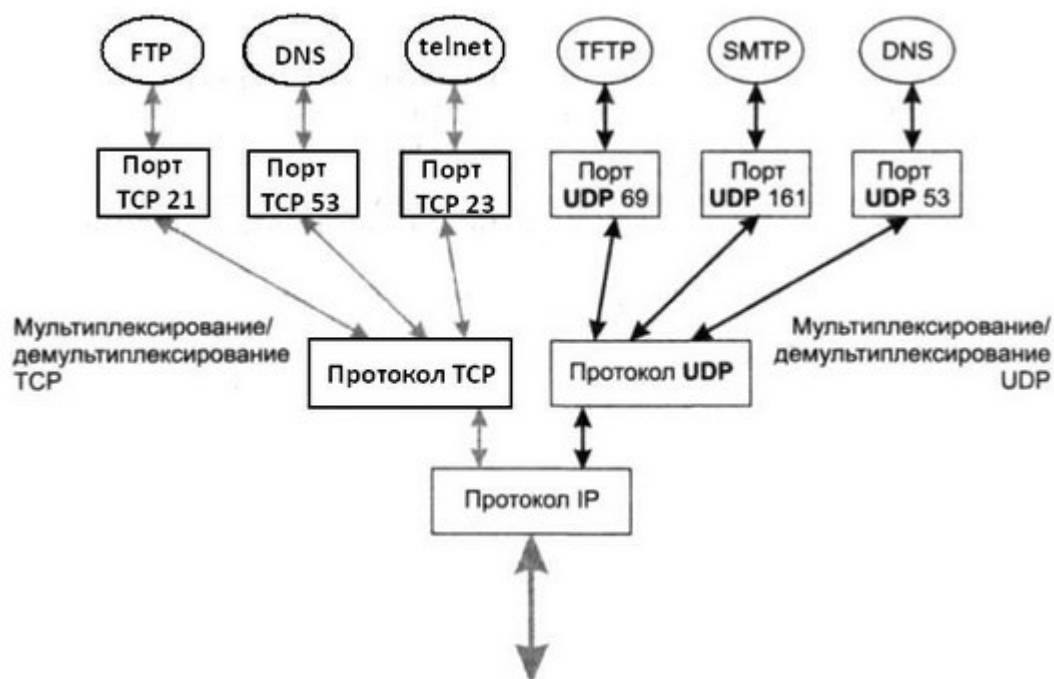
и закрепляются за приложениями централизованно.

Номера из диапазона 1024 до 65 535 могут назначаться *локально* разработчиками собственных приложений

или операционной системой в ответ на поступление запроса

от приложения. На каждом компьютере операционная система

ведет список занятых и свободных номеров портов.



Никакой связи между назначенными номерами TCP- и UDP-портов нет.

Даже если номера TCP- и UDP-портов совпадают, они идентифицируют разные приложения. Например, одному приложению может быть назначен TCP-порт 1750, а другому — UDP-порт 1750.

С точки зрения именования (адресации) сокетов, существует две различные схемы, называемые *address families* (семейства адресации).

Наиболее часто используемая, схема адресации носит название *Internet domain addressing*. Эта схема базируется на 32-битном IP адресе хоста, на котором адресуемый сокет (процесс создавший сокет) находится и на 16-битном номере порта, идентифицирующем процесс на хосте. То есть адрес сокета в этом случае ассоциируется с парой :

IP address/Port number.

Другая, упрощенная, схема адресации, называемая *Unix domain address family*, подразумевает, что сокет ассоциируется с файлом, имя и путь к которому попадает в системный список каталогов и помечается символом *s* (*socket*). Естественно, данная схема не пригодна для взаимодействия процессов по сети, то есть клиент и сервер обязаны находиться на одной и той же машине.

Для обоих видов адресации определяются соответствующие им структуры. Системные вызовы, проводящие операции обмена, для обращения к нужным им сокетам используют указатели на ассоциированные с этими сокетами адресные структуры.

