# 7 задача Сеть

1. UDP - эхо сервер:
   a. Сделайте UDP-сервер, который принимает данные от клиентов и пересылает их обратно клиенту.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>



#define BUFFER_SIZE 1024

#define PORT 8888

int main() {

int sockfd;

struct sockaddr_in server_addr, client_addr;

socklen_t client_len;

char buffer[BUFFER_SIZE];

// Создаем UDP сокет

if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {

perror("socket creation failed");

exit(EXIT_FAILURE);

}

memset(&server_addr, 0, sizeof(server_addr));

memset(&client_addr, 0, sizeof(client_addr));

// Настраиваем адрес сервера

server_addr.sin_family = AF_INET;

server_addr.sin_addr.s_addr = INADDR_ANY;

server_addr.sin_port = htons(PORT);

// Привязываем сокет к адресу

if (bind(sockfd, (const struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {

perror("bind failed");
```

```
close(sockfd);

exit(EXIT_FAILURE);

}

printf("UDP Echo Server is listening on port %d...\n", PORT);

while (1) {

client_len = sizeof(client_addr);

// Получаем данные от клиента

ssize_t recv_len = recvfrom(sockfd, buffer, BUFFER_SIZE, 0,

(struct sockaddr *)&client_addr, &client_len);

if (recv_len < 0) {

perror("recvfrom failed");

continue;

}

buffer[recv_len] = '\0';

printf("Received from %s:%d: %s\n",

inet_ntoa(client_addr.sin_addr),

ntohs(client_addr.sin_port),

buffer);

// Отправляем данные обратно клиенту

if (sendto(sockfd, buffer, recv_len, 0,

(const struct sockaddr *)&client_addr, client_len) < 0) {

perror("sendto failed");

}

}

close(sockfd);

return 0;

}

//gcc udp_echo_server.c -o udp_server
```

**Сокеты** - это конечные точки для межпроцессного взаимодействия, позволяющие программам обмениваться данными по сети. В данном случае мы используем **UDP сокеты** (SOCK_DGRAM), которые работают без установления соединения.

## UDP (User Datagram Protocol)

UDP - это протокол транспортного уровня, который:

- Не требует установления соединения
- Не гарантирует доставку данных
- Не обеспечивает порядок доставки пакетов
- Имеет меньшие накладные расходы, чем TCP
- Идеален для приложений, где важна скорость, а не надежность (например, видеостриминг, DNS) ( ютуб на udp работает :0 )

---

## По коду:

### `<sys/socket.h>` - библиотека для (1) Основных функций и структур для работы с сокетами:

- Определяет типы сокетов ( `SOCK_STREAM` -надёжный потоковый (TCP) , `SOCK_DGRAM` - датаграммный (UDP))
- Содержит функции `socket()` , `bind()` , `listen()` , `accept()` , `connect()` , `send()` , `recv()`
- Определяет структуры `sockaddr` (общий адрес) и `sockaddr_in` (для IPv4)

# IPv4 сокет

```c
struct sockaddr_in {
    sa_family_t    sin_family;  // Семейство адресов (AF_INET)
    in_port_t      sin_port;    // Номер порта (в сетевом порядке байт)
    struct in_addr sin_addr;    // IP-адрес
    char           sin_zero[8]; // Дополнение до размера sockaddr
};

struct in_addr {
    uint32_t s_addr;  // Адрес в сетевом порядке байт
};
```

В памяти:

```
| 2 байта | 2 байта  | 4 байта       | 8 байт           |
|---------|----------|---------------|------------------|
| AF_INET | порт     | IP-адрес      | нули (padding)   |
```

# IPv6 сокет ( `sockaddr_in6` ):

```c
struct sockaddr_in6 {
    sa_family_t     sin6_family;   // AF_INET6 (2 байта)
    in_port_t       sin6_port;     // Порт (2 байта)
    uint32_t        sin6_flowinfo; // Flow information (4 байта)
```

