Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

ОТЧЕТ
по лабораторной работе

«Разработка прикладного протокола»

Технологии компьютерных систем

Работу выполнил студент

группа 43501/3 Крылов И.С.

Работу принял преподаватель

_____ Зозуля А.В.

Санкт-Петербург

2018

Содержание

Техническое задание

Система терминального доступа

- **Задание**

  Разработать клиент-серверную систему терминального доступа, позволяющую клиентам подсоединяться к серверу и выполнять элементарные команды операционной системы.

- **Основные возможности серверного приложения**

  1. Прослушивание определенного порта
  2. Обработка запросов на подключение по этому порту от клиентов
  3. Поддержка одновременной работы нескольких терминальных клиентов через механизм нитей
  4. Проведение аутентификации клиента на основе полученных имени пользователя и пароля
  5. Выполнение команд пользователя:

     > ls – выдача содержимого каталога
     > cd – смена текущего каталога
     > who – выдача списка зарегистрированных пользователей с указанием их текущего каталога
     > kill – Привилегированная команда. Завершение сеанса другого пользователя
     > logout – выход из системы

  6. Принудительное отключение клиента

- **Клиентское приложение должно реализовывать следующие функции:**

  1. Установление соединения с сервером
  2. Посылка аутентификационных данных клиента (имя и пароль)
  3. Посылка одной из команд (ls, cd, who, kill, logout) серверу
  4. Получение ответа от сервера
  5. Разрыв соединения
  6. Обработка ситуации отключения клиента сервером или другим клиентом

- **Настройки приложений**

  Разработанное клиентское приложение должно предоставлять пользователю настройку IP-адреса или доменного имени удалённого терминального сервера и номера порта, используемого сервером. Разработанное серверное приложение должно хранить аутентификационные

данные для вех пользователей, а также списки разрешенных каждому пользователю команд.

- **Методика тестирования**

  Для тестирования приложений запускаетсятерминальный сервер и несколько клиентов. В процессе тестирования проверяются основные возможности сервера по параллельной работе нескольких клиентов, имеющих различные привилегии (списки разрешенных команд). Проверяется корректность выполнения всех команда в различных ситуациях.

Прикладной протокол

Для реализации технического задания был разработан прикладной протокол передачи данных.

Запрос

Протоколом задаётся два формата запроса для взаимодействия клиента с сервером:

- запрос аутентификации с помощью пары логин:пароль 2.1
- запрос выполнения определённой команды 2.2

Login:Password:	Package Index
[0 - 507]	[508 - 511]

Таблица 2.1: Формат запроса аутентификации

Message Length	Command Descriptor	Command Parameters	Package Index
[0 - 3]	[4]	[5 - 507]	[508 - 511]

Таблица 2.2: Формат запроса выполнения команды

Оба запроса имеют одинаковый размер - 512 байт.
В таблице формата аутентификации 2.1:

- Login:Password - поле, содержащее передаваемые клиентом логин и пароль, необходимые для подключения к серверу. Протоколом задается формат ввода пары следующим образом:

  Поле занимает 508 байт, задавая тем самым максимально возможную длинну пары логин:пароль равной 508 символам.

[login] : [password] :

- Package Index - поле, хранящее индекс пересылаемого пакета. Благодаря наличию этого поля протокол гарантирует последовательный приём пакетов (защита от перемешивания). Так же, контроль номера пакета усложняет возможность атаки с имитацией адреса клиента поторонними. Длина поля - 4 байта, что позволяет обеспечить до 9999 последовательных запросов клиента серверу. В условиях технического задания данная продолжительность взаимодействия клиента с сервером более чем достаточна. При превышении этого значения клиент будет отключён от сервера. Тем самым гарантируется пресечение чрезмерно активного трафика, исходящего от клиента, который может свидетельствовать о зловредном характере запросов клиента

В таблице запроса выполнения команды 2.2:

- Message Length - длина параметров команды. В случае если команда не имеет параметров, данное поле заполняется нулями. Под данное поле выделено 4 байта.
- Command Descriptor - целое число - дескриптор команды, однозначно определяющий требуемую команду:
  - 1 – выдача содержимого каталога ls
  - 2 – смена текущего каталога cd
  - 3 – выдача списка зарегистрированных пользователей с указанием их текущего каталога who
  - 4 – Привилегированная команда. Завершение сеанса другого пользователя kill
  - 5 – выход из системы logout

  Список поддерживаемых протоколом команд ограничиваются пятью, вследствии чего под дескриптор задачи выделено поле в 1 байт. Использование схемы взаимодействия клиента с сервером посредством передачи дескприптора команды снимает с сервера задачу проверки правильности введённой с консоли пользователем команды, тем самым минимизируя количество пересылаемых пакетов и ускоряя работу сервера.
- Command Parameters - поле длинной 499 байт, содержит параметры команд: cd и kill
- Package Index - поле, хранящее индекс пересылаемого пакета. Идентично одноимённому полю запроса аутентификации

Ответ

В соответствии с имеющимися форматами запросов, протоколом определено два формата ответа:

- ответ аутентификации
- ответ выполнения команды

Ответ аутентификации - пакет длинной 1 байт, содержащий код результата аутентификации (Таблица 2.3):

- 0 - неверная пара логин:пароль, отказ в доступе
- 1 - аутентификация прошла успешно, получен доступ к удалённому терминалу
- 2 - пользователь с данным логином уже вошёл в систему с другого устройства, отказ в доступе
- * - неверный формат ввода, отказ в доступе. Имеет специальное назначение для ответа выполнения команды

Ответ выполнения команды - пакет длинной 8192 байт, содержащий результат выполнения команды на удалённом терминале (Таблица 2.4).

Исключительной ситуацией является попытка запроса на выполнение команды клиентом, сеанс которого был завершён другим пользователем, обладающим правами администратора. Такая ситуация возможна исключительно при использовании протокола UDP, вследствие того, что отключение клиента происходит по таймауту запроса ответа от сервера. В таком случае в качестве ответа на запрос посылается однобайтовый ответ аутентификации "* обёрнутый в формат ответа выполнения команды с помещением на первый (с индексом [0]) байт. Особенности устройства файловых систем ОС Windows и UNIX-подобных исключает возможность использования символа "*"в названиях файлов и директорий. Протоколом же запрещено создание логинов, начинающихся с символа "*". Тем самым исключается наличие и ошибочное обнаружение символа "*"в стандартном ответе выполнения команды.

Response
[0 - 1]

Таблица 2.3: Формат ответа аутентификации

Response
[0 - 8191]

Таблица 2.4: Формат ответа выполнения команды

Описание архитектур

Приложение было разбито на несколько модулей, реализующих имплементацию протокола с TCP и UDP, а также саму реализацию протокола.

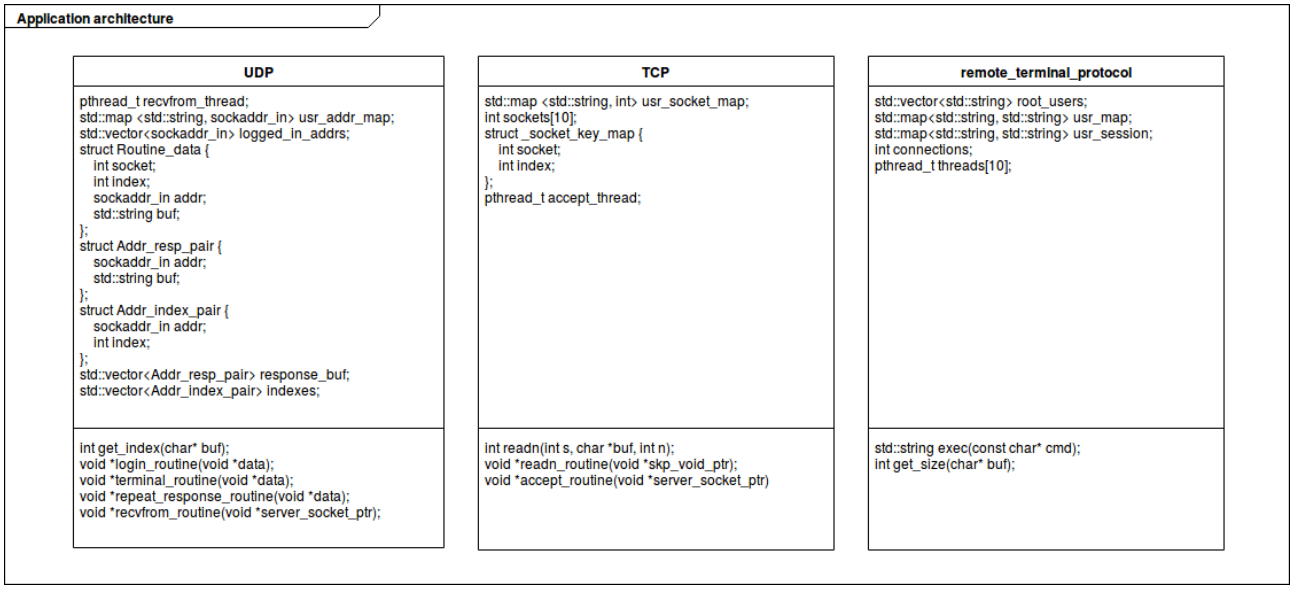

Рис. 3.1: Диаграмма модулей

Модуль UDP

- recvfrom_thread - дескриптор потока приёма всех входящих запросов
- usr_addr_map - структура данных, хранящая имя пользователя и соответсвующий данному имени адрес и порт подключения. Необходима для определения адреса отправки ответа по имени пользователя
- logged_in_addrs - список адресов подключенных пользователей. С помощью него можно определить - входящий пакет поступает от нового подключения или является пакетом сессии одного из уже подключенных пользователей
- Routine_data - структура данных, содержащая основную информацию для отправки ответа на входящих запрос. Используется для удобной траспортировки данных между потоками
- Addr_resp_pair - структура данных, объединяющая буфер ответа с адресом пользователя
- Addr_index_pair - структура данных для хранения данных нумерации пакетов в соответствии с адресом пользователя
- response_buf - буфер, хранящий предыдущий ответ для заданного адреса. Данные хранятся парами - каждому адресу ставится в соответствие

