Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Технологии компьютерных сетей

Отчет по лабораторной работе
Протоколы TCP и UDP

Работу выполнил:
Кыльчик И.В.
Группа: 43501/1
Преподаватель:
Алексюк А.О.

Санкт-Петербург
2018

Содержание

1. Цель работы

Изучение принципов программирования сокетов с использованием протоколов TCP и UDP.

2. Программа работы

1. Реализация TCP сервера и TCP клиента на ОС семейства Linux
2. Реализация TCP сервера и TCP клиента на Windows
3. Реализация UDP сервера и UDP клиента (ОС по выбору)
4. Реализация индивидуального задания
5. Задание по выбору - SOCKS5 Proxy на Java

3. Теоретическая информация

3.1. Разработка сетевых приложений с помощью технологии сокетов

С точки зрения архитектуры TCP/IP сокетом называется пара (IP-адрес, порт), однозначно идентифицирующая прикладное приложение в сети Internet.

С точки зрения операционной системы BSD-сокет (Berkley Software Distribution) или просто сокет — это выделенные операционной системой набор ресурсов, для организации сетевого взаимодействия. К таким ресурсам относятся, например, буфера для приёма/посылки данных или очереди сообщений. Технология сокетов была разработана в университете Беркли и впервые появилась в операционной системе BSD-UNIX.

В операционной системе MS Windows имеется аналогичная библиотека сетевого взаимодействия WinSock, реализованная на основе библиотеки BSD-сокетов. Существует две версии библиотеки WinSock1 и WinSock2. В подавляющем большинстве случаев функции и типы библиотеки WinSock совпадают с функциями и типами BSD-сокетов. Об имеющихся отличиях будет сказано отдельно.

3.2. Создание сокета

Для создания сокета в библиотеках BSD-socket и WinSock имеется системный вызов socket:

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

В случае успеха результат вызова функции – дескриптор созданного сокета, в случае ошибки (-1) в библиотеке BSD-socket и INVALID_SOCKET в библиотеке WinSock.

Параметр domain указывает на домен, в пространстве которого создаётся данный сокет. Домен AF_UNIX используется для межпроцессного взаимодействия, домен AF_INET – для передачи с использованием стека протоколов TCP/IP.

Параметр type определяет тип создаваемого сокета. Этот параметр может принимать значения:

- SOCK_STREAM – для организации надёжного канала связи с установлением соединений

- SOCK_DGRAM – для организации ненадёжного дейтаграммного канала связи
- SOCK_RAW – для организации низкоуровнего доступа на основе «сырых» сокетов

Параметр protocol – идентификатор используемого протокола. В большинстве случаев протокол однозначно определяется типом создаваемого сокета, и передаваемое значение этого параметра – 0. Если же это не так (например, в случае SOCK_RAW), то необходимо явно задавать идентификатор протокола. Для его получения имеются системные вызовы getprotobyname и getprotobynumber, которые разбирают файл /etc/protocols и получают идентификатор сетевого протокола:

```
1 struct protoent* getprotobyname(const char *name);
2 struct protoent* getprotobynumber(int proto);
```

Эти функции заполняют структуру protoent, поле p_proto которой следует использовать в качестве параметра protocol вызова socket:

```
1 struct protoent
2 {
3 char* p_name; // имя протокола из файла protocols
4 char** p_aliases; // список псевдонимов
5 int p_proto; // идентификатор протокола
6 }
```

Структура protoent описана в файле /usr/include/netdb.h

3.3. Организация соединения

Для установления TCP-соединения используется вызов connect:

```
1 int connect(int s, const struct sockaddr* serv_addr, int addr_len);
```

Результатом выполнения функции является установление TCP-соединения с TCP - сервером. Функция возвращает значение 0 в случае успеха и -1 в случае ошибки.

Параметр s – дескриптор созданного сокета.

Параметр serv_addr – указатель на структуру, содержащую параметры удалённого узла.

Параметр addr_len – размер в байтах структуры, на которую указывает параметр serv_addr.

При программировании сокетов из домена AF_INET вместо структуры sockaddr используется приводимая к ней структура sockaddr_in, находящаяся в подключаемом файле /usr/include/linux/in.h:

```
1 struct sockaddr_in
2 {
3 sa_family_t sin_family; // Коммуникационный домен
4 unsigned short int sin_port; // Номер порта
5 struct in_addr sin_addr; // IP-адрес
6 ...
7 };
```

Успех выполнения функции connect означает корректное установление логического канала связи и возможность начала передачи и приёма данных по протоколу TCP.

В случае использования вызова connect для протокола UDP установления соединения не происходит, а адрес и порт из структуры serv_addr используется как адрес по умолчанию для последующих вызовов send и recv.

Для более простого заполнения параметров структуры sockaddr_in используются системные вызовы htons и inet_addr, осуществляющие замену порядка следования байт в

номере порта и перевод IP-адреса из строкового вида в числовой соответственно, например:

```
1 serv_addr.sin_port = htons(3128);
2 serv_addr.sin_addr.s_addr = inet_addr“”(192.168.1.1);
```

3.4. Передача и приём данных по протоколу TCP

Передача и приём данных в рамках установленного TCP-соединения осуществляется вызовами send и recv:

```
1 int send(int s, const void *msg, size_t len, int flags);
2 int recv(int s, void *msg, size_t len, int flags);
```

Параметр s – дескриптор сокета, параметр msg – указатель на буфер, содержащий данные (вызов send), или указатель на буфер, предназначенный для приёма данных (вызов recv). Параметр len – длина буфера в байтах, параметр flags – опции посылки или приёма данных.

Возвращаемое значение – число успешно посланных или принятых байтов, в случае ошибки функция возвращает значение -1.

3.5. Передача и приём данных по протоколу UDP

В случае установленного адреса по умолчанию для протокола UDP (вызов connect) функции для передачи и приёма данных по протоколу UDP можно использовать вызовы send и recv.

Если адрес и порт по умолчанию для протокола UDP не установлен, то параметры удалённой стороны необходимо указывать или получать при каждом вызове операций записи или чтения. Для протокола UDP имеется два аналогичных вызова sendto и recvfrom:

```
1 int sendto(int s, const void *buf, size_t len, int flags, struct sockaddr *to,
    ↪ int* tolen);
2 int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, int
    ↪ * fromlen);
```

Параметры s, buf, len и flags имеют тот же смысл, что и в случае использования функций send и recv, параметры to и tolen – атрибуты адреса удалённого сокета при посылке данных, параметры from и fromlen – атрибуты структуры данных в которую помещаются параметры удалённого сокета при получении данных.

3.6. Завершение TCP-соединения

Завершение установленного TCP-соединения осуществляется в библиотеке BSD-socket с помощью вызова shutdown:

```
1 int shutdown(int s, int how);
```

Параметр s – дескриптор сокета, параметр how – определяет способ за- крытия:

- SHUT_RD – запрещён приём данных
- SHUT_WR – запрещена передача данных
- SHUT_RDWR – запрещены и приём и передача данных.

В библиотеке WinSock семантика вызова несколько отличается:

```
1 int shutdown(SOCKET s, int how);
```

Параметр s – дескриптор сокета, параметр how – определяет способ за- крытия:

- SD_RECEIVE – запрещён приём данных. В случае наличия данных в очереди соединение разрывается.
- SD_SEND – запрещена передача данных.
- SD_BOTH – запрещены и приём и передача данных.

3.7. Закрытие сокета

По окончании работы следует закрыть сокет, для этого в библиотеке BSD-socket предусмотрен вызов close:

```
1 int close(int s);
```

Аналогичный вызов в библиотеке WinSock имеет название closesocket:

```
1 int closesocket(SOCKET s);
```

Параметр s – дескриптор сокета.
Возвращаемое значение – 0, в случае успеха.

3.8. Привязывание сокета

Созданный сокет является объектом операционной системы, использующим её отдельные ресурсы. В то же время в большинстве случаев недостаточно просто выделить ресурсы операционной системы, а следует также связать эти ресурсы с конкретными сетевыми параметрами: сетевым адресом и номером порта. Особенно это важно для серверных сокетов, для которых такая связь – необходимое требование доступности разрабатываемого сетевого сервиса.

Организация привязки созданного вызовом socket() сокета к определённым IP-адресам и портам осуществляется с помощью функция bind:

```
1 int bind(int s, struct sockaddr *addr, socklen_t addrlen);
```

Параметр s – дескриптор сокета, параметр addr задаёт указатель на структуру, хранящую параметры адреса и порта, addrlen – размер структуры addr в байтах.

3.9. Перевод TCP-сокета в состояние прослушивания

Для перевода сокета в состояние прослушивания служит системный вызов listen:

```
1 int listen(int s, int backlog);
```

Параметр s – дескриптор сокета, параметр backlog – задаёт максимальную длину, до которой может расти очередь ожидающих соединений.

В случае успеха возвращаемое значение – 0. При ошибке возвращается -1.

3.10. Приём входящего TCP-соединения

В случае, когда сокет находится в состоянии прослушивания (listen) необходимо отслеживать поступление входящих соединений. Для этого предусмотрен системный вызов accept:

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

Параметр s – дескриптор прослушивающего сокета, параметр addr – указатель на структуру, содержащую параметры сокета, инициирующего соединение, addrlen – размер структуры addr в байтах.

