Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»

Кафедра Вычислительных Систем

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КОНТРОЛЬНОМУ ПРОЕКТУ НА ТЕМУ “РАЗРАБОТКА СЕТЕВОГО ПРИЛОЖЕНИЯ”

Работу выполнил: магистрант 1 курса
группы МГ-165
Гайдай А. В.

Работу принял: доцент, д.т.н.
Павский К. В.

Новосибирск 2016 г.

1. Постановка задачи

Необходимо реализовать сетевое приложение, используя сетевую модель стека сетевых протоколов OSI/ISO.
Требования и ограничения:

- Приложение должно быть отказоустойчивым;
- Язык программирования: C/C++;
- Протоколы транспортного уровня: TCP/UDP;
- Должна присутствовать возможность одновременного обслуживания нескольких клиентов (thread/select).

2. Описание средств разработки

2.1. Программная библиотека сокетов Беркли

2.1.1. Заголовочные файлы

Программная библиотека сокетов Беркли включает в себя множество связанных заголовочных файлов. Ниже перечислены те, что использовались при разработке клиент-серверного приложения для данного КП:

- <sys/socket.h> — базовые функции сокетов BSD и структуры данных;
- <sys/types.h> — различные типы данных;
- <netinet/in.h> — семейства адресов/протоколов AF_INET / PF_INET и AF_INET6 / PF_INET6;
- <sys/un.h> — семейство адресов AF_UNIX / AF_LOCAL. Используется для локального взаимодействия между программами, запущенными на одном компьютере;
- <arpa/inet.h> — функции для работы с числовыми IP-адресами;
- <netdb.h> — функции для преобразования протокольных имён и имён хостов в числовые адреса.

2.1.2. Основные структуры

Основные структуры программной библиотеки сокетов Беркли:

- sockaddr — обобщённая структура адреса, в которой, в зависимости от используемого семейства протоколов, проводится соответствующая структура, в данном КП sockaddr_in.

struct sockaddr

type	field name	description
sa_family_t	sa_family;	address family: AF_INET
char	sa_data[14];	other data

struct sockaddr_in

type	field name	description
sa_family_t	sin_family;	address family: AF_INET
in_port_t	sin_port;	port in network byte order
struct in_addr	sin_addr;	internet address

struct in_addr

type	field name	description
uint32_t	s_addr;	address in network byte order

2.1.3. Основные функции

socket()

Функция socket() создаёт конечную точку соединения и возвращает дескриптор. socket() принимает три аргумента:

- domain, указывающий семейство протоколов создаваемого сокета. Этот параметр задаёт правила использования именования и формат адреса. Например:
  - PF_INET для сетевого протокола IPv4 (используется в данном КП);
  - PF_INET6 для сетевого протокола IPv6;
  - PF_UNIX для локальных сокетов (используя файл).
- type, тип:
  - SOCK_STREAM надёжная потокоориентированная служба (TCP) или потоковый сокет (используется в данном КП);
  - SOCK_DGRAM служба датаграмм (UDP) или датаграммный сокет (используется в данном КП);
  - SOCK_SEQPACKET надёжная служба последовательных пакетов (SCTP);
  - SOCK_RAW сырой сокет — сырой протокол поверх сетевого уровня.
- protocol определяет используемый транспортный протокол. Самые распространённые — это IPPROTO_TCP, IPPROTO_SCTP, IPPROTO_UDP, IPPROTO_DCCP. Эти протоколы указаны в <netinet/in.h>. Значение «0» может

быть использовано для выбора протокола по умолчанию из указанного семейства (domain) и типа (type).

Функция возвращает −1 в случае ошибки. Иначе, она возвращает целое число, представляющее присвоенный дескриптор.

Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

connect()

Функция connect() устанавливает соединение с сервером. Возвращает целое число, представляющее код ошибки: 0 означает успешное выполнение, а -1 свидетельствует об ошибке. connect() принимает три аргумента:

- sockfd — дескриптор сокета;
- addr — адрес сервера (struct sockaddr);
- addrlen — длина адреса (sizeof (addr)).

Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

bind()

Функция bind() связывает сокет с конкретным адресом. Когда сокет создается при помощи socket(), он ассоциируется с некоторым семейством адресов, но не с конкретным адресом. До того как сокет сможет принять входящие соединения, он должен быть связан с адресом. bind() принимает три аргумента:

- sockfd — дескриптор сокета;
- addr — адрес, к которому привязать (struct sockaddr);
- addrlen — длина адреса (sizeof (addr)).

Возвращает 0 при успехе и −1 при возникновении ошибки.

Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

listen()

Функция listen() подготавливает привязываемый сокет к принятию входящих соединений. Данная функция применима только к типам сокетов SOCK_STREAM и SOCK_SEQPACKET. Принимает два аргумента:

- sockfd — дескриптор сокета;
- backlog — целое число, означающее число установленных соединений, которые могут быть обработаны в любой момент времени. Операционная система обычно ставит его равным максимальному значению.

После принятия соединения оно выводится из очереди. В случае успеха возвращается 0, в случае возникновения ошибки возвращается −1.

Прототип

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

accept()

Функция accept() используется для принятия запроса на установление соединения от удаленного хоста. Принимает следующие аргументы:

- sockfd — дескриптор слушающего сокета;
- cliaddr — указатель на структуру sockaddr, для принятия информации об адресе клиента;
- addrlen — указатель на socklen_t, определяющее размер структуры, содержащей клиентский адрес и переданной в accept(). Когда accept() возвращает некоторое значение, socklen_t указывает сколько байт структуры cliaddr использовано в данный момент.

Функция возвращает дескриптор сокета, связанный с принятым соединением, или −1 в случае возникновения ошибки.

Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

send()

Функция send() отправляет сообщение по установленному соединениею. Принимает следующие аргументы:

- sockfd — дескриптор сокета;
- buf — отправляемое сообщение;
- len — длина сообщения;
- flags — параметры отправки (можно указывать значение «0»).

В случае успеха, функция возвращает количество отправленных байт, в противном случае -1 и сохраняет подробности о коде ошибки в errno (<errno.h>).

Прототип

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

recv()

Функция recv() принимает сообщение по установленному соединению. Принимает следующие аргументы:

- sockfd — дескриптор сокета;
- buf — указатель на буфер, в который будет сохранено сообщение;
- len — размер буфера;
- flags — параметры получения(можно указывать значение «0»).

В случае успеха, функция возвращает количество принятых байт, в противном случае -1 и сохраняет подробности о коде ошибки в errno (<errno.h>).

Прототип

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

sendto()

Функция sendto() отправляет сообщение по указанному адресу. Принимает следующие аргументы:

- sockfd — дескриптор сокета;
- buf — отправляемое сообщение;
- len — длина сообщения;
- flags — параметры отправки (можно указывать значение «0»);
- dest_addr — адрес получателя (struct sockaddr);
- addrlen — длина адреса (sizeof (addr)).

В случае успеха, функция возвращает количество отправленных байт, в противном случае -1 и сохраняет подробности о коде ошибки в errno (<errno.h>).

Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockfd, const void *buf, size_t len, int flags,
           const struct sockaddr *dest_addr, socklen_t addrlen);
```

recvfrom()

Функция recv() принимает сообщение с указанного адреса. Принимает следующие аргументы:

- sockfd — дескриптор сокета;
- buf — указатель на буфер, в который будет сохранено сообщение;
- len — размер буфера;
- flags — параметры получения (можно указывать значение «0»);
- src_addr — указатель на структуру sockaddr, для принятия информации об адресе источника;
- addrlen — указатель на socklen_t, определяющее размер структуры, содержащей адрес источника.

В случае успеха, функция возвращает количество принятых байт, в противном случае -1 и сохраняет подробности о коде ошибки в errno (<errno.h>).

Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom(int sockfd, void *buf, size_t len, int flags,
             struct sockaddr *src_addr, socklen_t *addrlen);
```

2.2. Программная пользовательская библиотека

2.2.1. Заголовочные файлы

- <error.h> — информирующие функции;
- <list.h> — пользовательский связный список, используется на сервере для хранения информации о клиентах и истории;
- <msghandler.h> — обработчик сообщений;
- <network.h> — функции-обёртки для отправки и приёма сообщений;
- <shared_data.h> — разделяемые данные;
- <types.h> — пользовательские типы данных;
- <user_socket.h> — функции-обёртки для работы с неблокирующими сокетами;
- <client_func.h> — клиентские функции;
- <server_func.h> — серверные функции.

2.2.2. Основные структуры

Основные структуры программной пользовательской библиотеки:

- struct __header — структура заголовка сообщения, информирующая о типе и размере основного сообщения.

struct __header

type	field name	description
int	type;	main message type
long int	size;	main message size

- struct __login — структура идентифицирующая пользователя. Используется для регистрации пользователя на стороне сервера.

struct __login

type	field name	description
struct in_addr	addr;	internet address
int	pid;	process id

- struct __chat — структура сообщения.

struct __chat

type	field name	description
struct __login	login;	user id
char	buf[MAX_MSG_SIZE]	message

- struct __serv_msg — структура сообщения передаваемого между серверами.

struct __serv_msg

type	field name	description
struct __login	login;	user id
int	type;	message type
char	buf[MAX_MSG_SIZE]	message

- struct __user — структура клиента на сервере.

struct __serv_msg

type	field name	description
struct __login	login;	user id
int	sd;	connect socket description

- struct __recv_msg — структура клиента на сервере.

struct __serv_msg

type	field name	description
struct __user	user;	user id for server
int	ind;	index for client thread

2.2.3. Основные функции

start_chat() / stop_chat()

Функция start_chat() инициализирует необходимые данные на стороне клиента и производят соединение с сервером. Функция stop_chat() завершает сеанс. Аргументов функции не имеют и не возвращают никакого значения.

Прототип

```
#include "client_func.h"
void start_chat(void);
void stop_chat(void);
```

start_server / stop_server()

Функции start_server() и stop_server() запускают и останавливают сервер соответственно. Аргументов функции не имеют и не возвращают никакого значения.