определенный буфер с ответом. Таким образом можно однозначно получить предыдущий ответ для конкретного адреса, в случае повторного постуления пакета запроса

- indexes - массив пар адрес:номер пакета. Обеспечивает хранение индексов пакетов индивидуально для каждой сессии

> get_index - функция распаковки значения индекса принятого запроса

Модуль TCP

- usr_socket_map - структура данных для хранения пар имя_пользователя:дескриптор_сокета. Позволяет соотнести заданного пользователя к конкретному сокету
- sockets - массив дескрипторов активных сокетов
- socket_key_map - структура данных для удобной передачи информации между потоками
- accept_thread - дескриптор потока приёма всех входящих запросов

Модуль remote_terminal_protocol

- root_users - список имён пользователей, обладающих привилигированными правами
- usr_map - пары логин:пароль зарегистрированные для доступа к удалённому терминалу
- usr_session - пара имя_пользователя:текущая_директория
- connections - счётчик активных подключений к серверу. Контролирует входную нагрузку

> exec - функция вызова входящей команды в терминале сервера

> get_size - функция распаковки значения фактической длины параметров принятого запроса

Особенности реализацции

Приложение на основе TCP

Серверное приложение

При запуске серверного приложения, в главном потоке main thread создаётся сокет и устанавливается на прослушку порта 7500. После чего порождается поток принятия новых соединений accept thread с функцией обработчиком accept_routine(), куда передаётся дескприптор созданного сокета.

Accept_routine() в бесконечном цикле принимает новые подключения, порождая новые потоки readn thread с назначенной функцией-обработчиком readn_routine(). Readn_routine() в бесконечном цикле принимает входящие данные с помощью функции readn, которая гарантирует целостность принятых данных. Затем, в том же потоке происходит аутентификация пользователя, отправка дескриптора результата входа в систему и в случае успешного входа в систему - выполнение переданной команды с последующим отправлением результата. По окончанию соединения, все потоки завершаются. Это гарантируется вызовом функции pthread_join(). Чтобы избежать конфликтов доступа и перезаписи данных, используются мьютексы. Примерное отображение процесса порождения и завершения потоков отображено на Рис.4.1.

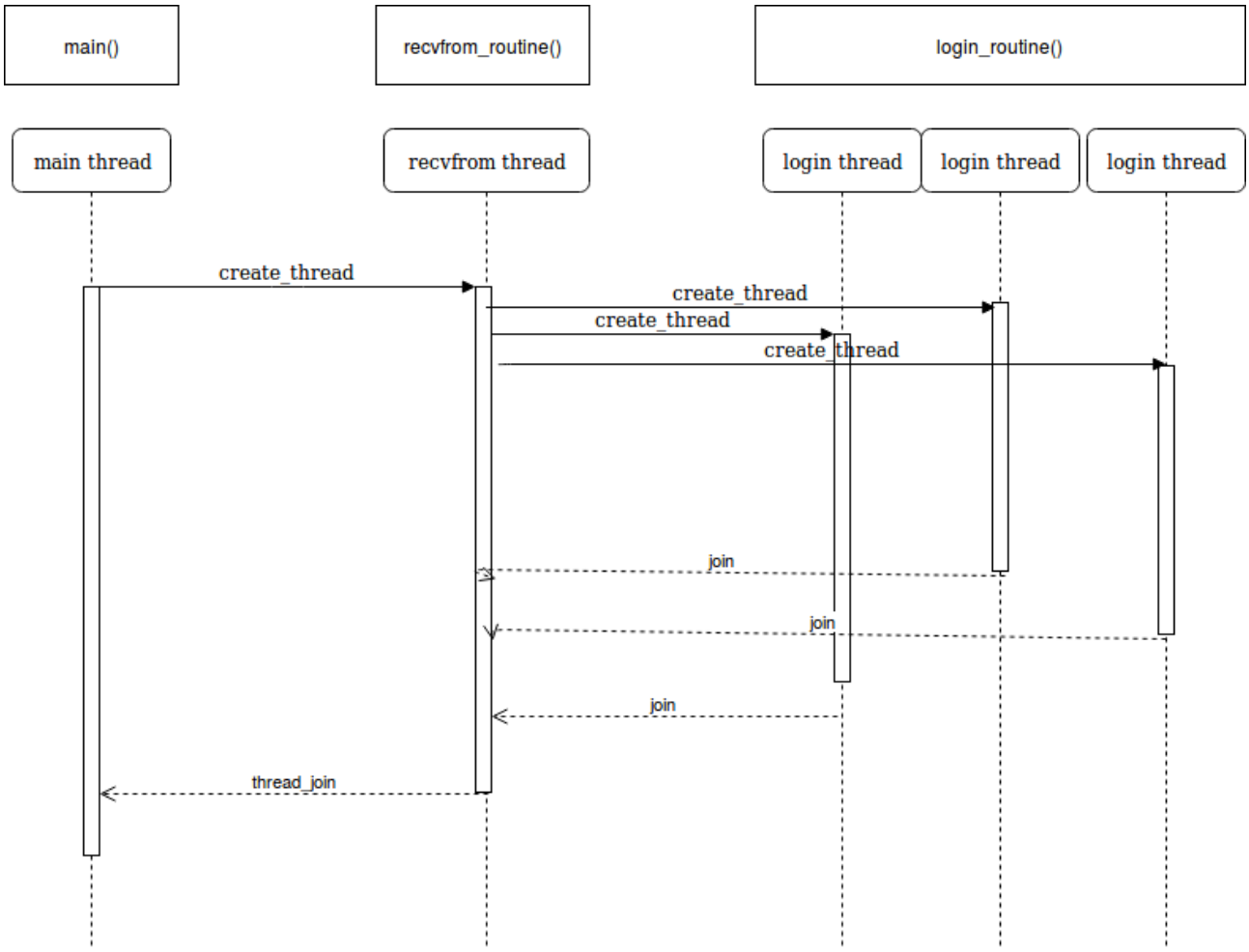

Рис. 4.1: Диаграмма примера потоков в серверном приложении TCP

Клиентское приложение

Клиентское приложение является однопоточным. Пользователю предоставляется возможность задать адрес и порт для установления соединения. Создаётся сокет, по которому происходит общение клиента с сервером. Адрес сервера и номер порта для подключения задаются явно. Взаимодействие с

сервером происходит последовательно в двух циклах. Первый цикл бесконечный с условием выхода по положительному ответу сервера на запрос доступа к удалённому терминалу. Во втором бесконечном цикле происходит обработка пользовательского ввода с клавиатуры, упаковка передаваемых данных в соответстви с протоколом 2.2 и отправка посредством команды send с последующим принятием ответа через функцию readn(). Выход из второго цикла и завершение работы клиента происходит либо при вводе пользователем команды logout, либо по решению администратора, либо при принудительной остановке работы сервера.

Приложение на основе UDP

Серверное приложение

В приложении с использованием UDP аналогично TCP создаётся сокет, привязываемый к конкретному порту, в нашем случае к порту 7500. В отличие от TCP никакого принятия новых соединений не происходит. В порождённом потоке recvfrom thread в функции recvfrom_routine напрямую принимаются UDP дэйтаграммы. Определение необходимого сеанса происходит по адрему отправителя. После принятия дэйтаграммы, определения её принадлежности к одной из активных сессий, а также проверки корректности индекса пакета, происходит порождение нового потока-обработчика с выполнением соответсвующей запросу функцией: login_routine или terminal_routine. Для ускорения работы сервера, в случае повторного приёма одного и того же пакета, порождается поток с функцией-обработчиком repeat_response_routine() - дублирующей предыдущий ответ по данному запросу. Пример порождения и завершения работы потоков представлен на Рис.4.2. Аналогично приложению на TCP все данные защищены мьютексами.