В случае применения *Unix domain address family* образуемый сокет для обмена между процессами представляет собой улучшенный вариант программного канала (*pipe*), но работающего в обоих направлениях, то есть *full-duplex pipe* .

Самый простой способ организации обмена на сокетах базируется на системном вызове *socketpair()*. Естественным ограничением данного способа является невозможность взаимодействия процессов, запущенных на разных хостах, а также родство процессов. В системном вызове *socketpair()* в первом параметре задается семейство адресации, второй параметр указывает на тип (*SOCK_STREAM* либо *SOCK_DGRAM*) создаваемого сокета, третий параметр конкретизирует тип протокола (0 - протокол выбирается самой ОС) заданного семейства адресации. Четвертый параметр вызова *socketpair()* указывает на двухэлементный целый массив, который заполняется (при успешном вызове) дескрипторами создаваемой пары сокетов. Программа *socketpair* иллюстрирует применение этого системного вызова.

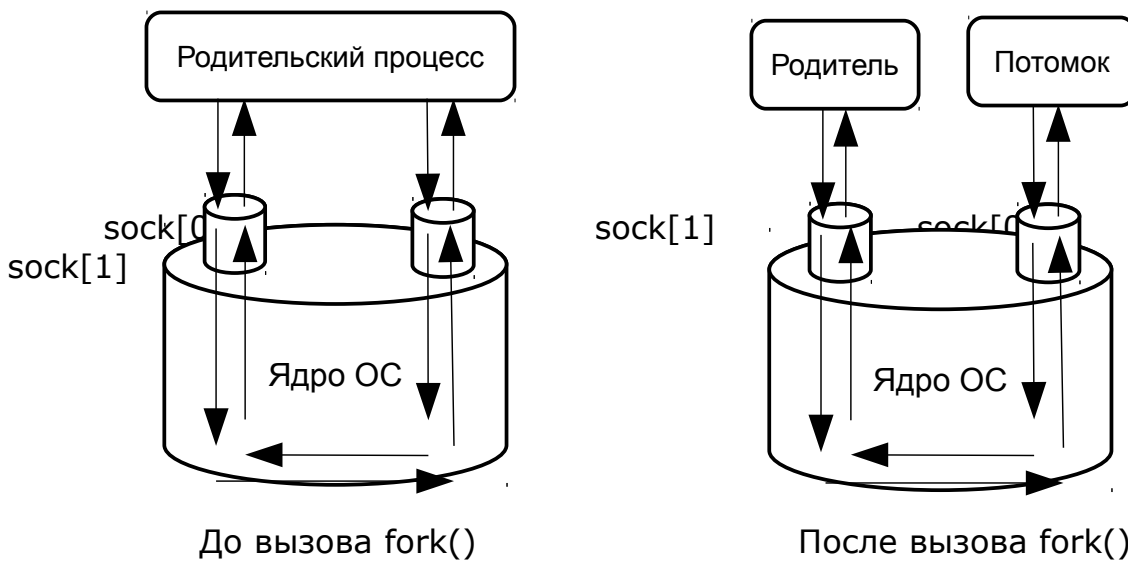
```

/* Программа socketpair.cpp */

/* Создает пару сокетов, запускает дочерний процесс и
 * использует пару сокетов для обмена информацией
 * между родительским и дочерним процессами.
 * Компилировать с опцией -lsocket
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types>
#include <sys/socket.h>
#define BUF_SZ 10
main(void){
int  sock[2];
int  cpid, i;
static char buf[BUF_SZ];
if (socketpair(PF_UNIX, SOCK_STREAM, 0, SOCK)<0){
    perror("Generation error");
    exit(1);
}
switch (cpid = (int) fork()){
    case -1:
        perror("Bad fork");
        exit(2);
    case 0: /* Дочерний процесс */
        close(sock[1]);
        for (i=0; i<10; i+=2){
            sleep(1);
            sprintf(buf, "c:%d\n", i);
            write(sock[0], buf, sizeof(buf));
            read(sock[0], buf, BUF_SZ);
            printf("c->%s", buf); /* Сообщение от родительского процесса */
        }
        close(sock[0]);
        break;
    default: /* Родительский процесс */
        close(sock[0]);
        for(i=1; i<10; i+=2){
            sleep(1);
            read(sock[1], buf, BUF_SZ);
            printf("p->%s", buf); /* Сообщение от дочернего процесса */
            sprintf(buf, "p: %d\n", i);
            write(sock[1], buf, sizeof(buf));
        }
        close(sock[1]);
    }
    return 0;
}