Прототип

```
#include "server_func.h"
void start_server(void);
void stop_server(void);
```

handler_msg()

Функция handler_msg() обрабатывает сообщения согласно их типу (на стороне клиента). Принимает следующие аргументы:

- type — тип сообщения;
- buf — сообщение.

Функция возвращает значение «0» в случае успеха, или полученное значение, если тип сообщения «INFO».

Прототип

```
#include "msg_handler.h"
int handler_msg(int type, void *buf);
```

handler_client_msg()

Функция handler_client_msg() обрабатывает сообщения от клиентов согласно их типу (на стороне сервера). Принимает следующие аргументы:

- user — информация о клиенте;
- type — тип сообщения;
- buf — сообщение.

Функция возвращает значение «0» в случае успеха, в противном случае отрицательное число.

Прототип

```
#include "msg_handler.h"
int handler_client_msg(struct __user user, int type, void *buf);
```

handler_server_msg()

Функция handler_server_msg() обрабатывает сообщения от серверов согласно их типу (на стороне сервера). Принимает следующие аргументы:

- user — информация о клиенте;
- type — тип сообщения;
- buf — сообщение.

Функция возвращает значение «0» в случае успеха, в противном случае отрицательное число.

Прототип

```
#include "msg_handler.h"
int handler_server_msg(struct __user user, int type, void *buf);
```

send_msg_nblck()

Функция send_msg_nblck() передаёт сообщение указанного типа, используя неблокирующий сокет. Принимает следующие аргументы:

- sd — дескриптор сокета;
- type — тип сообщения;
- buf — сообщение;
- size — размер сообщения;
- timeout — максимальное время отправки сообщения.

Функция возвращает количество отправленных байт; в случае возникновения ошибки — отрицательное число.

Прототип

```
#include "network.h"
int send_msg_nblck (int sd, int type, void *buf,
                    long int size, int timeout);
```

recv_msg_nblck()

Функция recv_msg_nblck() принимает сообщение, используя неблокирующий сокет. Аргументы:

- sd — дескриптор сокета;
- type — указатель на переменную, в которую необходимо сохранить тип сообщения;
- buf — указатель на буфер для сообщение;
- timeout — максимальное время получения сообщения.

Функция возвращает количество полученных байт; в случае возникновения ошибки — отрицательное число.

Прототип

```
#include "network.h"
int recv_msg_nblck (int sd, int type, void *buf, int timeout);
```

nblck_sock_mode ()

Функция nblck_sock_mode () переключает сокет в блокирующий/неблокирующий режим. Аргументы:

- sd — дескриптор сокета;
- mode — режим сокета («0» — блокирующий; «1» — неблокирующий).

Функция возвращает значение «0» в случае успеха, в противном случае — отрицательное число.

Прототип

```
#include "user_socket.h"
int nblck_sock_mode (int sd, int mode);
```

connect_nblck()

Функция connect_nblck() устанавливает соединение с сервером, используя неблокирующий сокет. Аргументы:

- sd — дескриптор сокета;
- server — адрес сервера;
- size — размер структуры, хранящей адрес;
- try — количество попыток установления соединения с сервером.

Функция возвращает целое число, представляющее код ошибки: 0 означает успешное выполнение, а -1 свидетельствует об ошибке.

Прототип

```
#include "user_socket.h"
int connect_nblck (int sd, struct sockaddr *server, socklen_t size,
                   int timeout, int try);
```

accept_nblck()

Функция accept_nblck() используется для принятия запроса на установление соединения от удаленного хоста, с помощью неблокирующего сокета. Аргументы:

- sd — дескриптор сокета;
- addr — указатель на структуру sockaddr, для принятия информации об адресе клиента;
- addr_len — указатель на socklen_t, определяющее размер структуры, содержащей клиентский адрес;
- timeout — максимальное время на принятие запроса.

Функция возвращает дескриптор сокета, связанный с принятым соединением, или −1 в случае возникновения ошибки.

Прототип

```
#include "user_socket.h"
int accept_nblck (int sd, struct sockaddr *addr, socklen_t *addr_len,
                  int timeout);
```

send_nblck()

Функция send_nblck() отправляет сообщение по установленному соединению, используя неблокирующий сокет. Аргументы:

- sd — дескриптор сокета;
- content — отправляемое сообщение;
- size — длина сообщения;
- timeout — максимальное время отправки сообщения.

В случае успеха, функция возвращает количество отправленных байт, в противном случае отрицательное значение.

Прототип

```
#include "user_socket.h"
int send_nblck(int sd, void *content, long int len, int timeout);
```

recv_nblck()

Функция recv_nblck() принимает сообщение по установленному соединению, используя неблокирующий сокет. Аргументы:

- sd — дескриптор сокета;
- content — указатель на буфер, в который будет сохранено сообщение;
- size — размер буфера;
- timeout — максимальное время получения сообщения.

В случае успеха, функция возвращает количество отправленных байт, в противном случае -1 и сохраняет подробности о коде ошибки в errno (<errno.h>).

Прототип

```
#include "user_socket.h"
int recv_nblck(int sd, void *content, long int size, int timeout);
```

sendto_broadcast()

Функция sendto_broadcast() отправляет сообщение указанного типа группе получателей. Аргументы:

- sd — дескриптор сокета;
- buf — сообщение;
- len — длина сообщения;
- flags — параметры отправки (можно указывать значение «0»);

- dest_addr — адреса получателей;
- addrlen — общий размер всех адресов.

Функция возвращает количество отправленных байт.

Прототип

```
#include "user_socket.h"
int sendto_broadcast(int sd, const void *buf, size_t len, int flags,
                     const struct sockaddr *dest_addr, socklen_t addrlen);
```

3. Описание протоколов

3.1. Internet Protocol (IP)

IP (англ. Internet Protocol — межсетевой протокол) — маршрутизируемый протокол сетевого уровня стека TCP/IP модели OSI. Неотъемлемой частью протокола является адресация сети.

IP объединяет сегменты сети в единую сеть, обеспечивая доставку пакетов данных между любыми узлами сети через произвольное число промежуточных узлов (маршрутизаторов). Он классифицируется как протокол третьего уровня по сетевой модели OSI. IP не гарантирует надёжной доставки пакета до адресата — в частности, пакеты могут прийти не в том порядке, в котором были отправлены, продублироваться (приходят две копии одного пакета), оказаться повреждёнными (обычно повреждённые пакеты уничтожаются) или не прийти вовсе. Гарантию безошибочной доставки пакетов дают некоторые протоколы более высокого уровня — транспортного уровня сетевой модели OSI, — например, TCP, которые используют IP в качестве транспорта.

IP-пакет — форматированный блок информации, передаваемый по компьютерной сети, структура которого определена протоколом IP. В отличие от них, соединения компьютерных сетей, которые не поддерживают IP-пакеты, такие как традиционные соединения типа «точка-точка» в телекоммуникациях, просто передают данные в виде последовательности байтов, символов или битов. При использовании пакетного форматирования сеть может передавать длинные сообщения более надежно и эффективно.

3.1.1. IPv4

В современной сети Интернет используется IP четвёртой версии, также известный как IPv4. В протоколе IP этой версии каждому узлу сети ставится в соответствие IP-адрес длиной 4 октета (4 байта). При этом компьютеры в подсетях объединяются общими начальными битами адреса. Количество этих бит, общее для данной подсети, называется маской подсети (ранее использовалось деление пространства адресов по классам — A, B, C; класс сети определялся диапазоном значений старшего октета и определял число адресуемых узлов в данной сети, сейчас используется бесклассовая адресация).

- Версия — для IPv4 значение поля должно быть равно 4;
- IHL — (Internet Header Length) длина заголовка IP-пакета в 32-битных словах. Именно это поле указывает на начало блока данных в пакете. Минимальное корректное значение для этого поля равно 5;
- Длина пакета — (Total Length) длина пакета в октетах, включая заголовок и данные. Минимальное корректное значение для этого поля равно 20, максимальное — 65 535;
- Идентификатор — (Identification) значение, назначаемое отправителем пакета и предназначенное для определения корректной последовательности фрагментов при сборке пакета. Для фрагментированного пакета все фрагменты имеют одинаковый идентификатор;
- 3 бита флагов. Первый бит должен быть всегда равен нулю, второй бит DF (don't fragment) определяет возможность фрагментации пакета и третий бит MF (more fragments) показывает, не является ли этот пакет последним в цепочке пакетов;
- Смещение фрагмента — (Fragment Offset) значение, определяющее позицию фрагмента в потоке данных. Смещение задается количеством восьмибайтовых блоков, поэтому это значение требует умножения на 8 для перевода в байты;
- Время жизни (TTL) — число маршрутизаторов, которые может пройти этот пакет. При прохождении маршрутизатора это число уменьшается на единицу. Если значение этого поля равно нулю, то пакет должен быть отброшен, и отправителю пакета может быть послано сообщение Time Exceeded (ICMP тип 11 код 0);
- Протокол — идентификатор интернет-протокола следующего уровня указывает, данные какого протокола содержит пакет, например, TCP, UDP, или ICMP;
- Контрольная сумма заголовка — (Header Checksum) вычисляется в соответствии с RFC 1071.

Таблица 3.1 — Структура IP-пакета в протоколе IPv4

Октет	**0**	...	**3**	**4**	...	**7**	**8**	...	**15**	**16**	...	**18**	**19**	...	**31**
0	Версия			IHL			Тип обслуживания			Длина пакета					
4	Идентификатор									Флаги			Смещение фрагмента		
8	Время жизни (TTL)						Протокол			Контрольная сумма загололвка					
12	IP-адрес отправителя														
16	IP-адрес получателя														
20	Параметры (от 0 до 10-и 32-х битных слов)														
	Данные														

3.1.2. IPv6

IPv6 (англ. Internet Protocol version 6) — новая версия протокола IP, призванная решить проблемы, с которыми столкнулась предыдущая версия (IPv4) при её использовании в Интернете, за счёт использования длины адреса 128 бит вместо 32.