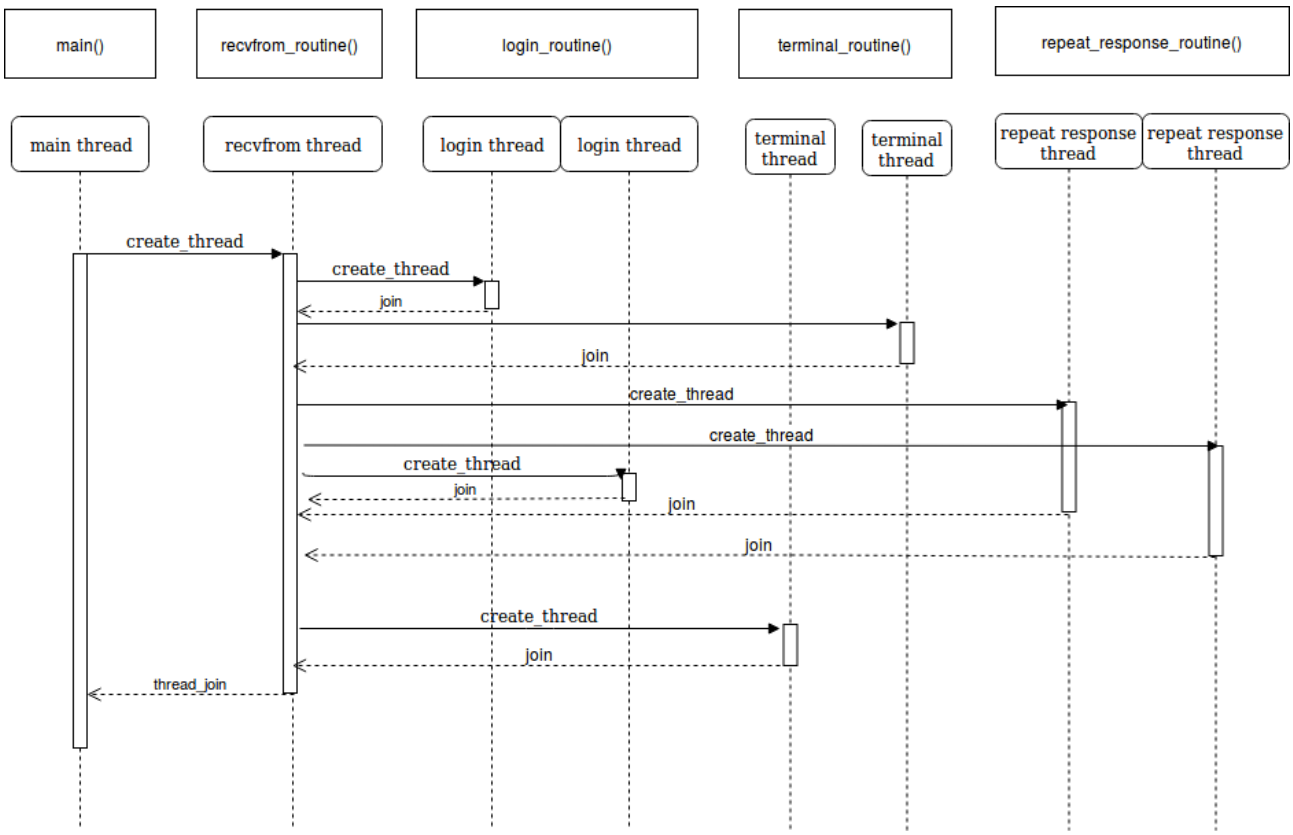

Рис. 4.2: Диаграмма примера потоков в серверном приложении UDP

Клиентское приложение

Клиентское приложение UDP похоже на TCP. Основным отличием является наличие таймаута ожидания ответа от сервера. В случае превышения таймаута, запрос отсылается ещё 10 раз, после чего приложение уведомляет пользователя о недоступности сервера. Принудительное завершение сеанса другим пользователем обеспечивается засчёт посылки сервером сообщения с дескриптором "*"означающим завершение сеанса.

Результаты тестирования

Приложение на основе TCP

Приложение было протестировано с помощью подключения трёх клиентов к работающему серверу. Были рассмотрены различные сценарии аутентификации, после чего протестированы все предусмотренные протоколом команды. Сначала был подключен пользователь ivan - с привилигированными правами. Затем была попытка повторной аутентификации с другого адреса, однако сервером было отказано в доступе, так как уже была активная сессия данного пользователя. Затем был подключен пользователь valik. С консоли ivan вызвана команда who - которая вывела обоих пользователей

и их текущие директории. Далее был подключен третий пользователь. Все три пользователя попробовали завершить чужой сеанс, однако завершение произошло только при вызове команды kill valik пользователем ivan, так он единственный обладал привилигированными правами. Затем ivan закончил свой сеанс, вызвав команду logout. Далее сервер прекратил работу по решению администратора со стороны сервера и пользовтель vaddya получил уведомление о закрытии сессии. Было протестировано завершение серверного приложения, клиентские сокеты и клиентские потоки успешно завершались, при этом происходило ожидание завершения потоков. Результаты проделанного эксперимента представлены в 1.

```
(ivan - priviliged user)
Log in: {user_name}:{password}:
ivan:0000:
Succesfully logged in
who
Received:
ivan /home/ivan/Documents/nets/remote_terminal/cmake-build-debug
valik /home/ivan/Documents/nets/remote_terminal/cmake-build-debug

ls
Received:
CMakeCache.txt
CMakeFiles
cmake_install.cmake
Makefile
remote_terminal
remote_terminal.cbp

kill ivan
Received:
You can not kill your own session. Use 'logout' command instead
kill valik
Received:
Succesful murder
logout
Session closed, logged out
Process finished with exit code 0
------------------------------------------------------------------
(valik)
Log in: {user_name}:{password}:
ivan:0000:
Already logged in in another session
Log in: {user_name}:{password}:
valik:1228:
Succesfully logged in
kill valik
Received:
Permission denied
ls
recv call failed: Success
```

```
Session closed, logged
------------------------------------------------------------------

(vaddya)
Log in: {user_name}:{password}:
vaddya:32283228:
Succesfully logged in
who
Received:
ivan /home/ivan/Documents/nets/remote_terminal/cmake-build-debug
vaddya /home/ivan/Documents/nets/remote_terminal/cmake-build-
   debug
valik /home/ivan/Documents/nets/remote_terminal/cmake-build-debug

kill valik
Received:
Permission denied
ls
Received:
CMakeCache.txt
CMakeFiles
cmake_install.cmake
Makefile
remote_terminal
remote_terminal.cbp

who
Received:
vaddya /home/ivan/Documents/nets/remote_terminal/cmake-build-
   debug

ls
recv call failed: Success
Session closed, logged out
```

Листинг 1: Проверка работоспособности приложения на TCP

Приложение на основе UDP

Приложение UDP было протестировано аналогичным образом. Были в отдельности рассмотрены ситуации:

- Нарушения порядка пакетов

```
(client)
Log in: {user_name}:{password}:
ivan:0000:
Succesfully logged in
ls
Server unreachable

Process finished with exit code 0
```

```
---------------------------------
(server)
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
```

Листинг 2: Нарушение порядка пакетов на UDP

Увеличим специально индекс отсылаемого пакета с запросом не на единицу, а на 2. В итоге можем наблюдать как клиент 10 раз достигает таймаута ожидания ответа от сервера, который отбрасывает пакет из за неверного значения индекса пакета.

- Отправка пакета-дубликата

```
(client)
Log in: {user_name}:{password}:
ivan:0000:
Succesfully logged in
ls
Server unreachable

Process finished with exit code 0
---------------------------------
(server)
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
Index error, package dropped with no response
```

Листинг 3: Отправка пакета-дубликата на UDP

Специально оставим индекс посылаемого пакета равным предыдущему. В итоге видим как сервер у себя выводит сообщение "Again"информирующее о повторном пакете-запросе, а на клиент приходит 1 - соответсвующая

дескриптору успешного логирования - ответа на предыдущий запрос клиента.

Выводы

В процессе выполнения работы был разработан прикладной протокол для удалённого доступа к терминалу: были определены форматы запроса и ответа. По описанию протокола были составлены модули клиентского и серверного приложений.

Было реализовано два клиент-серверных приложения, поддерживающих обмен данными по прикладному протоколу, поверх протокола TCP и UDP. Засчёт переноса части логики на сторону клиента, серверные приложения имеют схожую логику на TCP и UDP. Тем не менее принципиальным различием стала обработка запросов. В TCP это реализовано засчёт порождения отдельного единого потока, привязанного к выделенному сокету, посредством которого и происходит взаимодействие клиента с сервером в пределах одного сеанса. В случае с UDP такая модель оказалась невозможна, так как приём всех запросов приходится на один и тот же сокет. Работать с UDP оказалось несколько сложнее из за необходимости явно определять сеанс по адресу отправителя запроса.