Возвращаемое значение – дескриптор сокета, созданного для нового соединения. Большинство параметров нового сокета соответствуют параметрам слушающего сокета. Полученный сокет в дальнейшем может использоваться для передачи и приёма данных.

В случае если входящих соединений нет, то функция accept ожидает поступления запроса на входящее соединение.

3.11. Дополнительные функции

Для получения текстового представления ошибки в операционной системе Linux используется функция strerror:

```
1 char* strerror(int errnum);
```

Функция слежения за изменением файлового дескриптора select:

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

Эта функция может использоваться для определения наличия информации в приёмном сокете.

Функции, предназначенные для установки и чтения значений опций сокетов setsockopt и getsockopt:

```
1 int setsockopt(int s, int level, int optname, const void *optval, socklen_t
    ↪ optlen);
2 int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
```

Параметр s – дескриптор сокета, level – уровень, на котором должна происходить интерпретация флага, optname – идентификатор опции. Параметры optval и optlen используются в функции setsockopt для доступа к значениям флагов, для getsockopt они задают буфер, в который нужно поместить запрошенное значение.

3.12. Структура TCP-клиента

Перечисленные в предыдущих параграфах функции используются для организации клиентских и серверных приложений.

Клиент протокола TCP создаёт экземпляр сокета, необходимый для взаимодействия с сервером, организует соединение, осуществляет обмен данными, в соответствии с протоколом прикладного уровня.

Типичная структура TCP-клиента представлена на рисунке ниже.

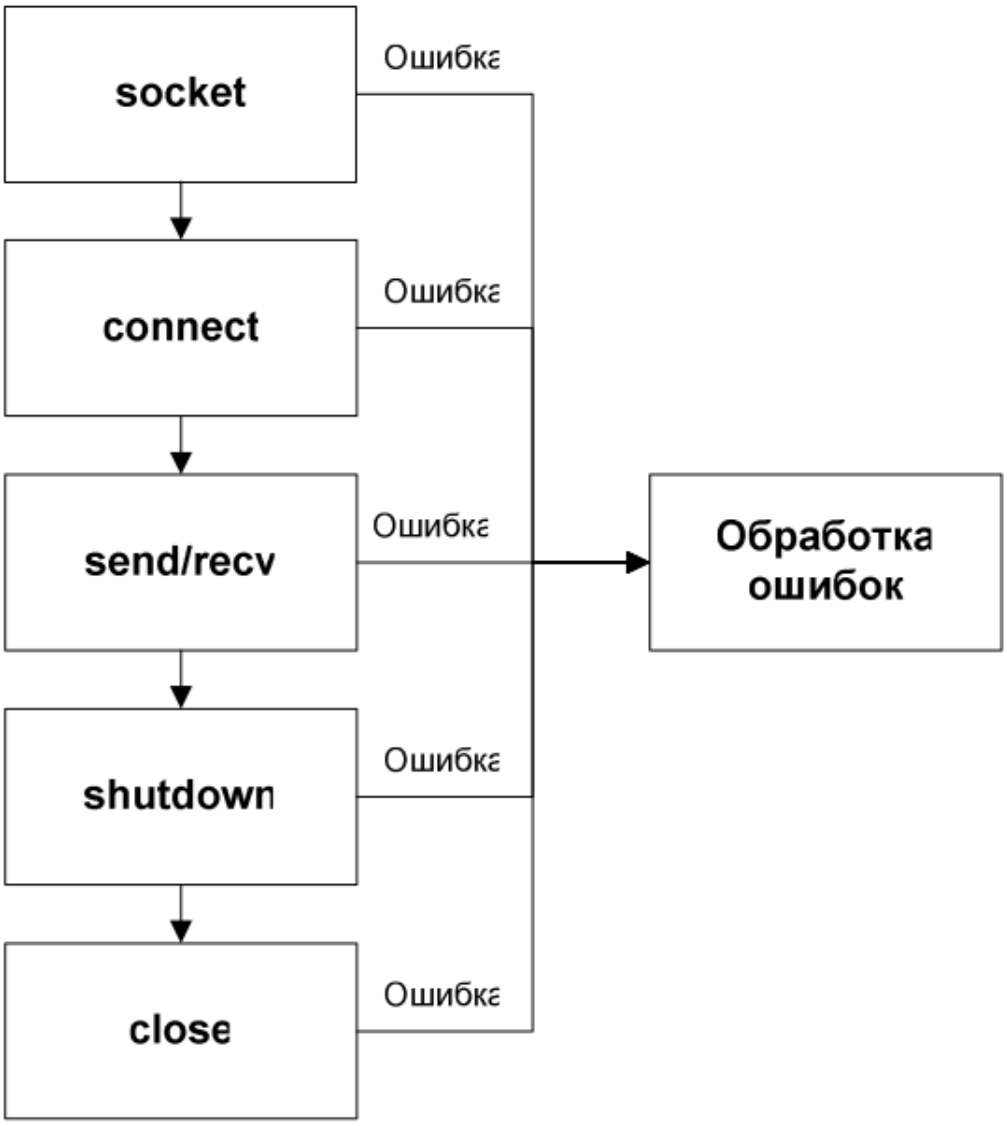

Рисунок 3.1. Типичная структура TCP-клиента

Если инициатором разрыва соединения является клиентское приложение (например, канал данных протокола FTP), то далее следует вызвать функцию shutdown и после этого закрыть сокет.

Каждый вызов функций библиотеки сокетов должен сопровождаться проверкой на наличие ошибочной ситуации и обработкой этой ситуации.

3.13. Структура TCP-сервера

Организация TCP-сервера отличается от TCP-клиента в первую очередь созданием слушающего сокета (см. рис. 1.2 а). Такой сокет находится в состоянии listen и предназначен только для приёма входящих соединений. В случае прихода запроса на соединение создаётся дополнительный сокет, который и занимается обменом данными с клиентом. Типичная структура TCP-сервера и взаимосвязь сокетов изображена на рис. 3.2 а) и б).

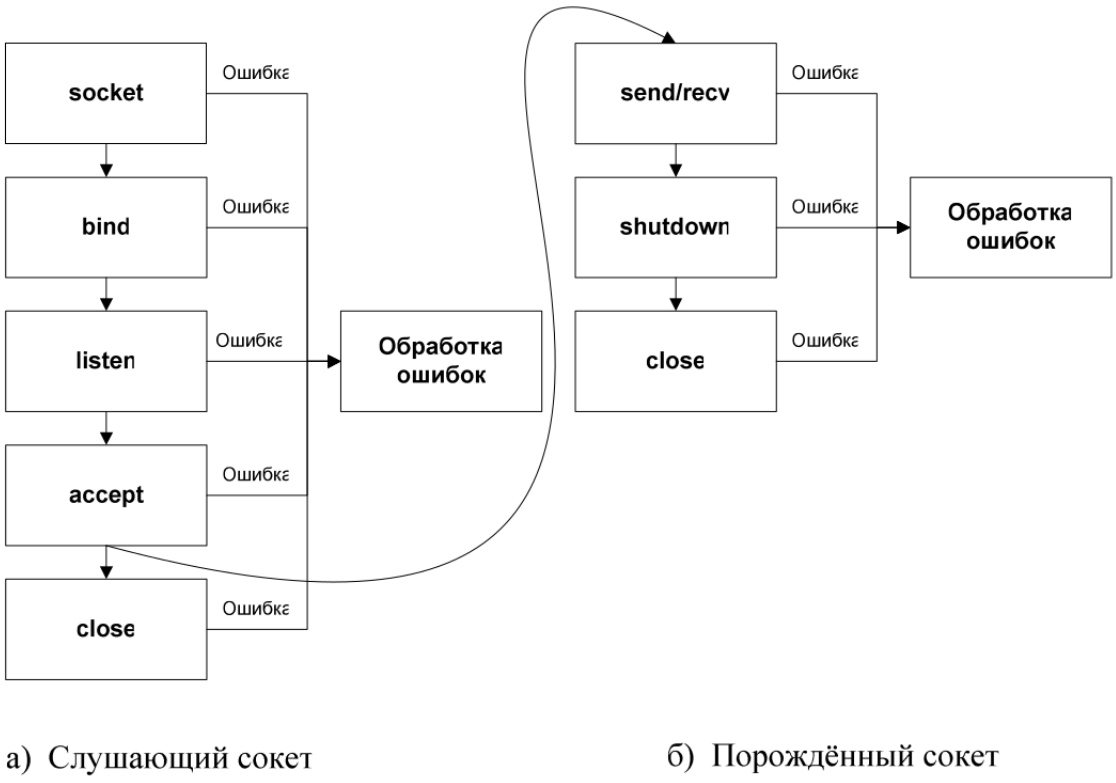

Рисунок 3.2. Типичная структура TCP-сервера

3.14. Структура UDP-клиента

Структура UDP-клиента ещё более простая, чем у TCP-клиента, так как нет необходимости создавать и разрывать соединение. Варианты организации UDP-клиента изображены на рис. 3.3.

Наличие двух вариантов организации связано с возможностью в UDP- приложениях использовать вызов connect, устанавливающий значения по умолчанию для IP-адреса и порта сервера.