```

Иллюстрация использования системного вызова *socketpair()* в приведенной программе :



Для организации взаимодействия процессов на сокетах с постоянным соединением (*stream sockets*), без ограничений на выбор семейства адресации, на стороне процесса сервера и процесса клиента должна произойти целая цепочка системных вызовов и событий.

Системный вызов *socket()* на сервере создает экземпляр нового сокета. Параметры этого вызова задают семейство адресации, тип сокета и протокол. При успешном выполнении вызов возвращает дескриптор созданного сокета. Однако изначально с сокетом не ассоциируется какое-либо имя или адрес/порт.

Привязка к сокету имени или адреса выполняется следующим системным вызовом *bind()*, параметры которого содержат:
дескриптор созданного сокета,
указатель на адресную структуру
и длину адреса.

На серверной стороне вслед за *bind()* производится системный вызов *listen()*, создающий очередь для входящих запросов на соединение от клиентов.

В параметрах этого вызова указываются дескриптор сокета и максимально возможная длина очереди запросов.

На этот момент сервер уже готов к соединению с клиентами. Системный вызов *accept()* переводит, по умолчанию, сервер в состояние ожидания получения запросов на соединение от клиентов. Параметрами этого вызова являются
дескриптор сокета и
указатели на его адресную структуру
и ее длину.

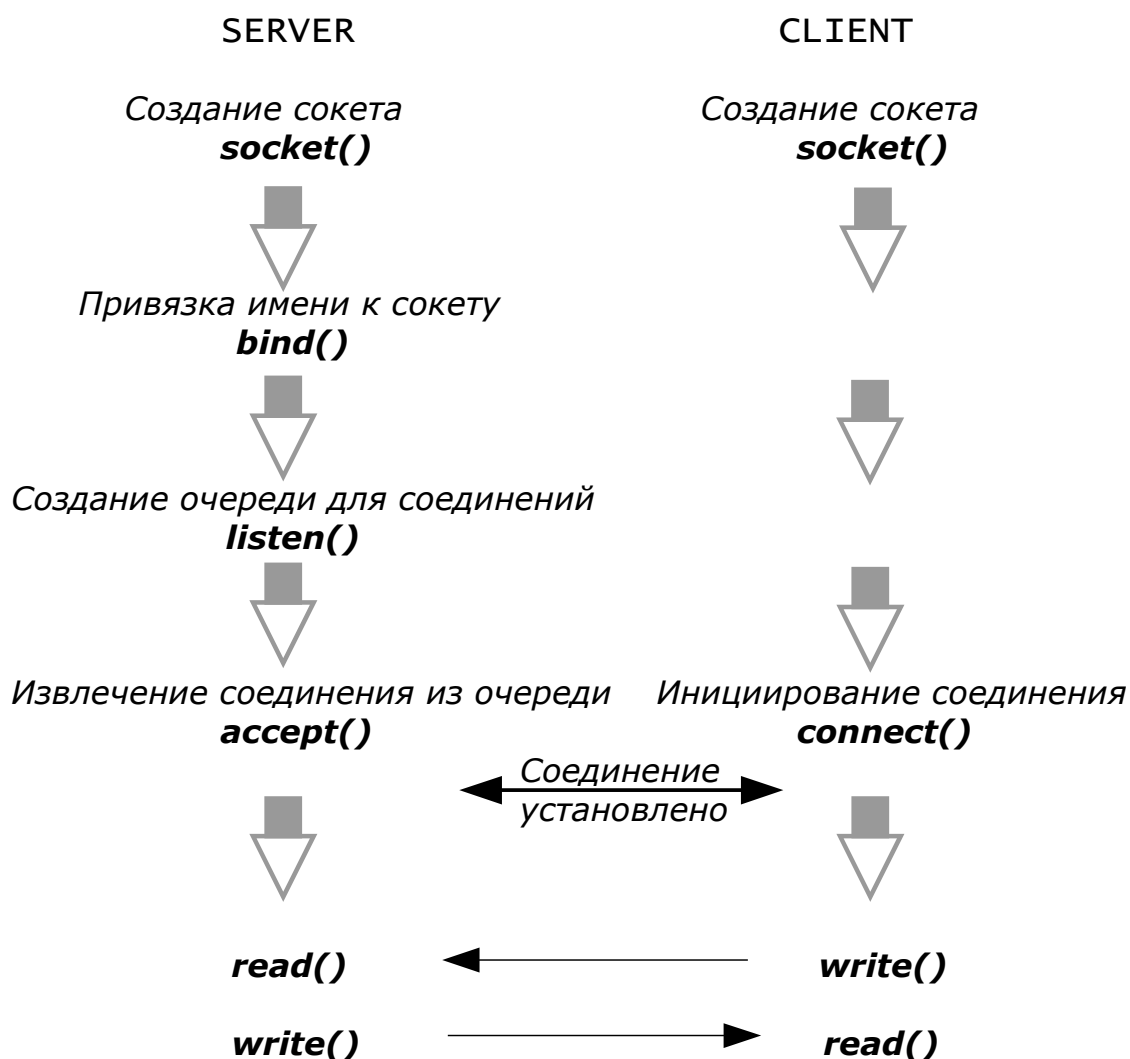
При наличии в очереди запроса от клиента (или при поступлении нового запроса) вызов *accept()* , воспринимает первый в очереди запрос, уменьшает очередь на единицу и создает новый сокет специально для организации соединения с данным клиентом.

