

Федеральное государственное автономное образовательное учреждение
высшего образования «Самарский национальный исследовательский университет
имени академика С.П. Королева»
(Самарский университет)



Лабораторная работа №4 СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ

Самара, 2022

Содержание

1. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ	3
2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ	3
2.1. IP-адреса, порты и порядок байтов	3
2.2. Сокеты	4
2.3. Установление соединения	5
2.4. Передача данных	7
3. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ	7
ПРИЛОЖЕНИЯ	9
Приложение 1. Пример приложения-сервера	9
Приложение 2. Пример приложения-клиента.	10
Приложение 3. Файл common.h	11

1. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

1. **Компьютерная сеть** – множество компьютеров, связанных между собой каналами передачи информации.
2. **Узел сети** – устройство, участвующее в процессе передачи данных по сети.
3. **Протокол передачи данных** – стандарт, описывающий процесс передачи данных по сети (структуры данных, последовательность действий сторон, возможные состояния и переходы между ними).
4. **Конечная точка** – узел сети либо процесс на узле сети, являющийся исходным отправителем или итоговым получателем данных. В сетях IP обычно определяется триадой (IP-адрес, протокол транспортного уровня, порт).
5. **Протокол TCP** – протокол транспортного уровня, предназначенный для *надежной* передачи *поточковых данных*.
6. **Протокол UDP** – протокол транспортного уровня, предназначенный для *ненадежной* передачи *отдельных сообщений*.
7. **Протокол IP** – протокол сетевого уровня, ответственный за передачу данных между узлами сети.
8. **IP адрес** – 4-байтовая структура, однозначно идентифицирующая узел в данной сети.
9. **Порядок байтов** – порядок, в котором байты многобайтовых чисел хранятся в памяти. Может быть прямым или обратным.
10. **Сетевой порядок байтов** – порядок байтов, который должны использовать устройства при работе в сети. Эквивалентен прямому порядку байтов.
11. **Сокет** – программная абстракция конечной точки.

2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1. IP-адреса, порты и порядок байтов

IP-адрес может быть представлен в виде целого 4-байтового числа (unsigned int). Порт – число в диапазоне [0,65535] – также является 2-байтовым целым числом.

Существует ряд макросов, представляющих особые значения портов и адресов.

```
/*#include <arpa/inet.h>*/

#define INADDR_NONE ((in_addr_t) 0xffffffff) /*ошибочный адрес*/
#define INADDR_LOOPBACK ((in_addr_t) 0x7f000001) /*127.0.0.1*/

/*верхняя граница системных портов, порты с меньшими номерами не могут
быть назначены без root-прав*/
#define IPPORT_RESERVED      1024

/*нижняя граница пользовательских портов, порты с большими номерами не
зарезервированы IANA для приложений*/
#define IPPORT_USERRESERVED  5000 /* граница системных портов*/
```

Поскольку и IP-адрес, и порт являются многобайтовыми числами, при работе с сетью они должны быть представлены в сетевом порядке байтов. Если же они были прочитаны (например, через вызов `sockopt`), то для использования внутри программы они должны быть представлены в порядке байтов, используемых процессором. Для преобразования данных чисел из сетевого порядка байтов в порядок байтов хоста и наоборот используется ряд функций:

```
#include <arpa/inet.h>
unsigned int  htonl(unsigned int hostlong);    /*to network order*/
unsigned short htons(unsigned short hostshort); /*to network order*/
unsigned int  ntohl(unsigned int netlong);     /*to host order*/
unsigned short ntohs(unsigned short netshort); /*to host order*/
```

IP-адрес представляется в виде структуры `in_addr`. Данная структура имеет единственный адрес, представляющий IP-адрес в формате одного целого числа:

```
#include <arpa/inet.h>

typedef unsigned int in_addr_t;

struct in_addr {
    in_addr_t s_addr;
};
```

Преобразование порядка байтов из строки/в строку осуществляется функциями `inet_addr/inet_ntoa`:

```
#include <arpa/inet.h>

in_addr_t inet_addr(const char* cp); /* INADDR_NONE при ошибке */
char* inet_ntoa(in_addr in); /* NULL при ошибке */
```

Важно отметить, что `inet_addr` возвращает только `in_addr_t`, в то время как `inet_ntoa` принимает на вход `in_addr`. Функция `inet_addr` возвращает адрес в сетевом порядке байтов – дальнейшие преобразования не требуются.