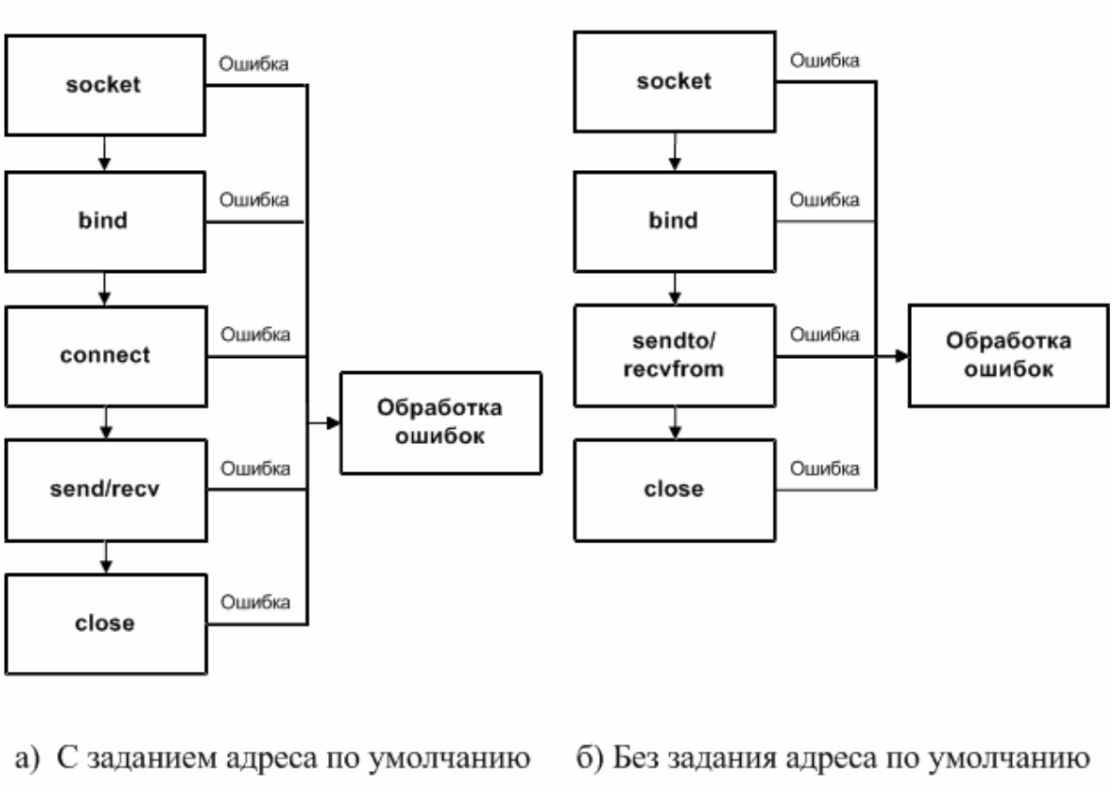

Рисунок 3.3. Типичная структура UDP-клиента

3.15. Структура UDP-сервера

Ввиду того, что в протоколе UDP не устанавливается логический канал связи между клиентом и сервером, то для обмена данными между несколькими клиентами и сервером нет необходимости использовать со стороны сервера несколько сокетов. Для определения источника полученной дейтаграммы серверный сокет может использовать поля структуры from вызова recvfrom. Типичный способ организации UDP-сервера приведён на рис. 3.4.

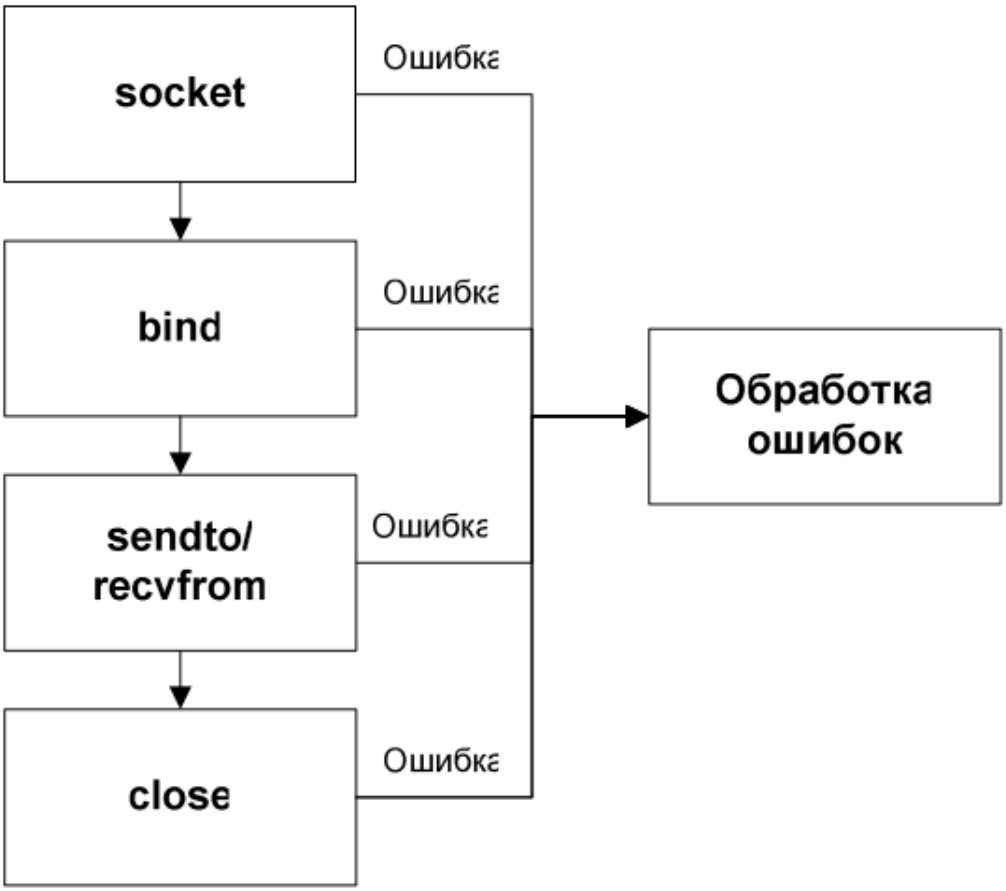

Рисунок 3.4. Типичная структура UDP-сервера

4. Ход выполнения работы

4.1. Реализация TCP на Linux

Листинг 1: TCP сервер на Linux

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include <netdb.h>
5  #include <netinet/in.h>
6  #include <unistd.h>
7
8  #include <string.h>
9
10 #include "../util_linux/util_linux.h"
11
12 int main() {
13   int sockfd, newsockfd;
14   uint16_t portno;
15   unsigned int clilen;
16   char* buffer;
17   struct sockaddr_in serv_addr, cli_addr;
```

```
18
19    // Сперва вызываем функцию socket()
20    sockfd = socket(AF_INET, SOCK_STREAM, 0);
21
22    //Если возникла ошибка открытия сокета то выходим из программы
23    if (sockfd < 0) {
24      perror("ERROR opening socket");
25      exit(1);
26    }
27
28    // Инициализируем структуру сокета
29    bzero((char *) &serv_addr, sizeof(serv_addr));
30    portno = 5001;  //указываем порт
31    //заносим данные в стуктуру сокета
32    serv_addr.sin_family = AF_INET; //семейство адресов
33    serv_addr.sin_addr.s_addr = INADDR_ANY; //IP
34    serv_addr.sin_port = htons(portno); //порт
35
36    //указываем дополнительные опции сокета для повторного открытия нового сокета на том же
        порту
37    //https://serverfault.com/questions/329845/how-to-forcibly-close-a-socket-in-time-wait
38    if(setsockopt(sockfd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR), &(int){ 1 },
        ↪ sizeof(int)) < 0){
39      closeSocket((int[]){sockfd}, 1, "ERROR on setsockopt");
40    }
41
42    // Привязываем адрес через функцию bind()
43    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
44      closeSocket((int[]){sockfd}, 1, "ERROR on binding");
45    }
46
47    //слушаем клиентов, стэк установлен на 5 соединений
48    listen(sockfd, 5);
49    clilen = sizeof(cli_addr);
50
51    //создаем новую ветку программы которая ожидает нажатия клавиши q и зыкрывает сервер
52    if(fork() > 0){
53      close(sockfd);
54      while(getchar() != 'q'){
55      }
56      shutdown(sockfd, SHUT_RDWR);
57      exit(0);
58    }
59
60    while(1) {
61
62      // Принимаем соединение от клиента
63      newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
64      //в случае оишибки закрываем сервер
65      if (newsockfd < 0) {
66        closeSocket((int[]){sockfd, newsockfd}, 1, "ERROR on accept");
67      }
68      //создаем новый потом
69      switch(fork()) {
70        case -1:
71          perror("ERROR on fork");
72          break;
73        case 0:
74          //дочерний поток работает с клиентом
75          close(sockfd);
76          //если соединение было установлено то начинаем общение
```