Дескриптором этого нового сокета является целое значение, возвращаемое вызовом *accept()*.

Новый сокет имеет тот же тип, что и основной сокет, его дескриптор ассоциируется с дескриптором указанным в первом параметре вызова *accept()*, и на нем происходит коммуникация в стиле *read()/write()* с клиентом, запрос от которого таким образом был воспринят.

При этом, оригинальный сокет остается в том же виде и может использоваться для восприятия новых запросов от клиентов с целью организации соединений.

Последовательность системных вызовов при организации клиент-серверного взаимодействия, ориентированного на соединение :



На стороне клиентского процесса вслед за вызовом *socket()*, создающим сокет на клиенте, следует вызов *connect()*, с помощью которого клиент запрашивает сервер об организации соединения для обмена данными.

Первым параметром *connect()* является дескриптор созданного на клиенте сокета, а второй указывает на адресную структуру того сокета, с которым выполняется попытка соединиться, т.е. сокета сервера.

Третий параметр содержит длину адресной структуры.

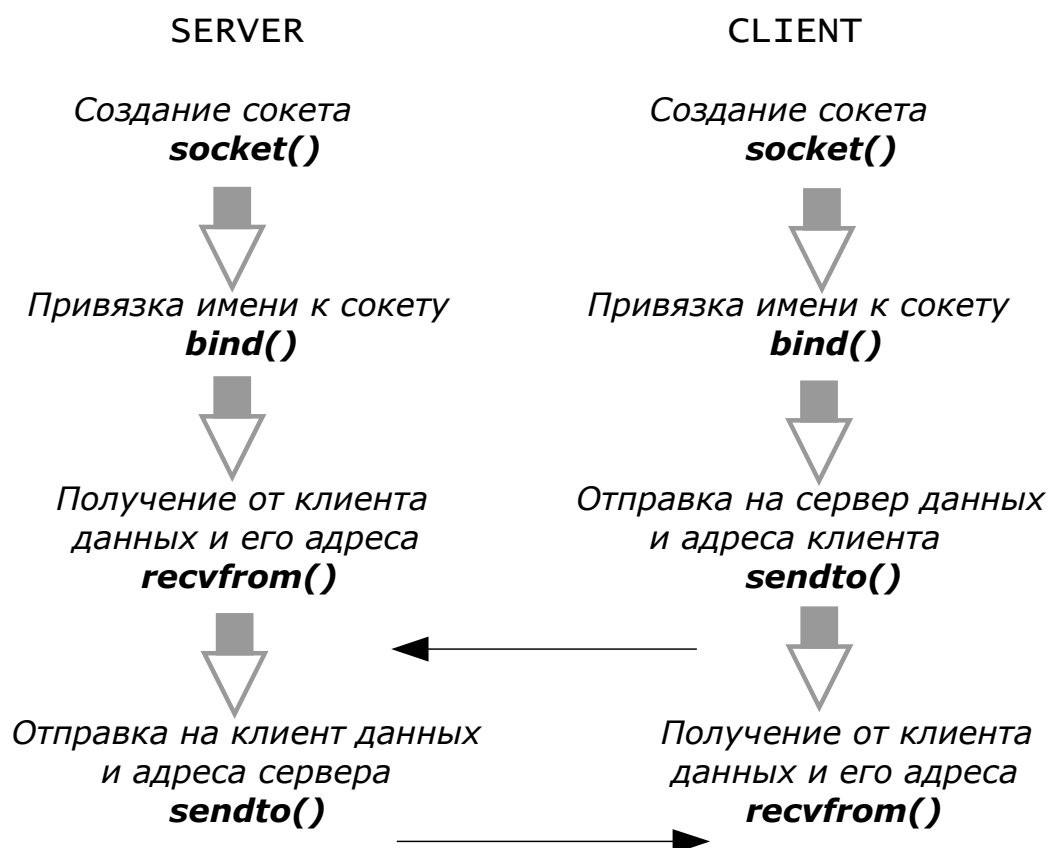
Запрос, порождаемый на клиентской стороне вызовом *connect()*, на сервере, в конечном итоге, воспринимается вызовом *accept()*.

По окончании сеанса обмена в стиле *read()/write()*, сокеты на обеих сторонах закрываются традиционным вызовом *close()*.

Обычно, при организации обмена с постоянным соединением (*stream socket*) сокет соединяется только раз.

В случае обмена без постоянного соединения (*connectionless socket*) один и тот же сокет может соединяться несколько раз, при этом вместо *read()/write()* используются вызовы *sendto()* / *recvfrom()*.

Последовательность системных вызовов при организации клиент-серверного взаимодействия, без постоянного соединения :



