

Сети ЭВМ и телекоммуникации

Ю. С. Арсёнов

24 декабря 2014 г.

Глава 1

Задание

Задание: разработать приложение–клиент и приложение–сервер электронного магазина. Товар в электронном магазине имеет уникальный идентификатор, именование, цену.

1.1 Функциональные требования

Серверное приложение должно реализовывать следующие функции:

1. Прослушивание определенного порта
2. Обработка запросов на подключение по этому порту от клиентов
1. электронного магазина
2. Поддержка одновременной работы нескольких клиентов электронного магазина через механизм нитей
3. Прием запросов на добавление или покупку товара
4. Осуществление добавления товара, учет количества единиц товара
5. Передача клиенту электронного магазина информации о товарах и
6. подтверждений о совершении покупки
7. Обработка запроса на отключение клиента
8. Принудительное отключение клиента

Клиентское приложение должно реализовывать следующие функции:

1. Установление соединения с сервером
2. Передача запросов о добавлении, покупке товаров серверу
3. Получение ответов на запросы от сервера
4. Разрыв соединения
5. Обработка ситуации отключения клиента сервером

1.2 Нефункциональные требования

Для сервера:

1. Прослушивание определенного порта
2. Поддержка одновременной работы нескольких клиентов через механизм нитей
3. Обработка запросов на подключение по этому порту от клиентов

Для клиента:

1. Установление соединения с сервером

1.3 Настройки приложений

Разработанное клиентское приложение должно предоставлять пользователю просмотр каталога товаров интернет-магазина и их покупку. Разработанное серверное приложение должно хранить количество доступных товаров на складе и автоматически учитывать их при покупке.

1.4 Накладываемые ограничения

В данной реализации пакеты для обмена по протоколу TCP будут иметь размер 256 байт. В начале пакета указывается длина всей посылки, для этого выделяется 4 байта. Таким образом на саму информационную часть сообщения остается 252 байта, что вполне достаточно для стабильной работы электронного магазина. Минимальное значение длины текста – 0. Максимальное – $2^{32}-1$. Таким образом, максимальный теоретический размер текста – 4 Гбайт.

Глава 2

Реализация для работы по протоколу TSP

2.1 Прикладной протокол

Клиент отправляет команды в виде строки с текстом:

Показать все доступные в магазине товары	show index
Показать содержимое корзины	show cart
Купить товар под номером N	buy N
Добавить в каталог товар с именем N по цене P в количестве C	add N P C
Закрыть сессию	close

2.2 Архитектура приложения

Взаимодействие сервера и клиента:

1. Сервер отправляет приветственное сообщение “Please enter the command :” в котором клиенту предлагается ввести команду.
2. Клиент отправляет серверу какую-либо из команд.
3. Сервер отсылает клиенту информацию в соответствии с введенной командой.
4. Клиент производит дальнейшее взаимодействие с сервером по средством ввода команд из списка выше.
5. Клиент завершает работу с сервером.

TSP – протокол с обеспечением надежности передачи. TSP гарантирует, что данные не потеряются в пути, придут в правильном порядке и не придут дважды. Однако, так как TSP – потоковый протокол, могут возникнуть следующие проблемы:

- Сервер или клиент может считать лишь часть сообщения. Таким образом, для чтения одного сообщения может потребоваться несколько вызовов команды чтения.
- Один вызов команды чтения может вернуть несколько сообщений.

Есть разные варианты решения этой проблемы. Ниже приведены некоторые из них:

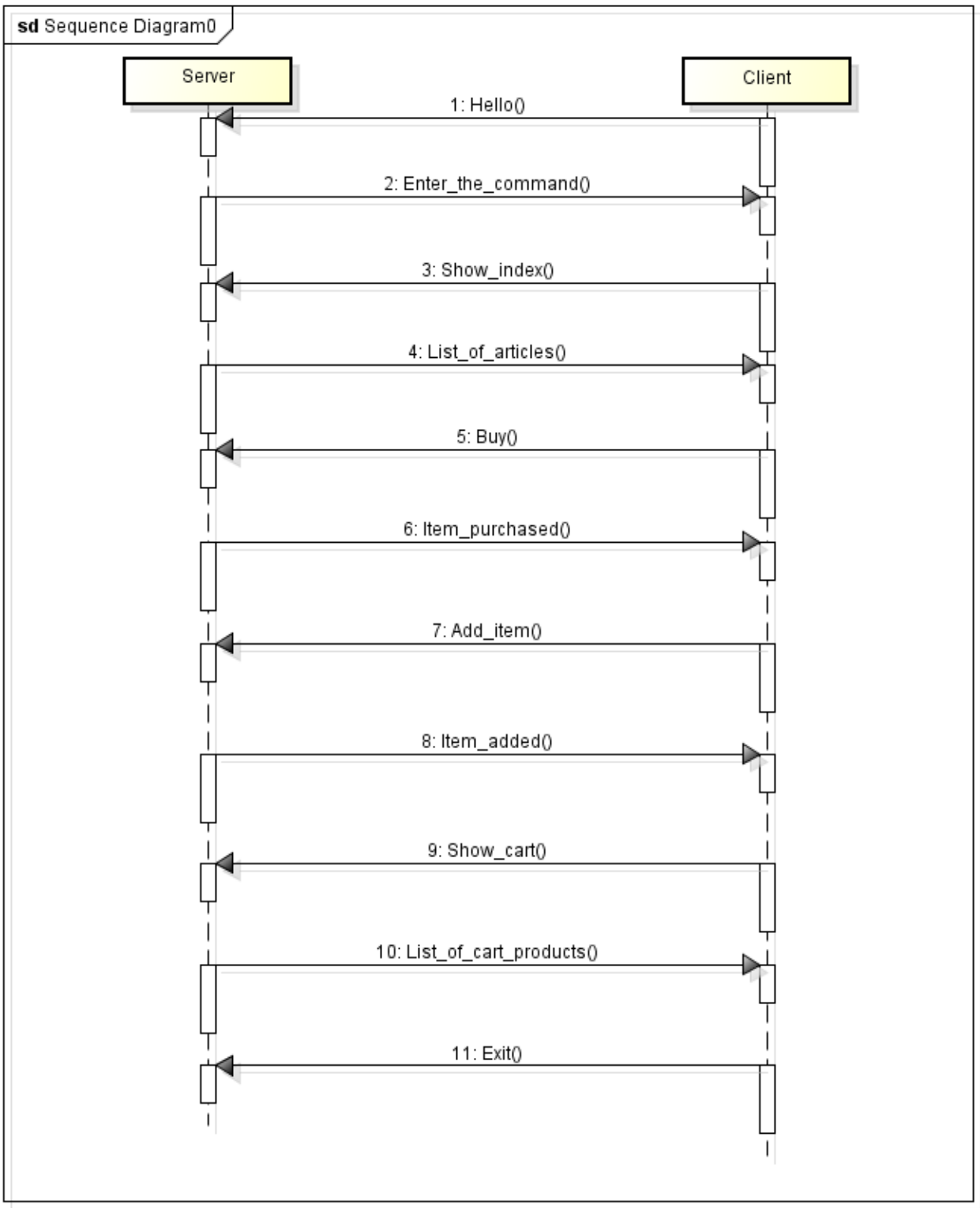
Способ решения	Недостатки
Использовать протокол с фиксированной длиной сообщения	Малая гибкость
Добавлять в конец сообщения разделитель (например, перенос строки)	Необходимость посимвольной обработки входящего сообщения
В начале сообщения указывать его длину.	Ограничение на максимальную длину сообщения, неэффективность в случае коротких сообщений

Был выбран третий способ.

Формат сообщения для TCP:

4 байта	0 байт – 4 Гбайта
Длина текста	Текст сообщения

Sequence диаграмма, демонстрирующая возможное взаимодействие клиента и сервера:



2.3 Тестирование

2.3.1 Описание тестового стенда и методики Тестирования

Тестирование проводилось на виртуальной машине Debian 7.4. Было запущено приложение сервера, затем - несколько приложений клиента. Таким образом сервер и клиент работали на одном компьютере.

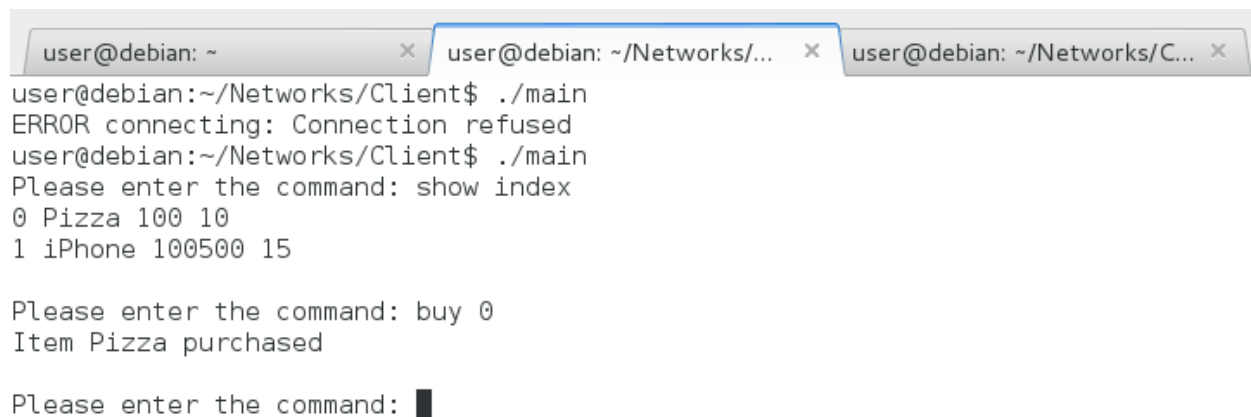
2.3.2 Тестовый план и результаты тестирования