- Версия — для IPv6 значение поля должно быть равно 6;
- Класс трафика — определяет приоритет трафика (QoS, класс обслуживания);
- Метка потока — уникальное число, одинаковое для однородного потока пакетов;
- Длина полезной нагрузки — длина данных в октетах (заголовок IP-пакета не учитывается);
- Следующий заголовок — задаёт тип расширенного заголовка, который идёт следующим. В последнем расширенном заголовке поле Next header задаёт тип транспортного протокола (TCP, UDP и т. д.) и определяет следующий инкапсулированный уровень;
- Число переходов — максимальное число маршрутизаторов, которые может пройти пакет. При прохождении маршрутизатора это значение уменьшается на единицу и по достижении нуля пакет отбрасывается.

Таблица 3.2 — Структура IP-пакета в протоколе IPv6

<table>
<tr><th>Позиция в октетах</th><th></th><th colspan="6">0</th><th colspan="6">1</th><th colspan="3">2</th><th colspan="3">3</th></tr>
<tr><th></th><th>Позиция в битах</th><th>0</th><th>...</th><th>3</th><th>4</th><th>...</th><th>7</th><th>8</th><th>...</th><th>11</th><th>12</th><th>...</th><th>15</th><th>16</th><th>...</th><th>23</th><th>24</th><th>...</th><th>31</th></tr>
<tr><td>0</td><td>0</td><td colspan="3">Версия</td><td colspan="6">Класс трафика</td><td colspan="12">Метка потока</td></tr>
<tr><td>4</td><td>32</td><td colspan="12">Длина полезной нагрузки</td><td colspan="6">Следующий заголовок</td><td colspan="6">Число переходов</td></tr>
<tr><td>8</td><td>64</td><td colspan="18" rowspan="4">IP-адрес отправителя</td></tr>
<tr><td>12</td><td>96</td></tr>
<tr><td>16</td><td>128</td></tr>
<tr><td>20</td><td>160</td></tr>
<tr><td>24</td><td>192</td><td colspan="18" rowspan="3">IP-адрес получателя</td></tr>
<tr><td>28</td><td>224</td></tr>
<tr><td>32</td><td>256</td></tr>
</table>

36	288	

3.2. Transmission Control Protocol (TCP)

TCP (англ. Transmission Control Protocol — протокол управления передачей) — один из основных протоколов передачи данных интернета, предназначенный для управления передачей данных. Сети и подсети, в которых совместно используются протоколы TCP и IP называются сетями TCP/IP.

В стеке протоколов IP TCP выполняет функции протокола транспортного уровня модели OSI.

Механизм TCP предоставляет поток данных с предварительной установкой соединения, осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного пакета, гарантируя тем самым, в отличие от UDP, целостность передаваемых данных и уведомление отправителя о результатах передачи.

Реализации TCP обычно встроены в ядра ОС. Существуют реализации TCP, работающие в пространстве пользователя.

Когда осуществляется передача от компьютера к компьютеру через Интернет, TCP работает на верхнем уровне между двумя конечными системами, например, браузером и веб-сервером. TCP осуществляет надежную передачу потока байтов от одной программы на некотором компьютере к другой программе на другом компьютере (например, программы для электронной почты, для обмена файлами). TCP контролирует длину сообщения, скорость обмена сообщениями, сетевой трафик.

3.3. User Datagram Protocol (UDP)

UDP (англ. User Datagram Protocol — протокол пользовательских датаграмм) — один из ключевых элементов TCP/IP, набора сетевых протоколов для Интернета. С UDP компьютерные приложения могут посылать сообщения (в данном случае называемые датаграммами) другим хостам по IP-сети без необходимости предварительного сообщения для установки специальных каналов передачи или путей данных. Протокол был разработан Дэвидом П. Ридом в 1980 году и официально определён в RFC 768.

UDP использует простую модель передачи, без неявных «рукопожатий» для обеспечения надёжности, упорядочивания или целостности данных. Таким образом, UDP предоставляет ненадёжный сервис, и датаграммы могут прийти не по порядку, дублироваться или вовсе исчезнуть без следа. UDP подразумевает, что проверка ошибок и исправление либо не нужны, либо должны исполняться в приложении. Чувствительные ко времени приложения часто используют UDP, так как предпочтительнее сбросить пакеты, чем ждать задержавшиеся пакеты, что может оказаться невозможным в системах реального времени. При необходимости исправления ошибок на сетевом уровне интерфейса приложение может задействовать TCP или SCTP, разработанные для этой цели.

Природа UDP как протокола без сохранения состояния также полезна для серверов, отвечающих на небольшие запросы от огромного числа клиентов, например DNS и потоковые мультимедийные приложения вроде IPTV, Voice over IP, протоколы туннелирования IP и многие онлайн-игры.

4. Реализация программы

В качестве клиент-серверного отказоустойчивого приложение был реализован чат с функцией хранения истории. Система является децентрализованной. В ходе разработки использовались описанные выше средства и протоколы.

4.1. Клиентская часть

Для клиента набор серверов представляется как одно целое. Информация о имеющихся серверах (IP-адреса и порты) заранее записываются в конфигурационный файл servers.conf (Листинг 4.1).

Листинг 4.1 — servers.conf

```
127.0.0.1          7778
192.168.122.1      7780
5.255.255.55       7779
127.0.0.1          7777
127.0.0.1          7781
```

Логика работы клиента:

1. Установка соединения со случайно выбранным сервером из имеющихся.
2. Если установить соединение не удалось выбрать следующий сервер (выбор сервера происходит до тех пор пока не удастся установить соединение или пока не будут перебраны все имеющиеся адреса из servers.conf).
3. Вход в чат. Взаимодействие с сервером по TCP.
4. Если сервер перестаёт функционировать, происходит автоматическое переподключение к другому серверу из имеющихся. Переподключение невидимо для клиента.
5. Выход из чата.

4.2. Серверная часть

Каждый сервер имеет как минимум два активных сокета: для прослушки подключений (TCP сокет) и для обмена клиентской информацией между другими серверами (UDP сокет). Каждый сервер имеет конфигурационный файл с информацией о других серверах (servers.conf см. листинг 4.1).

Логика работы сервера:

1. Один поток ожидает запросы от клиентов на подключение.

2. Если запрос получен, данные о клиенте заносятся в связный список и передаются другим серверам. Таким образом информация о всех клиентах имеется на каждом сервере. Каждый сервер знает какие клиенты из списка подключены непосредственно к нему, а какие нет.
3. При получении сообщения чата от клиента, сервер отправляет это сообщение всем своим клиентам и другим серверам, для того чтобы они (сервера) переслали это сообщение своим клиентам.
4. При отключении клиента информация о нём удаляется из списка. Сервер, к которому был подключен клиент, отправляет информацию об отключении данного клиента другим серверам, для того чтобы они (сервера) также удалили данные об этом клиенте из своих списков.

Таким образом достигается децентрализованность и отказоустойчивость системы.

5. Текст программы

5.1. Клиентская часть

Листинг 5.1 — main.c

```
#include <stdio.h>
#include <string.h>
#include "client_func.h"

int main (int argc, char *argv[])
{
    start_chat ();

    char   buf[10];

    do {
        fscanf (stdin, "%s", buf);
    } while (strncmp (buf, "/exit/", 6));

    stop_chat ();

    return 0;
}
```