В файлах *local_c_i* , *sock_c_i_srv* и *sock_c_i_clt* содержится пример создания сокетов с постоянным соединением (*stream sockets*) и адресацией типа Internet Domain family для коммуникации клиентского процесса с сервером, запущенном на другом компьютере. Оба процесса запускаются в обычном *foreground* режиме, либо на разных хостах, либо на одном и том же, но на разных терминалах (или в разных CDE). Клиенту при запуске в качестве параметра командной строки (*cmd*) сообщается имя хоста сервера. Номер порта задается константой PORT в общем заголовочном файле. На сервере и клиенте выполняется цепочка системных вызовов и событий, предваряющая создание постоянных соединений на сокетах. Сервер, получая запросы от клиентов на соединение, порождает дочерние процессы образуя в них постоянные соединения на отдельных сокетах для ведения коммуникации с клиентом. Клиентский процесс после образования соединения отправляет серверу сообщения, вводимые пользователем с консоли клиентской машины.. Сервер считывает сообщения, преобразует и отправляет клиенту обратно. Завершение диалога клиента с сервером, закрытие соответствующих сокетов и завершение самого клиентского процесса происходит при вводе и обработке строки с символом '.' в первой позиции.

```
/* Заголовочный файл file local_c_i.h
 * общий для sock_c_i_srv.cpp и sock_c_i_clt.cpp
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT 6996
static char buf[BUFSIZ];          /* Буфер для сообщений */


/* Программа sock_c_i_srv.cpp */
/*
 * Сервер – сокеты с Internet domain адресацией, с установлением
 * соединения
 */
#include "local_c_i.h"
```

```

main(void) {
int      orig_sock, /* Дескриптор сокета на сервере */
          new_sock, /* Дескриптор сокета соединения */
          clnt_len; /* Длина адреса клиента */
struct sockaddr_in
          clnt_adr, /* Интернет адрес клиента */
          serv_adr; /* Интернет адрес сервера */
int      len,i;     /* Счетчики*/
if((orig_sock=socket(AF_INET,SOCK_STREAM,0)) < 0) {/* Сокет */
    perror("generate error");
    exit(1);
};
memset(&serv_adr, 0 , sizeof(serv_adr)); /* Очистка структуры */
serv_adr.sin_family =AF_INET;           /* Вид адресации Internet
domain */
serv_adr.sin_addr.s_addr=htonl(INADDR_ANY); /* Интерфейс любой */
serv_adr.sin_port=htons(PORT);           /* Фиктивный порт */
                                           /* BIND */
if (bind(orig_sock, (struct sockaddr *) &serv_adr,
          sizeof(serv_adr)) < 0){
    perror("bind error");
    close(orig_sock);
    exit(2);
};