1. Запустим первого клиента и с помощью команды “show index” попросим его вывести список доступных товаров для покупки.
2. Запустим второго клиента и с помощью команды “show index” так же попросим его вывести список доступных товаров для покупки.
3. Для обоих клиентов сервер дал ответ, что на складе есть два товара: Pizza – 10 штук, iPhone – 15 штук.




```
user@debian: ~  
user@debian: ~/Networks/Client$ ./main  
Please enter the command: show index  
0 Pizza 100 10  
1 iPhone 100500 15  
  
Please enter the command: █
```

4. Купим с первого клиента один товар “Pizza” :



```
user@debian: ~  
user@debian: ~/Networks/Client$ ./main  
ERROR connecting: Connection refused  
user@debian: ~/Networks/Client$ ./main  
Please enter the command: show index  
0 Pizza 100 10  
1 iPhone 100500 15  
  
Please enter the command: buy 0  
Item Pizza purchased  
  
Please enter the command: █
```

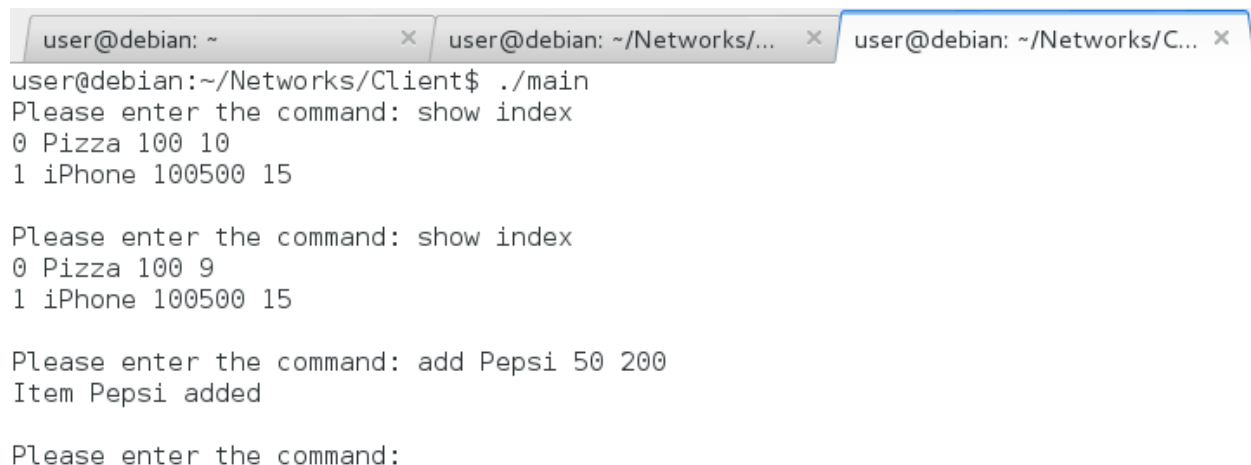
5. Теперь посмотрим со второго клиента список доступных товаров:



```
user@debian: ~  
user@debian: ~/Networks/Client$ ./main  
Please enter the command: show index  
0 Pizza 100 10  
1 iPhone 100500 15  
  
Please enter the command: show index  
0 Pizza 100 9  
1 iPhone 100500 15  
  
Please enter the command:
```

Товаров “Pizza” стало 9 штук, вместо 10.

6. С помощью второго клиента добавим в каталог товаров новый товар:



```
user@debian: ~  
user@debian: ~/Networks/Client$ ./main  
Please enter the command: show index  
0 Pizza 100 10  
1 iPhone 100500 15  
  
Please enter the command: show index  
0 Pizza 100 9  
1 iPhone 100500 15  
  
Please enter the command: add Pepsi 50 200  
Item Pepsi added  
  
Please enter the command:
```

7. Попросим первого клиента вывести список доступных товаров. В нем мы увидим новый товар, добавленный со второго клиента:

```
user@debian: ~ user@debian: ~/Networks/... user@debian: ~/Networks/C...
user@debian:~/Networks/Client$ ./main
ERROR connecting: Connection refused
user@debian:~/Networks/Client$ ./main
Please enter the command: show index
0 Pizza 100 10
1 iPhone 100500 15

Please enter the command: buy 0
Item Pizza purchased

Please enter the command: show index
0 Pizza 100 9
1 iPhone 100500 15
2 Pepsi 50 200

Please enter the command:
```

8. Осуществим выход из обоих клиентов.

Сервер корректно работает с несколькими клиентами. При вводе некорректных команд клиентское приложение сообщает об этом пользователю и продолжает работу. Серверное приложение так же выводит на консоль некорректные команды от пользователя.

Тестирование работы сервера с большим количеством клиентов:

1. Был запущен сервер.
2. Из различных терминалов было осуществлено подключение 5 клиентов.
3. Для каждого из клиентов были осуществлены операции показа каталога товаров, покупки и добавления новых товаров в каталог.
4. Для каждого из клиентов была проверена корректность работы.
5. Был осуществлен выход клиентами.

Сервер корректно обрабатывает запросы от клиентов и ведет учет купленных и добавленных товаров, а клиенты используют всю функциональность сервера без ограничений.