Листинг 5.2 — client_func.c

```
/*
 * Library containing user functions for working with network
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

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

#include <pthread.h>

#include "shared_data.h"
#include "user_socket.h"
#include "msghandler.h"
#include "network.h"
#include "types.h"
#include "error.h"


pthread_t    thr_recv_chat_msg;
pthread_t    thr_send_chat_msg;


int          stop_chat_flg = 0;
int          stop          = 0;

int init_shared_data (void)
{
    sigset_t newset;

    sigemptyset (&newset);
    sigaddset (&newset, SIGPIPE);
    sigprocmask (SIG_BLOCK, &newset, 0);


    extern int   CLIENT_SOCKET;

    CLIENT_SOCKET = socket (PF_INET, SOCK_STREAM, 0);
    if (CLIENT_SOCKET < 0) {
        return print_err ("CLIENT_SOCKET = socket ()!");
    }
    if (nblck_sock_mode (CLIENT_SOCKET, 1) == -1) {
        close (CLIENT_SOCKET);
        return print_err ("nblck_sock_mode ON!");
    }

    extern struct sockaddr_in  SERVERS[];

    int         port;
    char        addr[16];
    FILE       *fp;

    fp = fopen (CONF_FNAME, "r");
    if (fp == NULL) {
        close (CLIENT_SOCKET);
        return print_err ("read info about servers!");
    }

    for (int i = 0; i < NUM_OF_SERVERS; ++i) {
        fscanf (fp, "%s", addr);
        fscanf (fp, "%d", &port);

        bzero (&SERVERS[i], sizeof (SERVERS[i]));
```

```
        SERVERS[i].sin_family = AF_INET;
        SERVERS[i].sin_port   = htons (port);
        inet_aton (addr, &SERVERS[i].sin_addr);
    }

    extern int  PID;
    extern int  CUR_SERVER;

    PID = getpid ();

    CUR_SERVER = PID % NUM_OF_SERVERS;

    return 0;
}




int connect_to_server (int mode)
{
    extern int     CLIENT_SOCKET;
    extern int     CUR_SERVER;

    int            status;

    if (mode == DISCONNECT) {
        status = send_msg_nblck ( CLIENT_SOCKET, DISCONNECT,
                                  NULL         , 0         , TIMEOUT_SEND
);
        if (status <= 0) {
            return print_err ("send_msg_nblck ()!");
        }
        return 0;
    } else if (mode == RECONNECT) {
        close (CLIENT_SOCKET);
        CLIENT_SOCKET = socket (PF_INET, SOCK_STREAM, 0);
        if (CLIENT_SOCKET < 0) {
            return print_err ("CLIENT_SOCKET = socket ()!");
        }
        if (nblck_sock_mode (CLIENT_SOCKET, 1) == -1) {
            close (CLIENT_SOCKET);
            return print_err ("nblck_sock_mode ON!");
        }
        CUR_SERVER = (CUR_SERVER + 1) % NUM_OF_SERVERS;
    } else if (mode != CONNECT) {
        return print_err ("incorrect mode connect_to_server ()!");
    }

    socklen_t   addr_len;

    for (int i = CUR_SERVER; i < NUM_OF_SERVERS * TRY_CONNECT; ++i) {
        CUR_SERVER = i % NUM_OF_SERVERS;
        addr_len   = sizeof (SERVERS[CUR_SERVER]);
        status = connect_nblck ( CLIENT_SOCKET                          ,
                                 (struct sockaddr *) &SERVERS[CUR_SERVER],
                                 addr_len                               ,
                                 TIMEOUT_CONNECT
```

```
);
        if (status == 0) {
            break;
        }
        close (CLIENT_SOCKET);
        CLIENT_SOCKET = socket (PF_INET, SOCK_STREAM, 0);
        if (CLIENT_SOCKET < 0) {
            return print_err ("CLIENT_SOCKET = socket ()!");
        }
        if (nblck_sock_mode (CLIENT_SOCKET, 1) == -1) {
            close (CLIENT_SOCKET);
            return print_err ("nblck_sock_mode ON!");
        }
    }
    if (status != 0) {
        return print_info ("All servers disabled!");
    }

    struct __login    login;
    int               type;
    char              buf[MAX_MSG_SIZE];

    login.pid = PID;
    inet_aton (ADDR, &login.addr);

    if (mode == RECONNECT) {
        status = send_msg_nblck ( CLIENT_SOCKET, RECONNECT     ,
                                  &login       , sizeof (login),
                                  TIMEOUT_SEND                  );
    } else {
        status = send_msg_nblck ( CLIENT_SOCKET, CONNECT       ,
                                  &login       , sizeof (login),
                                  TIMEOUT_SEND                  );
    }

    if (status <= 0) {
        return print_err ("send_msg_nblck ()!");
    }

    status = recv_msg_nblck (CLIENT_SOCKET, &type, buf, TIMEOUT_RECV);
    if (status <= 0) {
        return print_err ("recv_msg_nblck ()!");
    }

    return handler_msg (type, buf);
}


void *recv_chat_msg (void *arg)
{
    char    buf[MAX_MSG_SIZE + 10];
    int     type;
    int     status;

    while (!stop_chat_flg) {
        status = recv_msg_nblck (CLIENT_SOCKET, &type, buf, TIMEOUT_RECV);
        if (status == -4 && !stop) {
```

```
            stop = 1;
            if (connect_to_server (RECONNECT)) {
                stop = 0;
                break;
            }
            stop = 0;
        }

        if (status > 0) {
            handler_msg (type, buf);
        }
    }

    pthread_exit (NULL);
}


void *send_chat_msg (void *arg)
{
    FILE  *fp;
    char   buf[MAX_MSG_SIZE];
    int    status;


    while (!stop_chat_flg) {
        bzero (buf, sizeof (buf));
        fp = fopen (".buf.txt", "r");
        if (fp == NULL) {
            continue;
        }
        while (stop) sleep (1);
        fgets (buf, MAX_MSG_SIZE, fp);
        fclose (fp);
        if (strlen (buf) > 1) {
            fp = fopen (".buf.txt", "w");
            fclose (fp);
            status = send_msg_nblck ( CLIENT_SOCKET, CHAT        ,
                                      buf          , sizeof (buf),
                                      TIMEOUT_SEND                );

            if (status <= 0) {
                continue;
            }

        }
    }

    pthread_exit (NULL);
}


void start_chat (void)
{
    int status;

    init_shared_data ();
    status = connect_to_server (CONNECT);
```

```
    if (status != 0) {
        close (CLIENT_SOCKET);
        exit (1);
    }

    system ("/bin/gnome-terminal --command ./send_msg.out");
    pthread_create (&thr_recv_chat_msg, NULL, recv_chat_msg, NULL);
    pthread_create (&thr_send_chat_msg, NULL, send_chat_msg, NULL);
}


void stop_chat (void)
{
    stop = 1;
    connect_to_server (DISCONNECT);
    stop_chat_flg = 1;
    pthread_join (thr_recv_chat_msg, NULL);
    pthread_join (thr_send_chat_msg, NULL);
    close (CLIENT_SOCKET);
}
```

Листинг 5.3 — client_func.h

```
#ifndef CLIENT_FUNC
#define CLIENT_FUNC

void start_chat (void);
void stop_chat (void);

#endif
```

Листинг 5.4 — msghandler.c

```
/*
 * Library containing user functions for working with network
 */
#include <stdio.h>

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

#include <arpa/inet.h>

#include "shared_data.h"
#include "types.h"
#include "error.h"

int info_msg_handler (char *buf)
{
    struct __chat   chat;

    memcpy (&chat, buf, sizeof (struct __chat));

    printf ( "\n\E[35m----------------------------\E[0m");
    printf ( "\E[35m----------------------------\E[0m\n");
```

```
    printf ( "\E[32m[ ip: %s ; pid: %d ] "
             "\E[35m%s\E[0m\n",
             inet_ntoa (chat.login.addr), chat.login.pid, chat.buf );
    printf ( "\n\E[35m----------------------------\E[0m");
    printf ( "\E[35m----------------------------\E[0m\n");

    return 0;
}

int chat_handler (char *buf)
{
    struct __chat   chat;

    memcpy (&chat, buf, sizeof (struct __chat));

    printf ( "[ \E[32mip: %s\E[0m ;"
             " \E[33mpid: %d \E[0m ]"
             " \E[34m-->\E[0m %s",
             inet_ntoa (chat.login.addr), chat.login.pid, chat.buf );

    return 0;
}

int info_handler (void *buf)
{
    printf ("\n\E[H\E[2J\n");
    printf ("\n\E[H\E[2J\n");

    return *(int *) buf;
}

int handler_msg (int type, void *buf)
{
    if (type == CHAT) {
        return chat_handler (buf);
    } else if (type == INFO) {
        return info_handler (buf);
    } else if (type == INFO_MSG) {
        return info_msg_handler (buf);
    }

    return -1;
}
```

Листинг 5.5 — msghandler.h

```
#ifndef MSGHANDLER_H
#define MSGHANDLER_H

int handler_msg (int, void *);

#endif
```

Листинг 5.6 — send_msg.c

```
#include <stdio.h>
#include "shared_data.h"

int main (int argc, char *argv[])
{
    FILE *fp;
    char  buf[MAX_MSG_SIZE];

    fp = fopen (".buf.txt", "w");
    fclose (fp);

    while (1) {
        printf ("\E[32mInput text\E[34m > \E[0m");
        fgets (buf, MAX_MSG_SIZE, stdin);
        fp = fopen (".buf.txt", "w");
        if (!strncmp (buf, "/exit/", 6)) {
            fclose (fp);
            break;
        }
        fputs (buf, fp);
        fclose (fp);
    }
    return 0;
}
```

Листинг 5.7 — shared_data.h

```
#ifndef SHARED_DATA_H
#define SHARED_DATA_H

#include <sys/un.h>
#include <resolv.h>

enum {
        NO_TYPE   , INFO      , CONNECT   , RECONNECT ,
        DISCONNECT, CHAT      , INFO_MSG
     };

#define NUM_OF_SERVERS   5

int                      CUR_SERVER;

#define TIMEOUT_CONNECT  5
#define TIMEOUT_SEND     5
#define TIMEOUT_RECV     5
#define CONF_FNAME       "servers.conf"

#define MAX_MSG_SIZE     50

#define ADDR             "127.0.0.1"

int                      PID;

#define TRY_CONNECT      3
#define TRY_SEND         1
```

```
#define TRY_RECV         1

int                      CLIENT_SOCKET;

struct sockaddr_in       SERVERS[NUM_OF_SERVERS];

#endif
```

Листинг 5.8 — types.h

```
#ifndef USER_TYPES_H
#define USER_TYPES_H

#include <netinet/in.h>
#include "shared_data.h"

struct __header {
    int              type;
    long int         size;

};

struct __login {
    struct in_addr   addr;
    int              pid;
};

struct __chat {
    struct __login   login;
    char             buf[MAX_MSG_SIZE];
};

#endif
```

5.2. Серверная часть

Листинг 5.9 — main.c

```
#include <stdio.h>
#include <string.h>
#include "server_func.h"

int main (int argc, char *argv[])
{
    start_server ();

    char   buf[10];

    do {
        fscanf (stdin, "%s", buf);
    } while (strncmp (buf, "/exit/", 6));

    stop_server ();
```

```
    return 0;
}
```