```
    struct in6_addr sin6_addr;     // IPv6-адрес (16 байт)
    uint32_t        sin6_scope_id; // Scope ID (4 байта)
};

struct in6_addr {
    unsigned char s6_addr[16]; // 128-битный IPv6-адрес
};
```

## Unix-сокет

```
struct sockaddr_un {
    sa_family_t sun_family;    // AF_UNIX (1 байт)
    char        sun_path[108]; // Путь к сокету
};
```

## Функция `socket()`

```
int socket(int domain, int type, int protocol);
```

**Создаёт конечную точку связи (сокет)**:

**Параметры**:

| Параметр | Тип | Возможные значения |
|---|---|---|
| domain | `int` | `AF_INET` (IPv4), `AF_INET6` (IPv6) |
| type | `int` | `SOCK_STREAM` (TCP), `SOCK_DGRAM` (UDP) |
| protocol | `int` | Обычно 0 (автовыбор) |

- `domain` (домен):
  - `AF_INET` — IPv4
  - `AF_INET6` — IPv6
  - `AF_UNIX` — локальные сокеты (Unix)
- `type` (тип сокета):
  - `SOCK_STREAM` — надёжный потоковый (TCP)
  - `SOCK_DGRAM` — датаграммный (UDP)
  - `SOCK_RAW` — низкоуровневый доступ
- `protocol`:
  - Обычно `0` (автовыбор для указанного типа)
  - Например, `IPPROTO_TCP` для TCP

Пример:

```
int udp_socket = socket(AF_INET, SOCK_DGRAM, 0);  // UDP-сокет
int tcp_socket = socket(AF_INET, SOCK_STREAM, 0); // TCP-сокет
```

## Функция `bind()` - привязка сокета к адресу

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

**Параметры**:

| Параметр | Тип | Описание |
|---|---|---|
| sockfd | `int` | Файловый дескриптор сокета, возвращённый `socket()` |
| addr | `struct sockaddr*` | Указатель на структуру с адресом (для IPv4 - `sockaddr_in`) |
| addrlen | `socklen_t` | Размер структуры адреса (обычно `sizeof(struct sockaddr_in)` для IPv4) |

Пример:

```
struct sockaddr_in addr = {
    .sin_family = AF_INET,
    .sin_port = htons(8080),
    .sin_addr.s_addr = INADDR_ANY
};
bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
```

## Функция `listen()` - начало прослушивания (только TCP)

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

**Параметры**:

| Параметр | Тип | Описание |
|---|---|---|
| sockfd | `int` | Файловый дескриптор TCP-сокета (тип `SOCK_STREAM`) |
| backlog | `int` | Максимальная длина очереди ожидающих соединений (типичные значения 5-10) |

**Пример**:

```
listen(sockfd, 5); // Очередь на 5 соединений
```

## Функция `accept()` - принятие соединения (только TCP)

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

**Параметры**:

| Параметр | Тип | Описание |
|---|---|---|
| sockfd | `int` | Дескриптор слушающего сокета |
| addr | `struct sockaddr*` | Указатель, куда будет записан адрес клиента (может быть NULL) |

| Параметр | Тип | Описание |
|---|---|---|
| addrlen | `socklen_t*` | Указатель на переменную с размером структуры addr (входное и выходное) |

**Пример**:

```
struct sockaddr_in client_addr;
socklen_t client_len = sizeof(client_addr);
int client_sock = accept(sockfd, (struct sockaddr*)&client_addr, &client_len);
```

## Функция `connect()` - подключение к серверу

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

**Параметры**:

| Параметр | Тип | Описание |
|---|---|---|
| sockfd | `int` | Дескриптор сокета |
| addr | `struct sockaddr*` | Указатель на структуру с адресом сервера |
| addrlen | `socklen_t` | Размер структуры адреса |

**Пример**:

```
connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr));
```

## Функции `send()` / `write()` - отправка данных (TCP)

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t write(int sockfd, const void *buf, size_t len);
```