Глава 3

Реализация для работы по протоколу UDP

3.1 Прикладной протокол

Прикладной протокол понес небольшие изменения по сравнению с UDP. Однако, для пользователя взаимодействие с сервером осалось аналогичным.

3.2 Архитектура приложения

Взаимодействие сервера и клиента:

1. Клиент отправляет серверу приветственное сообщение для начала взаимодействия.
2. Сервер отправляет приветственное сообщение “Please enter the command :” в котором клиенту предлагается ввести команду.
3. Клиент отправляет серверу какую-либо из команд.
4. Сервер отсылает клиенту информацию в соответствии с введенной командой.
5. Клиент производит дальнейшее взаимодействие с сервером по средством ввода команд из списка выше.
6. Клиент завершает работу с сервером.

UDP – протокол без обеспечения надежности. Таким образом, прикладной протокол должен решать следующие проблемы:

- Данные могут потеряться в пути
- Данные могут прийти в неправильном порядке
- Данные могут прийти дважды

Для решения этих проблем в каждое сообщение добавляется уникальный номер. Первый запрос содержит номер 0, каждый следующий запрос использует номер на единицу больше. При этом ответ содержит номер соответствующего запроса.

Другая проблема – UDP – это протокол без установления соединения, для обработки нескольких клиентов используется один сокет. Поэтому сервер должен уметь самостоятельно различать клиентов. Есть два варианта:

1. Различать клиентов на основе адреса и порта, откуда пришел запрос.
2. В каждом сообщении указывать номер сессии.

В данном протоколе используется второй вариант. Сессии выделяются сервером. Первый запрос содержит сессию равную -1, в ходе формирования ответа на такой запрос сервер создает новую сессию и указывает её номер в ответе. Последующие запросы используют назначенный номер сессии.

UDP-дейтаграммы всегда приходят целиком, поэтому нет необходимости указывать длину сообщения, как это было в случае TCP.

Формат сообщения для UDP:

4 байта	4 байта	0 – 65499 байт
Номер сообщения	Номер сессии	Текст сообщения

3.3 Тестирование

2.3.1 Описание тестового стенда и методики Тестирования

Тестирование проводилось на виртуальной машине Debian 7.4. Было запущено приложение сервера, затем - несколько приложений клиента. Таким образом сервер и клиент работали на одном компьютере.

2.3.2 Тестовый план и результаты тестирования

1. Запустим первого клиента и с помощью команды “show index” попросим его вывести список доступных товаров для покупки.
2. Запустим второго клиента и с помощью команды “show index” так же попросим его вывести список доступных товаров для покупки.
3. Для обоих клиентов сервер дал ответ, что на складе есть два товара: Pizza – 10 штук, iPhone – 15 штук.

```
show index
>> 0 0
<< Received message with num 0 session 0
<< 0 Pizza 100 10
1 iPhone 100500 15
```

4. Купим с первого клиента один товар “Pizza” :

```
ClientUDP Default [C/C++ Application] /home/user/Networks
show index
>> 0 0
<< Received message with num 0 session 0
<< 0 Pizza 100 10
1 iPhone 100500 15

buy 0
|>> 1 0
<< Received message with num 1 session 0
<< Item Pizza purchased
```

5. Теперь посмотрим со второго клиента список доступных товаров:

```
show index
>> 0 0
<< Received message with num 0 session 1
<< 0 Pizza 100 9
1 iPhone 100500 15
```

Товаров “Pizza” стало 9 штук, вместо 10.

6. С помощью второго клиента добавим в каталог товаров новый товар:

```
show index
>> 0 0
<< Received message with num 0 session 1
<< 0 Pizza 100 9
1 iPhone 100500 15

add Fanta 50 400
>> 1 0
<< Received message with num 1 session 1
<< Item Fanta added
```

7. Попросим первого клиента вывести список доступных товаров. В нем мы увидим новый товар, добавленный со второго клиента:

```
show index
>> 0 0
<< Received message with num 2 session 0
<< 0 Pizza 100 9
1 iPhone 100500 15
2 Fanta 50 400
```

8. Осуществим выход из обоих клиентов.

Сервер корректно работает с несколькими клиентами. При вводе некорректных команд клиентское приложение сообщает об этом пользователю и продолжает работу. Серверное приложение так же выводит на консоль некорректные команды от пользователя.

Тестирование работы сервера с большим количеством клиентов:

1. Был запущен сервер.
2. Из различных терминалов было осуществлено подключение 5 клиентов.
3. Для каждого из клиентов были осуществлены операции показа каталога товаров, покупки и добавления новых товаров в каталог.
4. Для каждого из клиентов была проверена корректность работы.
5. Был осуществлен выход клиентами.

Сервер корректно обрабатывает запросы от клиентов и ведет учет купленных и добавленных товаров, а клиенты используют всю функциональность сервера без ограничений.

