Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Курсовой проект по курсу
«Операционные системы»

Сервер сообщений

Студент	Гришин Алексей Юрьевич
Группа	М8О-208Б-21
Вариант	25
Преподаватель	Соколов Андрей Алексеевич
Оценка	5
Дата	05.01.2023
Подпись	

Москва, 2022.

Постановка задачи

Целью работы является:

- Приобретение практических навыков в использовании знаний, полученных в течении курса
- Проведение исследования в выбранной предметной области

Задание

Необходимо спроектировать и реализовать программный прототип в соответствии с выбранным вариантом. Произвести анализ и сделать вывод на основании данных, полученных при работе программного прототипа.

Вариант 25

Клиент-серверная система для передачи мгновенных сообщений. Базовый функционал должен быть следующим:

- Клиент может присоединиться к серверу, введя логин
- Клиент может отправить сообщение другому клиенту по его логину
- Клиент в реальном времени принимает сообщения от других клиентов

Необходимо предусмотреть возможность хранения истории переписок (на сервере) и поиска по ним. Связь между сервером и клиентом должна быть реализована при помощи pipe'ов

Общие сведения о программе

Описание

Общение клиента с сервером

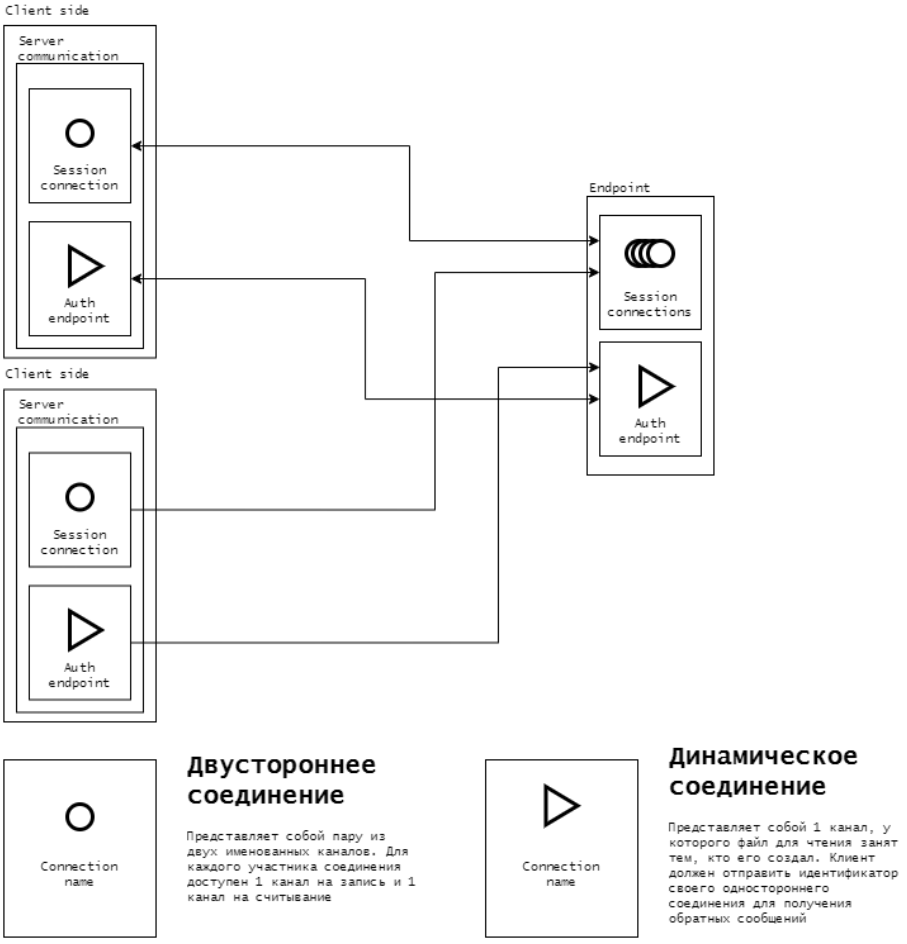
Так как одной из моих задач была реализация общения клиентов с сервером через именованные каналы, то начнем с них. Особенность именованных каналов состоит в том, что он реализует только одностороннее общение. Для реализации двустороннего общения необходимо иметь два именованных канала.

В ходе проектирования системы я выявил 2 вида возможных соединений, которые будут встречаться во время работы. Назовем их динамическим и двусторонним соединениями.

Динамическое соединение заключается в следующем. У двух участников открыто по одному каналу на чтение. Далее, зная название канала второго участника, первый канал

подключается к нему и отправляет сообщение, в котором также хранится название входного канала 1-го участника. После обработки сообщения 2-м участником он подсоединяется к входному каналу 1-го участника и отправляет ему результат.

Двустороннее соединение состоит из двух именованных каналов, названия которых должны быть схожими, так как они созданы для одной цели. Далее участник, создавший двустороннее соединение, отправляет названия созданных каналов 2-му участнику общения. После отправки первый участник открывает 1-й канал на запись, а 2-й – на чтение. Второй участник делает аналогичные действия, но с противоположными каналами.



Собственно, динамическое соединение я использовал для канала авторизации. Перед тем как подключиться к серверу сообщений необходимо идентифицировать подключившегося клиента, чтобы знать от чьего лица приходят сообщения и кому их отправлять. При получении сообщения на авторизацию, сервер проверяет, нет ли уже существующей сессии (о сессии более подробно будет сказано далее) с таким же логином. Если сессия уже существует, сервер отправляет сообщение об ошибке обратно клиенту. Если же никто не заходил под этим логином, то создается сессия и необходимые данные отправляются обратно клиенту.

Под сессией подразумевается совокупность из логина и двустороннего соединения. Через 1-й канал клиент посылает запросы серверу, а через 2-й канал сервер посылает ответы на запросы.

Формат сообщений

При организации общения между клиентом и сервером возникла задача регламентации формата сообщений. Изначально была идея внедрения JSON формата сообщений, однако такой способ был бы слишком трудозатратным в силу того, что проект писался на языке программирования C. Поэтому было принято реализовать свой формат сообщений. Основные требования к формату – минимальная но достаточно полная функциональность, которая позволит отправить сообщения разного рода. Учитывая требование было принято использовать формат “key – value” сообщений наподобие записей в Redis. В качестве ключей и значений выступали строки, так как такой тип данных является самым полным (используя строки можно охватить максимальное множество видов передаваемой информации: числа, булевы значения и т. д.). Идея разделения всех сообщений на поля – уже проверенная временем идея, который используется повсеместно (в таком формате представлены структуры в C, этот формат использует JSON, реляционные базы данных).