Листинг 5.10 — server_func.c

```
/*
 * Library containing user functions for working with network
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <string.h>

#include <sys/socket.h>
#include <arpa/inet.h>

#include <pthread.h>

#include "shared_data.h"
#include "user_socket.h"
#include "msghandler.h"
#include "network.h"
#include "types.h"
#include "error.h"
#include "list.h"


pthread_t    thr_accept_clients;
pthread_t    thr_recv_servers;
pthread_t    thr_recv_client[MAX_USERS];

pthread_mutex_t   mut = PTHREAD_MUTEX_INITIALIZER;

int stop_accept_clients = 0;
int stop_recv_servers   = 0;


int init_shared_data (void)
{
    sigset_t newset;

    sigemptyset (&newset);
    sigaddset (&newset, SIGPIPE);
    sigprocmask (SIG_BLOCK, &newset, 0);


    extern int   SERVER_TO_CLIENT_SOCKET;

    SERVER_TO_CLIENT_SOCKET = socket (PF_INET, SOCK_STREAM, 0);
    if (SERVER_TO_CLIENT_SOCKET < 0) {
        return print_err ("SERVER_TO_CLIENT_SOCKET = socket ()!");
    }
    if (nblck_sock_mode (SERVER_TO_CLIENT_SOCKET, 1) == -1) {
        return print_err ("nblck_sock_mode ON!");
```

```
}

extern struct sockaddr_in  SERVER;
extern struct sockaddr_in  SERVERS[];

int         port;
char        addr[16];
FILE       *fp;

fp = fopen (CONF_FNAME, "r");
if (fp == NULL) {
    return print_err ("read info about servers!");
}

for (int i = 0, j = 0; i < NUM_OF_SERVERS; ++i) {
    fscanf (fp, "%s", addr);
    fscanf (fp, "%d", &port);

    if (i == CUR_SERVER) {
        bzero (&SERVER, sizeof (SERVER));
        SERVER.sin_family = AF_INET;
        SERVER.sin_port   = htons (port);
        inet_aton (addr, &SERVER.sin_addr);
        continue;
    }

    bzero (&SERVERS[j], sizeof (SERVERS[j]));
    SERVERS[j].sin_family = AF_INET;
    SERVERS[j].sin_port   = htons (port + 1000);
    inet_aton (addr, &SERVERS[j].sin_addr);
    ++j;
}

int status;

status = bind ( SERVER_TO_CLIENT_SOCKET    ,
                (struct sockaddr *) &SERVER,
                sizeof (SERVER)              );
if (status < 0) {
    return print_err ("SERVER_TO_CLIENT_SOCKET bind ()!");
}

port = ntohs (SERVER.sin_port);
SERVER.sin_port = htons (port + 1000);

extern int   SERVER_TO_SERVER_SOCKET;

SERVER_TO_SERVER_SOCKET = socket (PF_INET, SOCK_DGRAM, 0);
if (SERVER_TO_SERVER_SOCKET < 0) {
    return print_err ("SERVER_TO_SERVER_SOCKET = socket ()!");
}

status = bind ( SERVER_TO_SERVER_SOCKET    ,
                (struct sockaddr *) &SERVER,
                sizeof (SERVER)              );
if (status < 0) {
    return print_err ("SERVER_TO_SERVER_SOCKET bind ()!");
```

```
    }


    extern struct list  *history;
    extern struct list  *users;

    history = (struct list *) malloc (sizeof (struct list));
    users   = (struct list *) malloc (sizeof (struct list));

    list_init (history);
    list_init (users);
    for (int i = 0; i < MAX_USERS; ++i) {
        state_thr[i] = 0;
    }

    return 0;
}


void *recv_client (void *arg)
{
    struct __recv_msg   info;
    int                 type;
    int                 status;
    char                buf[MAX_MSG_SIZE];

    info = *(struct __recv_msg *) arg;

    while (state_thr[info.ind] == 1) {
        bzero (buf, sizeof (buf));
        status = recv_msg_nblck (info.user.sd, &type, buf, TIMEOUT_RECV);
        if (status == -4) {
            break;
        } else if (status <= 0) {
            continue;
        }

        if (type == DISCONNECT) {
            break;
        }

        handler_client_msg (info.user, type, buf);
    }

    type = REMOVE_USER;
    handler_client_msg (info.user, type, &info.user);

    close (info.user.sd);
    pthread_mutex_lock (&mut);
    state_thr[info.ind] = 0;
    pthread_mutex_unlock (&mut);

    printf ( "\E[32m[ ip: %s ; pid: %d ; sd: %d ]\E[35m "
             "has left the chat\E[0m\n", inet_ntoa (info.user.login.addr),
                                         info.user.login.pid, info.user.sd
);
```

```
    pthread_exit (NULL);
}


void *recv_servers (void *arg)
{
    struct __serv_msg   msg;
    struct __user       user;

    while (!stop_recv_servers) {
        bzero (&msg, sizeof (msg));
        msg.type = -1;
        recvfrom (SERVER_TO_SERVER_SOCKET, &msg, sizeof (msg), 0, NULL,
NULL);
        if (msg.type == -1) {
            continue;
        }

        user.login = msg.login;
        user.sd = -1;
        handler_server_msg (user, msg.type, msg.buf);
    }

    pthread_exit (NULL);
}


void *accept_clients (void *arg)
{
    extern int          state_thr[];

    int                 i;
    int                 client;
    int                 status;
    int                 type;
    char                buf[MAX_MSG_SIZE];
    struct __login      login;
    struct __recv_msg   info;

    if (listen (SERVER_TO_CLIENT_SOCKET, 20) != 0) {
        print_err ("listen ()!");
        pthread_exit (NULL);
    }


    while (!stop_accept_clients) {
        for (i = 0; i < MAX_USERS; ++i) {
            if (state_thr[i] == 0) {
                break;
            }
        }

        client = accept_nblck ( SERVER_TO_CLIENT_SOCKET, NULL          ,
                                NULL                   , TIMEOUT_ACCEPT  )
        if (client <= 0) {
            continue;
        }
```

```
        status = recv_msg_nblck (client, &type, buf, TIMEOUT_RECV);
        if (status <= 0) {
            continue;
        }

        info.user.sd      = client;
        info.ind          = i;

        memcpy (&login, buf, sizeof (struct __login));
        info.user.login = login;

        if (type == CONNECT) {
            type = ADD_USER;
        } else if (type == RECONNECT) {
/* print */
 printf ("EDIT_USER\n");
            type = EDIT_USER;
        } else {
            continue;
        }

        status = handler_client_msg (info.user, type, &info.user);

/* print */
 printf ("status handler edit user %d\n", status);
        if (status != 0) {
            handler_client_msg (info.user, REMOVE_USER, &info.user);
            continue;
        }

        printf ( "\E[32m[ addr: %s ; pid: %d ; sd: %d ]",
                 inet_ntoa (login.addr), login.pid, client );
        if (type == ADD_USER) {
            printf ("\E[35m connected new client\E[0m\n");
        } else if (type == EDIT_USER) {
            printf ("\E[35m reconnected client\E[0m\n");
        }

        pthread_mutex_lock (&mut);
        state_thr[i] = 1;
        pthread_mutex_unlock (&mut);
        pthread_create (&thr_recv_client[i], NULL, recv_client, &info);
    }

    pthread_exit (NULL);
}

int close_users (void *elem, long int size)
{
    struct __user   user;

    memcpy (&user, elem, size);

    if (user.sd > 0) {
        close (user.sd);
    }
```

```
    return 0;
}


int start_server (void)
{
    if (init_shared_data ()) {
        print_err ("Can't start server!");
        exit (1);
    } else {
        printf ("*************** START SERVER ***************\n");
        printf (" ip    : %s              \n",    inet_ntoa
(SERVER.sin_addr));
        printf (" port 1: %d (for clients)\n", ntohs (SERVER.sin_port) -
1000);
        printf (" port 2: %d (for servers)\n",        ntohs
(SERVER.sin_port));
        printf ("*******************************************\n");
    }

    pthread_create (&thr_accept_clients, NULL, accept_clients, NULL);
    pthread_create (&thr_recv_servers, NULL, recv_servers, NULL);

    return 0;
}

int stop_server (void)
{
    close (SERVER_TO_CLIENT_SOCKET);
    close (SERVER_TO_SERVER_SOCKET);

    pthread_mutex_lock (&mut);
    for (int i = 0; i < MAX_USERS; ++i) {
        state_thr[i] = 0;
    }
    pthread_mutex_unlock (&mut);

    sleep (TIMEOUT_RECV);

    pthread_mutex_lock (&mut);
    stop_recv_servers   = 1;
    stop_accept_clients = 1;
    pthread_mutex_unlock (&mut);
    pthread_join (thr_accept_clients, NULL);

    extern struct list  *history;
    extern struct list  *users;

    list_elem_act (users, sizeof (struct __user), close_users);
    list_free (history);
    free (history);
    list_free (users);
    free (users);

    return 0;
}
```

Листинг 5.11 — server_func.h

```
#ifndef SERVER_FUNC
#define SERVER_FUNC

int start_server (void);
int stop_server (void);

#endif
```

