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**](#_td21ualyqni)[**3**](#_td21ualyqni)

[**2. Описание средств разработки**](#_ww5nb37a3qvl)[**3**](#_ww5nb37a3qvl)

[2.1. Программная библиотека сокетов Беркли](#_qz551n1i6pz) [3](#_qz551n1i6pz)

[2.1.1. Заголовочные файлы](#_btn83apnv2eh) [3](#_btn83apnv2eh)

[2.1.2. Основные структуры](#_12dj0f90xkf7) [3](#_12dj0f90xkf7)

[2.1.3. Основные функции](#_13wqw45prwnz) [4](#_13wqw45prwnz)

[2.2. Программная пользовательская библиотека](#_4u7vgn75d1ui) [8](#_4u7vgn75d1ui)

[2.2.1. Заголовочные файлы](#_68fxa4gfgreh) [8](#_68fxa4gfgreh)

[2.2.2. Основные структуры](#_3huwdtal1pve) [8](#_3huwdtal1pve)

[2.2.3. Основные функции](#_ik2vf67wlxo) [10](#_ik2vf67wlxo)

[**3. Описание протоколов**](#_w76hp8d7qe9c)[**14**](#_w76hp8d7qe9c)

[3.1. Internet Protocol (IP)](#_16smephys2gf) [14](#_16smephys2gf)

[3.1.1. IPv4](#_plbx8ulel6rd) [14](#_plbx8ulel6rd)

[3.1.2. IPv6](#_1ji5qvgm2n2i) [16](#_1ji5qvgm2n2i)

[3.2. Transmission Control Protocol (TCP)](#_krg96rlt8gfj) [17](#_krg96rlt8gfj)

[3.3. User Datagram Protocol (UDP)](#_g0sp9j6ne7a3) [17](#_g0sp9j6ne7a3)

[**4. Реализация программы**](#_bd6kz1rhlvbc)[**18**](#_bd6kz1rhlvbc)

[4.1. Клиентская часть](#_myxngtx29kbf) [18](#_myxngtx29kbf)

[4.2. Серверная часть](#_npu50namzbtb) [18](#_npu50namzbtb)

[**5. Текст программы**](#_yj5xl39lsim3)[**19**](#_yj5xl39lsim3)

[5.1. Клиентская часть](#_34s1wrp8n9nc) [19](#_34s1wrp8n9nc)

[5.2. Серверная часть](#_t2z6sr1o1c5) [27](#_t2z6sr1o1c5)

[5.2. Другие файлы](#_oxee3h1k8ih8) [41](#_oxee3h1k8ih8)

[**6. Скан-сессии**](#_ncmomb18r726)[**52**](#_ncmomb18r726)

[**7. Список литературы**](#_9juoio9k2ogc)[**55**](#_9juoio9k2ogc)

# 1. Постановка задачи

Необходимо реализовать сетевое приложение, используя сетевую модель стека сетевых протоколов OSI/ISO.

Требования и ограничения:

* Приложение должно быть отказоустойчивым;
* Язык программирования: С/С++;
* 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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Позиция в октетах** |  | **0** | | | | | | **1** | | | | | | **2** | | | **3** | | |
|  | **Позиция в битах** | **0** | **...** | **3** | **4** | **...** | **7** | **8** | **...** | **11** | **12** | **...** | **15** | **16** | **...** | **23** | **24** | **...** | **31** |
| **0** | **0** | Версия | | | Класс трафика | | | | | | Метка потока | | | | | | | | |
| **4** | **32** | Длина полезной нагрузки | | | | | | | | | Следующий заголовок | | | | | | Число переходов | | |
| **8** | **64** | IP-адрес отправителя | | | | | | | | | | | | | | | | | |
| **12** | **96** |
| **16** | **128** |
| **20** | **160** |
| **24** | **192** | IP-адрес получателя | | | | | | | | | | | | | | | | | |
| **28** | **224** |
| **32** | **256** |
| **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

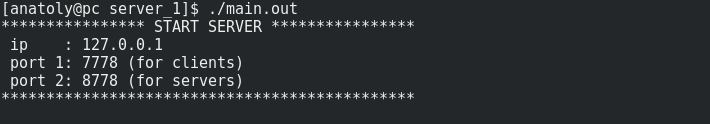


Рисунок 6.1 — Старт 1-ого сервера

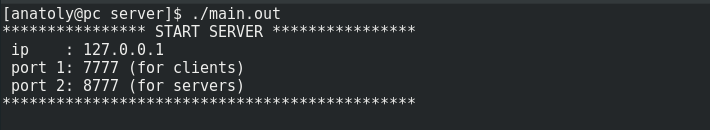


Рисунок 6.2 — Старт 2-ого сервера

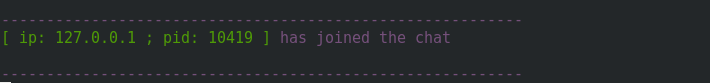


Рисунок 6.3 — Подключение 1-ого клиента (сторона клиента)