С целью программной поддержки такого формата сообщений были реализованы модули `message` и `message/io`. Последний внедряет функционал для записи и считывания сообщений из файлов.

База данных

Приведенный выше формат сообщений значительно повлиял на дальнейшее проектирование программы. Так, при проектировании базы данных хранения сообщений использовались обычные текстовые файлы, каждая строка которых является записью в базе данных. Каждая запись имеет тот же формат, что и у описанных выше сообщений.

Таким образом, мы обеспечиваем гибкость при изменении баз данных (потребовалось добавить новое поле в запись – просто добавляем новую пару при записи сообщений в базу данных).

Также, при проектировании базы данных я опирался на архитектуру такого брокера сообщений, как Kafka, так как он примечателен своим свойством сохранения сообщений. В итоге, база данных является своего рода log файлом и, следовательно, наследует все идеи такого формата файлов (что записано в log файл, уже никак нельзя удалить или изменить). Это также повлияло и на набор методов для работы с базой данных. В ней нет функциональности для изменения записей или чтения. Мы можем только добавлять записи или итерироваться по ним.

Потоки

Хоть именованные каналы и поддерживают неблокирующие операции чтения и записи, их использование лишь усложняет задачу и по опыту работы с ними создает накладные расходы на вычислительные ресурсы, так как для отслеживания получения сообщений необходимо совершать polling, что вытекает в использование бесконечно `while` цикла с операцией чтения из файла, что является крайне трудозатратным занятием.

И, казалось бы, блокирующие операции значительно превосходят неблокирующие. Однако, стоит учитывать, что при блокирующем подходе операции занимают поток исполнения до тех пор, пока не возникнет определенная ситуация. Но мы не знаем точного времени, через которое данное событие произойдет. Более того, мы не можем вообще гарантировать возникновения данной ситуации. Например, если мы будем использовать блокирующие операции для канала авторизации в главном потоке исполнения, то сервер будет простаивать на моментах, когда продолжительное время не приходило соответствующих запросов. Однако, данная проблема решается путем введения многопоточности в программы.

Такое решение является хорошим по той причине, что с одной стороны мы используем блокирующие операции, которые оказывают положительное влияние на грамотное использование ресурсов ЭВМ, с другой стороны блокирующие операции останавливают лишь тот поток, в котором они выполняются, в то время как другие потоки могут спокойно выполняться параллельно.

Таким образом, клиентское приложение имеет 2 потока: один на считывание входных сообщений от сервера, другой – для ввода команд пользователем. Тем самым мы обеспечиваем асинхронность при работе с сервером. Клиенту не надо ждать обработки сообщения A, чтобы отправить сообщение B. Он может отправить 2 сообщения сразу, после чего они выстроятся в очередь на сессионном канале в силу поддержки способа организации данных FIFO именованными каналами (из-за чего, кстати, именованные каналы можно считать своего рода очередями сообщений).

Исходный код

`connection/conneciton.c`

```
#include "connection.h"
#include <stddef.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
```

```
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

char * path_generate(char * name) {
  const char * path_tmp = "/tmp/%s.fifo";
  size_t result_str_size = strlen(name) + strlen(path_tmp);
  char * result_str = (char * ) malloc(result_str_size + 1);

  sprintf(result_str, path_tmp, name);
  return result_str;
}

int get_use_file_mode(ConnectionMode mode) {
  switch (mode) {
  case READ:
    return O_RDONLY;
  case WRITE:
    return O_WRONLY;
  }
}

int connection_create(PipeConnection * connection, char * name,
ConnectionMode mode) {
  char * connection_path;
  int descriptor;

  connection_path = path_generate(name);

  connection -> name = strdup(name);
  connection -> descriptor = -1;
  connection -> use_mode = mode;

  if (mkfifo(connection_path, 0777) == -1)
    return -1;

  free(connection_path);

  return 0;
}

int connection_connect(PipeConnection * connection, char * name,
ConnectionMode mode) {
  char * connection_filepath;
  int descriptor;

  connection_filepath = path_generate(name);
  descriptor = open(connection_filepath, get_use_file_mode(mode));

  connection -> name = strdup(name);
  connection -> descriptor = descriptor;
  connection -> use_mode = mode;
```

```
  if (descriptor == -1)
    return -1;

  return 0;
}

bool connection_exists(char * name) {
  char * path;
  bool result;

  path = path_generate(name);
  result = access(path, F_OK) == 0;

  free(path);
  return result;
}

void connection_close(const PipeConnection * connection) {
  char * connection_filepath;

  connection_filepath = path_generate(connection -> name);

  close(connection -> descriptor);
  remove(connection_filepath);

  free(connection -> name);
  free(connection_filepath);
}

int connection_descriptor(PipeConnection * connection) {
  if (connection -> descriptor == -1) {
    char * filepath = path_generate(connection -> name);

    connection -> descriptor = open(filepath,
get_use_file_mode(connection -> use_mode));
    printf("[ConnectionLib] Generate descriptor from '%s': %d\n",
connection -> name, connection -> descriptor);

    free(filepath);
  }

  return connection -> descriptor;
}
```

connection/conneciton.h

```
#ifndef __CONNECTION_H__
#define __CONNECTION_H__
```

```
#include <stdbool.h>

typedef enum {
  WRITE,
  READ
} ConnectionMode;

typedef struct {
  char * name;
  int descriptor;
  ConnectionMode use_mode;
} PipeConnection;

int connection_create(PipeConnection * connection, char * name,
ConnectionMode mode);
int connection_connect(PipeConnection * connection, char * name,
ConnectionMode mode);
int connection_descriptor(PipeConnection * connection);
bool connection_exists(char * name);
void connection_close(const PipeConnection * connection);

#endif
```