2.2. Сокеты

Сокет является абстракцией конечной точки некоторого соединения. Сокеты создаются вызовом `socket`:

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

int fd = socket(int domain, int type, int protocol /*=0*/);
```

Функция возвращает дескриптор сокета или -1 в случае ошибки.

Аргумент `domain` должен содержать константу `AF_INET` в случае сети IPv4 или `AF_UNIX` в случае создания сокета UNIX (см. далее).

Аргумент `type` предназначен для выбора типа сокета.

Аргумент `protocol` предназначен для передачи конкретного протокола транспортного уровня, однако поскольку протокол определяется типом сокета, здесь должен передаваться 0.

Тип сокета должен быть одним из 4 констант:

`SOCK_STREAM` – сокет потоковой передачи данных (TCP).

`SOCK_DGRAM` – сокет ненадежной передачи сообщений (UDP).

SOCK_SEQPACKET – сокет надежной передачи сообщений [сокет UNIX].

SOCK_RAW – «сырой» сокет [прямой доступ к сетевому уровню].

Созданный сокет не привязан ни к одной конечной точке. Для связывания сокета с конечной точкой с заданным адресом используется вызов `bind`:

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

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

Аргумент `sockfd` предназначен для передачи дескриптора сокета.

Аргумент `addr` предназначен для передачи адреса структуры, содержащей информацию об адресе конечной точки. При этом тип данной структуры скорее всего отличается от `sockaddr`, поэтому вызову требуется знать размер структуры.

Аргумент `addrlen` должен содержать реальный размер структуры, которая передается в `addr`.

IP-адрес представляется структурой `sockaddr_in`:

```
struct sockaddr_in {
    sa_family_t sin_family; /* = AF_INET */
    in_port_t sin_port; /* порт */
    in_addr sin_addr; /* адрес */
};
```

Поля данной структуры должны быть числами в прямом порядке байтов.

2.3. Установление соединения

Процедура установления соединения необходима только для сокетов типа `SOCK_STREAM` и `SOCK_SEQPACKET`.

Формально, для установления соединения требуется только вызов `connect`:

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

int connect(int sockfd, const struct sockaddr* addr, socklen_t addrlen);
```

Аргументы аналогичны аргументам вызова `bind` с тем исключением, что в качестве аргумента `addr` передается адрес конечной точки, к которой происходит подключение.

Если сокет не был привязан к определенному адресу, он будет привязан автоматически в ходе вызова `connect`.

В момент `connect` процесс будет приостановлен до приема соединения, если соответствующий порт конечной точки открыт (т.е. к данной конечной точке вызовом `bind` был привязан некоторый сокет). Если *на принимающей стороне* порт закрыт (т.е. к данной конечной точке не привязан сокет) либо запрос на соединение явно отвергнут (об этом см. `listen`), вызов `connect` завершается с ошибкой.

Для установления соединения достаточно, чтобы два процесса с адресами А и В (то есть, для которых был заранее вызван `bind`), одновременно вызвали `connect(..., В, ...)` /

`connect(..., A, ...)`. Однако необходимость вызывать `bind` до `connect` порождает состояние гонки. Поэтому при организации обмена данными по сети используется другой способ.

При установлении соединения явно выделяют сторону, ожидающую соединение (сервер), и сторону, иницилирующую соединение (клиент).

На клиенте выполняется вызов `connect`.