Листинг 5.12 — msghandler.c

```
/*
 * Library containing user functions for working with network
 */
#include <stdio.h>
#include <unistd.h>

#include <string.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <pthread.h>

#include "shared_data.h"
#include "user_socket.h"
#include "network.h"
#include "types.h"
#include "error.h"


pthread_mutex_t    mut_users = PTHREAD_MUTEX_INITIALIZER;


struct __user   global_user;
struct __chat   global_msg;


int send_history (void *elem, long int size)
{
    send_msg_nblck (global_user.sd, CHAT, elem, size, TIMEOUT_SEND);

    return 0;
}

int users_info (void *elem, long int size)
{
    int             status;
    struct __user   user;

    memcpy (&user, elem, size);
    if (user.sd > 0) {
```

```
        status = send_msg_nblck ( user.sd    , INFO_MSG           ,
                                  &global_msg, sizeof (global_msg),
                                  TIMEOUT_SEND                      );
        if (status <= 0) {
           close (user.sd);
        }
    }

    return 0;
}

int add_user_c_handler (struct __user user, void *buf)
{
    extern struct list   *users;

    int                  status, val = 0;

    memcpy (&user, buf, sizeof (struct __user));
    pthread_mutex_lock (&mut_users);
    if (list_push_front (users, &user, sizeof (user))) {
        val = -1;
        return print_err ("Don't added user!");
    }

    status = send_msg_nblck ( user.sd, INFO        ,
                              &val   , sizeof (int), TIMEOUT_SEND );
    if (status <= 0 || val == -1) {
        return -1;
    }

    extern struct list   *history;

    global_user.sd = user.sd;
    list_elem_act (history, sizeof (struct __chat), send_history);

    bzero (&global_msg, sizeof (global_msg));
    global_msg.login = user.login;
    memcpy (global_msg.buf, "has joined the chat", 19);
    list_elem_act (users, sizeof (struct __user), users_info);
    pthread_mutex_unlock (&mut_users);

    struct __serv_msg   msg;

    msg.type  = ADD_USER;
    msg.login = user.login;
    user.sd   = -1;
    memcpy (msg.buf, &user, sizeof (user));
    sendto_broadcast ( SERVER_TO_SERVER_SOCKET    , &msg     ,
                       sizeof (msg)               , 0        ,
                       (struct sockaddr *) SERVERS,
                       sizeof (SERVER) * (NUM_OF_SERVERS - 1)  );

    return 0;
}

int add_user_s_handler (struct __user user, void *buf)
{
```

```
    extern struct list   *users;

    pthread_mutex_lock (&mut_users);
    if (list_push_front (users, buf, sizeof (struct __user))) {
        return print_err ("Don't added user!");
    }
    printf ("add user %s\n", inet_ntoa (user.login.addr));
    bzero (&global_msg, sizeof (global_msg));
    global_msg.login = user.login;
    memcpy (global_msg.buf, "has joined the chat", 19);
    list_elem_act (users, sizeof (struct __user), users_info);
    pthread_mutex_unlock (&mut_users);

    return 0;
}

int remove_user_c_handler (struct __user user, void *buf)
{
    extern struct list   *users;

    int              status;

    pthread_mutex_lock (&mut_users);
    status = list_elem_find ( users, &user                 ,
                              buf  , sizeof (struct __user),
                              NULL                          );
    if (status == 0) {
        status = list_elem_remove (users, &user, sizeof (user));
    }
    bzero (&global_msg, sizeof (global_msg));
    global_msg.login = user.login;
    memcpy (global_msg.buf, "has left the chat", 17);
    list_elem_act (users, sizeof (struct __user), users_info);
    pthread_mutex_unlock (&mut_users);

    struct __serv_msg   msg;

    msg.type  = REMOVE_USER;
    msg.login = user.login;
    user.sd   = -1;
    memcpy (msg.buf, &user, sizeof (user));
    sendto_broadcast ( SERVER_TO_SERVER_SOCKET    , &msg     ,
                       sizeof (msg)               , 0        ,
                       (struct sockaddr *) SERVERS,
                       sizeof (SERVER) * (NUM_OF_SERVERS - 1)  );

    return 0;
}

int remove_user_s_handler (struct __user user, void *buf)
{
    extern struct list   *users;

    int                  status;

    pthread_mutex_lock (&mut_users);
    status = list_elem_find ( users, &user                 ,
```

```
                                buf  , sizeof (struct __user),
                                NULL                          );
    if (status == 0) {
        list_elem_remove (users, &user, sizeof (struct __user));
    }
    bzero (&global_msg, sizeof (global_msg));
    global_msg.login = user.login;
    memcpy (global_msg.buf, "has left the chat", 17);
    list_elem_act (users, sizeof (struct __user), users_info);
    pthread_mutex_unlock (&mut_users);

    return 0;
}

int edit_func (void *elem, long int size)
{
    struct __user   user;

    memcpy (&user, elem, size);
    if (!memcmp (&user.login, &global_user.login, sizeof (struct __login))
{
/* print */
printf ("EDIT! %s\n", inet_ntoa (user.login.addr));
        memcpy (elem, &global_user, size);
        return -1;
    }

    return 0;
}

int edit_user_handler (struct __user user, void *buf)
{
    extern struct list    *users;

    int                   status, val = 0;

    pthread_mutex_lock (&mut_users);
    memcpy (&global_user, buf, sizeof (struct __user));
    status = list_elem_act (users, sizeof (struct __user), edit_func);

    if (status == -1) {
        val = 0;
    } else {
        val = -1;
    }

    status = send_msg_nblck ( user.sd, INFO        ,
                              &val   , sizeof (int), TIMEOUT_SEND );
    if (status <= 0 || val == -1) {
        return -1;
    }
    extern struct list   *history;

    global_user.sd = user.sd;
    list_elem_act (history, sizeof (struct __chat), send_history);
    pthread_mutex_unlock (&mut_users);
```

```
    return 0;
}

int chat_func (void *elem, long int size)
{
    int             status;
    struct __user   user;

    memcpy (&user, elem, size);
    if (user.sd > 0) {
        status = send_msg_nblck ( user.sd    , CHAT               ,
                                  &global_msg, sizeof (global_msg),
                                  TIMEOUT_SEND                      );
        if (status <= 0) {
           close (user.sd);
        }
    }

    return 0;
}

int chat_c_handler (struct __user user, void *buf)
{
    extern struct list   *users;
    extern struct list   *history;

    struct __serv_msg     msg;

    pthread_mutex_lock (&mut_users);
    bzero (&global_msg, sizeof (global_msg));
    memcpy (global_msg.buf, buf, MAX_MSG_SIZE);
    global_msg.login = user.login;
    list_elem_act (users, sizeof (struct __user), chat_func);

    list_push_back (history, &global_msg, sizeof (global_msg));
    if (history->size > MAX_HISTORY_SIZE) {
        list_first_elem_remove (history);
    }

    pthread_mutex_unlock (&mut_users);

    msg.type  = CHAT;
    msg.login = user.login;
    memcpy (msg.buf, buf, MAX_MSG_SIZE);
    sendto_broadcast ( SERVER_TO_SERVER_SOCKET    , &msg     ,
                       sizeof (msg)               , 0        ,
                       (struct sockaddr *) SERVERS,
                       sizeof (SERVER) * (NUM_OF_SERVERS - 1)  );

    return 0;
}

int chat_s_handler (struct __user user, void *buf)
{
    extern struct list   *users;
    extern struct list   *history;
```

```
    pthread_mutex_lock (&mut_users);
    bzero (&global_msg, sizeof (global_msg));
    memcpy (global_msg.buf, buf, MAX_MSG_SIZE);
    global_msg.login = user.login;
    list_elem_act (users, sizeof (struct __user), chat_func);

    list_push_back (history, &global_msg, sizeof (global_msg));
    if (history->size > MAX_HISTORY_SIZE) {
        list_first_elem_remove (history);
    }
    pthread_mutex_unlock (&mut_users);

    return 0;
}

int handler_client_msg (struct __user user, int type, void *buf)
{
    if (type == ADD_USER) {
/* print */
printf ("add_user %s\n", __func__);
        return add_user_c_handler (user, buf);
    } else if (type == REMOVE_USER) {
        return remove_user_c_handler (user, buf);
    } else if (type == EDIT_USER) {
        return edit_user_handler (user, buf);
    } else if (type == CHAT) {
        return chat_c_handler (user, buf);
    }

    return -1;
}

int handler_server_msg (struct __user user, int type, void *buf)
{
    if (type == ADD_USER) {
        return add_user_s_handler (user, buf);
    } else if (type == REMOVE_USER) {
        return remove_user_s_handler (user, buf);
    } if (type == CHAT) {
        return chat_s_handler (user, buf);
    }

    return -1;
}
```

Листинг 5.13 — msghandler.h

```
#ifndef MSGHANDLER_H
#define MSGHANDLER_H

#include "types.h"

int handler_client_msg (struct __user, int, void *);
int handler_server_msg (struct __user, int, void *);

#endif
```

Листинг 5.14 — shared_data.h

```
#ifndef SHARED_DATA_H
#define SHARED_DATA_H

#include <resolv.h>
#include "list.h"

enum {
        NO_TYPE    , INFO       , CONNECT    , RECONNECT  ,
        DISCONNECT , CHAT       , INFO_MSG
     };

enum {
        ADD_USER   , REMOVE_USER, EDIT_USER
     };

#define NUM_OF_SERVERS   5

#define CUR_SERVER       3

#define TIMEOUT_ACCEPT   60
#define TIMEOUT_SEND     5
#define TIMEOUT_RECV     5
#define CONF_FNAME       "servers.conf"

#define MAX_MSG_SIZE     50
#define MAX_HISTORY_SIZE 10

#define MAX_USERS        20

#define TRY_SEND         1
#define TRY_RECV         1

struct list             *history;
struct list             *users;

int                      state_thr[MAX_USERS];

int                      SERVER_TO_CLIENT_SOCKET;
int                      SERVER_TO_SERVER_SOCKET;

struct sockaddr_in       SERVER;
struct sockaddr_in       SERVERS[NUM_OF_SERVERS - 1];

#endif
```

Листинг 5.15 — types.h

```
#ifndef USER_TYPES_H
#define USER_TYPES_H

#include <netinet/in.h>
#include "shared_data.h"
```

```
struct __header {
    int               type;
    long int          size;
};

struct __login {
    struct in_addr    addr;
    int               pid;
};

struct __chat {
    struct __login    login;
    char              buf[MAX_MSG_SIZE];
};

struct __serv_msg {
    struct __login    login;
    int               type;
    char              buf[MAX_MSG_SIZE];
};

struct __user {
    struct __login    login;
    int               sd;
};

struct __recv_msg {
    struct __user     user;
    int               ind;
};

#endif
```

5.2. Другие файлы

Листинг 5.16 — error.c

```
/*
 * Library containing the functions, whitch print error messages
 */
#include <stdio.h>


/*
 * Print error message *mes and returning value -1
 * This function reset background and font color
 */
int print_err (char *mes)
{
    fprintf (stderr, "\E[31mERROR:\E[0m %s\n", mes);
    return -1;
}
```

```
/*
 * Print warning message *mes and returning value -1
 * This function reset background and font color
 */
int print_war (char *mes)
{
    fprintf (stdout, "\E[35;1mWARNING:\E[0m %s\n", mes);
    return -2;
}

/*
 * Print info message *mes and returning value -1
 * This function reset background and font color
 */
int print_info (char *mes)
{
    fprintf (stdout, "\E[32;1mINFO:\E[0m %s\n", mes);
    return -2;
}
```

Листинг 5.17 — error.h

```
#ifndef ERROR_H
#define ERROR_H

int print_err (char *);
int print_war (char *);
int print_info (char *);

#endif
```