message/io/io.c

```
#include "io.h"
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

int message_write(int descriptor, Message * message) {
  int res;
  char * message_str;

  message_str = message_repr(message);
  res = write(descriptor, message_str, strlen(message_str));
  res = write(descriptor, "\n", 1);

  if (res == -1) {
    printf("MessageWriteError: Can't write message");
    return -1;
  }

  free(message_str);
  return 0;
}
```

```
int message_read(int descriptor, Message * message) {
  char * read_data;
  int read_bytes;
  int buff_size = 256;

  message_create(message);
  read_data = (char * ) malloc(buff_size);
  memset(read_data, '\0', buff_size);
  read_bytes = 0;

  while (1) {
    char read_symb;
    int res;

    while (1) {
      res = read(descriptor, & read_symb, sizeof(read_symb));

      if (res == -1) {
        if (errno == EAGAIN) continue;
        printf("MessageReadError: Can't read data\n");
        free(read_data);
        return -1;
      }

      break;
    }

    if (read_bytes == buff_size) {
      buff_size += 256;
      read_data = realloc(read_data, buff_size);
    }

    read_data[read_bytes] = read_symb;
    read_bytes++;

    if (read_symb == '\0' || read_symb == '\n' || read_symb == EOF)
      break;
  }

  read_data[read_bytes - 1] = '\0';
  int res = message_from_string(message, read_data);

  free(read_data);
  if (res == -1) return -1;
  return 0;
}
```

message/io/io.h

```
#ifndef __MESAGE_IO_H__
```

```
#define __MESSAGE_IO_H__

#include "../message.h"

int message_write(int descriptor, Message *message);
int message_read(int descriptor, Message *message);


#endif
```

message/pair/pair.c

```
#include "pair.h"
#include <stddef.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

void pair_create(Pair * pair, char * key, char * value) {
  pair -> key = strdup(key);
  pair -> value = strdup(value);
}

void pair_delete(Pair * pair) {
  free(pair -> key);
  free(pair -> value);
}

char * pair_repr(Pair * pair) {
  size_t repr_str_len;
  char * repr_str;

  repr_str_len = strlen(pair -> key) + strlen(pair -> value) + 1;
  repr_str = (char * ) malloc(repr_str_len + 1);

  sprintf(repr_str, "%s:%s", pair -> key, pair -> value);
  return repr_str;
}

int pair_from_string(Pair * pair, char * str) {
  char * key;
  char * value;
  int key_length;
  int value_length;
  int str_length;
  int bracket_pos;

  bracket_pos = strcspn(str, ":");
  str_length = strlen(str);
```

```
  key_length = bracket_pos;
  value_length = str_length - key_length - 1;

  if (key_length == str_length) {
    errno = PAIR_INVALID_FORMAT;
    printf("FormatPairStringError: symbol ':' doesn't exist. String:
%s\n", str);
    return -1;
  }

  key = (char * ) malloc(key_length + 1);
  value = (char * ) malloc(value_length + 1);

  memset(key, '\0', key_length + 1);
  memset(value, '\0', value_length + 1);

  memcpy(key, str, key_length);
  memcpy(value, str + key_length + 1, value_length);

  pair -> key = key;
  pair -> value = value;

  return 0;
}
```

message/pair/pair.h

```
#ifndef __MESSAGE_PAIR_H__
#define __MESSAGE_PAIR_H__

#define PAIR_INVALID_FORMAT 1

typedef struct {
  char * key;
  char * value;
} Pair;

void pair_create(Pair * pair, char * key, char * value);
void pair_delete(Pair * pair);
char * pair_repr(Pair * pair);
int pair_from_string(Pair * pair, char * str);

#endif
```

message/storage/storage.c

```
#include "storage.h"
#include "../../support/support.h"
```

```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int message_storage_init(MessageStorage * storage, char * filepath) {
  storage -> add = fopen(filepath, "a+");
  strcpy(storage -> path, filepath);

  if (storage -> add == NULL)
    return -1;

  return 0;
}

void message_storage_clear(MessageStorage * storage) {
  FILE * file = fopen(storage -> path, "w");
  fclose(file);
  fclose(storage -> add);
  storage -> add = fopen(storage -> path, "a+");
}

int message_storage_add(MessageStorage * storage, MessageRecord * record)
{
  int result = fprintf(storage -> add, "\"%s,%s,%s\"\n", record -> from,
record -> to, record -> content);
  fclose(storage -> add);
  storage -> add = fopen(storage -> path, "a+");

  if (result < 0)
    return -1;

  return 0;
}

MessageReader message_storage_select(MessageStorage * storage) {
  MessageReader reader;

  reader.file = fopen(storage -> path, "r");
  reader.have_info = true;

  return reader;
}

int message_reader_next(MessageReader * reader, MessageRecord * record) {
  if (!reader -> have_info) return -1;

  if (feof(reader -> file)) {
    reader -> have_info = false;
    return -1;
  }

  char line[1024];
```

```
  fscan_string(reader -> file, line);

  int i = strcspn(line, ",");

  if (i == strlen(line))
    return -1;

  int j = i + 1 + strcspn(line + i + 1, ",");

  if (j == strlen(line))
    return -1;

  line[i] = '\0';
  line[j] = '\0';

  strcpy(record -> from, line);
  strcpy(record -> to, line + i + 1);
  strcpy(record -> content, line + j + 1);

  return 0;
}
```

message/storage/storage.h

```
#ifndef __MESSAGE_STORAGE_H__
#define __MESSAGE_STORAGE_H__

#include <stdio.h>
#include <stdbool.h>

#define PATH_SIZE 100
#define RECORD_STR_FIELD_SIZE 100
#define RECORD_CONTENT_SIZE 256

typedef struct {
  char path[PATH_SIZE];
  FILE * add;
} MessageStorage;

typedef struct {
  char from[RECORD_STR_FIELD_SIZE];
  char to[RECORD_STR_FIELD_SIZE];
  char content[RECORD_CONTENT_SIZE];
} MessageRecord;

typedef struct {
  FILE * file;
  bool have_info;
} MessageReader;
```

```
int message_storage_init(MessageStorage * storage, char * filepath);
int message_storage_add(MessageStorage * storage, MessageRecord *
record);
void message_storage_clear(MessageStorage * storage);
MessageReader message_storage_select(MessageStorage * storage);
int message_reader_next(MessageReader * reader, MessageRecord * record);

#endif
```