Тестирование на сбой:

При вводе слишком большой команды, сервер её обрабатывает, как 2 разные команды независимо друг от друга. Это случается, потому что максимально возможная длина сообщения ограничена 253 символами.

[illegible]

Т.е. теоритически возможная максимальная длина команды 253 символа.

[illegible]

Аналогично будет и с принятыми пакетами, Если длина принятого сообщения больше 253 символов, то оно просто обрежется.

Глава 4

Выводы

В ходе работы я спроектировал протокол прикладного уровня и написал две реализации клиента и сервера – для TCP и UDP. В большинстве случаев более оправдан выбор TCP за счет обеспечения надежности передачи. Также TCP осуществляет контроль порядка доставки сообщений и данные считываются как единый поток байтов. В TCP для взаимодействия с несколькими клиентами серверу необходимо создавать новый сокет для каждого из них и передавать управление в отдельную нить, организуемую для обслуживания клиента.

Протокол UDP менее надежен, чем протокол TCP. Он использует механизм обмена дэйтаграммами для взаимодействия с клиентом и не основан на установлении соединения. Дэйтаграммы имеют определенные границы, поэтому интерпретируются на приемной стороне однозначно, и их целостность проверяется только после получения. При реализации работы с несколькими клиентами сервер использует только один поток, что является дополнительной сложностью при разработке. UDP логичнее использовать в следующих случаях:

- Если клиент или сервер имеет мало памяти
- Если требуется минимальная задержка между отправкой и получением сообщений (VoIP или онлайн-игры)
- Если имеется большое количество клиентов, но сеанс связи с каждым клиентом непродолжителен (как в случае DNS).

Глава 5

Приложения

5.1 Описание среды разработки

Linux Debian 7.6: Среда разработки - Eclipse.

Windows 8.1: Среда разработки - Visual Studio 2013.

5.2 Листинги

5.2.1 Сервер TCP. Основной файл программы main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <pthread.h>
#include <netinet/in.h>
#include <strings.h>
#include <string.h>
#include <errno.h>

struct item {
    char* name;
    unsigned int price;
    unsigned int count;
};

void my_write(int conn_socket, const void* send_buff, unsigned int size) {
    int n1, n2;
    n1 = write(conn_socket, &size, sizeof(unsigned int));
    n2 = write(conn_socket, send_buff, size);
    if (n1 < 0 || n2 < 0)
    {
        perror("ERROR writing to socket");
        exit(1);
    }
}

void show(int items_num, struct item* items, int conn_socket) {
    unsigned int i;
    char send_buff[256];
    unsigned int buff_size = 0;
```

```

    for (i=0; i < items_num; i++) {
        int written = snprintf(send_buff + buff_size, 255 - buff_size,
                                "%d %s %d %d\n", //
                                i, //
                                items[i].name, //
                                items[i].price, //
                                items[i].count); //
        buff_size += written;
    }

    my_write(conn_socket, send_buff, buff_size);
}

#define max_conn 100
#define max_items 20

int items_num = 2;
struct item items[max_items];

int worker_count = 0;
int worker_socket[max_conn];

ssize_t          /* Read "n" bytes from a descriptor. */
readn(int fd, void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nread;
    char *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0; /* and call read() again */
            else
                return (-1);
        } else if (nread == 0)
            break; /* EOF */

        nleft -= nread;
        ptr += nread;
    }
    return (n - nleft); /* return >= 0 */
}

static void* worker(void* arg) {
    int conn_socket = *(int*)arg;
    printf("Worker for %d is up\n", conn_socket);
    char recv_buffer[255];
    char send_buffer[255];
    int recv_msg_len;
    int n;
    int i;

    const int max_cart = 20;
    int cart_item_num = 0;
    struct item cart[max_cart];

```

```

while (1) {
    n = read(conn_socket, &recv_msg_len, sizeof(int));
    if (n < 0) {
        perror("ERROR getting message length");
        return 0;
    }
    printf("Receiving message with length %d\n", recv_msg_len);

    if (recv_msg_len == 0) {
        my_write(conn_socket, " ", 1);
        continue;
    }

    bzero(recv_buffer, 255);
    n = readn(conn_socket, recv_buffer, recv_msg_len);
    if (n < 0) {
        perror("ERROR reading message");
        return 0;
    }

    printf("Received %s\n", recv_buffer);

    if (strcmp(recv_buffer, "close") == 0) { //
        printf("Close\n");
        break; //
    }

    if (strcmp(recv_buffer, "show index") == 0) {
        show(items_num, items, conn_socket);
        continue;
    }

    if (strcmp(recv_buffer, "show cart") == 0) {
        show(cart_item_num, cart, conn_socket);
        continue;
    }

    unsigned int req_item;

    if (sscanf(recv_buffer, "buy %d", &req_item) == 1) {
        if (req_item >= items_num) {
            my_write(conn_socket, "No such item\n",
                strlen("No such item\n"));
        } else {
            snprintf(send_buffer, 255, "Item %s purchased\n",
                items[req_item].name);

            my_write(conn_socket, send_buffer, strlen(send_buffer));

            items[req_item].count--;

            int already_in_cart = 0;

            for (i = 0; i < cart_item_num; i++) {
                if (cart[i].name == items[req_item].name) {
                    cart[i].count++;
                    already_in_cart = 1;
                    break;
                }
            }
        }
    }
}

```