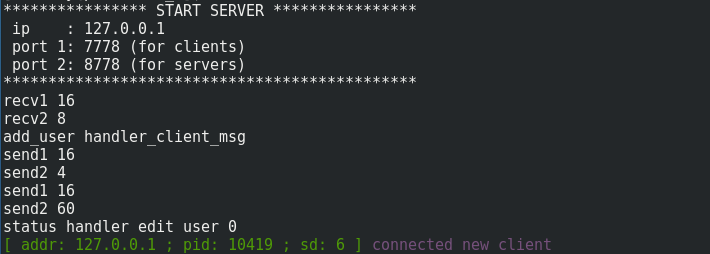


Рисунок 6.4 — Подключение 1-ого клиента (сторона первого сервера)

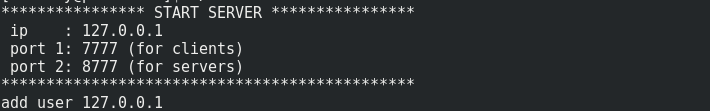


Рисунок 6.5 — Передача другому серверу информации о подключившемся клиенте

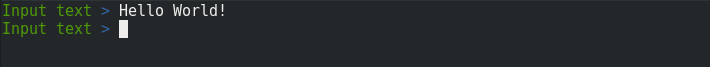


Рисунок 6.6 — Ввод сообщения 1-ым клиентом

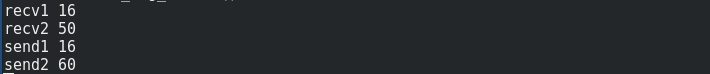


Рисунок 6.7 — Получение сообщения на стороне сервера

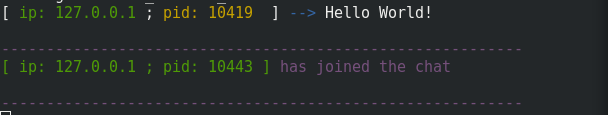


Рисунок 6.8 — Подключение 2-ого клиента (со стороны клиента)



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

info_for_connect_client_2.png

Рисунок 6.10 — Передача другому серверу информации о подключившемся клиенте

client_2_write_1.png

Рисунок 6.11 — Ввод сообщения 1-ым клиентом

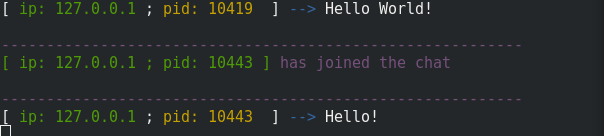


Рисунок 6.12 — Состояние чата 2-ого клиента

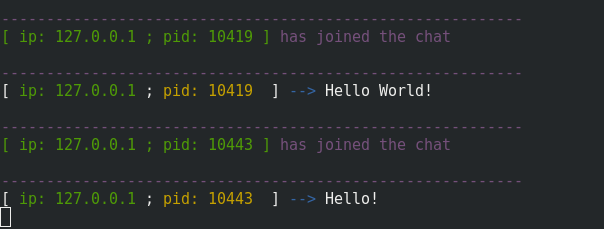


Рисунок 6.13 — Состояние чата 1-ого клиента

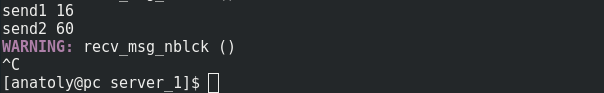


Рисунок 6.14 — Остановка 1-ого сервера

reconnect_cli_1.png

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

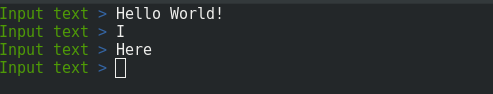


Рисунок 6.16 — Ввод сообщения 1-ым клиентом

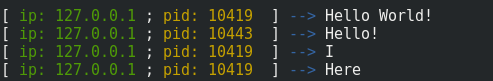


Рисунок 6.17 — Состояние чата 1-ого клиента

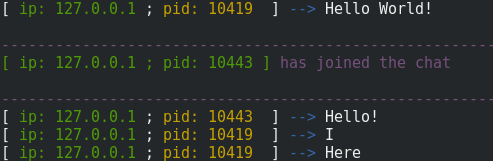


Рисунок 6.18 — Состояние чата 2-ого клиента

client_2_write_2.png

Рисунок 6.19 — Ввод сообщения 2-ым клиентом

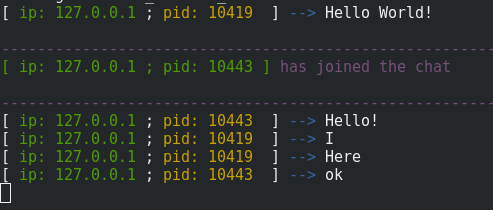


Рисунок 6.20 — Состояние чата 2-ого клиента

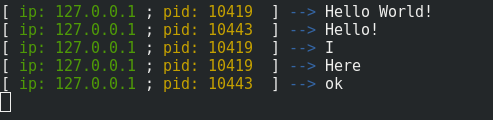


Рисунок 6.21 — Состояние чата 1-ого клиента

# 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