message/message.c

```
#include "message.h"
#include <stddef.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void message_create(Message * message) {
  message -> pairs = NULL;
  message -> pairs_amount = 0;
}

void message_add_pair(Message * message, char * key, char * value) {
  int i = message -> pairs_amount;

  if (i == 0)
    message -> pairs = (Pair * ) malloc(sizeof(Pair));
  else
    message -> pairs = realloc(message -> pairs, sizeof(Pair) * (i + 1));

  pair_create( & (message -> pairs[i]), key, value);
  message -> pairs_amount++;
}

char * message_get(Message * message, char * key) {
  for (int i = 0; i < message -> pairs_amount; i++) {
    if (strcmp(key, message -> pairs[i].key) == 0)
      return message -> pairs[i].value;
  }

  return NULL;
}

void message_delete(Message * message) {
  for (int i = 0; i < message -> pairs_amount; i++)
    pair_delete( & (message -> pairs[i]));

  free(message -> pairs);
}
```

```
char * message_repr(Message * message) {
  char * result_str = strdup("");

  for (int i = 0; i < message -> pairs_amount; i++) {
    char * pair_str = pair_repr( & (message -> pairs[i]));
    size_t new_str_len = strlen(result_str) + strlen(pair_str) + 2;

    result_str = realloc(result_str, new_str_len);
    strcat(result_str, pair_str);
    strcat(result_str, ";");

    free(pair_str);
  }

  return result_str;
}

bool message_empty(Message * message) {
  return message -> pairs_amount == 0;
}

int message_from_string(Message * message, char * str) {
  int pos = 0;
  size_t str_len = strlen(str);

  message_create(message);

  while (pos < str_len) {
    int pair_str_len;
    char * substr;
    Pair pair;

    pair_str_len = strcspn(str + pos, ";");

    if (pair_str_len == str_len - pos) {
      errno = MESSAGE_INVALID_FORMAT;
      printf("FormatMessageStringError: symbol ';' doesn't exist. String:
%s\n", str);
      return -1;
    }

    substr = (char * ) malloc(pair_str_len + 1);
    memset(substr, '\0', pair_str_len + 1);
    memcpy(substr, str + pos, pair_str_len);
    pair_from_string( & pair, substr);
    message_add_pair(message, pair.key, pair.value);

    pos += pair_str_len + 1;

    free(substr);
    pair_delete( & pair);
```

```
  }

  return 0;
}
```

message/message.h

```
#ifndef __MESSAGE_H__
#define __MESSAGE_H__

#include "./pair/pair.h"
#include <stdbool.h>

#define MESSAGE_INVALID_FORMAT 2;

typedef struct {
  Pair * pairs;
  int pairs_amount;
}
Message;

void message_create(Message * message);
int message_from_string(Message * message, char * str);
void message_delete(Message * message);

void message_add_pair(Message * message, char * key, char * value);
bool message_empty(Message * message);
char * message_get(Message * message, char * key);
char * message_repr(Message * message);

#endif
```

session/storage/storage.c

```
#include "storage.h"
#include <stddef.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define FULL_LOGIN_SIZE 100

void __login_full_name(char * result, char * server_id, char * login) {
  sprintf(result, "%s_%s", server_id, login);
}

void session_storage_init(SessionStorage * storage, char * server_id) {
  storage -> sessions = NULL;
```

```
  storage -> sessions_amount = 0;
  strcpy(storage -> server_id, server_id);
}

bool session_storage_exists(SessionStorage * storage, char * login) {
  char full_login[FULL_LOGIN_SIZE];

  __login_full_name(full_login, storage -> server_id, login);

  for (int i = 0; i < storage -> sessions_amount; i++)
    if (strcmp(storage -> sessions[i].login, full_login) == 0)
      return true;

  return false;
}

ClientSession * session_storage_create(SessionStorage * storage, char *
login) {
  char full_login[FULL_LOGIN_SIZE];

  __login_full_name(full_login, storage -> server_id, login);

  if (storage -> sessions_amount == 0)
    storage -> sessions = (ClientSession * )
malloc(sizeof(ClientSession));
  else
    storage -> sessions = (ClientSession * ) realloc(storage -> sessions,
sizeof(ClientSession) * (storage -> sessions_amount + 1));

  session_init(storage -> sessions + storage -> sessions_amount,
full_login);
  storage -> sessions_amount++;

  return storage -> sessions + (storage -> sessions_amount - 1);
}

ClientSession * session_storage_get(SessionStorage * storage, char *
login) {
  char full_login[FULL_LOGIN_SIZE];

  __login_full_name(full_login, storage -> server_id, login);

  for (int i = 0; i < storage -> sessions_amount; i++)
    if (strcmp(storage -> sessions[i].login, full_login) == 0)
      return storage -> sessions + i;

  return NULL;
}

void session_storage_remove(SessionStorage * storage, char * login) {
  char full_login[FULL_LOGIN_SIZE];
```

```
  __login_full_name(full_login, storage -> server_id, login);

  for (int i = 0; i < storage -> sessions_amount; i++) {
    if (strcmp(storage -> sessions[i].login, full_login) == 0) {
      ClientSession * session = & (storage -> sessions[i]);
      ClientSession * arr;

      session_close(session);

      if (storage -> sessions_amount > 1) {
        arr = (ClientSession * ) malloc(sizeof(ClientSession) * (storage
-> sessions_amount - 1));
        memcpy(arr, storage -> sessions, sizeof(ClientSession) * i);
        memcpy(arr + i, storage -> sessions + i + 1,
sizeof(ClientSession) * (storage -> sessions_amount - i - 1));
      } else arr = NULL;

      free(storage -> sessions);

      storage -> sessions_amount--;
      storage -> sessions = arr;

      break;
    }
  }
}

void session_storage_delete(SessionStorage * storage) {
  for (int i = 0; i < storage -> sessions_amount; i++)
    session_close(storage -> sessions + i);

  free(storage -> sessions);
}

SessionIterator session_storage_iter(SessionStorage * storage) {
  SessionIterator iterator = {
    .cursor = 0,
    .storage = storage
  };
  return iterator;
}

int session_storage_next(SessionIterator * iterator, SessionPair * pair)
{
  if (iterator -> cursor >= iterator -> storage -> sessions_amount)
    return -1;

  ClientSession * session = iterator -> storage -> sessions + iterator ->
cursor;
  pair -> login = session -> login + (strlen(iterator -> storage ->
server_id) + 1);
  pair -> session = session;
```

```
  iterator -> cursor++;

  return 0;
}
```

session/storage/storage.h

```
#ifndef __SESSION_STORAGE_H__
#define __SESSION_STORAGE_H__

#include "../session.h"
#include <stdbool.h>

#define SERVER_STR_SIZE 50

typedef struct {
  ClientSession * sessions;
  int sessions_amount;
  char server_id[SERVER_STR_SIZE];
} SessionStorage;

typedef struct {
  int cursor;
  SessionStorage * storage;
} SessionIterator;

typedef struct {
  char * login;
  ClientSession * session;
} SessionPair;

void session_storage_init(SessionStorage * storage, char * server_id);
void session_storage_delete(SessionStorage * storage);

bool session_storage_exists(SessionStorage * storage, char * login);
ClientSession * session_storage_create(SessionStorage * storage, char *
login);
ClientSession * session_storage_get(SessionStorage * storage, char *
login);
void session_storage_remove(SessionStorage * storage, char * login);

SessionIterator session_storage_iter(SessionStorage * storage);
int session_storage_next(SessionIterator * iterator, SessionPair * pair);

#endif
```

session/session.c

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

#define SESSION_CONECTION_NAME_SIZE 100

void __get_connection_name(char * result, char * login, char * type) {
  sprintf(result, "%s_%s", login, type);
}

int session_init(ClientSession * session, char * login) {
  char input_name[SESSION_CONECTION_NAME_SIZE];
  char output_name[SESSION_CONECTION_NAME_SIZE];