```
77          buffer = readAll((int[]){newsockfd, sockfd});
78
79          printf("Here is the message: %s\n", buffer);
80
81          //отправляет клиенту ответ
82          sendAll((int[]){newsockfd, sockfd}, "I got your message");
83          //зывавершаем дочерний поток
84          closeSocket((int[]){newsockfd}, 0, "");
85          break;
86        default:
87          //родительский поток продалжает ожидать новых клиентов
88          close(newsockfd);
89        }
90
91    }
92
93    closeSocket((int[]){sockfd, newsockfd}, 0, "");
94
95    return 0;
96 }
```

Листинг 2: TCP клиент на Linux

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include <netdb.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7
8 #include <string.h>
9
10 #include "../util_linux/util_linux.h"
11
12
13 int main(int argc, char *argv[]) {
14    int sockfd;
15    uint16_t portno;
16    struct sockaddr_in serv_addr;
17    struct hostent *server;
18
19    char* buffer = (char*)malloc(256);
20    //обазательно в качестве аргументов указать IP/имя хоста и номер порта
21    if (argc < 3) {
22       fprintf(stderr, "usage %s hostname port\n", argv[0]);
23       exit(0);
24    }
25
26    portno = (uint16_t) atoi(argv[2]);
27
28    // создаем сокет
29    sockfd = socket(AF_INET, SOCK_STREAM, 0);
30    //Если возникла ошибка открытия сокета то выходим из программы
31    if (sockfd < 0) {
32       perror("ERROR opening socket");
33       exit(1);
34    }
35    //получаем ip адрес по имени хоста
36    server = gethostbyname(argv[1]);
37
38    if (server == NULL) {
```

```
39      closeSocket((int[]){sockfd}, 0, "ERROR, no such host\n");
40    }
41    // Инициализируем структуру сокета
42    bzero((char *) &serv_addr, sizeof(serv_addr));
43    //заносим данные в стуктуру сокета
44    serv_addr.sin_family = AF_INET;
45    bcopy(server->h_addr, (char *) &serv_addr.sin_addr.s_addr, (size_t) server->
        ↪ h_length);
46    serv_addr.sin_port = htons(portno);
47
48    //соединяемся с сервером
49    if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
50      closeSocket((int[]){sockfd}, 1, "ERROR connecting");
51    }
52
53    //ожидаем ввода сообщния от польщователя для последующей отправки на сервер
54    printf("Please enter the message: ");
55    fgets(buffer, 255, stdin);
56
57    //отправляем сообщение на сервер
58    sendAll((int[]){sockfd}, buffer);
59
60    //читаем ответ с сервера
61    buffer = readAll((int[]){sockfd});
62
63    printf("%s\n", buffer);
64
65    free(buffer);
66    buffer = NULL;
67
68    closeSocket((int[]){sockfd}, 0, "");
69
70    return 0;
71  }
```

Листинг 3: Подключаемый файл для дополнительных функций для TCP на Linux

```
1  #ifndef UTIL_LINUX_H
2  #define UTIL_LINUX_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  #include <netdb.h>
8  #include <netinet/in.h>
9  #include <unistd.h>
10
11 #include <string.h>
12
13 void closeSocket(int socks[], int error, char* errorMsg);
14 char* readAll(int socks[]);
15 void sendAll(int socks[], char* buffer);
16
17 #endif
```

Листинг 4: Исходный файл для дополнительных функций для TCP на Linux

```
1  #include "util_linux.h"
2
3  //функция закрытия сокетов и завершение программы
4  void closeSocket(int socks[], int error, char* errorMsg){
```

```
5     if(strcmp(errorMsg,"") != 0){
6       perror(errorMsg);
7     }
8     int lenght = (int)(sizeof(socks)/sizeof(socks[0]));
9     for(int i = 0; i < lenght; i++) {
10      shutdown(socks[i], SHUT_RDWR);
11      close(socks[i]);
12    }
13    exit(error);
14  }
15  //функция чтения сообщения
16  //данная функция сперва считывает 4 байта в которых хранится длина последующего сообщения
17  //затем считывается нужное количество байт указанное в предыдущем сообщении
18  char* readAll(int socks[]){
19    char *buffer = (char*)calloc(256, sizeof(char));
20    if(buffer == NULL){
21      closeSocket(socks, 1, "ERROR allocation");
22    }
23    char strLenght[4];
24
25    int n = read(socks[0], strLenght, 4);
26    if (n < 0) {
27      closeSocket(socks, 1, "ERROR reading from socket");
28    }
29    int lenght = atoi(strLenght);
30
31    int recieved = 0;
32    while(recieved < lenght){
33      n = read(socks[0], buffer, 256);
34      recieved += n;
35      if (n < 0) {
36        closeSocket(socks, 1, "ERROR reading from socket");
37      }
38    }
39
40    return buffer;
41  }
42  //функция записи сообщения
43  //сперва отправляет 4 байта длины последующего сообщения
44  //затем отправляет само сообщение
45  void sendAll(int socks[], char* buffer){
46    int messageLength = strlen(buffer);
47    char toSend[4 + 256];
48
49    snprintf(toSend, 4 + 256, "%04d%s", messageLength, buffer);
50
51    int n = write(socks[0], toSend, strlen(toSend));
52    if (n < 0) {
53      closeSocket(socks, 1, "ERROR writing to socket");
54    }
55  }
```

4.2. Реализация TCP на Windows

Листинг 5: TCP сервер на Windows

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
```

```
4 #include <unistd.h>
5 #include <windows.h>
6 #include <winsock2.h>
7
8 #include <string.h>
9
10 #define SHUT_RDWR 2 //на Windows нет такой переменной, приходится объявлять самому
11
12 //функция закрытия сокетов в случае ошибки
13 void closeSocket(SOCKET socks[], int lenght, int error, char* errorMsg);
14 //чтение всех посланных данных
15 char* readAll(SOCKET socks[]);
16 //корректая отправка данных
17 void sendAll(SOCKET socks[], char* buffer);
18 //функция ожидания соединения клиента, запускается в отдельном потоке
19 DWORD WINAPI mainLoop();
20 //функция для работы с конкретным клиентом, запускается в отдельном потоке
21 DWORD WINAPI clientCommunication(LPVOID data);
22
23 //флаг который показывает что определенный клиент был принят
24 //используется потому что в Windows для работы с сокетом он должен быть создан в том же
      потоке
25 volatile BOOL readyToAccept = TRUE;
26
27 int main() {
28   WSADATA WSStartData;
29
30   //включаем сокеты
31   if (WSAStartup(MAKEWORD(2, 2), &WSStartData) != 0) {
32     perror("ERROR on WSAStartup");
33     exit(1);
34   }
35   //создаем новый поток который инициализирует серверный слушающий сокет
36   CreateThread(NULL, 0, mainLoop, NULL, 0, NULL);
37   //для выхода ожидаем нажатие клавиши q
38   while(getchar() != 'q'){
39   }
40
41     return 0;
42 }
43 //функция для закрытия сокета
44 //SOCKET socks[] - сокеты для закрытия
45 //int lenght - длина массива socks[]
46 //int error - код ошибки
47 //char* errorMsg - сообщение ошибки
48 void closeSocket(SOCKET socks[], int lenght, int error, char* errorMsg){
49   if(strcmp(errorMsg,"") != 0){
50     printf(strcat(errorMsg, " with code : %ld\n"), WSAGetLastError());
51   }
52   for(int i = 0; i < lenght; i++) {
53     shutdown(socks[i], SHUT_RDWR);
54     closesocket(socks[i]);
55   }
56   WSACleanup();
57   exit(error);
58 }
59 //функция для чтения из сокета
60 //SOCKET socks[] - сокеты для закрытия в случае ошибки
61 //возвращаемое значение - считанные данные
62 char* readAll(SOCKET socks[]){
63   char *buffer = (char*)malloc(256);
```

```
64    char strLenght[4];
65
66    //сперва считываем длину сообщения в 4 байта
67    int n = recv(socks[1], strLenght, 4, 0);
68    if (n < 0) {
69      closeSocket(socks, 2, 1, "ERROR reading header from socket");
70    }
71    int lenght = ((strLenght[0] - '0') << 24) + ((strLenght[1] - '0') << 16) + ((
        ↪ strLenght[2] - '0') << 8) + (strLenght[3] - '0');
72
73    //считываем само сообщение
74    memset(buffer, 0, 256);
75    int recieved = 0;
76    while(recieved < lenght){
77      n = recv(socks[1], buffer, 256, 0);
78      recieved += n;
79      if (n < 0) {
80        closeSocket(socks, 2, 1, "ERROR reading from socket");
81      }
82    }
83
84    return buffer;
85  }
86  //функция для записи в сокет
87  //SOCKET socks[] - сокеты для закрытия в случае ошибки
88  //char* buffer - данные для записи
89  void sendAll(SOCKET socks[], char* buffer){
90      int messageLength = strlen(buffer);
91      char toSend[4 + 256];
92    //сперва формируем длину сообщения
93    memset(toSend, 0, 4 + 255);
94      toSend[0] = ((messageLength >> 24) & 0xff) + '0';
95      toSend[1] = ((messageLength >> 16) & 0xff) + '0';
96      toSend[2] = ((messageLength >> 8) & 0xff) + '0';
97      toSend[3] = ((messageLength >> 0) & 0xff) + '0';
98
99      strcat(toSend, buffer);
100   //в одном сообщении посылаем длину сообщения и само сообщение
101     int n = send(socks[1], toSend, strlen(toSend), 0);
102     if (n < 0) {
103         closeSocket(socks, 2, 1, "ERROR writing to socket");
104     }
105 }
106
107 //основной цикл сервера
108 //инициализируем слушающий сокет и ждем подключения
109 DWORD WINAPI mainLoop() {
110   SOCKET sockfd;
111   unsigned int portno;
112   struct sockaddr_in serv_addr;
113
114   /* First call to socket() function */
115   sockfd = socket(AF_INET, SOCK_STREAM, 0);
116
117   if (sockfd == INVALID_SOCKET) {
118     perror("ERROR opening socket");
119     exit(1);
120   }
121
122   /* Initialize socket structure */
```