```

    }
}

if (!already_in_cart) {
    cart[cart_item_num] = items[req_item];
    cart[cart_item_num].count = 1;
    cart_item_num++;
}
}
continue;
}

char* name = malloc(21);
unsigned int price;
unsigned int count;

if (sscanf(recv_buffer, "add %s %d %d", name, //
    &price, //
    &count) == 3) {
    if (strlen(name) > 20) {
        my_write(conn_socket, "Too long name\n",
            strlen("Too long name\n"));
    } else {
        snprintf(send_buffer, 255, "Item %s added\n", name);

        my_write(conn_socket, send_buffer, strlen(send_buffer));

        struct item new_item = { name, price, count };
        items[items_num] = new_item;
        items_num++;
    }
    continue;
}

snprintf(send_buffer, 255, "Incorrect command");

/* Write a response to the client */
my_write(conn_socket, send_buffer, strlen(send_buffer));
}

shutdown(conn_socket, 2);
close(conn_socket);
}

void* accept_loop(void* arg) {
    int accept_socket = *(int*)arg;

    pthread_t worker_thread[max_conn];

    int newsockfd;
    int cliilen;
    struct sockaddr_in cli_addr;

    int i;

    cliilen = sizeof(cli_addr);

    while (worker_count < max_conn) {
        printf("Waiting\n");

```



```

/* Accept actual connection from the client */
newsockfd = accept(accept_socket, (struct sockaddr *) & cli_addr,
    &clilen);
if (newsockfd <= 0) {
    perror("ERROR on accept");
    break;
}
printf("Connection %d\n", newsockfd);

worker_socket[worker_count] = newsockfd;

int* sock = &(worker_socket[worker_count]);

pthread_create(&(worker_thread[worker_count]),
    NULL,
    worker,
    (void*) sock);

worker_count++;
}

printf("Closed accept socket\n");
for (i = 0; i < worker_count; i++) {
    shutdown(worker_socket[i], 2);
    close(worker_socket[i]);
}

for (i = 0; i < worker_count; i++) {
    pthread_join(worker_thread[i], NULL);
}
}

/*
*
*/
int main(int argc, char** argv) {
    int sockfd, portno, clilen;

    struct sockaddr_in serv_addr, cli_addr;
    int n;
    int optval;

    items[0].name = "Pizza";
    items[0].price = 100;
    items[0].count = 10;
    items[1].name = "iPhone";
    items[1].price = 100500;
    items[1].count = 15;

    char command;

    pthread_t accept_thread;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        perror("ERROR opening socket");
        exit(1);
    }

```

```

/* Initialize socket structure */
bzero((char *) &serv_addr, sizeof(serv_addr));
portno = 5000;
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

optval = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval);

/* Now bind the host address using bind() call.*/
if (bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0)
{
    perror("ERROR on binding");
    exit(1);
}

/* Now start listening for the clients, here process will
 * go in sleep mode and will wait for the incoming connection
 */
listen(sockfd,5);

pthread_create(&(accept_thread),
    NULL,
    accept_loop,
    (void*) &sockfd);

while (1) {
    command = getchar();
    if (command == 'q') {
        break;
    } else if (command == 'd') {
        int client;
        scanf("%d", &client);
        shutdown(worker_socket[client], 2);
        close(worker_socket[client]);
    }
}

printf("Exit...\n");
shutdown(sockfd, 2);
close(sockfd);
pthread_join(accept_thread, NULL);

printf("Done\n");

return 0;
}

```

5.2.2 Клиент TCP. Основной файл программы main.c

```

#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <netdb.h>
#include <strings.h>
#include <string.h>
#include <errno.h>

ssize_t          /* Read "n" bytes from a descriptor. */
readn(int fd, void *vp, size_t n)
{
    size_t nleft;
    ssize_t nread;
    char *ptr;

    ptr = vp;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0; /* and call read() again */
            else
                return (-1);
        } else if (nread == 0)
            break; /* EOF */

        nleft -= nread;
        ptr += nread;
    }
    return (n - nleft); /* return >= 0 */
}

/*
 *
 */
int main(int argc, char** argv) {
    int sockfd, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];

    const int portno = 5000;
    /* Create a socket point */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        perror("ERROR opening socket");
        exit(1);
    }
    server = gethostbyname("localhost");
    if (server == NULL) {
        fprintf(stderr, "ERROR, no such host\n");
        exit(0);
    }

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,

```

```

        (char *)&serv_addr.sin_addr.s_addr,
        server->h_length);
serv_addr.sin_port = htons(portno);