**Параметры**:

| Параметр | Тип | Описание |
|---|---|---|
| sockfd | `int` | Дескриптор подключённого TCP-сокета |
| buf | `const void*` | Указатель на данные для отправки |
| len | `size_t` | Размер данных в байтах |
| flags | `int` | Флаги (например, `MSG_NOSIGNAL`) |

## Функции `recv()` / `read()` - прием данных (TCP)

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

**Параметры**:

| Параметр | Тип | Описание |
|---|---|---|
| sockfd | `int` | Дескриптор подключённого TCP-сокета |
| buf | `const void*` | буфер для приёма данных |
| len | `size_t` | Размер данных в байтах |
| flags | `int` | Флаги (например, `MSG_NOSIGNAL`) |

## Функция `sendto()` - отправка (UDP)

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

**Параметры**:

| Параметр | Тип | Описание |
|---|---|---|
| sockfd | `int` | Дескриптор подключённого TCP-сокета |
| buf | `const void*` | Указатель на данные для отправки |
| len | `size_t` | Размер данных в байтах |
| flags | `int` | Флаги (например, `MSG_NOSIGNAL`) |
| dest_addr | struct sockaddr* | Адрес получателя |
| addrlen | socklen_t* | Указатель на размер структуры адреса получателя |

## Функция `recvfrom()` - прием (UDP)

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

**Параметры**:

| Параметр | Тип | Описание |
|---|---|---|
| sockfd | `int` | Дескриптор подключённого TCP-сокета |
| buf | `const void*` | Указатель на данные для отправки |
| len | `size_t` | Размер данных в байтах |
| flags | `int` | Флаги (например, `MSG_NOSIGNAL`) |
| src_addr | struct sockaddr* | Буфер для адреса отправителя |
| addrlen | socklen_t* | Указатель на размер структуры адреса |

## Функции `close()` / `shutdown()`

```
int close(int sockfd); // Полное закрытие
int shutdown(int sockfd, int how); // Частичное закрытие (how: SHUT_RD, SHUT_WR, SHUT_RDWR)
```

**Параметры**:

| Параметр | Тип | Описание |
|---|---|---|
| sockfd | `int` | Дескриптор подключённого TCP-сокета |
| **Параметр how**:<br><br>- `SHUT_RD` - закрыть на чтение<br><br>- `SHUT_WR` - закрыть на запись<br><br>- `SHUT_RDWR` - закрыть полностью | | |

Все эти функции возвращают `0` при успехе и `-1` при ошибке (кроме `socket()`, `accept()`, `recv()`, `send()` и др., которые возвращают специальные дескрипторы/размеры).

## Сравнение TCP и UDP функций

| Функция | TCP | UDP | Примечание |
|---|---|---|---|
| `socket()` | Да | Да | С разными типами ( `SOCK_STREAM` / `DGRAM` ) |
| `bind()` | Да | Да | Обязательно для сервера |
| `listen()` | Да | Нет | Только для TCP |
| `accept()` | Да | Нет | Только для TCP |
| `connect()` | Да | Опц. | В UDP задает адрес по умолчанию |
| `send()` | Да | Нет | |
| `recv()` | Да | Нет | |
| `sendto()` | Нет | Да | |
| `recvfrom()` | Нет | Да | |

## Примеры использования

**TCP сервер**:

```
socket() -> bind() -> listen() -> accept() -> recv()/send() -> close()
```

**TCP клиент**:

```
socket() -> connect() -> send()/recv() -> close()
```

**UDP сервер**:

```
socket() -> bind() -> recvfrom()/sendto() -> close()
```

**UDP клиент**:

```
socket() -> [connect()] -> sendto()/recvfrom() -> close()
```

(2) **Интернет-протоколы (IP) и порты**:

- Определяет структуры для работы с IPv4 ( `sockaddr_in` )
- Содержит константы ( `INADDR_ANY` , `INADDR_LOOPBACK` )
- Определяет функции для работы с портами ( `htons()` , `ntohs()` )