```
123   memset((char *)&serv_addr, 0, sizeof(serv_addr));
124   portno = 5001;
125
126   serv_addr.sin_family = AF_INET;
127   serv_addr.sin_addr.s_addr = INADDR_ANY;
128   serv_addr.sin_port = htons(portno);
129
130   //устанавливаем опции для повторного октрытия сокета на том же порту
131   BOOL value = TRUE;
132   if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (char *) &value, sizeof(BOOL)
        ↪ ) < 0) {
133     closeSocket((SOCKET[]) {sockfd}, 1, 1, "ERROR on setsockopt");
134   }
135
136   /* Now bind the host address using bind() call.*/
137   if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
138     closeSocket((SOCKET[]) { sockfd }, 1, 1, "ERROR on binding");
139   }
140
141   /* Now start listening for the clients, here process will
142   * go in sleep mode and will wait for the incoming connection
143   */
144
145   listen(sockfd, 5);
146
147   while (1) {
148     if (readyToAccept == TRUE) {
149       readyToAccept = FALSE;
150       CreateThread(NULL, 0, clientCommunication, &sockfd, 0, NULL);
151     }
152   }
153 }
154
155 //поток контролирующий работу с клиентом
156 DWORD WINAPI clientCommunication(LPVOID data) {
157   int sockfd = *(int *)data;
158   struct sockaddr_in cli_addr;
159   int clilen = sizeof(cli_addr);
160
161   /* Accept actual connection from the client */
162   SOCKET newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
163
164   if (newsockfd == INVALID_SOCKET) {
165     closeSocket((SOCKET[]) { sockfd, newsockfd }, 2, 1, "ERROR on accept");
166   }
167
168   readyToAccept = TRUE;
169   char* buffer = (char*)malloc(256);
170
171   /* If connection is established then start communicating */
172   buffer = readAll((SOCKET[]) { sockfd, newsockfd});
173   printf("Here is the message: %s\n", buffer);
174
175   /* Write a response to the client */
176   sendAll((SOCKET[]) { sockfd, newsockfd }, "I got your message");
177
178   closesocket(newsockfd);
179
180   return 0;
181 }
```

Листинг 6: TCP клиент на Windows

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include <unistd.h>
5
6 #include <winsock2.h>
7
8 #include <string.h>
9
10 #define SHUT_RDWR 2 //на Windows нет такой переменной, приходится объявлять самому
11
12 //функция закрытия сокетов в случае ошибки
13 void closeSocket(SOCKET sockfd, int error, char* errorMsg);
14 //корректая отправка данных
15 void sendAll(SOCKET sockfd, char buffer[]);
16 //чтение всех посланных данных
17 char* readAll(SOCKET sockfd);
18
19 int main(int argc, char *argv[]) {
20     SOCKET sockfd;
21     unsigned int portno;
22     struct sockaddr_in serv_addr;
23     struct hostent *server;
24   char* buffer = (char*)malloc(256);
25   WSADATA WSStartData;
26
27   //включаем сокеты
28   if (WSAStartup(MAKEWORD(2, 2), &WSStartData) != 0) {
29     perror("ERROR on WSAStartup");
30     exit(1);
31   }
32
33   //обазательно три аргумента, два из которых IP port
34     if (argc < 3) {
35         fprintf(stderr, "usage %s hostname port\n", argv[0]);
36         exit(0);
37     }
38   //преобразуем символьное значение порта в числовое
39     portno = (unsigned int) atoi(argv[2]);
40
41     /* Create a socket point */
42     sockfd = socket(AF_INET, SOCK_STREAM, 0);
43
44     if (sockfd == INVALID_SOCKET) {
45         perror("ERROR opening socket");
46         exit(1);
47     }
48
49   //получаем IP хоста по имени
50     server = gethostbyname(argv[1]);
51
52     if (server == NULL) {
53         closeSocket(sockfd, 0, "ERROR, no such host\n");
54     }
55
56   //инициализируем буффер
57   memset((char *)&serv_addr, 0, sizeof(serv_addr));
58     serv_addr.sin_family = AF_INET;
59   memmove((char *)&serv_addr.sin_addr.s_addr, server->h_addr, (size_t)server->
```

```
    ↪ h_length);
60      serv_addr.sin_port = htons(portno);
61
62      /* Now connect to the server */
63      if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) ==
    ↪ SOCKET_ERROR) {
64          closeSocket(sockfd, 1, "ERROR connecting");
65      }
66
67      /* Now ask for a message from the user, this message
68         * will be read by server
69      */
70
71      printf("Please enter the message: ");
72      fgets(buffer, 255, stdin);
73
74      /* Send message to the server */
75      sendAll(sockfd, buffer);
76
77      /* Now read server response */
78      buffer = readAll(sockfd);
79
80      printf("%s\n", buffer);
81
82      closeSocket(sockfd, 0, "");
83
84      return 0;
85  }
86
87  //функция для закрытия сокета
88  //SOCKET socks - сокет для закрытия
89  //int error - код ошибки
90  //char* errorMsg - сообщение ошибки
91  void closeSocket(SOCKET sockfd, int error, char* errorMsg){
92      if(strcmp(errorMsg,"") != 0){
93          printf(strcat(errorMsg, " with code : %ld\n"), WSAGetLastError());
94      }
95      shutdown(sockfd, SHUT_RDWR);
96      closesocket(sockfd);
97    WSACleanup();
98    exit(error);
99  }
100
101 //функция для записи в сокет
102 //SOCKET socks - клиентский сокет для закрытия в случае ошибки
103 //char* buffer - данные для записи
104 void sendAll(SOCKET sockfd, char buffer[]){
105     int messageLength = strlen(buffer);
106     char toSend[4 + 256];
107   memset(toSend, 0, 4 + 255);
108     toSend[0] = ((messageLength >> 24) & 0xff) + '0';
109     toSend[1] = ((messageLength >> 16) & 0xff) + '0';
110     toSend[2] = ((messageLength >> 8) & 0xff) + '0';
111     toSend[3] = ((messageLength >> 0) & 0xff) + '0';
112
113     strcat(toSend, buffer);
114
115     int n = send(sockfd, toSend, strlen(toSend), 0);
116     if (n < 0) {
117         closeSocket(sockfd, 1, "ERROR writing to socket");
```

```
118     }
119 }
120
121 //функция для чтения из сокета
122 //SOCKET socks - клиентский сокет для закрытия в случае ошибки
123 //возвращаемое значение - считанные данные
124 char* readAll(SOCKET sockfd){
125     char *buffer = (char*)malloc(256);
126     char strLenght[4];
127
128     int n = recv(sockfd, strLenght, 4, 0);
129     if (n < 0) {
130         closeSocket(sockfd, 1, "ERROR reading from socket");
131     }
132     int lenght = ((strLenght[0] - '0') << 24) + ((strLenght[1] - '0') << 16) +
    ↪ ((strLenght[2] - '0') << 8) + (strLenght[3] - '0');
133
134   memset(buffer, 0, 256);
135     int recieved = 0;
136     while(recieved < lenght){
137         n = recv(sockfd, buffer, 256, 0);
138         recieved += n;
139         if (n < 0) {
140             closeSocket(sockfd, 1, "ERROR reading from socket");
141         }
142     }
143
144     return buffer;
145 }
```

4.3. Реализация UDP на Linux

Листинг 7: UDP сервер на Linux

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include <netdb.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7
8 #include <string.h>
9
10 //функция закрытия сокетов в случае ошибки
11 void closeSocket(int sockfd, int error, char* errorMsg);
12 int main(int argc, char *argv[]) {
13     int sockfd;
14     uint16_t portno = 5001;
15     char buffer[256];
16     struct sockaddr_in serv_addr, cli_addr;
17     unsigned int clilen = sizeof(cli_addr);
18     ssize_t n;
19
20     /* First call to socket() function */
21     sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
22
23     if (sockfd < 0) {
24         perror("ERROR opening socket");
25         exit(1);
```