  __get_connection_name(input_name, login, "input");
  __get_connection_name(output_name, login, "output");

  int res1 = connection_create( & (session -> input), input_name, READ);
  int res2 = connection_create( & (session -> output), output_name,
WRITE);
  strcpy(session -> login, login);

  if (res1 == -1 || res2 == -1)
    return -1;

  return 0;
}

int session_restore(ClientSession * session) {
  char output_name[SESSION_CONECTION_NAME_SIZE];

  if (session -> output.descriptor == -1) {
    __get_connection_name(output_name, session -> login, "output");
    int res = connection_connect( & (session -> output), output_name,
WRITE);
    return res;
  }

  return 0;
}

void session_close(ClientSession * session) {
  connection_close( & (session -> input));
  connection_close( & (session -> output));
  strcpy(session -> login, "");
}

PipeConnection * session_input(ClientSession * session) {
  return & (session -> input);
}
```

```
PipeConnection * session_output(ClientSession * session) {
  return & (session -> output);
}

char * session_login(ClientSession * session) {
  return session -> login;
}
```

session/session.h

```
#ifndef __SESSION_H__
#define __SESSION_H__

#include "../connection/connection.h"

#define LOGIN_STR_SIZE 50

typedef struct {
  char login[LOGIN_STR_SIZE];
  PipeConnection input;
  PipeConnection output;
} ClientSession;

int session_init(ClientSession * session, char * login);
int session_restore(ClientSession * session);
void session_close(ClientSession * session);

PipeConnection * session_input(ClientSession * session);
PipeConnection * session_output(ClientSession * session);
char * session_login(ClientSession * session);

#endif
```

support/support.c

```
#include "support.h"
#include <string.h>
#include <stdlib.h>

void fscan_string(FILE * stream, char * buff) {
  size_t read_symbols = 0;

  fscanf(stream, "%s", buff);
  read_symbols = strlen(buff);

  if (buff[0] == '"') {
    memcpy(buff, buff + 1, read_symbols);
```

```
    read_symbols--;

    while (buff[read_symbols - 1] != '"') {
      buff[read_symbols] = ' ';
      fscanf(stream, "%s", buff + read_symbols + 1);
      read_symbols = strlen(buff);
    }

    buff[read_symbols - 1] = '\0';
  }
}
```

support/support.h

```
#ifndef __SUPPORT_H__
#define __SUPPORT_H__

#include <stdio.h>

void fscan_string(FILE * stream, char * buff);

#endif
```

client.c

```
#include "connection/connection.h"
#include "message/message.h"
#include "message/io/io.h"
#include "support/support.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <pthread.h>
#include <signal.h>

PipeConnection callback;
PipeConnection session_input;
PipeConnection session_output;
char * login;

pthread_t handle_server_reply_thread;

void close_session() {
  pthread_cancel(handle_server_reply_thread);
  pthread_join(handle_server_reply_thread, NULL);
  connection_close( & session_input);
  connection_close( & session_output);
```

```
}

void interrupt_handler(int sig) {
  exit(0);
}

char * server_path(char * server_id, char * conn_local_name) {
  size_t fullname_len = strlen(server_id) + strlen(conn_local_name) + 2;
  char * fullname = (char * ) malloc(fullname_len);

  sprintf(fullname, "%s_%s", server_id, conn_local_name);
  return fullname;
}

void connect_to_server(char * server_id) {
  PipeConnection auth;
  Message message;
  char * auth_name;

  auth_name = server_path(server_id, "auth");
  message_create( & message);

  if (!connection_exists(auth_name)) {
    printf("Server doesn't exist\n");
    exit(2);
  }

  if (connection_connect( & auth, auth_name, WRITE) == -1) {
    printf("Server unactive\n");
    exit(2);
  }

  message_add_pair( & message, "callback", callback.name);
  message_add_pair( & message, "login", login);

  if (message_write(connection_descriptor( & auth), & message) == -1) {
    printf("[Client] Can't write data to server: %s", strerror(errno));
  }

  printf("[Client] Listen callback...\n");
  while (1) {
    Message recv_msg;
    message_read(connection_descriptor( & callback), & recv_msg);

    if (!message_empty( & recv_msg)) {
      char * type = message_get( & recv_msg, "type");

      if (strcmp(type, "ok") == 0) {
        printf("[Client] Connected!\n");
        printf("[Client] Connecting to input/output pipes...\n");

        printf("\t1.Input ");
```

```
        connection_connect( & session_input, message_get( & recv_msg,
"input"), WRITE);
        printf("[OK]\n");

        printf("\t2.Output ");
        connection_connect( & session_output, message_get( & recv_msg,
"output"), READ);
        printf("[OK]\n");
      }

      if (strcmp(type, "error") == 0) {
        printf("[Client] Error: %s\n", message_get( & recv_msg,
"message"));
        exit(2);
      }

      break;
    }

    message_delete( & recv_msg);
  }

  message_delete( & message);
  free(auth_name);
  connection_close( & callback);
}

char * get_server_id(int argc, char * argv[]) {
  if (argc < 2) {
    printf("Server id not specified\n");
    exit(1);
  }

  return argv[1];
}

char * get_login(int argc, char * argv[]) {
  if (argc < 3) {
    printf("Login not specified\n");
    exit(1);
  }

  return argv[2];
}

void init_callback() {
  char callback_name[256];
  int client_id = 0;

  while (1) {
    sprintf(callback_name, "client%d_callback", client_id);
```

```
    if (!connection_exists(callback_name))
      break;

    client_id++;
  }

  if (connection_create( & callback, callback_name, READ) == -1) {
    printf("[Client] Can't create callback connection: %s\n",
strerror(errno));
    exit(1);
  }
}