if (listen(orig_sock,5)<0){                /* LISTEN */
    perror("listen error");
    exit(3);
};
do {
    clnt_len = sizeof(clnt_adr);
    if ((new_sock = accept(orig_sock,(struct sockaddr *) &clnt_adr,
                           &clnt_len)) < 0) { /* ACCEPT */
        perror("accept error");
        close(orig_sock);
        exit(4);
    };
    if ( fork () == 0) { /* Дочерний процесс */
        while ((len=read(new_sock,buf,BUFSIZ))>0){
            for(i=0;i<len;++i) /* Изменение букв на
прописные */
                buf[i] = toupper(buf[i]);
            write(new_sock,buf,len); /* Ответная запись */
            if (buf[0] == '.') break; /* Исчерпана? */
        }
        close(new_sock); /* Дочерний процесс */
        exit(0);
    } else close(new_sock); /* Родительский процесс */
} while(1); /* Заикливание */
}

```

```

/* Программа sock_c_i_clt.cpp */
/*
 * Клиент - сокеты с Internet domain адресацией, с установлением
 соединения
 */
#include "local_c_i.h"
main(int argc, char *argv[] ){
    int    orig_sock,          /* Дескриптор сокета клиента */
           len;               /* Длина адреса сервера */
    struct sockaddr_in
           serv_adr;          /* Интернет адрес сервера */
    struct hostent *host;      /* Хост (сервер) */

    if ( argc != 2){ /* Expect name of host on cmd line */
        fprintf(stderr, "usage: %s server\n", argv[0]);
        exit(1);
    }
    host = gethostbyname(argv[1]); /* Информация о хосте */
    if (host == (struct hostent *) NULL ){
        perror("gethostbyname");
        exit(2);
    }
    memset(&serv_adr, 0, sizeof(serv_adr)); /* Очистка структуры */
    serv_adr.sin_family = AF_INET; /* Вид адресации Internet domain */
    memcpy(&serv_adr.sin_addr, host->h_addr, host->h_length); /* Адрес */
    serv_adr.sin_port = htons(PORT); /* Используется фиктивный порт */

    if ((orig_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) { /* Сокет */
        perror("generate error");
        exit(3);
    }

    /* CONNECT */
    if (connect(orig_sock, (struct sockaddr *) &serv_adr,
               sizeof(serv_adr)) < 0 ) {
        perror("connect error");
        exit(4);
    };
    do {
        write(fileno(stdout), "> ", 3); /* Подсказка */
        if((len=read(fileno(stdin), buf, BUFSIZ))>0) { /* Ввод */
            write(orig_sock, buf, len);
            if ((len=read(orig_sock, buf, len))>0) /* Вывод, */
                write(fileno(stdout), buf, len); /* если не пусто */
        }
    } while (buf[0] != '.');
    close(orig_sock);