```
26      }
27
28      /* Initialize socket structure */
29      bzero((char *) &serv_addr, sizeof(serv_addr));
30
31      serv_addr.sin_family = AF_INET;
32      serv_addr.sin_addr.s_addr = INADDR_ANY;
33      serv_addr.sin_port = htons(portno);
34
35      if(setsockopt(sockfd, SOL_SOCKET,(SO_REUSEPORT | SO_REUSEADDR),&(int){ 1 },
    ↪ sizeof(int)) < 0){
36          closeSocket(sockfd, 1, "ERROR on setsockopt");
37      }
38
39      /* Now bind the host address using bind() call.*/
40      if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
41          closeSocket(sockfd, 1, "ERROR on binding");
42      }
43
44      if(fork() > 0){
45          while(getchar() != 'q'){
46          }
47          closeSocket(sockfd, 0, "");
48      }
49
50      while (1) {
51          bzero(buffer, 256);
52          n = recvfrom(sockfd, buffer, 256, 0, (struct sockaddr *) &cli_addr, &
    ↪ clilen);
53          if (n < 0) {
54              closeSocket(sockfd, 1, "ERROR reading from socket");
55          }
56          printf("Here is the message: %s\n", buffer);
57
58          n = sendto(sockfd, "I got your message", 18, 0, (struct sockaddr *) &
    ↪ cli_addr, clilen);
59          if (n < 0) {
60              closeSocket(sockfd, 1, "ERROR writing to socket");
61          }
62      }
63
64      return 0;
65  }
66
67  //функция для закрытия сокета
68  //int socks - сокет для закрытия
69  //int error - код ошибки
70  //char* errorMsg - сообщение ошибки
71  void closeSocket(int sockfd, int error, char* errorMsg){
72      if(strcmp(errorMsg,"") != 0){
73          perror(errorMsg);
74      }
75      shutdown(sockfd, SHUT_RDWR);
76      close(sockfd);
77      exit(error);
78  }
```

Листинг 8: UDP клиент на Linux

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```
3
4 #include <netdb.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include<arpa/inet.h>
8
9 #include <string.h>
10
11 //функция закрытия сокетов в случае ошибки
12 void closeSocket(int sockfd, int error, char* errorMsg);
13 int main(int argc, char *argv[]) {
14     int sockfd, n;
15     uint16_t portno;
16     struct sockaddr_in serv_addr;
17     unsigned int servlen = sizeof(serv_addr);
18
19     char buffer[256];
20
21     if (argc < 3) {
22         fprintf(stderr, "usage %s hostname port\n", argv[0]);
23         exit(0);
24     }
25
26     portno = (uint16_t) atoi(argv[2]);
27
28     /* Create a socket point */
29     sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
30
31     if (sockfd < 0) {
32         perror("ERROR opening socket");
33         exit(1);
34     }
35
36     bzero((char *) &serv_addr, sizeof(serv_addr));
37     serv_addr.sin_family = AF_INET;
38     serv_addr.sin_port = htons(portno);
39
40     if (inet_aton(argv[1], &serv_addr.sin_addr) == 0){
41         closeSocket(sockfd, 1, "ERROR connecting");
42     }
43
44     /* Now ask for a message from the user, this message
45        * will be read by server
46     */
47
48     printf("Please enter the message: ");
49     bzero(buffer, 256);
50     fgets(buffer, 255, stdin);
51
52     /* Send message to the server */
53     n = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr *) &serv_addr
    ↪ , servlen);
54
55     if (n < 0) {
56         closeSocket(sockfd, 1, "ERROR writing to socket");
57     }
58
59     /* Now read server response */
60     bzero(buffer, 256);
61     n = recvfrom(sockfd, buffer, 255, 0, (struct sockaddr *) &serv_addr, &
```

```
    ↪ servlen);
62
63      if (n < 0) {
64          closeSocket(sockfd, 1, "ERROR reading from socket");
65      }
66
67      printf("%s\n", buffer);
68
69      close(sockfd);
70      return 0;
71  }
72
73  //функция для закрытия сокета
74  //int socks - сокет для закрытия
75  //int error - код ошибки
76  //char* errorMsg - сообщение ошибки
77  void closeSocket(int sockfd, int error, char* errorMsg){
78      if(strcmp(errorMsg,"") != 0){
79          perror(errorMsg);
80      }
81      shutdown(sockfd, SHUT_RDWR);
82      close(sockfd);
83      exit(error);
84  }
```

4.4. Индивидуальное задание. Сервис коммунальных платежей

Задание. Разработать приложение–клиент и приложение–сервер Сервиса коммунальных платежей. Управляющая компания регистрирует счетчики с уникальными номерами на Абонента. Абонент вносит показания Счетчиков.

Основные возможности. Серверное приложение должно реализовывать следующие функции:

1. Прослушивание определенного порта
2. Обработка запросов на подключение по этому порту от пользователей Сервиса (Управляющая компания, Абонент)
3. Поддержка одновременной работы нескольких пользователей Сервиса через механизм нитей
4. Осуществление добавления счетчиков Управляющей компанией, назначение Счетчиков на Абонентов
5. Прием запросов на:
   (a) фиксацию показаний счетчиков Абонентом на заданное число;
   (b) отображение истории показаний счетчика для Абонента и Управляющей компании с отображением приращения;
   (c) для Управляющей компании: отображение счетчиков и Абонентов, которые не вносили показания на заданное число;
6. Обработка запроса на отключение клиента
7. Принудительное отключение клиента

Клиентское приложение должно реализовывать следующие функции:

- Установление соединения с сервером
- Передача запросов серверу
- Получение ответов на запросы от сервера
- Разрыв соединения
- Обработка ситуации отключения клиента сервером

4.4.1. Протокол работы

Сперва решим какой протокол на лучше использовать для нашего задания TCP или UDP. Принципиально нет никакой разницы. TCP гарантированно доставит данные и не требует от разработчика дополнительных усилий для получения данных в правильном порядке и в принципе для их получения (не считая того, что сперва мы отправляем длину посылки). Еще одна причина - автоматически поддерживается соединение с клиентом. UDP тоже может быть основой данного задания, но потребуется больше затрат времени и логика программы усложнится.

Следующим шагом необходимо организовать протокол общения. В нашей системе два вида клиентов: управляющие и абоненты. Необходимо разграничивать их и предоставлять соответствующие права доступа. Первым делом запускается сервер и ожидает клиентов. Каждому клиенту выделяется отдельный поток для работы с ним. Далее клиенту отправляется приветственная строка и запрашивается логин и пароль. Запрос происходит до тех пор пока клиент не введет корректный логин и не войдет в систему. На стороне клиента происходит валидация логина и пароля на корректность.

В случае успешной авторизации клиенту отправляется сообщение двух видов:

- для пользователя

```
1 Вы вошли в систему как пользователь. Выберете дальнейшее действие:
```

- для администратора

```
1 Вы вошли в систему как администратор. Выберете дальнейшее действие:
```

Далее отправляется меню для управления своими данными.

- для пользователя

```
1 1. Фиксация показаний счетчиков на заданное число;
2 2. Отображение истории показаний счетчика;
3 3. Выход из системы;
```

- для администратора

```
1 1. Отображение истории показаний счетчика для заданного абонента;
2 2. Отображение счетчиков и Абонентов, которые не вносили показания на заданное
    ↪ число;
3 3. Выход из системы;
```

На этом шаге нужно сделать важное замечание. И сервер, и клиент в своих сообщениях, в начале указывают 4 байта размера сообщения. Каждый байт данных, включая размер сообщение интерпретируются как символы.

Каждый из клиентов должен отправить номер интересующего его пункта меню. В случае некорректного ввода сервер отправляет одно из следующих сообщений:

- Неверный ввод, введите число;
- Клиент не найден;
- На сервере возникли неполадки, пожалуйста попробуйте позже;
- Неверный формат ввода;
- Неверный ввод, введите число в заданном пределе.

После выбора пункта меню сервер отправляет дополнительную информацию о выбранном пункте и затем еще одну строку в которой запрашивается требуемая информация. При вводе некорректной информации сервер отправит одну из ошибок описанных выше и вновь отправит меню. Все вычисления происходят на стороне сервера, клиент получает уже готовый вариант.

Код клиента реализован предельно просто. Каждый этап обмена с сервером состоит из 3-х шагов: чтение с сервера, запись на сервер, опять чтение. Каждое действие на сервере выполняется в соответствии с этим условием, что позволяет написать один небольшой цикл на клиенте не заботясь о содержимом принимаемой или отправляемой информации.

Данный факт позволяет нам сделать из консольного приложения графическое изменив лишь код клиента. Сервер посылает нам определенные текстовые запросы, понятные человеку. Анализируя эти сообщения в коде, мы можем понять, что пришло от сервера. Пользователь на стороне клиента может использовать GUI интерфейс прозрачно, не заботясь о протоколе обмена.

При отключении сервера каждому клиенту отправляется сообщение об отключении, тем самым достигается корректное завершение работы.

4.4.2. Тонкости реализации

Данные для авторизации отправляются на сервер и происходит поиск в базе данных. Для данного задания была использована база данных PostgreSQL. Схема созданной БД показана на рисунке ниже.

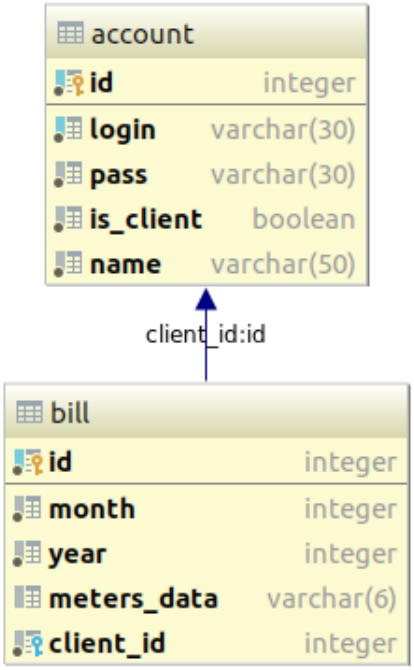

Рисунок 4.1. Диаграмма базы данных

На этой схеме представлены две таблицы. Таблица account отвечает за хранение пользовательских логинов и паролей, имени и тип клиента (абонент либо управляющий). Таблица bill содержит только абонентов. Там содержаться данные об оплате счетов.