/* Now connect to the server */
if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
{
    perror("ERROR connecting");
    exit(1);
}

while (1) {
    /* Now ask for a message from the user, this message
     * will be read by server
     */
    printf("Please enter the command: ");
    bzero(buffer, 256);
    fgets(buffer, 255, stdin);
    int buffer_len = strlen(buffer);

    if (buffer_len == 0) {
        perror("Empty message, try again");
        break;
    }
    // Remove EOL
    buffer[strlen(buffer) - 1] = '\0';
    buffer_len--;
    /* Send length to the server */
    write(sockfd, &buffer_len, sizeof(int));
    /* Send message to the server */
    n = write(sockfd, buffer, buffer_len);
    if (n < 0) {
        perror("ERROR writing to socket");
        exit(1);
    }
    /* Now read server response */
    read(sockfd, &buffer_len, sizeof(int));
    bzero(buffer, 256);
    n = readn(sockfd, buffer, buffer_len);
    if (n < 0) {
        perror("ERROR reading from socket");
        exit(1);
    }
    printf("%s\n", buffer);
}
return 0;
}

```

5.2.3 Сервер UDP. Основной файл программы main.c

```

#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <string.h>
#include <assert.h>

```

```

#include "stdafx.h"

#pragma comment(lib, "Ws2_32.lib")
#pragma comment(lib, "Mswsock.lib")
#pragma comment(lib, "AdvApi32.lib")

#define snprintf _snprintf

#define bzero(b,len) (memset((b), '\0', (len)), (void) 0)
#define bcopy(b1,b2,len) (memmove((b2), (b1), (len)), (void) 0)

struct wallet {
    char name[20];
    unsigned long money;
};

struct item {
    char* name;
    unsigned int price;
    unsigned int count;
};

#define max_conn 100
#define max_items 20

int items_num = 2;
struct item items[max_items];

const int max_cart = 20;
struct item cart[max_conn][max_cart];
int cart_size[max_conn];

struct session_st {
    int msg_num;
    char send_buffer[256];
};

struct session_st sessions[max_conn];

int session_count = 0;

void show(int items_num, struct item* items, char* send_buffer) {
    int i;
    unsigned int buff_size = 0;

    for (i=0; i < items_num; i++) {
        int written = snprintf(send_buffer + buff_size, 255 - buff_size,
                               "%d %s %d %d\n", //
                               i, //
                               items[i].name, //
                               items[i].price, //
                               items[i].count); //
        buff_size += written;
    }
}

int process_message(char* recv_buffer, char* send_buffer,
                    int recv_msg_len, int buff_size, int* session) {

```

```

int i;
int user_id;

if (sscanf(recv_buffer, "login %d", &user_id) == 1) {
    printf("OK\n");
    return 0;
}

if (strcmp(recv_buffer, "close") == 0) { //
printf("Close\n");
return 0;
}

if (strcmp(recv_buffer, "show index") == 0) {
    show(items_num, items, send_buffer);
    return 1;
}

if (strcmp(recv_buffer, "show cart") == 0) {
    show(cart_size[*session], cart[*session], send_buffer);
    return 1;
}

int req_item;

if (sscanf(recv_buffer, "buy %d", &req_item) == 1) {
    if (req_item >= items_num) {
        snprintf(send_buffer, 255, "No such item\n");
    } else {
        snprintf(send_buffer, 255, "Item %s purchased\n",
            items[req_item].name);

        items[req_item].count--;

        int already_in_cart = 0;

        for (i = 0; i < cart_size[*session]; i++) {
            if (cart[*session][i].name == items[req_item].name) {
                cart[*session][i].count++;
                already_in_cart = 1;
                break;
            }
        }

        if (!already_in_cart) {
            cart[*session][cart_size[*session]] = items[req_item];
            cart[*session][cart_size[*session]].count = 1;
            (cart_size[*session])++;
        }
    }
    return 1;
}

char* name = (char*)malloc(21);
unsigned int price;
unsigned int count;

if (sscanf(recv_buffer, "add %s %d %d", name, //
    &price, //

```

```

        &count) == 3) {
    if (strlen(name) > 20) {
        snprintf(send_buffer, 255, "Too long name\n");
    } else {
        snprintf(send_buffer, 255, "Item %s added\n", name);

        struct item new_item = { name, price, count };
        items[items_num] = new_item;
        items_num++;
    }
    return 1;
}

    snprintf(send_buffer, 255, "Incorrect command");
}

static void* worker(void* arg) {
    int sockfd = *(int*)arg;
    printf("Worker for %d is up\n", sockfd);
    char recv_buffer[256];
    char send_buffer[256];
    int n;
    int i;

    char* recv_text_buffer = recv_buffer + 8;
    char* send_text_buffer = send_buffer + 8;
    int cart_item_num = 0;

    while (1) {
        bzero(recv_buffer, 255);
        bzero(send_buffer, 255);

        struct sockaddr_in client_addr;
        socklen_t addrlen = sizeof(struct sockaddr_in);
        n = recvfrom(sockfd, recv_buffer, 255, 0, (struct sockaddr*)&client_addr, &addrlen);

        if (n < 0) {
            perror("ERROR getting message length");
            continue;
        }

        int recv_num = *(int*)recv_buffer;
        int session_id = *(int*)(recv_buffer + 4);

        printf("Received message length %d num %d session %d\n", n, recv_num, session_id);

        if (session_id >= 0) {
            if (recv_num != sessions[session_id].msg_num) {
                printf("Response was lost\n");
                sendto(sockfd, sessions[session_id].send_buffer,
                    strlen(sessions[session_id].send_buffer + 8) + 8, 0,
                    (struct sockaddr*)&client_addr, addrlen);
            }
        } else {
            session_id = session_count;
            session_count++;
        }

        printf("Received %s\n", recv_buffer + 8);
    }
}

```

```

int keep = process_message(recv_text_buffer, send_text_buffer, n, 255 - 8, &session_id);

