Архитектура ОС UNIX

Санкт-Петербургский государственный политехнический университет

3 ноября 2011

Обмен датаграммами

В большинстве случаев надежность при передаче данных критична поэтому используется надежный транспортный протокол TCP, но в некоторых случаях, например, для передаче видео или аудио подтверждение о доставке пакета не требуется т.к. данные поступают в реальном времени.

Обмен датаграммами

Создав сокет при помощи функций socket и bind можно сразу воспользоваться им для отправки данных, используя функции sendto и recvfrom, не вызывая метода connect.

```
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);
```

Обмен датаграммами

Два дополнительных параметра to и tolen используются для указания адреса получателя.
Для задания адреса используется структура sockaddr.
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.

Присоединённые датаграммные сокеты

Для сокета с типом SOCK_DGRAM тоже можно вызвать функцию connect, а затем использовать send и recv для обмена данными.
При этом никакого соединения при этом не устанавливается.
Операционная система просто запоминает адрес, который вы передали функции connect, а затем использует его при отправке данных.
Присоединённый сокет может получать данные только от сокета, с которым он соединён.

UDP sender

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

UDP sender

```
    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;
}
```

UDP receiver

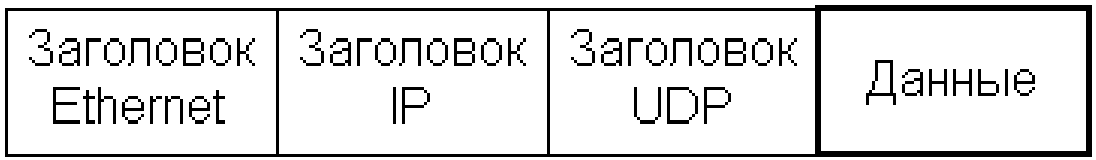
```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
int main()
{
    int sock;
    struct sockaddr_in addr;
    char buf[1024];
    int bytes_read;
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if(sock < 0)
    {
        perror("socket");
        exit(1);
    }
    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425);
}
```

UDP receiver

```
    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;
}
```

Использование низкоуровневых сокетов

Низкоуровневые сокеты в отличие от обычных позволяют в ручную формировать заголовки сообщения, т.е. предоставляют программисту полный контроль над содержимым пакетов, которые отправляются по сети.
Часто используются при разработке системных утилит типа ping и traceroute.



Использование низкоуровневых сокетов

При работе с низкоуровневыми сокетами необходимо указывать в третьем параметре функции socket тот протокол, к заголовкам которого нужно получить доступ.
Константы для основных протоколов Internet объявлены в файле netinet/in.h. Они имеют вид IPPROTO_XXX, где XXX-название протокола: IPPROTO_TCP, IPPROTO_UDP, IPPROTO_RAW (в последнем случае появляется возможность поработать с «сырым» IP и формировать IP-заголовки вручную).

Использование низкоуровневых сокетов

Все числовые данные в заголовках должны записываться в сетевом формате. Поэтому надо не забывать использовать функции htons и htonl.
Заголовок протокола UDP:

Порт отправителя (16 бит)	Порт получателя (16 бит)
Длина (заголовок+данные) (16 бит)	Контрольная сумма (16 бит)

Sender RAW socket

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
struct UdpHeader
{
    u_short src_port;
    u_short targ_port;
    u_short length;
    u_short checksum;
};
char message[] = "Hello there!\n";
char msgbuf[1024];
int main()
{
    int sock;
    struct sockaddr_in addr;
    struct UdpHeader header;
    sock = socket(AF_INET, SOCK_RAW, IPPROTO_UDP);
```

Sender RAW socket

```
if(sock < 0)
{
    perror("socket");
    exit(1);
}
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
header.targ_port = htons(3425);
header.length = htons(sizeof(header)+sizeof(
    message));
header.checksum = 0;
memcpy((void *)msgbuf, (void *)&header, sizeof(
    header));
memcpy((void *)(msgbuf+sizeof(header)), (void
    *)message, sizeof(message));
sendto(sock, msgbuf, sizeof(header)+sizeof(
    message), 0,
        (struct sockaddr *)&addr, sizeof(addr));
close(sock);
```

Функции для работы с адресами и DNS

Для идентификации хостов в сети Internet используются доменные имена, которые необходимо преобразовать в IP-адрес для доступа к узлу. Для разрешения имен используются следующие функции:

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

int inet_aton(const char *cp, struct in_addr *in_p)
    ;
unsigned long int inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);
```

Функции для работы с адресами и DNS

Функция inet_addr часто используется в программах.
Она принимает строку и возвращает адрес (уже с сетевым порядком следования байтов).
Проблема с этой функцией состоит в том, что значение -1, возвращаемое ею в случае ошибки, является в то же время корректным адресом 255.255.255.255.
Для решения этой проблемы рекомендуется использовать более новую функцию inet_aton (Ascii TO Network), которая возвращает заполненную структуру in_addr. Эта функция возвращает 0 в случае ошибки.
Для обратного преобразования используется функция inet_ntoa (Network TO Ascii). Эта функция также возвращает 0 в случае ошибки.