4.4.3. Исходный код индивидуального задания

Весь код расположен по адресу https://github.com/vana06/NetworksLab2018/tree/individual.

4.5. Реализация SOCKS5 на Java

SOCKS (сокращение от «SOCKet Secure») — сетевой протокол, который позволяет пересылать пакеты от клиента к серверу через прокси-сервер прозрачно (незаметно для них) и таким образом использовать сервисы за межсетевыми экранами (фаерволами).

Более поздняя версия SOCKS5 предполагает аутентификацию, так что только авторизованные пользователи получают доступ к серверу.

Клиенты за межсетевым экраном, нуждающиеся в доступе к внешним серверам, вместо этого могут быть соединены с SOCKS-прокси-сервером. Такой прокси-сервер контролирует права клиента на доступ к внешним ресурсам и передаёт клиентский запрос внешнему серверу. SOCKS может использоваться и противоположным способом, осуществляя контроль прав внешних клиентов соединяться с внутренними серверами, находящимися за межсетевым экраном (брандмауэром).

SOCKS 5 расширяет модель SOCKS 4, добавляя к ней поддержку UDP, обеспечение универсальных схем строгой аутентификации и расширяет методы адресации, добавляя поддержку доменных имен и адресов IPv6. Начальная установка связи теперь состоит из следующего:

- Клиент подключается, и посылает приветствие, которое включает перечень поддерживаемых методов аутентификации

- Сервер выбирает из них один (или посылает ответ о неудаче запроса, если ни один из предложенных методов не приемлем)
- В зависимости от выбранного метода, между клиентом и сервером может пройти некоторое количество сообщений
- Клиент посылает запрос на соединение, аналогично SOCKS 4
- Сервер отвечает, аналогично SOCKS 4

Методы аутентификации пронумерованы следующим образом:

0x00	Аутентификация не требуется
0x01	GSSAPI
0x02	Имя пользователя / пароль
0x03-0x7F	Зарезервировано IANA
0x80-0xFE	Зарезервировано для методов частного использования

Рисунок 4.2. Методы аутентификации

Из данного списка был реализован только метод не требующий аутентификации.

Размер	Описание
1 байт	Номер версии SOCKS (должен быть 0x05 для этой версии)
1 байт	Количество поддерживаемых методов аутентификации
n байт	Номера методов аутентификации, переменная длина, 1 байт для каждого поддерживаемого метода

Рисунок 4.3. Начальное приветствие от клиента

Размер	Описание
1 байт	Номер версии SOCKS (должен быть 0x05 для этой версии)
1 байт	Выбранный метод аутентификации или 0xFF, если не было предложено приемлемого метода

Рисунок 4.4. Сервер сообщает о своём выборе

Последующая идентификация зависит от выбранного метода.

Размер	Описание
1 байт	Номер версии SOCKS (должен быть 0x05 для этой версии)
1 байт	Код команды: • 0x01 = установка TCP/IP соединения • 0x02 = назначение TCP/IP порта (binding) • 0x03 = ассоциирование UDP-порта
1 байт	Зарезервированный байт, должен быть 0x00
1 байт	Тип адреса: • 0x01 = адрес IPv4 • 0x03 = имя домена • 0x04 = адрес IPv6
Зависит от типа адреса	Назначение адреса: • 4 байта для адреса IPv4 • Первый байт — длина имени, затем следует имя домена без завершающего нуля на конце • 16 байт для адреса IPv6
2 байта	Номер порта, в порядке от старшего к младшему (big-endian)

Рисунок 4.5. Запрос клиента

Поддерживаются только коды команд для установки TCP/IP соединения, а тип адреса только IPv4.

Размер	Описание
1 байт	Номер версии SOCKS (0x05 для этой версии)
1 байт	Код ответа: • 0x00 = запрос предоставлен • 0x01 = ошибка SOCKS-сервера • 0x02 = соединение запрещено набором правил • 0x03 = сеть недоступна • 0x04 = хост недоступен • 0x05 = отказ в соединении • 0x06 = истечение TTL • 0x07 = команда не поддерживается / ошибка протокола • 0x08 = тип адреса не поддерживается
1 байт	Байт зарезервирован, должен быть 0x00
1 байт	Тип последующего адреса: • 0x01 = адрес IPv4 • 0x03 = имя домена • 0x04 = адрес IPv6
Зависит от типа адреса	Назначение адреса: • 4 байта для адреса IPv4 • Первый байт — длина имени, затем следует имя домена без завершающего нуля на конце • 16 байт для адреса IPv6
2 байта	Номер порта, в порядке от старшего к младшему (big-endian)

Рисунок 4.6. Ответ сервера

Листинг программы приведен ниже.