Листинг 5.18 — list.c

```
/*
 * list
 */
#include <stdlib.h>
#include <string.h>
#include "error.h"

/*
 * Struct the element of the list
 */
struct node {

    void *content;     // Content

    struct node *next; // Pointer to next element

};

/*
 * Struct the list
 */
struct list {
```

```
    int size;          // List size

    struct node *head; // Pointer to first element in list
    struct node *tail; // Pointer to last element in list

};


/*
 * List initializer
 * list_ptr -- pointer to list
 */
void list_init (struct list *list_ptr)
{
   /*
    * List is empty
    */
    list_ptr->size = 0;
    list_ptr->head = NULL;
    list_ptr->tail = NULL;
}

/*
 * Adding element to the top of the list
 * list_ptr -- pointer to list
 * content  -- adding element
 * len      -- size of element
 */
int list_push_front (struct list *list_ptr, void *content, long int len)
{
    struct node *node_ptr;

   /*
    * Allocating memory for node of the list
    */
    node_ptr = (void *) malloc (sizeof (struct node));
    if (!node_ptr) {
        return print_err ("list_elem_add ()");
    }
   /*
    * Allocating memory for content of the node
    * Adding content
    */
    node_ptr->content = (void *) malloc (len);
    memcpy (node_ptr->content, content, len);

   /*
    * If list is empty
    */
    if (list_ptr->head == NULL) {
        list_ptr->tail = node_ptr;
    }

   /*
    * List is no empty
    */
```

```
    node_ptr->next = list_ptr->head;
    list_ptr->head = node_ptr;
    ++list_ptr->size;
    return 0;
}

/*
 * Adding element to the end of the list
 * list_ptr -- pointer to list
 * content  -- adding element
 * len      -- size of element
 */
int list_push_back (struct list *list_ptr, void *content, long int len)
{
    struct node *node_ptr;

   /*
    * Allocating memory for node of the list
    */
    node_ptr = (void *) malloc (sizeof (struct node));
    if (!node_ptr) {
        return print_err ("list_elem_add ()");
    }
   /*
    * Allocating memory for content of the node
    * Adding content
    */
    node_ptr->content = (void *) malloc (len);
    memcpy (node_ptr->content, content, len);
    node_ptr->next = NULL;

   /*
    * If list is empty
    */
    if (list_ptr->head == NULL) {
        list_ptr->head = node_ptr;
        list_ptr->tail = node_ptr;
   /*
    * If list is no empty
    */
    } else {
        list_ptr->tail->next = node_ptr;
        list_ptr->tail = node_ptr;
    }
    ++list_ptr->size;
    return 0;
}

/*
 * Free memory (remove list)
 * list_ptr -- pointer to list
 */
void list_free (struct list *list_ptr)
{
    struct node *node_ptr;

    for ( node_ptr  = list_ptr->head;
```

```
          node_ptr != NULL          ;
          node_ptr = list_ptr->head   ) {
        list_ptr->head = node_ptr->next;
        free (node_ptr->content);  // Free content of the node
        free (node_ptr);           // Free node of the list
    }
   /*
    * List is empty
    */
    list_ptr->head = NULL;
    list_ptr->tail = NULL;
    list_ptr->size = 0;
}


/*
 * Finding element of the list
 * list_ptr -- pointer to list
 * dest     -- found element
 * src      -- finding element
 * len      -- size of element
 * func_cmp -- pointer to function for to compare of the elements
 *             param1 -- first parameter (element of the list)
 *             param2 -- second parameter (element --//--)
 *             l      -- size of parameters (size of elements --//--)
 */
int list_elem_find ( struct list *list_ptr                          ,
                     void *dest                                     ,
                     void *src                                      ,
                     long int len                                   ,
                     int (*func_cmp)
                         (void *param1, void *param2, long int l)  )
{
    struct node *node_ptr;

   /*
    * Default compare
    */
    if (func_cmp == NULL) {
        for ( node_ptr  = list_ptr->head;
              node_ptr != NULL          ;
              node_ptr  = node_ptr->next  ) {
            if (!memcmp (node_ptr->content, src, len)) {
               /*
                * Found element
                */
                break;
            }
        }
        if (node_ptr == NULL) {
           /*
            * Don't found element
            */
            return -1;
        }
   /*
    * User copmare function
```

```
    */
    } else {
        for ( node_ptr  = list_ptr->head;
              node_ptr != NULL          ;
              node_ptr  = node_ptr->next  ) {
            if (!func_cmp (node_ptr->content, src, len)) {
               /*
                * Found element
                */
                break;
            }
        }
        if (node_ptr == NULL) {
           /*
            * Don't found element
            */
            return -1;
        }
    }
   /*
    * Copy found element
    */
    memcpy (dest, node_ptr->content, len);
    return 0;
}

/*
 * Remove element from the list
 * list_ptr -- pointer to list
 * content  -- removing element
 * len      -- size of element
 */
int list_elem_remove (struct list *list_ptr, void *content, long int len)
{
    struct node *node_ptr, *node_prev;

    for ( node_ptr  = list_ptr->head, node_prev = NULL;
          node_ptr != NULL                           ;
          node_ptr  = node_ptr->next                   ) {
        if (memcmp (content, node_ptr->content, len) == 0) {
            if (node_prev == NULL) {
                if (node_ptr == list_ptr->tail) {
                    list_ptr->tail = NULL;
                }
                list_ptr->head = node_ptr->next;
                free (node_ptr->content);  // Free content of the node
                free (node_ptr);           // Free node of the list
                --list_ptr->size;          // Dicrement number elements
                return 0;                  // Success
            } else {
                if (node_ptr == list_ptr->tail) {
                    list_ptr->tail = node_prev;
                }
                node_prev->next = node_ptr->next;;
                free (node_ptr->content);  // Free content of the node
                free (node_ptr);           // Free node of the list
                --list_ptr->size;          // Dicrement number elements
```

```
                return 0;                   // Success
            }
        } else {
            node_prev = node_ptr;           // Next iteration
        }
    }
    return -1;  // Element not found
}

int list_first_elem_remove (struct list *list_ptr)
{
    if (!list_ptr->size) {
        return print_war ("List is empty!");
    }
    struct node *node_ptr;

    node_ptr = list_ptr->head;
    list_ptr->head = node_ptr->next;
    if (list_ptr->size == 1) {
        list_ptr->tail = NULL;
    }
    free (node_ptr->content);
    free (node_ptr);
    --list_ptr->size;
    return 0;
}

/*
 * Action to all elements of the list
 * list_ptr -- pointer to list
 * len      -- size of element
 * func_act -- pointer to action function
 *             elem -- element of the list
 *             size -- size of element
 */
int list_elem_act ( struct list *list_ptr                       ,
                    long int len                                ,
                    int (*func_act) (void *elem, long int size)  )
{
    struct node *node_ptr;
    int          status;

   /*
    * Non action
    */
    if (func_act == NULL) {
            return 0;
    }
   /*
    * User action function
    */
    for ( node_ptr  = list_ptr->head;
          node_ptr != NULL          ;
          node_ptr  = node_ptr->next  ) {
        status = func_act (node_ptr->content, len);
        if (status == -1) {
            return -1;
```

```
        }
    }
    return 0;
}
```

Листинг 5.19 — list.h

```
#ifndef LIST_H
#define LIST_H

struct node {
    void *content;     // Content
    struct node *next; // Pointer to next element
};

struct list {
    int size;          // List size
    struct node *head; // Pointer to first element in list
};

/*
 * Prototype
 */
void list_init (struct list *);
int  list_push_front (struct list *, void *, long int);
int  list_push_back (struct list *, void *, long int);
void list_free (struct list *);
int  list_elem_find ( struct list *, void *, void *, long int,
                      int (*func_cmp) (void *, void *, long int) );
int  list_elem_remove (struct list *, void *, long int);
int  list_first_elem_remove (struct list *);
int  list_elem_act ( struct list *, long int,
                     int (*func_act) (void *, long int) );

#endif
```

Листинг 5.20 — user_socket.c

```
/*
 * Library containing user functions for working with network
 */
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/socket.h>
#include "error.h"

int nblck_sock_mode (int sd, int mode)
{
    int arg;

    arg = 1;

    if (setsockopt (sd, SOL_SOCKET, SO_REUSEADDR, &arg, sizeof (int)) < 0)
{
```

```
        return print_err ("SO_REUSEADDR!");
    }


    arg = fcntl (sd, F_GETFL, 0);

    if (mode % 2) {
        if (fcntl (sd, F_SETFL, arg | O_NONBLOCK) == -1) {
            return print_err ("fcntl ()!");
        }
    } else {
        if (fcntl (sd, F_SETFL, arg & ~O_NONBLOCK) == -1) {
            return print_err ("fcntl ()!");
        }
    }
    return 0;
}

int connect_nblck ( int sd        , struct sockaddr *server,
                    socklen_t size, int timeout            ,
                    int try                                  )
{
    struct timeval  tv;
    fd_set          fdset;
    int             status;

    status = connect (sd, server, size);
    if (status == -1 && errno == EINPROGRESS) {
        tv.tv_sec  = timeout;
        tv.tv_usec = 0;
        FD_ZERO (&fdset);
        FD_SET (sd, &fdset);
        status = select (sd + 1, NULL, &fdset, NULL, timeout ? &tv : NULL)
        if (status == 1) {
            if ( getsockopt ( sd      , SOL_SOCKET,
                              SO_ERROR, &status   , &size ) == -1 ) {
                return print_err ("getsockopt ()!");
            }
        } else {
            return -1;
        }
    } else {
        return -1;
    }
    return status;
}

int accept_nblck ( int sd             , struct sockaddr *addr,
                   socklen_t *addr_len, int timeout           )
{
    int               status;
    struct timeval    tv;
    fd_set            fdset;

    tv.tv_sec  = timeout;
    tv.tv_usec = 0;
    FD_ZERO (&fdset);
```

```
    FD_SET  (sd, &fdset);

    status = select (sd + 1, &fdset, NULL, NULL, timeout ? &tv : NULL);

    if (status != 1) {
        return -2;
    }

    status = accept (sd, addr, addr_len);
    if (status == EWOULDBLOCK) {
        return -2;
    }
    return status;
}

/*
 * Send message
 * type    -- type message
 * content -- body message
 * size    -- size body
 */
int send_nblck (int sd, void *content, long int size, int timeout)
{
    if (size == 0 || content == NULL) {
        return print_err ("Sending empty message!");
    }

    int             status;
    struct timeval  tv;
    fd_set          fdset;

    tv.tv_sec  = timeout;
    tv.tv_usec = 0;
    FD_ZERO (&fdset);
    FD_SET  (sd, &fdset);

    status = select (sd + 1, NULL, &fdset, NULL, timeout ? &tv : NULL);

    if (status != 1) {
        return -2;
    }

    status = send (sd, content, size, 0);
    if (status == EWOULDBLOCK) {
        return -2;
    } else if (status != size) {
        return -2;
    }
    return status;
}

/*
 * Receive message
 * type    -- type message
 * content -- body message
 */
int recv_nblck (int sd, void *content, long int size, int timeout)
```

```
{
    int             status;
    struct timeval  tv;
    fd_set          fdset;

    tv.tv_sec  = timeout;
    tv.tv_usec = 0;
    FD_ZERO (&fdset);
    FD_SET  (sd, &fdset);

    status = select (sd + 1, &fdset, NULL, NULL, timeout ? &tv : NULL);

    if (status != 1) {
        return -2;
    }

    status = recv (sd, content, size, 0);
    if (status == EWOULDBLOCK) {
        return -2;
    } else if (status != size) {
        return -4;
    }
    return status;
}

int sendto_broadcast ( int sd, const void *buf, size_t len, int flags,
                       const struct sockaddr *dest_addr, socklen_t addrlen
)
{
    int N = addrlen / sizeof (struct sockaddr);
    int ret = 0;

    for (int i = 0; i < N; ++i) {
        ret += sendto ( sd           , buf                  ,
                        len          , flags                ,
                        &dest_addr[i], sizeof (dest_addr[i])  );
    }
    return ret;
}
```