## Структура `sockaddr_in` (IPv4)

Полное определение структуры:

```
struct sockaddr_in {
    sa_family_t    sin_family;  // Семейство адресов (AF_INET)
    in_port_t      sin_port;    // Номер порта (сетевой порядок)
    struct in_addr sin_addr;    // IP-адрес
    unsigned char  sin_zero[8]; // Дополнение (нули)
};

struct in_addr {
    uint32_t s_addr; // 32-битный IPv4-адрес
};
```

## Поля структуры:

| Поле | Размер | Описание |
|---|---|---|
| `sin_family` | 2 байта | Всегда `AF_INET` для IPv4 |
| `sin_port` | 2 байта | Номер порта в **сетевом порядке байт** (используйте `htons()` !) |
| `sin_addr` | 4 байта | Структура с IP-адресом (сетевой порядок) |
| `sin_zero` | 8 байт | Дополнение (всегда обнуляйте через `memset()` !) |

## Пример заполнения:

```
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr)); // Важно! ниже написано почему!
addr.sin_family = AF_INET;
addr.sin_port = htons(8080); // Порт 8080
inet_pton(AF_INET, "192.168.1.1", &addr.sin_addr);
```

## Ключевые константы

### `INADDR_ANY`

```
#define INADDR_ANY ((in_addr_t)0x00000000)
```

- Специальный адрес `0.0.0.0`
- **Использование**: Сервер будет слушать все доступные сетевые интерфейсы

```
addr.sin_addr.s_addr = htonl(INADDR_ANY); // Лучше чем INADDR_ANY напрямую
```

### `INADDR_LOOPBACK`

```
#define INADDR_LOOPBACK ((in_addr_t)0x7f000001)
```

- Соответствует `127.0.0.1`
- **Использование**: Для локальных соединений( на одном устройстве)

```
addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
```

## Другие полезные константы:

- `INADDR_BROADCAST` ( `255.255.255.255` ) - широковещательный адрес
- `INADDR_NONE` ( `0xffffffff` ) - индикатор ошибки

## Функции работы с портами

### `htons()` (Host TO Network Short)

```
uint16_t htons(uint16_t hostshort);
```

- Преобразует 16-битное число (порт) из хостового порядка в сетевой
- **Обязательно** используйте для портов!

```
uint16_t port = 8080;
addr.sin_port = htons(port); // Правильно
// addr.sin_port = 8080;     // ОШИБКА! Порядок байт может быть неправильным
```

### `ntohs()` (Network TO Host Short)

```
uint16_t ntohs(uint16_t netshort);
```

- Обратное преобразование из сетевого порядка в хостовый

```
uint16_t port = ntohs(addr.sin_port);
```

## Аналоги для 32-битных чисел (IP-адреса):

- `htonl()` - для `uint32_t`
- `ntohl()` - обратное преобразование

# Как выглядит преобразование порядка байт( зачем что-то преобразовывать?)

Представим порт `8080` (в HEX: `0x1F90` ):

## 1. Хостовой порядок (зависит от процессора)

- **Little-endian** (x86, ARM): `0x90 0x1F` (младший байт сначала)
- **Big-endian**: `0x1F 0x90` (старший байт сначала)

### 2. Сетевой порядок (стандарт Big-endian)

Всегда: `0x1F 0x90`

### Функция `htons()` делает:

```
uint16_t htons(uint16_t hostshort) {
    return ((hostshort & 0xFF00) >> 8) | ((hostshort & 0x00FF) << 8);
}
```

Пример для порта 8080 ( `0x1F90` ):

```
uint16_t host_port = 0x1F90; // 8080 в хостовом порядке (Little-endian)
uint16_t net_port = htons(host_port); // Станет 0x901F (но это то же самое число 8080 в Big-endian)
```

## Визуализация

| Порядок | Байт 1 | Байт 2 | Как выглядит в памяти |
|---|---|---|---|
| Хостовой (LE) | 0×90 | 0×1F | `90 1F` |
| Сетевой (BE) | 0×1F | 0×90 | `1F 90` |

## Где возникает ошибка

1. **Прямое присвоение без `htons()`**:

   ```
   addr.sin_port = 8080; // ОШИБКА! Порядок байт неправильный
   ```

2. **На разных архитектурах**:
   - На x86 (Little-endian) без `htons()`:
     - Отправим: `90 1F` (неправильно)
     - Сервер прочитает как `0x901F` = 36895 (вместо 8080)
3. **При ручном разборе пакетов**:

   ```
   uint16_t port = *((uint16_t*)packet); // ОШИБКА: нужно ntohs()
   ```

## Практический пример

**Правильно:**

```
struct sockaddr_in addr;
addr.sin_port = htons(8080); // Всегда корректный сетевой порядок
```

**Ошибка (что будет):**

```
addr.sin_port = 8080;
// На x86 отправит 0x901F вместо 0x1F90
// Удалённая сторона получит 36895 вместо 8080
```

# Как проверить порядок байт

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

int main() {
    uint16_t port = 0x1234;
    printf("Хостовой порядок: %04x\n", port);
    printf("Сетевой порядок:  %04x\n", htons(port));

    if (htons(0x1234) == 0x3412) {
        printf("Система Little-endian\n");
    } else {
        printf("Система Big-endian\n");
    }
}
```

# Почему стандарт выбрал Big-endian

Исторически сетевые протоколы используют Big-endian, потому что:

1. Естественный порядок чтения чисел (слева направо)
2. Первые сетевые компьютеры (Sun, Motorola) были Big-endian
3. Упрощает визуальный анализ дампов пакетов

Всегда используйте:

- `htons()` - при записи порта в структуры
- `ntohs()` - при чтении порта из сетевых пакетов

Без этих функций **будет работать неправильно** на архитектурах с другим порядком байт!!!!!!!!!!

## Функции работы с IP-адресами

## `inet_pton()` (Presentation TO Network)

```
int inet_pton(int af, const char *src, void *dst);
```

- Преобразует текстовый IP в бинарный формат

- **Параметры**:
  - `af` : `AF_INET` (IPv4) или `AF_INET6` (IPv6)
  - `src` : Строка с IP ("192.168.1.1")
  - `dst` : Указатель на `struct in_addr`

```
inet_pton(AF_INET, "192.168.1.1", &addr.sin_addr);
```

### `inet_ntop()` (Network TO Presentation)

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

- Обратное преобразование (бинарный → текст)

```
char ip_str[INET_ADDRSTRLEN];
inet_ntop(AF_INET, &addr.sin_addr, ip_str, INET_ADDRSTRLEN);
printf("IP: %s\n", ip_str); // "192.168.1.1"
```

### Порядок байт (Endianness)

Проблема, которую решают `htons` / `ntohs` :

| Архитектура | Порядок байт | Пример числа 0x1234 |
|---|---|---|
| x86 (Little-endian) | Младший байт первый | 0×34 0×12 |
| Сетевой (Big-endian) | Старший байт первый | 0×12 0×34 |

**Почему это важно**:

- Без преобразования клиент и сервер на разных машинах будут по-разному интерпретировать порты

# Типичные ошибки

1. **Забыли `htons()` для порта**:

   ```
   addr.sin_port = 8080; // ОШИБКА! Должно быть htons(8080)
   ```

2. **Не обнулили структуру**:

   ```
   struct sockaddr_in addr; // Мусор в sin_zero
   // Всегда используйте memset()!
   ```

3. **Путаница с порядком байт**:

   ```
   uint32_t ip = 0x1020304; // Хостовой порядок
   addr.sin_addr.s_addr = ip; // ОШИБКА! Нужен сетевой порядок
   ```

b. Напишите UDP-клиента, для теста UDP-сервера.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define BUFFER_SIZE 1024
#define PORT 8888

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <server_ip>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int sockfd;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];

    // Создаем UDP сокет
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));

    // Настраиваем адрес сервера
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    if (inet_pton(AF_INET, argv[1], &server_addr.sin_addr) <= 0) {
        perror("invalid address");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("UDP Echo Client connected to %s:%d\n", argv[1], PORT);

    while (1) {
        printf("Enter message (or 'exit' to quit): ");
        fgets(buffer, BUFFER_SIZE, stdin);

        // Удаляем символ новой строки
        buffer[strcspn(buffer, "\n")] = '\0';

        if (strcmp(buffer, "exit") == 0) {
            break;
        }

        // Отправляем сообщение серверу
        if (sendto(sockfd, buffer, strlen(buffer), 0,
                   (const struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
            perror("sendto failed");
            continue;
        }

        // Получаем ответ от сервера
        socklen_t server_len = sizeof(server_addr);
        ssize_t recv_len = recvfrom(sockfd, buffer, BUFFER_SIZE, 0,
                                    (struct sockaddr *)&server_addr, &server_len);
```

```
        if (recv_len < 0) {
            perror("recvfrom failed");
            continue;
        }

        buffer[recv_len] = '\0';
        printf("Echo from server: %s\n", buffer);
    }

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

Этот код представляет собой **UDP-клиент для эхо-сервера**. Давайте разберём его работу пошагово:

---

# 1. Инициализация клиента