Листинг 9: SOCKS5 сервер

```
1 import java.io.IOException;
2 import java.net.*;
3 import java.nio.ByteBuffer;
4 import java.nio.channels.*;
```

```
5  import java.util.Arrays;
6  import java.util.Iterator;
7  import java.util.StringTokenizer;
8
9  public class Socks5Proxy {
10
11     private enum ERRORS {
12         ALL_OK,
13         SOCKS_SERVER_ERROR,
14         CONNECTION_DENIED,
15         NETWORK_UNAVAILABLE,
16         HOST_UNAVAILABLE,
17         CONNECTION_FAILURE,
18         TTL_EXPIRATION,
19         BAD_REQUEST,
20         ADDRESS_TYPE_UNAVAILABLE,
21         NONE
22     };
23
24     static class Attachment {
25         private static final byte SOCKS_VERSION = 0x05;
26         private static final byte[] OK = new byte[] { SOCKS_VERSION, 0x00 };
27         private static final byte[] FAIL_GREETINGS = new byte[]{ SOCKS_VERSION,
       ↪ (byte) 0xFF };
28         private static byte[] RESPONSE_TEMPLATE = new byte[] { SOCKS_VERSION, 0
       ↪ x01, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00};
29
30         boolean isFirstGreeting = true;
31         boolean closeAfterWrite = false;
32         /**
33          * Буфер для чтения, в момент проксирования становится буфером для
34          * записи для ключа хранимого в peer
35          */
36         ByteBuffer in;
37         /**
38          * Буфер для записи, в момент проксирования равен буферу для чтения для
39          * ключа хранимого в peer
40          */
41         ByteBuffer out;
42         /**
43          * Куда проксируем
44          */
45         SelectionKey peer;
46
47         Attachment(){
48             in = ByteBuffer.allocate(8192);
49             out = ByteBuffer.allocate(8192);
50         }
51
52         private void close(){
53             if(peer != null) {
54                 Attachment peerKey = ((Attachment) peer.attachment());
55                 if (peerKey != null && peerKey.peer != null) {
56                     peerKey.peer.cancel();
57                 }
58                 peer.cancel();
59             }
60         }
61
62         private void responseToGreetings (boolean accept, SelectionKey key){
```

```
63              closeAfterWrite = !accept;
64              if(accept){
65                  out.put(OK).flip();
66              } else {
67                  out.put(FAIL_GREETINGS).flip();
68              }
69              key.interestOps(SelectionKey.OP_WRITE);
70              isFirstGreeting = false;
71          }
72
73          private void responseToHeader (ERRORS responseCode, SelectionKey key,
    ↪ byte[] ip, byte[] port){
74              if(responseCode == ERRORS.NONE)
75                  return;
76
77              byte[] response = RESPONSE_TEMPLATE;
78              response[1] = (byte)responseCode.ordinal();
79              System.arraycopy(ip, 0, response, 4, ip.length);
80              System.arraycopy(port, 0, response, 8, port.length-2);
81
82              if(responseCode != ERRORS.ALL_OK){
83                  closeAfterWrite = true;
84                  out.put(response).flip();
85                  key.interestOps(SelectionKey.OP_WRITE);
86              } else {
87                  closeAfterWrite = false;
88                  in.put(response).flip();
89
90                  out = ((Attachment) peer.attachment()).in;
91                  ((Attachment) peer.attachment()).out = in;
92
93                  peer.interestOps(SelectionKey.OP_WRITE | SelectionKey.OP_READ);
94                  key.interestOps(0);
95              }
96          }
97
98
99      }
100
101     public static void main(String[] args) {
102         int port = 1080;
103         String host = "127.0.0.1";
104
105         // Открываем серверный канал
106         try (ServerSocketChannel serverChannel = ServerSocketChannel.open();
107              Selector selector = Selector.open()){
108             serverChannel.configureBlocking(false);
109             serverChannel.socket().bind(new InetSocketAddress(host, port));
110
111             // Регистрация в селекторе
112             serverChannel.register(selector, SelectionKey.OP_ACCEPT);
113             // Основной цикл работу неблокирующего сервер
114             while (selector.select() > -1) {
115                 Iterator<SelectionKey> iterator = selector.selectedKeys().
    ↪ iterator();
116                 while (iterator.hasNext()) {
117                     SelectionKey key = iterator.next();
118                     iterator.remove();
119                     System.out.println(key);
120                     if (key.isValid()) {
```

```
121                         // Обработка всех возможнных событий ключа
122                         try {
123                             if (key.isAcceptable()) {
124                                 // Принимаем соединение
125                                 accept(key);
126                             } else if (key.isConnectable()) {
127                                 // Устанавливаем соединение
128                                 connect(key);
129                             } else if (key.isReadable()) {
130                                 // Читаем данные
131                                 read(key);
132                             } else if (key.isWritable()) {
133                                 // Пишем данные
134                                 write(key);
135                             }
136                         } catch (Exception e) {
137                             e.printStackTrace();
138                             close(key);
139                         }
140                     }
141                 }
142             }
143         } catch (IOException e) {
144             e.printStackTrace();
145         }
146     }
147 
148     private static void accept(SelectionKey key) throws IOException {
149         // Приняли
150         SocketChannel newChannel = ((ServerSocketChannel) key.channel()).accept
    ↪ ();
151         // Неблокирующий
152         newChannel.configureBlocking(false);
153         // Регистрируем в селекторе
154         newChannel.register(key.selector(), SelectionKey.OP_READ);
155     }
156 
157     private static void read(SelectionKey key) throws IOException{
158         SocketChannel channel = (SocketChannel) key.channel();
159         Attachment attachment = ((Attachment) key.attachment());
160         if (attachment == null) {
161             // инициализируем буферы
162             key.attach(attachment = new Attachment());
163         }
164 
165         if (channel.read(attachment.in) < 1) {
166             // -1 - разрыв;
167             // 0 - нет места в буфере
168             close(key);
169         } else if (attachment.peer == null) {
170             // если нету второго конца значит мы читаем заголовок
171             System.out.println("header " + Arrays.toString(attachment.in.array()
    ↪ ));
172             if(attachment.isFirstGreeting){
173                 //читаем либо первый пакет
174                 attachment.responseToGreetings(readGreetings(attachment), key);
175             } else {
176                 //либо читаем второй пакет, с адресом
177                 attachment.responseToHeader(readHeader(key, attachment), key,
    ↪ new byte[4], new byte[2]);
```

```
178             }
179             attachment.in.clear();
180         } else {
181             System.out.println("data " + Arrays.toString(attachment.in.array()))
    ↪ ;
182             // ну а если мы проксируем, то добавляем ко второму концу интерес записать
183             attachment.peer.interestOps(attachment.peer.interestOps() |
    ↪ SelectionKey.OP_WRITE);
184             // а у первого убираем интерес прочитать, т.к пока не записали
185             // текущие данные, читать ничего не будем
186             key.interestOps(key.interestOps() ^ SelectionKey.OP_READ);
187             // готовим буфер для записи
188             attachment.in.flip();
189             ((Attachment)attachment.peer.attachment()).out = attachment.in;
190         }
191     }
192     private static ERRORS readHeader(SelectionKey key, Attachment attachment)
    ↪ throws IOException{
193         byte[] ar = attachment.in.array();
194 
195         if(ar.length < 10){
196             return ERRORS.BAD_REQUEST;
197         }
198         //версия обязательно пятая
199         if(ar[0] != 5 || ar[1] != 1 || ar[2] != 0){
200             return ERRORS.BAD_REQUEST;
201         }
202 
203         //обязательно ipv4
204         if(ar[3] != 1){
205             return ERRORS.ADDRESS_TYPE_UNAVAILABLE;
206         }
207 
208         // Создаём соединение
209         SocketChannel peer = SocketChannel.open();
210         peer.configureBlocking(false);
211         // Получаем из канала адрес и порт
212         byte[] addr = new byte[] { ar[4], ar[5], ar[6], ar[7] };
213         int p = (((int)ar[8] & 0xFF) << 8) + ((int)ar[9] & 0xFF);
214         // Начинаем устанавливать соединение
215         peer.connect(new InetSocketAddress(InetAddress.getByAddress(addr), p));
216         SelectionKey peerKey = peer.register(key.selector(), SelectionKey.
    ↪ OP_CONNECT);
217         // Обмен ключами
218         attachment.peer = peerKey;
219         Attachment peerAttachment = new Attachment();
220         peerAttachment.peer = key;
221         peerKey.attach(peerAttachment);
222 
223         return ERRORS.NONE;
224     }
225     private static boolean readGreetings(Attachment attachment){
226         byte[] ar = attachment.in.array();
227 
228         if(ar.length < 2){
229             return false;
230         }
231         //версия обязательно пятая
232         if(ar[0] != 5){
233             return false;
```

```
234         }
235         //количесто методов должно быть больше 0
236         int methodNumber = ar[1];
237         if(methodNumber <= 0){
238             return false;
239         }
240         //поддерживается единственный метод
241         for (int i = 2; i < methodNumber + 2; i++){
242             if(ar[i] == 0){
243                 return true;
244             }
245         }
246         return false;
247     }
248 
249     private static void write(SelectionKey key) throws IOException {
250         SocketChannel channel = ((SocketChannel) key.channel());
251         Attachment attachment = ((Attachment) key.attachment());
252         System.out.println("to write " + Arrays.toString(attachment.out.array())
    ↪ );
253 
254         if (channel.write(attachment.out) == -1) {
255             close(key);
256         } else if (attachment.out.remaining() ==  0 ) {
257             if(attachment.closeAfterWrite){
258                 //посылали ошибку и закрываем соединение
259                 close(key);
260             } else {
261                 if (attachment.peer == null) {
262                     //ждем посылку с адресом
263                     key.interestOps(SelectionKey.OP_READ);
264                 } else {
265                     // если всё записано, чистим буфер
266                     attachment.out.clear();
267                     // Добавялем ко второму концу интерес на чтение
268                     attachment.peer.interestOps(attachment.peer.interestOps() |
    ↪ SelectionKey.OP_READ);
269                     // А у своего убираем интерес на запись
270                     key.interestOps(key.interestOps() ^ SelectionKey.OP_WRITE);
271                 }
272             }
273         }
274 
275     }
276 
277     private static void connect(SelectionKey key) throws IOException {
278         SocketChannel channel = (SocketChannel) key.channel();
279         Attachment attachment = ((Attachment) key.attachment());
280         // Завершаем соединение
281         channel.finishConnect();
282         // Создаём буфер и отвечаем OK
283         StringBuilder sb = new StringBuilder(channel.getLocalAddress().toString
    ↪ ());
284         sb = sb.replace(0,1, "");
285 
286         StringTokenizer st = new StringTokenizer(sb.toString(), ":");
287         InetAddress ip = InetAddress.getByName(st.nextToken());
288         byte[] bytesIp = ip.getAddress();
289         byte[] port = ByteBuffer.allocate(4).putInt(Integer.valueOf(st.nextToken
    ↪ ())).array();
```

```
290
291             attachment.responseToHeader(ERRORS.ALL_OK, key, bytesIp, port);
292         }
293
294         private static void close(SelectionKey key) throws IOException {
295             key.cancel();
296             key.channel().close();
297             Attachment attachment = (Attachment) key.attachment();
298             if(attachment != null) {
299                 attachment.close();
300             }
301         }
302 }
```

5. Выводы

Сокеты оказались мощным средством организации обмена между процессами.
В данной работе мы создали простое приложение использующее TCP протокол.

1. При реализации индивидуального задания мы создали простой сервис на основе TCP, а также простой клиент для получения информации, которая хранится на сервере.

2. В качестве языка программирования был выбран C++, т.к. архитектуру данного сервиса проще отладить используя объекты и классы, а не функции.

3. Для хранения данных была выбрана СУБД, а не файлы. База данных это более гибкое хранилище. Оно позволяет удобно хранить и получать данные, выполнять транзакции, логировать действия клиентов. В случае потери данных, часть из них можно восстановить из логов. Разрабатываемая система может быть расширяемой, с множеством активных клиентов поэтому можно сказать, что использование СУБД оправдано.

4. Для логирования внутри системы была использована библиотека Boost.Log. Данная система может быть написана вручную, но для этого надо понимать все тонкости работы с потоками ввода/вывода. Boost.Log является универсальной, бесплатной, имеет гибкие настройки и самое главное просто запускается. Данные можно выводить в разном формате, в файл или на консоль и прочее.

Данную программу можно улучшить:

1. В данной работе используется принцип: 1 поток - 1 клиент. Это не самый лучший способ, но самый простой в реализации. Для улучшения можно использовать современные модели, например модель Actor. Наш сервис не занимается сложно обработкой данных, мы лишь получаем какие то данные из базы, слегка обрабатываем из и отправляем клиенту, т.е. нам не обязательно выделять целый поток на клиента. Актор в данной модели взаимодействует путём передачи сообщений с другими акторами, в ответ на получаемые сообщения может принимать локальные решения, создавать новые акторы, посылать свои сообщения, устанавливать, как следует реагировать на последующие сообщения. Объект актор занимает меньше места в памяти чем поток, поэтому мы можем создать гораздо большее число этих объектов чем потоков, тем самым мы можем выдерживать гораздо большее количество клиентов.

2. Еще один способ ускорения производительности это использование неблокирующих сокетов. В отличие от обычных, неблокирующие сокеты не останавливают поток на момент чтения, записи или ожидания подключения, тем самым мы можем более рационально использовать процессорное время.

3. При реализации сервиса, использующего аутентификацию, необходимо предусмотреть шифрование пересылаемых данных. В данной программ шифрование не применяется.

4. Хранения паролей нужно реализовывать в виде хэшей а не в открытом виде.