На сервере выполняются 3 вызова: `bind`, `listen` и `connect`.

Замечание: в случае сокетов типа `SOCK_DGRAM` явное установление соединения невозможно в силу особенностей протокола UDP. В этом случае вызов `connect` всего лишь запретит прием/передачу данных с адресов, отличных от адреса, переданного в аргументе `addr`.

Вызов `listen` переключает сокет в режим приема соединений.

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

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

Аргумент `sockfd` предназначен для передач дескриптора сокета. Если сокет не был явно привязан к конечной точке, он будет привязан автоматически к некоторому свободному порту. Однако в этом случае будет необходимо получить реальный адрес конечной точки вызовом `sockopt`, что является лишним дополнительным действием.

Аргумент `backlog` содержит длину очереди запросов на соединение. В случае, если `backlog == 3`, и пришло 4 запроса на соединение, первые 3 запроса будут в очередь ожидания, а 4-ый будет отвергнут.

Принять запрос на соединение (и тем самым установить соединение) можно вызовом `accept`:

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

int fd = accept(int sockfd, sockaddr* addr, socklen_t* addrlen);
```

Аргумент `sockfd` предназначен для передачи дескриптора сокета, ожидающего подключения.

Аргументы `addr` и `addrlen` служат для передачи адресов, по которым будет записан адрес конечной точки, с которой установлено соединение. В качестве их значений допустимо одновременно передавать `NULL`. Если `addr` – не `NULL`, переменная по адресу `addrlen` должна содержать размер буфера по указателю в `addr` (чтобы ядру был известен максимальный размер данных, которые можно записать).

Вызов возвращает дескриптор **нового сокета**, связанного с установленным соединением и только с ним. Закрытие дескриптора «слушающего» сокета, передаваемого в `sockfd`, никак не скажется на возвращенном «соединенном» дескрипторе `fd`.

Если в очереди нет ожидающих запросов на соединение, вызов `accept` блокирует вызывающий поток до получения запроса на соединение.

По завершении работы с соединением уведомить противоположную сторону о прекращении приема/передачи данных можно вызовом `shutdown`.

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

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

Аргумент `sockfd` предназначен для передачи дескриптора сокета, связанного с соединением.

Аргумент `how` предназначен для передачи одной из констант, определяющих, какой тип передачи запрещается:

- `SHUT_RD` – запретить прием новых данных;
- `SHUT_WR` – запретить отправку новых данных;
- `SHUT_RDWR` – разорвать соединение.

Окончательно закрыть сокет (и разорвать соединение) можно вызовом `close`.

```
#include <unistd.h>

int close(int fd);
```

2.4. Передача данных

Передача данных осуществляется парой вызовов `send/recv`.

```
#include <sys/socket.h>
ssize_t send(int sockfd, const void* buf, size_t len, int flags);
ssize_t recv(int sockfd, void* buf, size_t len, int flags);
```

Аргумент `sockfd` предназначен для передачи дескриптора сокета, связанного с соединением.

Аргументы `buf` и `len` служат для передачи буфера с данными.

Аргумент `flags` служит для указания флагов, среди которых в рамках лабораторной может пригодиться флаг `MSG_WAITALL` (ждать получения ровно `len` данных в случае сокета типа `SOCK_STREAM`).

Вызовы возвращают число отправленных/полученных байт.

Замечание: сокеты типа `SOCK_DGRAM` и `SOCK_SEQPACKET` ориентированы на передачу отдельных сообщений, в отличие от `SOCK_STREAM`. Каждый вызов `send` отправляет ровно 1 сообщение, каждый вызов `recv` читает не более одного сообщения (хотя может прочесть только часть одного сообщения, если размер буфера недостаточен). Подробнее см. пример в приложении 1.

3. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Написать 2 программы – клиент и сервер – для игры в «угадай число». Клиент может быть автоматическим (игрок – машина, как в л/р2) или интерактивным (игрок – человек). На стороне сервера игрок – всегда машина.

В качестве параметра сервер принимает порт, на котором он будет работать.

Клиент в качестве параметра принимает адрес сервера.

Сервер должен вести лог — выводить в стандартный вывод сообщения об подключении/отключении клиентов, а также ход всех игр в формате <адрес клиента>:<сообщение>.

Варианты:

Легкий: однопоточный сервер (только 1 клиент может быть подключен), после отключения сервер ожидает новое подключение.

Средний: многопоточный сервер, для обработки нового соединения создается новый поток (потребуется синхронизация вывода в лог любым удобным методом).

Сложный: многопоточный сервер, для обработки нового соединения создается отдельный процесс (потребуется синхронизация вывода в лог + предотвращение появления процессов-зомби).

ПРИЛОЖЕНИЯ

Приложение 1. Пример приложения-сервера