Приложения

```
#include <sys/types.h>
#include <sys/socket.h>
#include <pthread.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <zconf.h>
#include <map>
#include <algorithm>
#include <vector>
#include <iostream>
#include <stdexcept>
#include <string>
#include <cstring>
#include <stddef.h>
#define _POSIX_SOURCE
#include <unistd.h>
#undef _POSIX_SOURCE



const int max_connections = 10;
const int message_length = 10;

const int login_pair_size = 64;
const int buf_size = 256;
const int buf_out_size = 8192;

std::map <std::string, std::string> usr_map;
std::vector<std::string> root_users;
std::map <std::string, std::string> usr_session;
std::map <std::string, int> usr_socket_map;

pthread_t accept_thread;
pthread_t threads[10];

pthread_mutex_t mutex;

int sockets[10];


int connections = 0;

struct _socket_key_map {
    int socket;
    int index;
};
typedef struct _socket_key_map socket_key_map;

```

```
50 std::string exec(const char* cmd) {
51     char buffer[128];
52     std::string result = "";
53     FILE* pipe = popen(cmd, "r");
54     if (!pipe) throw std::runtime_error("popen() failed!");
55     try {
56         while (!feof(pipe)) {
57             if (fgets(buffer, 128, pipe) != NULL)
58                 result += buffer;
59         }
60     } catch (...) {
61         pclose(pipe);
62         throw;
63     }
64     pclose(pipe);
65     return result;
66 }
67
68
69 int get_size(char* buf) {
70     int size;
71     size = (buf[0] - '0') * 1000 + (buf[1] - '0') * 100 +
72             (buf[2] - '0') * 10 + (buf[3] - '0');
73     return size-1;
74 }
75
76
77 int readn(int s, char *buf, int n)
78 {
79     int rc;
80     char tmp_buf[n];
81     int read = 0;
82
83     while (read < n) {
84         rc = recv(s, tmp_buf, n - read, 0);
85         if (rc < 0)
86             return -1;
87         if (rc <= n) {
88             for (int j = 0; j < rc; ++j) {
89                 buf[j + read] = tmp_buf[j];
90             }
91             read += rc;
92         }
93     }
94     return 1;
95 }
96
97
98 void *readn_routine(void *skp_void_ptr)
99 {
100     int rc;
101     auto *skp = (socket_key_map *)skp_void_ptr;
```

```
102     int socket = skp->socket;
103     int index;
104     int input_size;
105     char buff[message_length];
106     std::string output;
107     bool logout;
108     char login[1];
109 
110     char login_buf[login_pair_size];
111     char buf[ buf_size ];
112     char buf_out[buf_out_size];
113     char sub_buf[ buf_size ];
114     char* running;
115     char* u_name;
116     char* password;
117     char cwd[256];
118     int socket_index;
119     int chdir_ret;
120     std::map<std::string, std::string>::iterator it;
121 
122     //login loop
123     while (1) {
124         rc = readn(socket, login_buf, login_pair_size);
125         if (rc < 0)
126         {
127             perror("recv call failed");
128             break;
129         }
130         running = strdupa(login_buf);
131         u_name = strsep(&running, ":");
132         password = strsep(&running, ":");
133         if (usr_session.find(u_name) != usr_session.end()) {
134             login[0] = '2';
135             rc = send(socket, login, 1, 0);
136             if (rc <= 0) {
137                 perror("send call failed");
138                 break;
139             }
140         } else {
141             if (usr_map.find(u_name)->second == password) {
142                 login[0] = '1';
143                 rc = send(socket, login, 1, 0);
144                 if (rc <= 0) {
145                     perror("send call failed");
146                     break;
147                 }
148                 break;
149             } else {
150                 login[0] = '0';
151                 rc = send(socket, login, 1, 0);
152                 if (rc <= 0) {
153                     perror("send call failed");
```

```
154                     break;
155                 }
156             }
157         }
158     }
159
160     pthread_mutex_lock(&mutex);
161     usr_socket_map.insert(std::pair <std::string, int> (u_name,
    socket));
162     if (getcwd(cwd, sizeof(cwd)) == NULL)
163         perror("getcwd() error");
164     else
165         usr_session.insert(std::pair <std::string, std::string> (
    u_name, std::string(cwd)));
166     pthread_mutex_unlock(&mutex);
167
168     while (1) {
169         rc = readn(socket, buf, buf_size);
170         if (rc < 0)
171         {
172             perror("recv call failed");
173             break;
174         }
175         input_size = get_size(buf);
176
177         switch (buf[4]) {
178             case '1': {
179                 output = exec("ls");
180                 break;
181             }
182             case '2': {
183                 strncpy(sub_buf, buf+5, input_size);
184                 chdir_ret = chdir(sub_buf);
185                 if (chdir_ret < 0) {
186                     perror("Chdir error");
187                     output = "Unsuccesfull";
188                 } else {
189                     usr_session.find(u_name)->second = std::
    string(sub_buf);
190                     output = std::string(sub_buf);
191                 }
192                 break;
193             }
194             case '3': {
195                 for (it=usr_session.begin(); it!=usr_session.end
    (); ++it)
196                     output = output + it->first + ' ' + it->
    second + '\n';
197                 break;
198             }
199             case '4': {
```

```
200                 if (std::find(root_users.begin(), root_users.end
    (), u_name) != root_users.end()) {
201                     strncpy(sub_buf, buf + 5, input_size);
202                     if ( std::string(sub_buf) == u_name ) {
203                         output = "You can not kill your own
    session. Use 'logout' command instead";
204                     } else {
205                         socket_index = usr_socket_map.find(std::
    string(sub_buf))->second;
206                         rc = shutdown(socket_index, 2);
207                         if (rc < 0)
208                             perror("Failed to shutdown socket");
209 
210                         rc = close(socket_index);
211                         if (rc < 0)
212                             perror("Failed to close socket");
213                         output = "Succesful murder";
214                     }
215                 } else
216                 {
217                     output = "Permission denied";
218                 }
219                 break;
220             }
221             case '5': {
222                 logout = 1;
223                 break;
224             }
225             default: {
226                 continue;
227             }
228         }
229 
230         if (logout) {
231             pthread_mutex_lock(&mutex);
232             usr_socket_map.erase(u_name);
233             usr_session.erase(u_name);
234             pthread_mutex_unlock(&mutex);
235             break;
236         }
237         std::strcpy(buf_out, output.c_str());
238         rc = send(socket, buf_out, buf_out_size, 0);
239         if (rc <= 0)
240         {
241             perror("send call failed");
242             break;
243         }
244         memset(&buf_out, 0, sizeof(buf_out));
245         output.clear();
246     }
247     pthread_mutex_lock(&mutex);
248     for (int i = 0; i < connections; ++i) {
```

```
249         if (sockets[i] == socket) {
250             index = i;
251             for (int j = index; j < connections - 1; ++j) {
252                 sockets[j] = sockets[j + 1];
253             }
254             break;
255         }
256     }
257     connections -= 1;
258     usr_socket_map.erase(u_name);
259     usr_session.erase(u_name);
260     pthread_mutex_unlock(&mutex);
261 
262 }
263 
264 void *accept_routine(void *server_socket_ptr)
265 {
266     socket_key_map skp;
267     int socket;
268     int result;
269     int local_connections;
270     auto *skp_ptr = (socket_key_map *)server_socket_ptr;
271     int server_socket = skp_ptr->socket;
272 
273     while (1)
274     {
275         if (connections < max_connections)
276         {
277             socket = accept(server_socket, NULL, NULL);
278             if (socket < 0) {
279                 perror("accept call failed");
280                 break;
281             }
282             skp.index = connections;
283             skp.socket = socket;
284 
285             pthread_mutex_lock(&mutex);
286             sockets[connections] = socket;
287             pthread_create(&threads[connections], NULL, &
    readn_routine, &skp);
288             connections = connections + 1;
289             pthread_mutex_unlock(&mutex);
290 
291 
292         }
293         else
294             sleep(1);
295     }
296     //free(ptr);
297 
298     pthread_mutex_lock(&mutex);
299     local_connections = connections;
```

```
300     pthread_mutex_unlock(&mutex);
301
302     for(int i = 0; i < local_connections; ++i)
303     {
304         result = shutdown(sockets[0], 2);
305         if (result < 0)
306             perror("shutdown call failed");
307
308         result = close(sockets[0]);
309         if (result < 0)
310             perror("close call failed");
311
312         pthread_join(threads[i], NULL);
313     }
314
315
316 }
317
318
319 int main(void)
320 {
321     struct sockaddr_in local;
322     int rc;
323     bool quit = 0;
324     char in;
325     int socket_index;
326     socket_key_map skp;
327     int server_socket;
328
329     usr_map.insert(std::pair <std::string, std::string> ("vaddya"
    , "32283228"));
330     usr_map.insert(std::pair <std::string, std::string> ("
    lamtev2000", "iluvclassmates"));
331     usr_map.insert(std::pair <std::string, std::string> ("
    mikle_undef", "arguetill_theend"));
332     usr_map.insert(std::pair <std::string, std::string> ("valik",
     "1228"));
333     usr_map.insert(std::pair <std::string, std::string> ("ivan",
    "0000"));
334     root_users.push_back("ivan");
335
336     local.sin_family = AF_INET;
337     local.sin_port = htons(7500);
338     local.sin_addr.s_addr = htonl(INADDR_ANY);
339
340     skp.socket = socket(AF_INET, SOCK_STREAM, 0);
341     server_socket = skp.socket;
342
343     if (server_socket < 0) {
344         perror("socket call failed");
345         exit(1);
346     }
```

```
347
348     rc = bind(server_socket, (struct sockaddr *) &local, sizeof(
    local));
349
350     if (rc < 0) {
351         perror("bind call failure");
352         exit(1);
353     }
354
355     rc = listen(server_socket, 10);
356
357     if (rc) {
358         perror("listen call failed");
359         exit(1);
360     }
361
362     pthread_mutex_init(&mutex, NULL);
363     pthread_create(&accept_thread, NULL, &accept_routine, &skp);
364
365     while (1) {
366         in = (char) getc(stdin);
367         switch (in) {
368             case 'c': {
369                 socket_index = (char) getc(stdin) - '0';
370                 if (socket_index > connections) {
371                     perror("Index out of range");
372                     break;
373                 }
374                 rc = shutdown(sockets[socket_index], 2);
375                 if (rc < 0)
376                     perror("Failed to shutdown socket");
377
378                 rc = close(sockets[socket_index]);
379                 if (rc < 0)
380                     perror("Failed to close socket");
381
382                 break;
383             }
384             case 'l': {
385                 printf("Connections = %d \n", connections);
386                 for (int i = 0; i < connections; ++i) {
387                     printf("%d\t%d\n", i, sockets[i]);
388                 }
389                 break;
390             }
391             case 'q': {
392
393                 quit = 1;
394                 break;
395             }
396         }
397         if (quit)
```

```
398                 break;
399         }
400
401         rc = shutdown(server_socket, 2);
402         if (rc < 0)
403             perror("shutdown call failed");
404
405         rc = close(server_socket);
406         if (rc < 0)
407             perror("close call failed");
408
409         pthread_join(accept_thread, NULL);
410         pthread_mutex_destroy(&mutex);
411
412         return 0;
413 }
```