*(int*)send_buffer = sessions[session_id].msg_num;
*(int*)(send_buffer + 4) = session_id;

int out_size = strlen(send_text_buffer) + 8;
        /* Write a response to the client */
n = sendto(socketfd, send_buffer, out_size, 0, (struct sockaddr*) &client_addr, addrlen);
if (n < 0) {
    perror("ERROR writing to socket");
    continue;
}

        bcopy(send_buffer, sessions[session_id].send_buffer, 256);

sessions[session_id].msg_num++;

if (!keep) {
    sessions[session_id].msg_num = 0;
}
}
}

/*
 *
 */
int _tmain(int argc, _TCHAR* argv[]) {
    int sockfd, portno, clilen;

    struct sockaddr_in serv_addr, cli_addr;
    int n;
    int optval;
    int i;

    items_num = 2;
    items[0].name = "Pizza";
    items[0].price = 100;
    items[0].count = 10;
    items[1].name = "iPhone";
    items[1].price = 100500;
    items[1].count = 15;

    char command;

        // Initialize Winsock
        WSADATA wsaData;
n = WSASocket(2, 2, 0, &wsaData, 0);
if (n != 0) {
    printf("WSAStartup failed with error: %d\n", n);
    return 1;
}

sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if (sockfd == INVALID_SOCKET)
{
    perror("ERROR opening socket");
    exit(1);
}

```



```

/* Initialize socket structure */
bzero((char *) &serv_addr, sizeof(serv_addr));
portno = 5000;
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

optval = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (char*)&optval, sizeof optval);

/* Now bind the host address using bind() call.*/
if (bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0)
{
    perror("ERROR on binding");
    exit(1);
}

/* Now start listening for the clients, here process will
 * go in sleep mode and will wait for the incoming connection
 */
listen(sockfd,5);

worker(&sockfd);
    /* pthread_create(&(accept_thread),
    NULL,
    worker,
    (void*) &sockfd);

while (1) {
    command = getchar();
    if (command == 'q') {
        break;
    }
}

printf("Exit...\n");
shutdown(sockfd, 2);
close(sockfd);
pthread_join(accept_thread, NULL); */

printf("Done\n");

return 0;
}

```

5.2.4 Клиент UDP. Основной файл программы main.c

```

#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <netdb.h>
#include <strings.h>
#include <string.h>
#include <errno.h>

ssize_t          /* Read "n" bytes from a descriptor. */

```

```

readn(int fd, void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nread;
    char *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0; /* and call read() again */
            else
                return (-1);
        } else if (nread == 0)
            break; /* EOF */

        nleft -= nread;
        ptr += nread;
    }
    return (n - nleft); /* return >= 0 */
}

/*
 *
 */
int main(int argc, char** argv) {
    int sockfd, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];

    const int portno = 5000;
    /* Create a socket point */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        perror("ERROR opening socket");
        exit(1);
    }
    server = gethostbyname("localhost");
    if (server == NULL) {
        fprintf(stderr, "ERROR, no such host\n");
        exit(0);
    }

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
        (char *)&serv_addr.sin_addr.s_addr,
        server->h_length);
    serv_addr.sin_port = htons(portno);

    /* Now connect to the server */
    if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
    {
        perror("ERROR connecting");
        exit(1);
    }

```

```

}

while (1) {
    /* Now ask for a message from the user, this message
     * will be read by server
     */
    printf("Please enter the command: ");
    bzero(buffer, 256);
    fgets(buffer, 255, stdin);
    int buffer_len = strlen(buffer);

    if (buffer_len == 0) {
        perror("Empty message, try again");
        break;
    }
    // Remove EOL
    buffer[strlen(buffer) - 1] = '\0';
    buffer_len--;
    /* Send length to the server */
    write(sockfd, &buffer_len, sizeof(int));
    /* Send message to the server */
    n = write(sockfd, buffer, buffer_len);
    if (n < 0) {
        perror("ERROR writing to socket");
        exit(1);
    }
    /* Now read server response */
    read(sockfd, &buffer_len, sizeof(int));
    bzero(buffer, 256);
    n = readn(sockfd, buffer, buffer_len);
    if (n < 0) {
        perror("ERROR reading from socket");
        exit(1);
    }
    printf("%s\n", buffer);
}
return 0;
}

```