```
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
```

- Создаётся **UDP-сокет** (SOCK_DGRAM) для IPv4 (AF_INET)
- При ошибке выводится сообщение и программа завершается

---

# 2. Настройка адреса сервера

```
struct sockaddr_in server_addr;
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT);
inet_pton(AF_INET, argv[1], &server_addr.sin_addr);
```

- `memset` : обнуление структуры (важно для поля `sin_zero` )
- Установка:
  - Семейства адресов (IPv4)
  - Порта (с преобразованием в сетевой порядок через `htons()` )
  - IP-адреса сервера (из аргумента командной строки)

---

# 3. Основной цикл работы

```
while (1) {
    // Ввод сообщения
    printf("Enter message...");
    fgets(buffer, BUFFER_SIZE, stdin);
    buffer[strcspn(buffer, "\n")] = '\0'; // Удаление '\n'

    if (strcmp(buffer, "exit") == 0) break;
```

- Пользователь вводит сообщение

- Удаляется символ новой строки ( `\n` )
- При вводе "exit" цикл прерывается

---

## 4. Отправка данных серверу

```
sendto(sockfd, buffer, strlen(buffer), 0,
      (struct sockaddr*)&server_addr, sizeof(server_addr));
```

- Отправка сообщения на сервер через UDP
- Параметры:
  - `sockfd` : дескриптор сокета
  - `buffer` : данные для отправки
  - `server_addr` : адрес сервера

---

## 5. Получение ответа

```
recvfrom(sockfd, buffer, BUFFER_SIZE, 0,
        (struct sockaddr*)&server_addr, &server_len);
```

- Ожидание ответа от сервера
- Блокирующая операция (программа ждёт, пока не придёт ответ)
- Ответ помещается в `buffer`

---

## 6. Завершение работы

```
close(sockfd);
```

- Закрытие сокета при завершении программы

---

## Как это работает на практике?

1. **Запуск клиента**:

   ```
   ./udp_client 127.0.0.1
   ```

2. **Пример сеанса**:

   ```
   Enter message (or 'exit' to quit): Hello
   Echo from server: Hello
   Enter message (or 'exit' to quit): Test
   ```

```
Echo from server: Test
Enter message (or 'exit' to quit): exit
```

---

## Что делает сервер для этого клиента?

Сервер (который должен быть запущен отдельно):

1. Получает сообщение через `recvfrom`
2. Отправляет его обратно через `sendto` тому же адресу

---

## Важные моменты:

1. **Безопасность**:
   - Нет проверки длины ввода (возможно переполнение буфера)
   - Нет таймаутов на `recvfrom` (может висеть вечно)
2. **Порядок байт**:
   - `htons()` гарантирует правильный порядок порта
3. **Универсальность**:
   - Работает с любым UDP-сервером на указанном IP/порту

c. Проверьте, что UDP-сервер, работает с несколькими клиентами.