Листинг 4: TCP Server

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  #include <netinet/in.h>
4  #include <arpa/inet.h>
5  #include <stdio.h>
6  #include <cstdlib>
7  #include <iostream>
8  #include <string>
9  #include <cstring>
10
11 const int login_pair_size = 64;
12 const int buf_size = 256;
13 const int buf_out_size = 8192;
14 const int message_length_buffer_field = 4;
15
16
17 void pack_size(char* buf, ulong size)
18 {
19     char c;
20     for ( int i = 3; i >= 0; --i )
21         if ( size > 0 ) {
22             c = char(size % 10) + '0';
23             buf[i] = c;
24         } else {
25             buf[i] = '0';
26         }
27         size /= 10;
28     }
29 }
30
31
32 int readn(int s, char *buf, int n)
33 {
```

```
34      int rc;
35      char tmp_buf[n];
36      int read = 0;
37
38      while (read < n) {
39          rc = recv(s, tmp_buf, n - read, 0);
40          if (rc <= 0)
41              return -1;
42          if (rc <= n) {
43              for (int j = 0; j < rc; ++j) {
44                  buf[j + read] = tmp_buf[j];
45              }
46              read += rc;
47          }
48      }
49      return 1;
50  }
51
52
53  int main( void )
54  {
55      struct sockaddr_in peer;
56      int s;
57      int rc;
58      int command_descriptor_index = message_length_buffer_field;
59      bool noInput = 1;
60      char logInRes[1];
61      std::string str;
62      std::string substring;
63      std::string login_str;
64      login_str.resize(64);
65
66      char login_buf[login_pair_size];
67      char buf[ buf_size ];
68      char buf_out[buf_out_size];
69
70      peer.sin_family = AF_INET;
71      peer.sin_port = htons( 7500 );
72      peer.sin_addr.s_addr = inet_addr( "127.0.0.1" );
73
74      s = socket( AF_INET, SOCK_STREAM, 0 );
75      if ( s < 0 )
76      {
77          perror( "socket call failed" );
78          exit( 1 );
79      }
80      rc = connect( s, ( struct sockaddr * )&peer, sizeof( peer ) )
    ;
81      if ( rc )
82      {
83          perror( "connect call failed" );
84          exit( 1 );
```

```
85      }
86
87      //Log in loop
88      while (1) {
89          printf("Log in: {user_name}:{password}: \n");
90          std::getline(std::cin, login_str);
91          if (login_str.size() > login_pair_size -
    message_length_buffer_field) {
92              perror("Incorrect authentication length, should be
    less than 64");
93          } else {
94              rc = send(s, login_str.c_str(), login_pair_size, 0);
95              if (rc <= 0) {
96                  perror("send call failed");
97                  exit(1);
98              }
99              rc = readn(s, logInRes, 1);
100             if (rc <= 0) {
101                 perror("recv call failed");
102                 exit(1);
103             }
104             if (logInRes[0] == '1') {
105                 printf("Succesfully logged in \n");
106                 break;
107             } else if (logInRes[0] == '2')
108                 printf("Already logged in in another session \n")
    ;
109             else
110                 printf("Wrong login:password pair \n");
111         }
112     }
113
114
115     //main workflow loop
116     while(1) {
117         std::getline (std::cin, str);
118
119         if (str == "ls") {
120             //packing message size 0001 to buffer
121             for (int i = 0; i < message_length_buffer_field - 1;
    ++i) {
122                 buf[i] = '0';
123             }
124             buf[command_descriptor_index - 1] = '1';
125             buf[command_descriptor_index] = '1';
126             noInput = 0;;
127         }
128
129         else if (str.compare(0,2,"cd") == 0) {
130             for (int i = 0; i < message_length_buffer_field - 1;
    ++i) {
131                 buf[i] = '0';
```

```
132             }
133             buf[command_descriptor_index - 1] = '1';
134             buf[command_descriptor_index] = '2';
135             if (str[2] != ' ') {
136                 std::cout << "Incorrect input format" << std::
    endl;
137                 continue;
138             } else {
139                 substring = str.substr(3);
140                 if (substring.size() > buf_size -
    message_length_buffer_field) {
141                     perror("Out of input size, max length is 30
    symbols");
142                     break;
143                 } else {
144                     for (int i = 0; i < substring.size(); ++i) {
145                         buf[i + message_length_buffer_field + 1]
    = substring[i];
146                     }
147                     pack_size(buf, substring.size() + 1);
148                 }
149                 substring.clear();
150             }
151             noInput = 0;
152         }
153 
154         else if (str=="who") {
155             for (int i = 0; i < message_length_buffer_field - 1;
    ++i) {
156                 buf[i] = '0';
157             }
158             buf[command_descriptor_index - 1] = '1';
159             buf[command_descriptor_index] = '3';
160             noInput = 0;
161         }
162 
163         else if (str.compare(0,4,"kill") == 0) {
164             buf[command_descriptor_index] = '4';
165             if (str[4] != ' ') {
166                 perror("Incorrect input format");
167                 break;
168             } else {
169                 substring = str.substr(5);
170                 if (substring.size() > buf_size -
    message_length_buffer_field) {
171                     perror("Out of input size, max length is 30
    symbols");
172                     //break;
173                 } else {
174                     for (int i = 0; i < substring.size(); ++i) {
175                         buf[i + message_length_buffer_field + 1]
    = substring[i];
```