Функции для работы с адресами и DNS

Для преобразования доменного имени в IP-адрес используется функция gethostbyname.

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

Эта функция получает имя хоста и возвращает указатель на структуру с его описанием.

Функции для работы с адресами и DNS

```
struct hostent {
    char    *h_name;
    char    **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
};
#define h_addr h_addr_list[0]
```

- h_name — имя хоста.
- h_aliases — массив строк, содержащих псевдонимы хоста. Завершается значением NULL.
- h_addrtype — тип адреса. Для Internet-домена - AF_INET.
- h_length — длина адреса в байтах.
- h_addr_list — массив, содержащий адреса всех сетевых интерфейсов хоста. Завершается нулём. Байты каждого адреса хранятся с сетевым порядке.

Функции для работы с адресами и DNS

```
struct hostent {
    char    *h_name;
    char    **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
};
#define h_addr h_addr_list[0]
```

- h_name — имя хоста.
- h_aliases — массив строк, содержащих псевдонимы хоста. Завершается значением NULL.
- h_addrtype — тип адреса. Для Internet-домена - AF_INET.
- h_length — длина адреса в байтах.
- h_addr_list — массив, содержащий адреса всех сетевых интерфейсов хоста. Завершается нулём.

Функции для работы с адресами и DNS

Для определения имени хоста по адресу используется функция gethostbyaddr. Вместо строки она получает адрес (в виде sockaddr) и возвращает указатель на ту же самую структуру hostent.
В случае ошибки gethostbyaddr и gethostbyname вместо указателя возвращают NULL, а расширенный код ошибки записывается в глобальную переменную h_errno.
Функции gethostbyname и gethostbyaddr возвращают указатель на статическую область памяти. Это означает, что каждое новое обращение к одной из этих функций приведёт к перезаписи данных, полученных при предыдущем обращении.

Функции для работы с адресами и DNS

```c
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

Функция gethostname используется для получения имени локального хоста. Далее его можно преобразовать в адрес при помощи gethostbyname. Это даёт нам способ в любой момент программно получить адрес машины, на которой выполняется наша программа, что может быть полезным во многих случаях.

Функции для работы с адресами и DNS

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

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

Функция getpeername позволяет в любой момент узнать адрес сокета на «другом конце» соединения. Она получает дескриптор сокета, соединённого с удалённым хостом, и записывает адрес этого хоста в структуру, на которую указывает addr. Фактическое количество записанных байт помещается по адресу addrlen.

Параллельное обслуживание клиентов

Одним из способов параллельного обслуживания клиентов является 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 клиента и завершается.

Server (fork)

```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;
```

Server (fork)

```
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);
    }
```

Server (fork)

```
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);
}
```

Server (fork)

```
    close(listener);
    return 0;
}
```

Задание 4

Реализовать Web-Server использующий многопроцессный способ обработки клиентов через fork(). Веб-сервер должен работать в виде демона и обеспечивать базовую поддержку протокола HTTP, на уровне метода GET (по желанию методы HEAD и POST). Проверку Web-Server'а необходимо осуществлять на статических текстовых и двоичных данных, таких как изображения. Предусмотреть контроль и журналирование ошибок (либо в файл, либо через syslog). Обеспечить одновременную работу сервера с множественным числом клиентов.

Параллельное обслуживание клиентов

Недостатками такого способа является создание чрезмерного количества процессов при достаточном большом количестве подлюченных клиентов, а также такой подход подразумевает что все клиенты должны обслуживаться независимо друг от друга, что будет помехай в некоторых типах приложений (например, чат-сервер).

Неблокирующие сокеты

Другим способом параллельного обслуживания многих клиентов является использование неблокирующих сокетов.
В стандартом случае при использовании блокирующих сокетов функции accept или recv ожидают поступления данных и программа блокируется.
Это поведение можно изменить переведя сокет в неблокирующий режим используя fcntl.