```
#include "common.h"

char buffer[64];

int main() {
    auto server_address = local_addr(SERVER_PORT);
    auto listening_socket = check(make_socket(SOCKET_TYPE));
    int connected_socket = 0;

    check(bind(listening_socket, (sockaddr*)&server_address, sizeof(server_address)));
    check(listen(listening_socket, 2));

    while (true) {
        sockaddr_in connected_address{};
        socklen_t addrlen = sizeof(connected_socket);
        connected_socket = check(accept(listening_socket, (sockaddr*)&connected_address,
                                         &addrlen));

        std::cout << "Connected from " << connected_address << std::endl;

        while (true) {
            int size = recv(connected_socket, buffer, sizeof(buffer), MSG_WAITALL);

            if (size == 0 || (size < 0 && errno == ENOTCONN))
                break; //disconnected

            check(size);
            std::cout
                << connected_address
                << " send a message of a size " << size
                << ":" << std::string_view(buffer, size) << std::endl;
        }
        std::cout << "Disconnected from " << connected_address << std::endl;
    }
}
```

Приложение 2. Пример приложения-клиента.

```
#include "common.h"

int main() {
    auto dest_address = local_addr(SERVER_PORT);
    int sock_fd = check(make_socket(SOCKET_TYPE));
    check(connect(sock_fd, (sockaddr*)&dest_address, sizeof(dest_address)));

    std::string message;
    while (1) {
        std::getline(std::cin, message);
        if (message == "q")
            break;
        send(sock_fd, message.c_str(), message.size() + 1, MSG_WAITALL);
    }

    shutdown(sock_fd, SHUT_RDWR);
    close(sock_fd);
}
```

Приложение 3. Файл common.h

```
#include <sys/socket.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <iostream>
#include "check.hpp"

constexpr unsigned short SERVER_PORT = 60002;
constexpr int SOCKET_TYPE = SOCK_STREAM;

inline std::ostream& operator<<(std::ostream& s, const sockaddr_in& addr) {
    union {
        in_addr_t x;
        char c[sizeof(in_addr)];
    }t{};
    t.x = addr.sin_addr.s_addr;
    return s << std::to_string(int(t.c[0]))
        << "." << std::to_string(int(t.c[1]))
        << "." << std::to_string(int(t.c[2]))
        << "." << std::to_string(int(t.c[3]))
        << ":" << std::to_string(ntohs(addr.sin_port));
}

inline int make_socket(int type) {
    switch (type)
    {
        case SOCK_STREAM:
            return socket(AF_INET, SOCK_STREAM, 0);
        case SOCK_SEQPACKET:
            return check(socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)); // analogue to
SOCK_SEQPACKET
        default:
            errno = EINVAL;
            return -1;
    }
}

inline sockaddr_in local_addr(unsigned short port) {
    sockaddr_in addr{};
    addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK); //127.0.0.1
    addr.sin_port = htons(port);
    addr.sin_family = AF_INET;
    return addr;
}
```