```
176                     }
177                     pack_size(buf, substring.size() + 1);
178                 }
179                 substring.clear();
180             }
181             noInput = 0;
182         }
183 
184         else if (str == "logout") {
185             for (int i = 0; i < message_length_buffer_field - 1;
    ++i) {
186                 buf[i] = '0';
187             }
188             buf[command_descriptor_index - 1] = '1';
189             buf[command_descriptor_index] = '5';
190             rc = send(s, buf, buf_size, 0);
191             if (rc <= 0)
192             {
193                 perror("send call failed");
194                 break;
195             }
196             break;
197         }
198 
199         else
200             noInput = 1;
201 
202         if (!noInput) {
203             rc = send(s, buf, buf_size, 0);
204             if (rc <= 0) {
205                 perror("send call failed");
206                 break;
207             }
208 
209 
210             rc = readn(s, buf_out, buf_out_size);
211             if (rc <= 0) {
212                 perror("recv call failed");
213                 break;
214             }
215 
216             std::cout << "Received: \n" << std::string(buf_out)
    << std::endl; //may not work */
217             memset(&buf_out, 0, sizeof(buf_out));
218         }
219 
220      }
221 
222      printf("Session closed, logged out");
223      exit( 0 );
224 }
```

Листинг 5: TCP Client

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>
#include <pthread.h>
#include <zconf.h>
#include <map>
#include <algorithm>
#include <vector>
#include <iostream>
#include <string>
#include <cstring>
#include <stddef.h>
#include <algorithm>

#define _POSIX_SOURCE
#include <unistd.h>
#include <arpa/inet.h>
#undef _POSIX_SOURCE
#define BUFLEN 512  //Max length of buffer

const int max_connections = 10;
const int buf_out_size = 8182;

pthread_t threads[10];
pthread_t recvfrom_thread;
int connections = 0;
pthread_mutex_t mutex;

std::map <std::string, std::string> usr_map;
std::vector<std::string> root_users;
std::map <std::string, std::string> usr_session;
std::map <std::string, sockaddr_in> usr_addr_map;
std::vector<sockaddr_in> logged_in_addrs;


struct Socket_key_map {
    int socket;
    int index;
};


struct Routine_data {
    int socket;
    int index;
```

```
50      sockaddr_in addr;
51      std::string buf;
52  };
53
54  struct Addr_resp_pair {
55      sockaddr_in addr;
56      std::string buf;
57  };
58
59  struct Addr_index_pair {
60      sockaddr_in addr;
61      int index;
62  };
63
64  std::vector<Addr_resp_pair> response_buf;
65  std::vector<Addr_index_pair> indexes;
66
67
68  std::string exec(const char* cmd) {
69      char buffer[128];
70      std::string result = "";
71      FILE* pipe = popen(cmd, "r");
72      if (!pipe) throw std::runtime_error("popen() failed!");
73      try {
74          while (!feof(pipe)) {
75              if (fgets(buffer, 128, pipe) != NULL)
76                  result += buffer;
77          }
78      } catch (...) {
79          pclose(pipe);
80          throw;
81      }
82      pclose(pipe);
83      return result;
84  }
85
86
87  int get_size(char* buf) {
88      int size;
89      size = (buf[0] - '0') * 1000 + (buf[1] - '0') * 100 +
90              (buf[2] - '0') * 10 + (buf[3] - '0');
91      return size-1;
92  }
93
94
95  int get_index(char* buf) {
96      int size;
97      size = (buf[BUFLEN - 4] - '0') * 1000 + (buf[BUFLEN - 3] - '0
    ') * 100 +
98              (buf[BUFLEN - 2] - '0') * 10 + (buf[BUFLEN - 1] - '0')
    ;
99      return size-1;
```

```
100 }
101
102
103 void *login_routine(void *data)
104 {
105     auto *rd_ptr = (Routine_data *)data;
106     int socket = rd_ptr->socket;
107     sockaddr_in addr = rd_ptr->addr;
108     unsigned int slen = sizeof(addr);
109     char login_buf[BUFLEN];
110     std::strcpy(login_buf, rd_ptr->buf.c_str());
111     char cwd[256];
112     std::string str_u_name;
113
114     int rc;
115     char login[1];
116     char* running;
117     char* u_name;
118     char* password;
119
120     std::map<std::string, std::string>::iterator it;
121     std::pair<std::map<std::string, sockaddr_in>::iterator, bool>
     res;
122
123     if (rd_ptr->buf.find(":") == std::string::npos)
124     {
125         login[0] = '*';
126         sendto(socket, login, 1, 0, (struct sockaddr *) &addr,
    slen);
127     } else {
128         running = strdupa(login_buf);
129         u_name = strsep(&running, ":");
130         password = strsep(&running, ":");
131         if (usr_session.find(u_name) != usr_session.end()) {
132             login[0] = '2';
133             rc = sendto(socket, login, 1, 0, (struct sockaddr *)
    &addr, slen);
134             if (rc <= 0) {
135                 perror("send call failed");
136             }
137         } else {
138             if (usr_map.find(u_name)->second == password) {
139                 login[0] = '1';
140                 rc = sendto(socket, login, 1, 0, (struct sockaddr
     *) &addr, slen);
141                 if (rc <= 0) {
142                     perror("send call failed");
143                 }
144             } else {
145                 login[0] = '0';
146                 rc = sendto(socket, login, 1, 0, (struct sockaddr
     *) &addr, slen);
```

```
147                 if (rc <= 0) {
148                     perror("send call failed");
149                 }
150             }
151         }
152 
153         pthread_mutex_lock(&mutex);
154         for (auto i = response_buf.begin(); i != response_buf.end
    ();) {
155             if ((inet_ntoa(i->addr.sin_addr) == inet_ntoa(addr.
    sin_addr)) &&
156                 (ntohs(i->addr.sin_port) == ntohs(addr.sin_port))
    ) {
157                 response_buf.erase(i);
158                 //break;
159             } else { ++i; }
160         }
161         if ( login[0] == '1' ) {
162             res = usr_addr_map.insert(std::pair<std::string,
    sockaddr_in>(u_name, addr));
163             if (!res.second) {
164                 usr_addr_map.erase(u_name);
165                 res = usr_addr_map.insert(std::pair<std::string,
    sockaddr_in>(u_name, addr));
166             }
167 
168             response_buf.emplace_back(Addr_resp_pair{addr, std::
    string(1, login[0])});
169             logged_in_addrs.push_back(addr);
170             if (getcwd(cwd, sizeof(cwd)) == NULL)
171                 perror("getcwd() error");
172             else {
173                 usr_session.insert(std::pair<std::string, std::
    string>(u_name, std::string(cwd)));
174             }
175         }
176     }
177 
178     connections--;
179     pthread_mutex_unlock(&mutex);
180 }
181 
182 
183 void *terminal_routine(void *data) {
184     int rc;
185     auto *rd_ptr = (Routine_data *) data;
186     int socket = rd_ptr->socket;
187     sockaddr_in addr = rd_ptr->addr;
188     unsigned int slen = sizeof(addr);
189     //struct sockaddr_in addr = rd_ptr->addr;
190     char buf[BUFLEN];
191     std::strcpy(buf, rd_ptr->buf.c_str());
```

```
192     int socket_index;
193     int chdir_ret;
194     char buf_out[buf_out_size];
195     char sub_buf[BUFLEN];
196     bool logout;
197     int input_size;
198     sockaddr_in victim_addr;
199     std::string output;
200     std::string u_name;
201     std::map<std::string, std::string>::iterator it;
202 
203     input_size = get_size(buf);
204     for (auto &i : usr_addr_map) {
205         if (inet_ntoa(i.second.sin_addr) == inet_ntoa(addr.
    sin_addr) &&
206             (ntohs(i.second.sin_port) == ntohs(addr.sin_port))) {
207             u_name = i.first;
208             break;
209         }
210     }
211 
212 
213     switch (buf[4]) {
214         case '1': {
215             output = exec("ls");
216             break;
217         }
218         case '2': {
219             strncpy(sub_buf, buf + 5, input_size);
220             chdir_ret = chdir(sub_buf);
221             if (chdir_ret < 0) {
222                 perror("Chdir error");
223                 output = "Unsuccesfull";
224             } else {
225                 usr_session.find(u_name)->second = std::string(
    sub_buf);
226                 output = std::string(sub_buf);
227             }
228             break;
229         }
230         case '3': {
231             for (it = usr_session.begin(); it != usr_session.end
    (); ++it)
232                 output = output + it->first + ' ' + it->second +
    '\n';
233             break;
234         }
235         case '4': {
236             if (std::find(root_users.begin(), root_users.end(),
    u_name) != root_users.end()) {
237                 strncpy(sub_buf, buf + 5, input_size);
238                 if (std::string(sub_buf) == u_name) {
```

```
239                     output = "You can not kill your own session.
    Use 'logout' command instead";
240                 } else {
241                     pthread_mutex_lock(&mutex);
242                     for (auto i = logged_in_addrs.begin(); i !=
    logged_in_addrs.end();) {
243                         if (inet_ntoa(i->sin_addr) == inet_ntoa(
    usr_addr_map.find(std::string(sub_buf))->second.sin_addr) &&
244                             (ntohs(i->sin_port) == ntohs(
    usr_addr_map.find(std::string(sub_buf))->second.sin_port))) {
245                             logged_in_addrs.erase(i);
246                         } else {++i;}
247                     }
248                     victim_addr = usr_addr_map.find(std::string(
    sub_buf))->second;
249                     usr_session.erase(std::string(sub_buf));
250                     usr_addr_map.erase(std::string(sub_buf));
251                     for (auto i = response_buf.begin(); i !=
    response_buf.end();) {