void * handle_server_reply(void * vargp) {
  printf("[Client] Waiting messages...\n");

  while (1) {
    Message reply;

    if (message_read(connection_descriptor( & session_output), & reply)
== -1) {
      printf("[Client] Error: read from session input pipe\n");
      exit(1);
    }

    if (!message_empty( & reply)) {
      char * msg_type = message_get( & reply, "type");

      if (strcmp(msg_type, "incoming_message") == 0)
        printf("[Client] '%s': %s\n", message_get( & reply, "sender"),
message_get( & reply, "content"));

      else if (strcmp(msg_type, "history") == 0) {
        for (int i = 0; i < reply.pairs_amount; i++) {
          Pair * pair = reply.pairs + i;

          if (strcmp(pair -> key, "_") == 0) {
            char * from;
            char * to;
            char * content;

            int pos1 = strcspn(pair -> value, ",");
            int pos2 = pos1 + 1 + strcspn(pair -> value + pos1 + 1, ",");

            pair -> value[pos1] = '\0';
            pair -> value[pos2] = '\0';

            from = pair -> value;
            to = pair -> value + (pos1 + 1);
            content = pair -> value + (pos2 + 1);

            printf("%s\t%s\t%s\n", from, to, content);
```

```
          }
        }
      }
    }

    message_delete( & reply);
  }
}

void send_message() {
  Message request;
  char target[256];
  char content[256];

  scanf("%s", target);
  fscan_string(stdin, content);

  message_create( & request);
  message_add_pair( & request, "type", "send");
  message_add_pair( & request, "target", target);
  message_add_pair( & request, "content", content);

  if (message_write(connection_descriptor( & session_input), & request)
== -1)
    printf("[Client] Error: can't write to server input pipe\n");

  message_delete( & request);
}

void parse_filter_parameter(char * value, char * key, Message * msg) {
  if (strcmp(value, "ALL") == 0) return;
  if (strcmp(value, "ME") == 0) {
    message_add_pair(msg, key, login);
    return;
  }
  message_add_pair(msg, key, value);
}

void get_history() {
  Message request;
  char target_filter[256];
  char sender_filter[256];

  scanf("%s %s", sender_filter, target_filter);

  message_create( & request);
  message_add_pair( & request, "type", "history");

  parse_filter_parameter(target_filter, "target", & request);
  parse_filter_parameter(sender_filter, "sender", & request);

  if (message_write(connection_descriptor( & session_input), & request)
```

```
== -1) {
    printf("[Client] Error: can't write to server input pipe\n");
    return;
  }

  message_delete( & request);
}

void exit_request() {
  Message request;

  message_create( & request);
  message_add_pair( & request, "type", "exit");

  printf("[Client] Send exit request...\n");

  if (message_write(connection_descriptor( & session_input), & request)
== -1) {
    printf("[Client] Error: can't write to server input pipe\n");
    return;
  }

  message_delete( & request);
  exit(0);
}

int main(int argc, char * argv[]) {
  char * server_id = get_server_id(argc, argv);
  login = get_login(argc, argv);

  signal(SIGINT, interrupt_handler);

  init_callback();
  connect_to_server(server_id);
  atexit(close_session);

  pthread_create( & handle_server_reply_thread, NULL,
handle_server_reply, NULL);

  while (1) {
    char cmd_type[256];

    printf("> ");
    scanf("%s", cmd_type);

    if (strcmp(cmd_type, "send") == 0) send_message();
    else if (strcmp(cmd_type, "history") == 0) get_history();
    else if (strcmp(cmd_type, "exit") == 0) exit_request();
    else printf("[Client] Unknown command\n");
  }

  pthread_exit(NULL);
```

```
  return 0;
}
```

server.c

```
#include "./connection/connection.h"
#include "message/message.h"
#include "message/io/io.h"
#include "session/storage/storage.h"
#include "message/storage/storage.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <dirent.h>
#include <stdbool.h>
#include <pthread.h>
#include <signal.h>

PipeConnection auth;
MessageStorage storage;
MessageStorage unwatched;
SessionStorage sessions;

pthread_t auth_thread;

void close_auth() {
  printf("[Server] Closing auth pipe...\n");
  pthread_cancel(auth_thread);
  pthread_join(auth_thread, NULL);
  connection_close( & auth);
}

void close_sessions() {
  printf("[Server] Closing sessions...\n");
  session_storage_delete( & sessions);
}

void interrupt_handler(int sig) {
  exit(0);
}

char * get_server_id(int argc, char * argv[]) {
  if (argc < 2) {
    perror("Please, write server name as first argument\n");
    exit(1);
  }

  return argv[1];
```

```
}

char * server_path(char * server_id, char * conn_local_name) {
  size_t fullname_len = strlen(server_id) + strlen(conn_local_name) + 2;
  char * fullname = (char * ) malloc(fullname_len);

  sprintf(fullname, "%s_%s", server_id, conn_local_name);
  return fullname;
}

void launch_server(char * server_id) {
  printf("[Server] Launching...\n");
  char * auth_connection_name = server_path(server_id, "auth");

  printf("\t1. Auth connection ");
  if (connection_exists(auth_connection_name)) {
    printf("[Error]\n");
    exit(1);
  } else {
    printf(": Create ");
    if (connection_create( & auth, auth_connection_name, READ) == -1) {
      printf("[Error]\n");
      exit(1);
    }
  }
  printf("[OK]\n");
  atexit(close_auth);

  printf("\t2. Session storage [OK]\n");
  session_storage_init( & sessions, server_id);
  atexit(close_sessions);

  printf("\t3. Message storage ");
  if (message_storage_init( & storage, "messages.db") == -1) {
    printf("[Error]\n");
    exit(1);
  }
  printf("[OK]\n");

  printf("\t4. Unwatched messages storage ");
  if (message_storage_init( & unwatched, "unwatched.db") == -1) {
    printf("[Error]\n");
    exit(1);
  }
  printf("[OK]");

  free(auth_connection_name);
}

void send_message_history(char * sender, char * target_filter, char *
sender_filter) {
  Message reply;
```