Листинг 5.21 — user_socket.h

```
#ifndef NBLCK_SOCKET_H
#define NBLCK_SOCKET_H

#include <sys/socket.h>

int  nblck_sock_mode (int, int);
int  connect_nblck (int, struct sockaddr *, socklen_t, int);
int  accept_nblck ( int sd, struct sockaddr *, socklen_t *, int);
int  send_nblck (int, void *, long int, int);
int  recv_nblck (int, void *, long int, int);

int  sendto_broadcast ( int, const void *, size_t, int ,
                        const struct sockaddr *, socklen_t );
```

```
#endif
```

6. Скан-сессии

```
[anatoly@pc server_1]$ ./main.out
**************** START SERVER ****************
 ip    : 127.0.0.1
 port 1: 7778 (for clients)
 port 2: 8778 (for servers)
**********************************************
```

Рисунок 6.1 — Старт 1-ого сервера

```
[anatoly@pc server]$ ./main.out
**************** START SERVER ****************
 ip    : 127.0.0.1
 port 1: 7777 (for clients)
 port 2: 8777 (for servers)
**********************************************
```

Рисунок 6.2 — Старт 2-ого сервера

```
------------------------------------------------------------
[ ip: 127.0.0.1 ; pid: 10419 ] has joined the chat

------------------------------------------------------------
```

Рисунок 6.3 — Подключение 1-ого клиента (сторона клиента)

```
**************** START SERVER ****************
 ip    : 127.0.0.1
 port 1: 7778 (for clients)
 port 2: 8778 (for servers)
**********************************************
recv1 16
recv2 8
add_user handler_client_msg
send1 16
send2 4
send1 16
send2 60
status handler edit user 0
[ addr: 127.0.0.1 ; pid: 10419 ; sd: 6 ] connected new client
```

Рисунок 6.4 — Подключение 1-ого клиента (сторона первого сервера)

```
**************** START SERVER ****************
 ip    : 127.0.0.1
 port 1: 7777 (for clients)
 port 2: 8777 (for servers)
**********************************************
add user 127.0.0.1
```

Рисунок 6.5 — Передача другому серверу информации о подключившемся клиенте

Рисунок 6.6 — Ввод сообщения 1-ым клиентом

Рисунок 6.7 — Получение сообщения на стороне сервера

Рисунок 6.8 — Подключение 2-ого клиента (со стороны клиента)

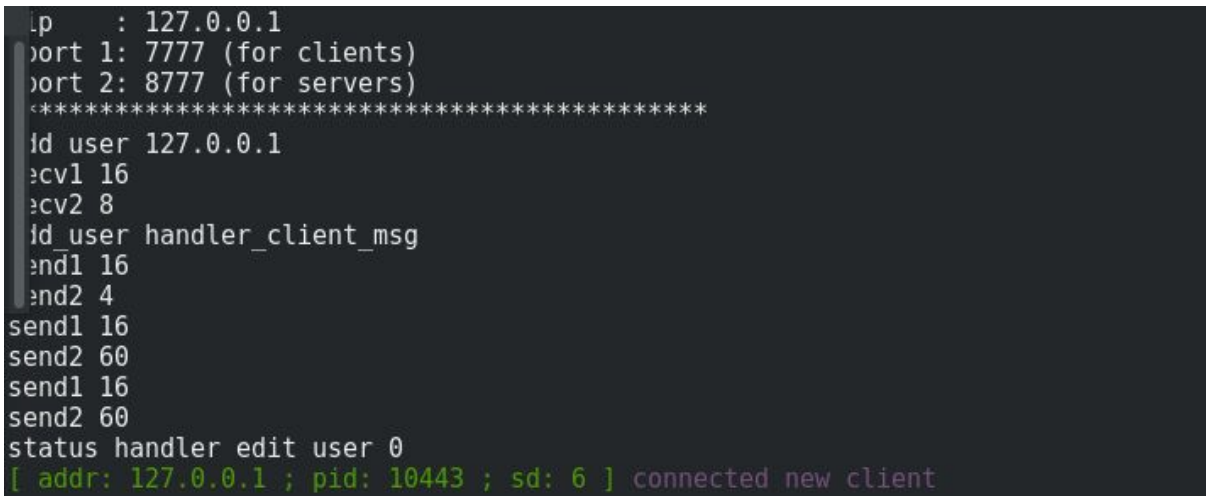

Рисунок 6.9 — Подключение 2-ого клиента (со стороны 2-ого сервера)

Рисунок 6.10 — Передача другому серверу информации о подключившемся клиенте

Рисунок 6.11 — Ввод сообщения 1-ым клиентом

Рисунок 6.12 — Состояние чата 2-ого клиента

Рисунок 6.13 — Состояние чата 1-ого клиента

Рисунок 6.14 — Остановка 1-ого сервера

Рисунок 6.15 — Состояние чата 1-ого клиента после автоматического переподключения

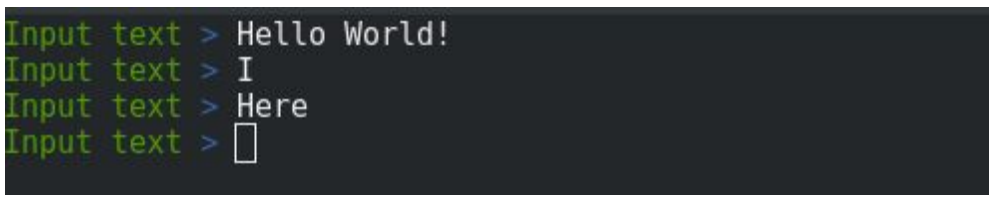

Рисунок 6.16 — Ввод сообщения 1-ым клиентом

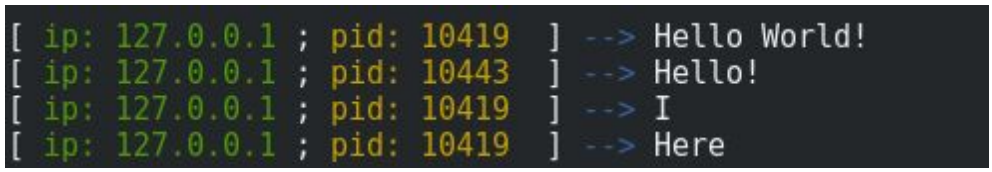

Рисунок 6.17 — Состояние чата 1-ого клиента

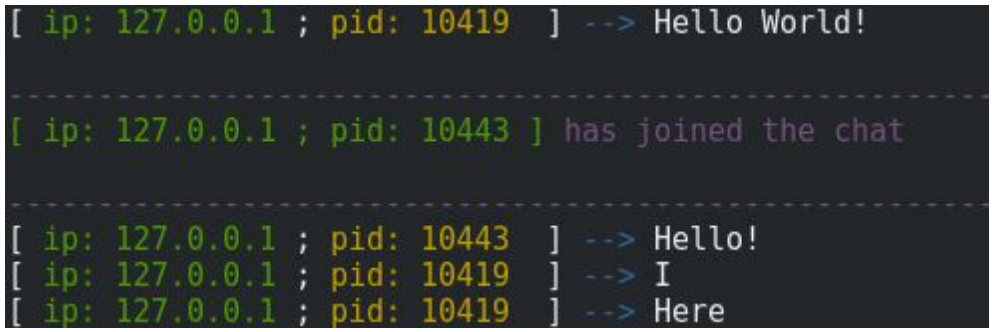

Рисунок 6.18 — Состояние чата 2-ого клиента

Рисунок 6.19 — Ввод сообщения 2-ым клиентом

Рисунок 6.20 — Состояние чата 2-ого клиента

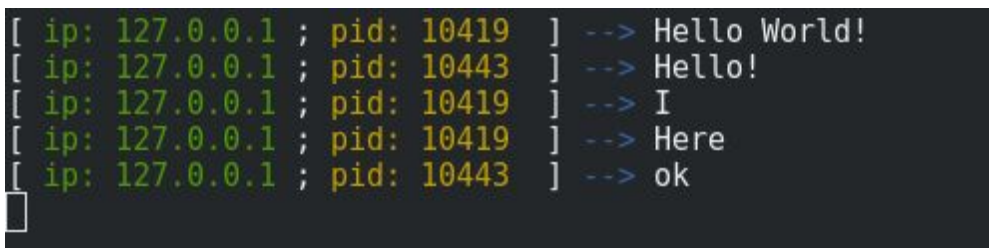

Рисунок 6.21 — Состояние чата 1-ого клиента

7. Список литературы

1. Стивенс У. Р., Феннер Б., Рудофф Э. М. UNIX: разработка сетевых приложений. 3-е изд. — СПБ.:Питер, 2007. — 1039 с.:ил. ISBN 5-94723-991-4