```
gcc udp_echo_server.c -o udp_server

gcc udp_echo_client.c -o udp_client

./udp_client 127.0.0.1
./udp_client 127.0.0.1
```

2. TCP - эхо сервер:
   a. Сделайте TCP-сервер, который принимает соединения от клиентов на заданном ip и port.
   b. создает новый процесс, в котором:
   i.
   читает данные от клиента;
   ii.
   пересылает их ему обратно.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/wait.h>
```

```c
#define BUFFER_SIZE 1024
#define PORT 8888

void handle_client(int client_sock) {
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;

    while ((bytes_read = read(client_sock, buffer, BUFFER_SIZE - 1)) > 0) {
        buffer[bytes_read] = '\0';
        printf("Received: %s\n", buffer);

        // Отправляем данные обратно клиенту
        if (write(client_sock, buffer, bytes_read) < 0) {
            perror("write failed");
            break;
        }
    }

    if (bytes_read < 0) {
        perror("read failed");
    }

    printf("Client disconnected\n");
    close(client_sock);
    exit(0);
}

int main() {
    int server_sock, client_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len;
    pid_t pid;

    // Создаем TCP сокет
    if ((server_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));
    memset(&client_addr, 0, sizeof(client_addr));

    // Настраиваем адрес сервера
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Привязываем сокет к адресу
    if (bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("bind failed");
        close(server_sock);
        exit(EXIT_FAILURE);
    }

    // Начинаем слушать соединения
    if (listen(server_sock, 5) < 0) {
        perror("listen failed");
        close(server_sock);
        exit(EXIT_FAILURE);
    }
```

```
    printf("TCP Echo Server is listening on port %d...\n", PORT);

    while (1) {
        client_len = sizeof(client_addr);

        // Принимаем новое соединение
        if ((client_sock = accept(server_sock, (struct sockaddr *)&client_addr, &client_len)) <
0) {
            perror("accept failed");
            continue;
        }

        printf("New connection from %s:%d\n",
               inet_ntoa(client_addr.sin_addr),
               ntohs(client_addr.sin_port));

        // Создаем новый процесс для обработки клиента
        pid = fork();
        if (pid < 0) {
            perror("fork failed");
            close(client_sock);
            continue;
        }

        if (pid == 0) { // Дочерний процесс
            close(server_sock); // Закрываем слушающий сокет в дочернем процессе
            handle_client(client_sock);
        } else { // Родительский процесс
            close(client_sock); // Закрываем клиентский сокет в родительском процессе
            // Очищаем завершенные дочерние процессы
            while (waitpid(-1, NULL, WNOHANG) > 0);
        }
    }

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

c. Напишите TCP-клиента для проверки TCP-сервера.
d. Проверьте, что TCP-сервер работает с несколькими клиентами и обрабатывает сессии в разных процессах.

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

#define BUFFER_SIZE 1024
#define PORT 8888

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <server_ip>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

```c
    int sockfd;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];

    // Создаем TCP сокет
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));

    // Настраиваем адрес сервера
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    if (inet_pton(AF_INET, argv[1], &server_addr.sin_addr) <= 0) {
        perror("invalid address");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    // Устанавливаем соединение с сервером
    if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("connection failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("TCP Echo Client connected to %s:%d\n", argv[1], PORT);

    while (1) {
        printf("Enter message (or 'exit' to quit): ");
        fgets(buffer, BUFFER_SIZE, stdin);

        // Удаляем символ новой строки
        buffer[strcspn(buffer, "\n")] = '\0';

        if (strcmp(buffer, "exit") == 0) {
            break;
        }

        // Отправляем сообщение серверу
        if (write(sockfd, buffer, strlen(buffer)) < 0) {
            perror("write failed");
            continue;
        }

        // Получаем ответ от сервера
        ssize_t bytes_read = read(sockfd, buffer, BUFFER_SIZE - 1);
        if (bytes_read < 0) {
            perror("read failed");
            continue;
        }

        buffer[bytes_read] = '\0';
        printf("Echo from server: %s\n", buffer);
    }

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

3. Реализуйте задачу из пункта 2 при помощи мультиплексирования ввода-вывода poll(2)/select(2).