```
#include <unistd.h>
#include <fcntl.h>
.
.
sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

Неблокирующие сокеты

Вызов любой функции с таким сокетом будет возвращать управление немедленно. Причём если затребованная операция не была выполнена до конца, функция вернёт -1 и запишет в errno значение EWOULDBLOCK.
Чтобы дождаться завершения операции, можно опрашивать все сокеты в цикле, пока какая-то функция не вернёт значение, отличное от EWOULDBLOCK. Как только это произойдёт, становится возможным выполнение следующей операции с этим сокетом и возврат к циклу.

select

Чтобы исправить ситуацию используется функция select, которая позволяет отслеживать состояние нескольких файловых дескрипторов.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int n, fd_set *readfds, fd_set *writefds
    ,
                fd_set *exceptfds, struct timeval *
                  timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(int fd);
```

select

Функция select работает с тремя множествами дескрипторов, каждое из которых имеет тип fd_set. В множество readfds записываются дескрипторы сокетов, из которых нам требуется читать данные.
Множество writefds должно содержать дескрипторы сокетов, в которые нужно писать данные, а exceptfds - дескрипторы сокетов, которые нужно контролировать на возникновение ошибки.
Параметр n указывает максимальное число всех дескрипторов + 1, а в timeout — величину таймаута.

select

Для работы с дескрипторами предусмотрены функции FD_XXX:

- FD_ZERO(fd_set *set) — очищает множество set
- FD_SET(int fd, fd_set *set) — добавляет дескриптор fd в множество set
- FD_CLR(int fd, fd_set *set) — удаляет дескриптор fd из множества set
- FD_ISSET(int fd, fd_set *set) — проверяет, содержится ли дескриптор fd в множестве set

select

Если хотя бы один сокет готов к выполнению заданной операции, select возвращает ненулевое значение, а все дескрипторы, которые привели к "срабатыванию"функции, записываются в соответствующие множества.
Если сработал таймаут, select возвращает ноль, а в случае ошибки -1. Расширенный код записывается в errno.

select

Программы, использующие неблокирующие сокеты вместе с select, получаются весьма запутанными, потомучто приходится отслеживать дескрипторы всех клиентов и работать с ними параллельно.

Server (select)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <algorithm>
#include <set>
using namespace std;
int main()
{
    int listener;
    struct sockaddr_in addr;
    char buf[1024];
    int bytes_read;
    listener = socket(AF_INET, SOCK_STREAM, 0);
```

Server (select)

```
if(listener < 0)
{
    perror("socket");
    exit(1);
}
fcntl(listener, F_SETFL, O_NONBLOCK);
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, 2);
set<int> clients;
clients.clear();
```

Server (select)

```
while(1)
{
    // Заполняем множество сокетов
    fd_set readset;
    FD_ZERO(&readset);
    FD_SET(listener, &readset);
    for(set<int>::iterator it = clients.begin()
        ; it != clients.end(); it++)
        FD_SET(*it, &readset);
    // Задаём таймаут
    timeval timeout;
    timeout.tv_sec = 15;
    timeout.tv_usec = 0;
    // Ждём события в одном из сокетов
    int mx = max(listener, *max_element(clients
        .begin(), clients.end()));
```

Server (select)

```
if(select(mx+1, &readset, NULL, NULL, &
    timeout) <= 0)
{
    perror("select");
    exit(3);
}
// Определяем тип события и выполняем
    соответствующие действия
if(FD_ISSET(listener, &readset))
{
    // Поступил новый запрос на соединение,
        используем accept
    int sock = accept(listener, NULL, NULL)
        ;
    if(sock < 0)
    {
        perror("accept");
        exit(3);
    }
```

Server (select)

```
            fcntl(sock, F_SETFL, O_NONBLOCK);
            clients.insert(sock);
        }
        for(set<int>::iterator it = clients.begin()
            ; it != clients.end(); it++)
        {
            if(FD_ISSET(*it, &readset))
            {
                // Поступили данные от клиента,
                    читаем их
                bytes_read = recv(*it, buf, 1024,
                    0);
                if(bytes_read <= 0)
                {
                    // Соединение разорвано,
                        удаляем сокет из множества
                    close(*it);
                    clients.erase(*it);
                    continue;
```

Server (select)

```
                    }
                    // Отправляем данные обратно
                        клиенту
                    send(*it, buf, bytes_read, 0);
                }
            }
        }
        return 0;
}
```

Задание 5

Реализовать приложение «сетевой чат». Сервер должен использовать неблокирующие сокеты, как способ параллельной обработки и передачи данных клиентам. Сетевой сервер должен быть реализован ввиде демона, с контролем и журналированием ошибок, либо через syslog, либо в файл журнала. Сервер должен обеспечивать одноадресную и широковещательную отправку текстовых сообщений, одному или всем участникам соответственно. Реализовать сетевой клиент для проверки работоспособности сервера. Сетевой клиент может быть реализован без неблокирующих сокетов, в самом простом варианте, можно взять готовый клиент: netcat или telnet.