252                         if ((inet_ntoa(i->addr.sin_addr) ==
    inet_ntoa(victim_addr.sin_addr)) &&
253                             (ntohs(i->addr.sin_port) == ntohs(
    victim_addr.sin_port))) {
254                             response_buf.erase(i);
255                         } else {++i;}
256                     }
257                     for (auto i = indexes.begin(); i != indexes.
    end();) {
258                         if ((inet_ntoa(i->addr.sin_addr) ==
    inet_ntoa(victim_addr.sin_addr)) &&
259                             (ntohs(i->addr.sin_port) == ntohs(
    victim_addr.sin_port))) {
260                             indexes.erase(i);
261                             //break;
262                         } else {++i;}
263                     }
264                     pthread_mutex_unlock(&mutex);
265                     output = "Succesful murder";
266                 }
267             } else {
268                 output = "Permission denied";
269             }
270             break;
271         }
272         case '5': {
273             logout = 1;
274             break;
275         }
276     }
277
278     if (logout) {
279         pthread_mutex_lock(&mutex);
```

```
280         for (auto i = logged_in_addrs.begin(); i !=
    logged_in_addrs.end();) {
281             if ((inet_ntoa(i->sin_addr) == inet_ntoa(addr.
    sin_addr)) &&
282                 (ntohs(i->sin_port) == ntohs(addr.sin_port))) {
283                 logged_in_addrs.erase(i);
284                 //break;
285             } else {++i;}
286         }
287         usr_session.erase(u_name);
288         pthread_mutex_unlock(&mutex);
289     } else {
290         std::strcpy(buf_out, output.c_str());
291         rc = sendto(socket, buf_out, buf_out_size, 0, (struct
    sockaddr *) &addr, slen);
292         if (rc <= 0) {
293             perror("send call failed");
294         }
295     }
296     memset(&buf_out, 0, sizeof(buf_out));
297     output.clear();
298
299     pthread_mutex_lock(&mutex);
300     connections--;
301     pthread_mutex_unlock(&mutex);
302 }
303
304
305 void *repeat_response_routine(void *data)
306 {
307     auto *rd_ptr = (Routine_data *)data;
308     int socket = rd_ptr->socket;
309     int index = rd_ptr->index;
310     sockaddr_in addr = rd_ptr->addr;
311     unsigned int slen = sizeof(addr);
312     char buf[BUFLEN];
313
314     for ( auto &response : response_buf ) {
315         if ( ( inet_ntoa(response.addr.sin_addr) == inet_ntoa(
    addr.sin_addr) ) &&
316              ( ntohs(response.addr.sin_port) == ntohs(addr.
    sin_port) ))
317         {
318             sendto(socket, response.buf.c_str(), buf_out_size, 0,
     (sockaddr *) &addr, slen);
319             std::cout << "Again" << std::endl;
320         }
321     }
322
323     pthread_mutex_lock(&mutex);
324     connections--;
325     pthread_mutex_unlock(&mutex);
```

```
326 }
327
328
329 void *recvfrom_routine(void *server_socket_ptr)
330 {
331     sockaddr_in si_other;
332     char buf[BUFLEN];
333     auto *skp_ptr = (Socket_key_map *)server_socket_ptr;
334     int s = skp_ptr->socket, recv_len;
335     unsigned int slen = sizeof(si_other);
336     bool is_new = true;
337     bool already_requested = false;
338     int local_connections;
339     int index;
340     bool index_error = 0;
341     Routine_data rd;
342     Addr_index_pair addr_index_pair;
343     std::string temp = "Session closed by administrator";
344
345     while(1) {
346         if (connections < max_connections)
347         {
348             recv_len = recvfrom(s, buf, BUFLEN, 0, (struct
    sockaddr *) &si_other, &slen);
349             if ( recv_len < 0 )
350             {
351                 perror("recv_from error");
352                 break;
353             }
354             index = get_index(buf);
355             rd = {s, index, si_other, buf};
356
357             if ( index < 0)
358             {
359                 std::cout << "Incorrect package index, droping
    the package" << index <<std::endl;
360                 continue;
361             }
362
363             for ( sockaddr_in &e : logged_in_addrs ) {
364                 if (inet_ntoa(e.sin_addr) == inet_ntoa(si_other.
    sin_addr) &&
365                     (ntohs(e.sin_port) == ntohs(si_other.sin_port
    )))
366                 {
367                     is_new = 0;
368                     break;
369                 }
370             }
371
372             for ( Addr_index_pair &i : indexes ) {
373                 if ( ( inet_ntoa(i.addr.sin_addr) == inet_ntoa(
```

```
si_other.sin_addr) ) &&
374                     ( ntohs(i.addr.sin_port) == ntohs(si_other.
sin_port) ))
375                 {
376                     if ( i.index == index )
377                     {
378                         already_requested = 1;
379                         break;
380                     } else if ( index - i.index == 1 )
381                     {
382                         i.index = index;
383                     } else if ( index - i.index > 1 )
384                     {
385                         index_error = 1;
386                     }
387                 }
388             }
389             if ( index_error )
390             {
391                 std::cout << "Index error, package dropped with
no response" << std::endl;
392                 index_error = 0;
393                 continue;
394             }
395 
396             if ( is_new )
397             {
398                 addr_index_pair = {si_other, index};
399                 pthread_mutex_lock(&mutex);
400                 indexes.push_back(addr_index_pair);
401                 pthread_create(&threads[connections], NULL, &
login_routine, &rd);
402                 connections++;
403                 pthread_mutex_unlock(&mutex);
404             } else {
405 
406                 if ( already_requested )
407                 {
408                     pthread_mutex_lock(&mutex);
409                     pthread_create(&threads[connections], NULL, &
repeat_response_routine, &rd);
410                     pthread_mutex_unlock(&mutex);
411                 } else {
412                     pthread_mutex_lock(&mutex);
413                     pthread_create(&threads[connections], NULL, &
terminal_routine, &rd);
414                     connections++;
415                     pthread_mutex_unlock(&mutex);
416                 }
417             }
418             already_requested = 0;
419             is_new = 1;
```

```
420         } else
421             sleep(1);
422     }
423 
424     pthread_mutex_lock(&mutex);
425     local_connections = connections;
426     pthread_mutex_unlock(&mutex);
427 
428     for (int i = 0; i < local_connections; ++i) {
429         pthread_join(threads[i], NULL);
430     }
431 }
432 
433 
434 int main(void)
435 {
436     struct sockaddr_in si_me;
437     int rc;
438     bool quit = 0;
439     char in;
440     int socket_index;
441     Socket_key_map skp;
442     int s;
443 
444     usr_map.insert(std::pair <std::string, std::string> ("vaddya"
    , "32283228"));
445     usr_map.insert(std::pair <std::string, std::string> ("
    lamtev2000", "iluvclassmates"));
446     usr_map.insert(std::pair <std::string, std::string> ("
    mikle_undef", "arguetill_theend"));
447     usr_map.insert(std::pair <std::string, std::string> ("valik",
     "1228"));
448     usr_map.insert(std::pair <std::string, std::string> ("ivan",
    "0000"));
449     root_users.emplace_back("ivan");
450 
451     //create a UDP socket
452     s = socket(AF_INET, SOCK_DGRAM, 0);
453     if (s < 0)
454     {
455         perror("socket call failed");
456         exit(1);
457     }
458 
459     skp = {s, 0};
460 
461     si_me.sin_family = AF_INET;
462     si_me.sin_port = htons(7500);
463     si_me.sin_addr.s_addr = htonl(INADDR_ANY);
464 
465     //bind socket to port
466     rc = bind(s , (struct sockaddr*)&si_me, sizeof(si_me) );
```

```
467     if( rc < 0 )
468     {
469         perror("bind");
470         exit(1);
471     }
472     pthread_mutex_init(&mutex, NULL);
473     pthread_create(&recvfrom_thread, NULL, &recvfrom_routine, &
    skp);
474 
475     //keep listening for data
476     while (1) {
477         in = (char) getc(stdin);
478         switch (in) {
479             case 'c': {
480                 socket_index = (char) getc(stdin) - '0';
481                 if ( socket_index > logged_in_addrs.size() )
482                 {
483                     perror("Index out of range");
484                     break;
485                 }
486                 pthread_mutex_lock(&mutex);
487                 logged_in_addrs.erase(logged_in_addrs.begin() +
    socket_index);
488                 pthread_mutex_unlock(&mutex);
489                 break;
490             }
491             case 'l': {
492                 pthread_mutex_lock(&mutex);
493                 std::cout << "Connections: " << logged_in_addrs.
    size() << std::endl;
494                 int i = 0;
495                 for ( auto &c : logged_in_addrs ) {
496                     std::cout << i << "\t" << inet_ntoa(c.
    sin_addr) << ":" << htons(c.sin_port) << std::endl;
497                     i++;
498                 }
499                 pthread_mutex_unlock(&mutex);
500                 break;
501             }
502             case 'q': {
503 
504                 quit = 1;
505                 break;
506             }
507         }
508         if ( quit )
509             break;
510     }
511 
512     rc = shutdown(s, 2);
513     if (rc < 0)
514         perror("shutdown call failed");
```

```
515     rc = close(s);
516     if (rc < 0)
517         perror("close call failed");
518 
519     pthread_join(recvfrom_thread, NULL);
520     pthread_mutex_destroy(&mutex);
521     return 0;
522 }
```