```
  ClientSession * sender_session;
  MessageReader reader;
  MessageRecord record;

  printf("[Server] Collecting message history of user '%s'...\n",
sender);

  message_create( & reply);
  sender_session = session_storage_get( & sessions, sender);
  reader = message_storage_select( & storage);

  if (sender_session == NULL) {
    message_delete( & reply);
    printf("[Server] User '%s' offline\n", sender);
    return;
  }

  if (session_restore(sender_session) == -1) {
    message_delete( & reply);
    printf("[Server] Can't restore connection\n");
    return;
  }

  message_add_pair( & reply, "type", "history");

  while (message_reader_next( & reader, & record) != -1) {
    char * repr;
    size_t repr_size;

    if (sender_filter != NULL)
      if (strstr(record.from, sender_filter) == NULL)
        continue;

    if (target_filter != NULL)
      if (strstr(record.to, target_filter) == NULL)
        continue;

    repr_size = strlen(record.from) + strlen(record.to) +
strlen(record.content) + 2;
    repr = (char * ) malloc(repr_size);

    sprintf(repr, "%s,%s,%s", record.from, record.to, record.content);

    message_add_pair( & reply, "_", repr);

    free(repr);
  }

  if (message_write(connection_descriptor( & (sender_session -> output)),
& reply) == -1) {
    message_delete( & reply);
    printf("[Server] Can't write to output pipe\n");
```

```
    return;
  }

  printf("[Server] Sent!\n");
  message_delete( & reply);
}

void save_message(char * sender, char * target, char * content) {
  printf("[Server] Save message...\n");

  MessageRecord record;
  strcpy(record.from, sender);
  strcpy(record.to, target);
  strcpy(record.content, content);

  if (message_storage_add( & storage, & record) == -1) {
    printf("[Server] Error: Can't add record to db. Reason: %s\n",
strerror(errno));
    exit(1);
  }

  printf("[Server] Message saved\n");
}

void send_message(char * sender, char * target, char * content) {
  ClientSession * target_session;
  Message target_message;

  message_create( & target_message);
  message_add_pair( & target_message, "type", "incoming_message");
  message_add_pair( & target_message, "sender", sender);
  message_add_pair( & target_message, "target", target);
  message_add_pair( & target_message, "content", content);

  printf("[Server] Send message to '%s'\n", target);

  if (session_storage_exists( & sessions, target)) {
    target_session = session_storage_get( & sessions, target);

    if (session_restore(target_session) != -1) {
      if
(message_write(connection_descriptor(session_output(target_session)), &
target_message) == -1) {
        printf("[Server] Can't write to output pipe\n");
        exit(1);
      }
    } else {
      printf("[Server] Can't restore connection\n");
      exit(1);
    }
  } else {
    printf("[Server] User '%s' offline\n", target);
```

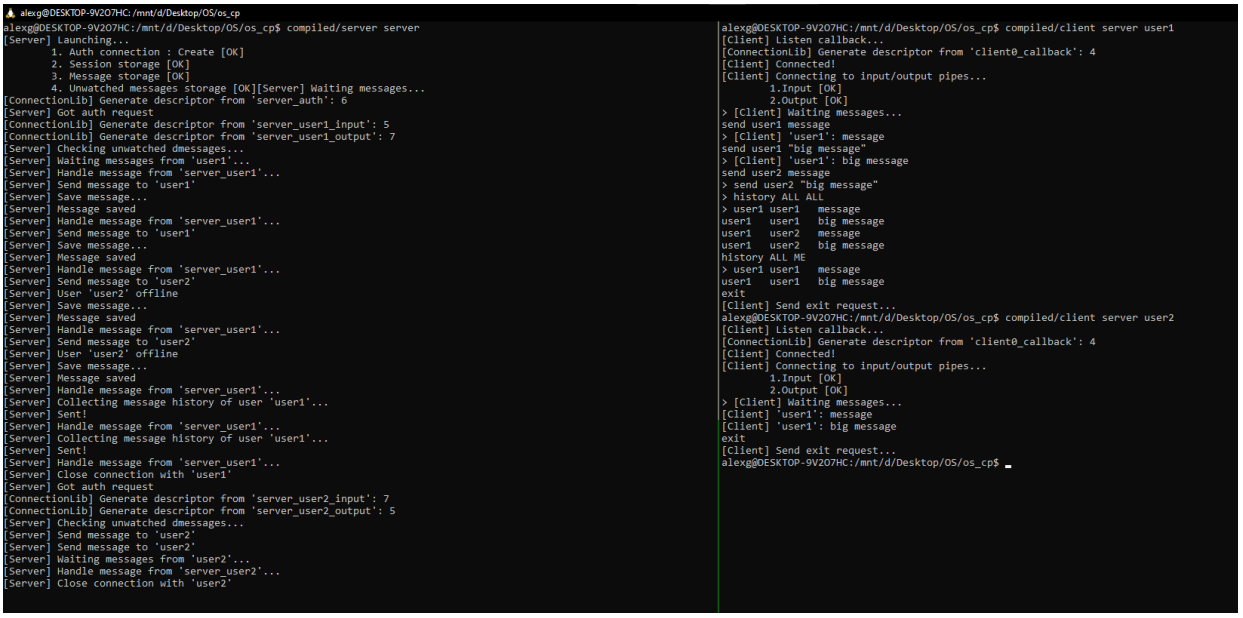
```
    MessageRecord record;
    strcpy(record.from, sender);
    strcpy(record.to, target);
    strcpy(record.content, content);
    message_storage_add( & unwatched, & record);
  }

  message_delete( & target_message);
}

void close_session(char * sender) {
  printf("[Server] Close connection with '%s'\n", sender);
  session_storage_remove( & sessions, sender);
}

void check_unwatched_messages(char * login, int output) {
  MessageReader reader;
  MessageStorage buffer;
  MessageRecord record;

  reader = message_storage_select( & unwatched);
  message_storage_init( & buffer, "buffer.db");

  while (message_reader_next( & reader, & record) != -1) {
    if (strcmp(record.to, login) == 0)
      send_message(record.from, record.to, record.content);
    else
      message_storage_add( & buffer, & record);
  }

  reader = message_storage_select( & buffer);
  message_storage_clear( & unwatched);

  while (message_reader_next( & reader, & record) != -1)
    message_storage_add( & unwatched, & record);

  message_storage_clear( & buffer);
}

void * handle_user_requests(void * vargp) {
  char * login = (char * ) vargp;
  ClientSession * session = session_storage_get( & sessions, login);
  int input_desc = connection_descriptor(session_input(session));
  int output_desc = connection_descriptor(session_output(session));

  printf("[Server] Checking unwatched dmessages...\n");

  check_unwatched_messages(login, output_desc);