```

```
exit(0);  
};
```

Пример применения сокетов, ориентированных на соединение, но с адресацией типа *Unix domain* (подразумевает нахождение сервера и клиента на одном компьютере), приведен в программах *sock_c_u_srv* и *sock_c_u_clt*.
Сервер запускается в фоновом (*background*) режиме, получая запрос на соединение от клиента, создает сокет для сеанса с клиентом, получает и выдает на консоль до 10-ти сообщений от клиента, после чего закрывает этот сокет.

```
/* Программа sock_c_u_srv.cpp */  
/*  
 * Сервер - сокеты с UNIX domain адресацией, с установлением  
соединения  
 */  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <string.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <sys/un.h>  
#define NAME "my_sock"  
  
void clean_up( int, char *);  
  
main( void ) {  
int      orig_sock,      /* Дескриптор сокета */  
         new_sock,      /* Дескриптор сокета соединения */  
         clnt_len,      /* Длина адреса клиента */  
         i,             /* Счетчик циклов */  
         cpid;          /* PID */  
static struct sockaddr_un  
         clnt_adr,      /* UNIX адреса клиента */  
         serv_adr;      /* сервера */  
static char  buf[10];    /* Буфер для сообщений */  
  
if((orig_sock = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) { /* SOCKET */  
    perror("Generate error");  
    exit(1);  
}  
serv_adr.sun_family = AF_UNIX;      /* Установка вида адресации */  
strcpy(serv_adr.sun_path,NAME);     /* Присвоение имени (108  
символов макс) */  
unlink(NAME);                       /* Удаление старой копии, если имеется  
*/
```

```

if(bind(orig_sock, (struct sockaddr *) &serv_adr,      /* BIND */
    sizeof(serv_adr.sun_family)+strlen(serv_adr.sun_path)) < 0) {
    perror("bind error");
    clean_up(orig_sock,NAME);
    exit(2);
};
listen(orig_sock,3);                                /* LISTEN */
clnt_len = sizeof(clnt_adr);

if ((new_sock = accept(orig_sock, (struct sockaddr *) &clnt_adr,
    &clnt_len)) < 0) {      /* ACCEPT */
    perror("accept error");
    clean_up(orig_sock, NAME);
    exit(3);

}
/*printf("new sock = %d\n",new_sock);*/
for(i=1; i<= 10; i++){      /* PROCESS */
    sleep(1);
    read(new_sock, buf, sizeof(buf));
    printf("s-> %s", buf);

}

close(new_sock);
clean_up(orig_sock, NAME);
exit(0);
}

void
clean_up(int sd, char *the_file ) {
    close(sd);      /* Закрыць, */
    unlink(the_file); /* удалить */
}

/* Программа sock_c_u_clt.cpp */
/*
 * Клиент - сокеты с UNIX domain адресацией, с установлением соединения
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

```

```

#define NAME "my_sock"

main(void) {
    int    orig_sock,      /* Дескриптор сокета клиента */
           i;              /* Счетчик циклов */
    static structsockaddr_un
           serv_adr;       /* Unix адрес сервера */
    static char  buf[10];  /* Буфер для сообщений */

    if ((orig_sock = socket(AF_UNIX, SOCK_STREAM,0)) < 0) { /* Сокет */
        perror("generate error");
        exit(1);
    }
    serv_adr.sun_family = AF_UNIX; /* Установка типа адреса */
    strcpy(serv_adr.sun_path,NAME); /* Назначение имени */

    if (connect(orig_sock, (struct sockaddr *) &serv_adr, /* CONNECT */
               sizeof(serv_adr.sun_family)+strlen(serv_adr.sun_path))<0) {
        perror("connect error");
        exit(1);
    }
    for(i=1;i<=10;i++){          /* Отправка сообщений */
        sprintf(buf,"c: %d\n",i);
        write(orig_sock,buf,sizeof(buf));
    }
    close(orig_sock);
    exit(0);
}

```