Листинг 6: UDP Server

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  #include <netinet/in.h>
4  #include <arpa/inet.h>
5  #include <stdio.h>
6  #include <cstdlib>
7  #include <iostream>
8  #include <string>
9  #include <cstring>
10 #include <stdlib.h>
11 #include <unistd.h>
12 #include <sys/socket.h>
13 #include <netinet/in.h>
14 #include <netdb.h>
15 #include <pthread.h>
16 #include <zconf.h>
17 #include <map>
18 #include <algorithm>
19 #include <vector>
20 #include <stddef.h>
21 #include <sys/time.h>
22 
23 #define _POSIX_SOURCE
24 #include <unistd.h>
25 #include <arpa/inet.h>
26 #undef _POSIX_SOURCE
27 
28 #define LOGIN_L 1
29 
30 const int buf_size = 512;
31 const int buf_out_size = 8192;
32 const int size_gap = 4;
33 const int index_gap = 4;
34 
35 
36 void pack_size(char* buf, ulong size)
37 {
38     char c;
39     for ( int i = 3; i >= 0; --i )
40         if ( size > 0 ) {
41             c = char(size % 10) + '0';
```

```
            buf[i] = c;
        } else {
            buf[i] = '0';
        }
        size /= 10;
    }
}


void pack_index(char* buf, ulong index)
{
    char c;
    for ( int i = buf_size - 1; i >= buf_size - 4; --i )
        if ( index > 0 ) {
            c = char(index % 10) + '0';
            buf[i] = c;
        } else {
            buf[i] = '0';
        }
        index /= 10;
    }
}


int main( void )
{
    int s;
    int rc;
    int command_descriptor_index = size_gap;
    int msg_size = 32;
    bool noInput = 1;
    char logInRes[1];
    int index = 0;
    std::string str;
    std::string substring;
    std::string login_str;
    timeval timeval1;
    timeval1.tv_sec = 3;
    timeval1.tv_usec = 0;

    int n;
    struct sockaddr_in from;
    unsigned int slen = sizeof(from);
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s < 0)
    {
        perror("socket");
        exit(1);
    }

    from.sin_family = AF_INET;
    from.sin_port = htons( 7500 );
```

```
94      from.sin_addr.s_addr = inet_addr( "127.0.0.1" );
95
96      char login_buf[buf_size];
97      char buf[ buf_size ];
98      char buf_out[buf_out_size];
99
100     if ( setsockopt(s, SOL_SOCKET, SO_RCVTIMEO, &timeval1, sizeof
    (timeval1)) < 0)
101     {
102         perror("Error setting timeout option \n");
103     }
104
105     //Log in loop
106     while (1) {
107         index = index + 1;
108         printf("Log in: {user_name}:{password}: \n");
109         std::getline(std::cin, login_str);
110         if (login_str.size() > buf_size - size_gap - index_gap) {
111             perror("Incorrect authentication length, should be
    less than 64");
112         } else {
113             login_str.resize(buf_size, 0);
114             strcpy(login_buf, login_str.c_str());
115             pack_index(login_buf, index);
116             rc = sendto(s, login_buf, buf_size, 0, (struct
    sockaddr *) &from, slen);
117             if (rc <= 0) {
118                 perror("send call failed");
119                 exit(1);
120             }
121             //ack loop
122             if ( recvfrom(s, logInRes, LOGIN_L, 0, (struct
    sockaddr *) &from, &slen) < 0 ) {
123                 for (int i = 0; i <= 10; ++i) {
124                     if (recvfrom(s, logInRes, LOGIN_L, 0, (struct
     sockaddr *) &from, &slen) < 0) {
125                         rc = sendto(s, login_buf, buf_size, 0, (
    struct sockaddr *) &from, slen);
126                         if (rc <= 0) {
127                             perror("send call failed");
128                             exit(1);
129                         }
130                         if (i > 9) {
131                             std::cout << "Aborting connection, no
     response" << std::endl;
132                             exit(0);
133                         }
134                     }
135                 }
136             }
137
138             if (logInRes[0] == '1') {
```

```
139                 printf("Succesfully logged in \n");
140                 break;
141             } else if (logInRes[0] == '2')
142                 printf("Already logged in in another session \n")
    ;
143             else if (logInRes[0] == '*')
144                 printf("Incorrect format \n");
145             else {
146                 printf("Wrong login:password pair \n");
147                 std::cout << logInRes << std::endl;
148             }
149         }
150      }
151
152
153      //main workflow loop
154      while(1) {
155          index--;
156
157          std::getline (std::cin, str);
158
159          if (str == "ls") {
160              //packing message size 0001 to buffer
161              for (int i = 0; i < size_gap - 1; ++i) {
162                  buf[i] = '0';
163              }
164              buf[command_descriptor_index - 1] = '1';
165              buf[command_descriptor_index] = '1';
166              noInput = 0;
167          }
168
169          else if (str.compare(0,2,"cd") == 0) {
170              for (int i = 0; i < size_gap - 1; ++i) {
171                  buf[i] = '0';
172              }
173              buf[command_descriptor_index - 1] = '1';
174              buf[command_descriptor_index] = '2';
175              if (str[2] != ' ') {
176                  std::cout << "Incorrect input format" << std::
    endl;
177                  continue;
178              } else {
179                  substring = str.substr(3);
180                  if (substring.size() > buf_size - size_gap -
    index_gap) {
181                      perror("Out of input size, max length is 30
    symbols");
182                      break;
183                  } else {
184                      for (int i = 0; i < substring.size(); ++i) {
185                          buf[i + size_gap + 1] = substring[i];
186                      }
```

```
187                     pack_size(buf, substring.size() + 1);
188                 }
189                 substring.clear();
190             }
191             noInput = 0;
192         }
193 
194         else if (str=="who") {
195             for (int i = 0; i < size_gap - 1; ++i) {
196                 buf[i] = '0';
197             }
198             buf[command_descriptor_index - 1] = '1';
199             buf[command_descriptor_index] = '3';
200             noInput = 0;
201         }
202 
203         else if (str.compare(0,4,"kill") == 0) {
204             buf[command_descriptor_index] = '4';
205             if (str[4] != ' ') {
206                 perror("Incorrect input format");
207                 break;
208             } else {
209                 substring = str.substr(5);
210                 if (substring.size() > buf_size - size_gap -
    index_gap) {
211                     perror("Out of input size");
212                     //break;
213                 } else {
214                     for (int i = 0; i < substring.size(); ++i) {
215                         buf[i + size_gap + 1] = substring[i];
216                     }
217                     pack_size(buf, substring.size() + 1);
218                 }
219                 substring.clear();
220             }
221             noInput = 0;
222         }
223 
224         else if (str == "logout") {
225             for (int i = 0; i < size_gap - 1; ++i) {
226                 buf[i] = '0';
227             }
228             buf[command_descriptor_index - 1] = '1';
229             buf[command_descriptor_index] = '5';
230             pack_index(buf, index);
231             rc = sendto(s, buf, buf_size, 0, (struct sockaddr *)
    &from, slen);
232             if (rc <= 0)
233             {
234                 perror("send call failed");
235                 break;
236             }
```

```
237             break;
238         }
239         else
240             noInput = 1;
241 
242         if (!noInput) {
243             //rc = send(s, buf, buf_size, 0);
244             index ++;
245             pack_index(buf, index);
246             rc = sendto(s, buf, buf_size, 0, (struct sockaddr *)
    &from, slen);
247             if (rc <= 0) {
248                 perror("send call failed");
249                 break;
250             }
251 
252             if ( recvfrom(s, buf_out, buf_out_size, 0, (struct
    sockaddr *) &from, &slen) < 0 ) {
253                 for (int i = 0; i <= 10; ++i) {
254                     if (recvfrom(s, buf_out, buf_out_size, 0, (
    struct sockaddr *) &from, &slen) < 0) {
255                         rc = sendto(s, buf, buf_size, 0, (struct
    sockaddr *) &from, slen);
256                         if (rc <= 0) {
257                             perror("send call failed");
258                             break;
259                         }
260                         if (i > 9) {
261                             std::cout << "Server unreachable" <<
    std::endl;
262                             exit(0);
263                         }
264                     }
265                 }
266             }
267 
268             if (buf_out[0] == '*')
269             {
270                 std::cout << "Session closed by administrator" <<
     std::endl;
271                 exit(0);
272             }
273             std::cout << "Received: \n" << std::string(buf_out)
    << std::endl; //may not work */
274             memset(&buf_out, 0, sizeof(buf_out));
275         }
276      }
277 
278      close(s);
279      printf("Session closed, logged out");
280      exit( 0 );
281 }
```

Листинг 7: UDP Client