  printf("[Server] Waiting messages from '%s'...\n", login);

  while (1) {
```

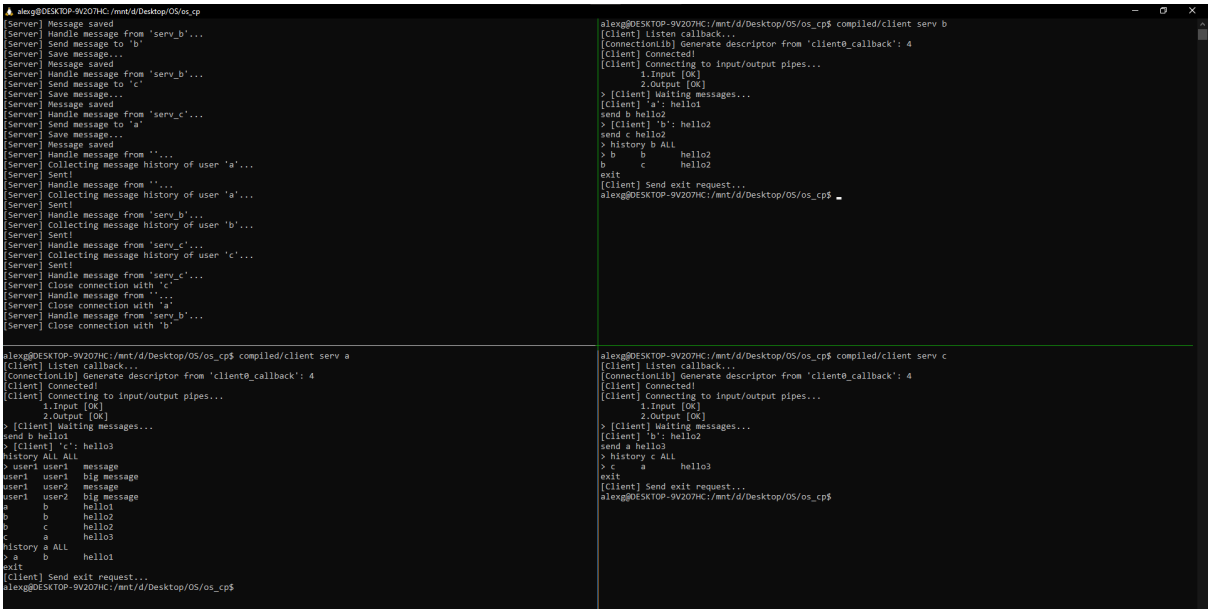
```
    Message request;

    if (message_read(input_desc, & request) == -1) {
      printf("[Server] Can't read message from '%s' input pipe. Reason:
%s\n", login, strerror(errno));
      printf("\tDescriptor: %d\n",
connection_descriptor(session_input(session)));
      exit(1);
    }

    if (!message_empty( & request)) {
      printf("[Server] Handle message from '%s'...\n",
session_login(session));
      char * type = message_get( & request, "type");

      if (strcmp(type, "send") == 0) {
        char * target = message_get( & request, "target");
        char * content = message_get( & request, "content");

        send_message(login, target, content);
        save_message(login, target, content);
      } else if (strcmp(type, "history") == 0) {
        char * target_filter = message_get( & request, "target");
        char * sender_filter = message_get( & request, "sender");
        send_message_history(login, target_filter, sender_filter);
      } else if (strcmp(type, "exit") == 0) {
        close_session(login);
        break;
      }
    }

    message_delete( & request);
  }

  free(login);
  return NULL;
}

void * auth_users(void * vargp) {
  printf("[Server] Waiting messages...\n");

  while (1) {
    Message message;

    if (message_read(connection_descriptor( & auth), & message) == -1) {
      printf("[Server] Error: can't read from auth pipe. Reason: %s\n",
strerror(errno));
      break;
    }

    if (!message_empty( & message)) {
      printf("[Server] Got auth request\n");
```

```
      PipeConnection callback;
      Message reply;
      char * callback_name;
      char * login;

      callback_name = message_get( & message, "callback");
      login = message_get( & message, "login");

      if (connection_connect( & callback, callback_name, WRITE) == -1) {
        printf("[Server] Error: can't connect to callback pipe. Reason:
%s\n", strerror(errno));
        break;
      }

      message_create( & reply);

      if (session_storage_exists( & sessions, login)) {
        message_add_pair( & reply, "type", "error");
        message_add_pair( & reply, "message", "login taken");
      } else {
        ClientSession * new_session = session_storage_create( & sessions,
login);
        message_add_pair( & reply, "type", "ok");
        message_add_pair( & reply, "input", new_session -> input.name);
        message_add_pair( & reply, "output", new_session -> output.name);

        pthread_t handle_thread;

        pthread_create( & handle_thread, NULL, handle_user_requests,
strdup(login));
      }

      message_write(connection_descriptor( & callback), & reply);
      connection_close( & callback);
      message_delete( & reply);
    }

    message_delete( & message);
  }

  return NULL;
}

int main(int argc, char * argv[]) {
  signal(SIGINT, interrupt_handler);

  char * server_id = get_server_id(argc, argv);
  Message message;

  launch_server(server_id);
```

```
  pthread_create( & auth_thread, NULL, auth_users, NULL);

  pthread_exit(NULL);
  return 0;
}
```

Пример работы





Вывод

В ходе выполнения данного курсового проекта я:

- использовал темы межпроцессорного взаимодействия, потоков в операционных системах, системных вызовах, которые изучал на протяжении курса “операционные системы”
- приобрел опыт в написании клиент серверных приложений на языке программирования C
- приобрел опыт в использовании именованных каналов. Изучил их особенности и отличия реализации в Unix системах и Windows.

Именованные каналы, на мой взгляд, отлично подходят для паттернов общения pipeline в программах (например, они бы отлично вписались в сервисах с нейронными сетями, так как все они строятся по одному шаблону – клиенты присылают запросы, все они выстраиваются в очередь, сервер постепенно обрабатывает сообщения).

Однако, для организации стандартного для сетевого взаимодействия Request – Reply они вряд ли подойдут. Здесь лидирующую позицию занимает такой способ межпроцессного взаимодействия, как сокеты, так как изначально они и были созданы для этих целей.

Отсутствие функциональности для создания именованных очередей в рамках сетевого общения является большим недостатком систем Unix по моему мнению. Примечательной становится реализация именованных очередей в операционной системе Windows, так как они поддерживают создание каналов в shared file system.