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**

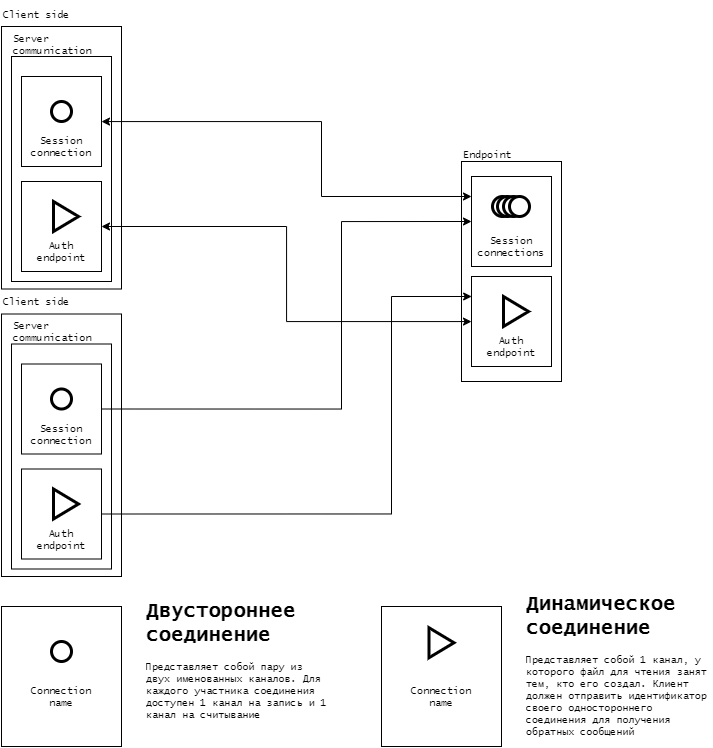
#### **Общение клиента с сервером**

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 |
| --- |

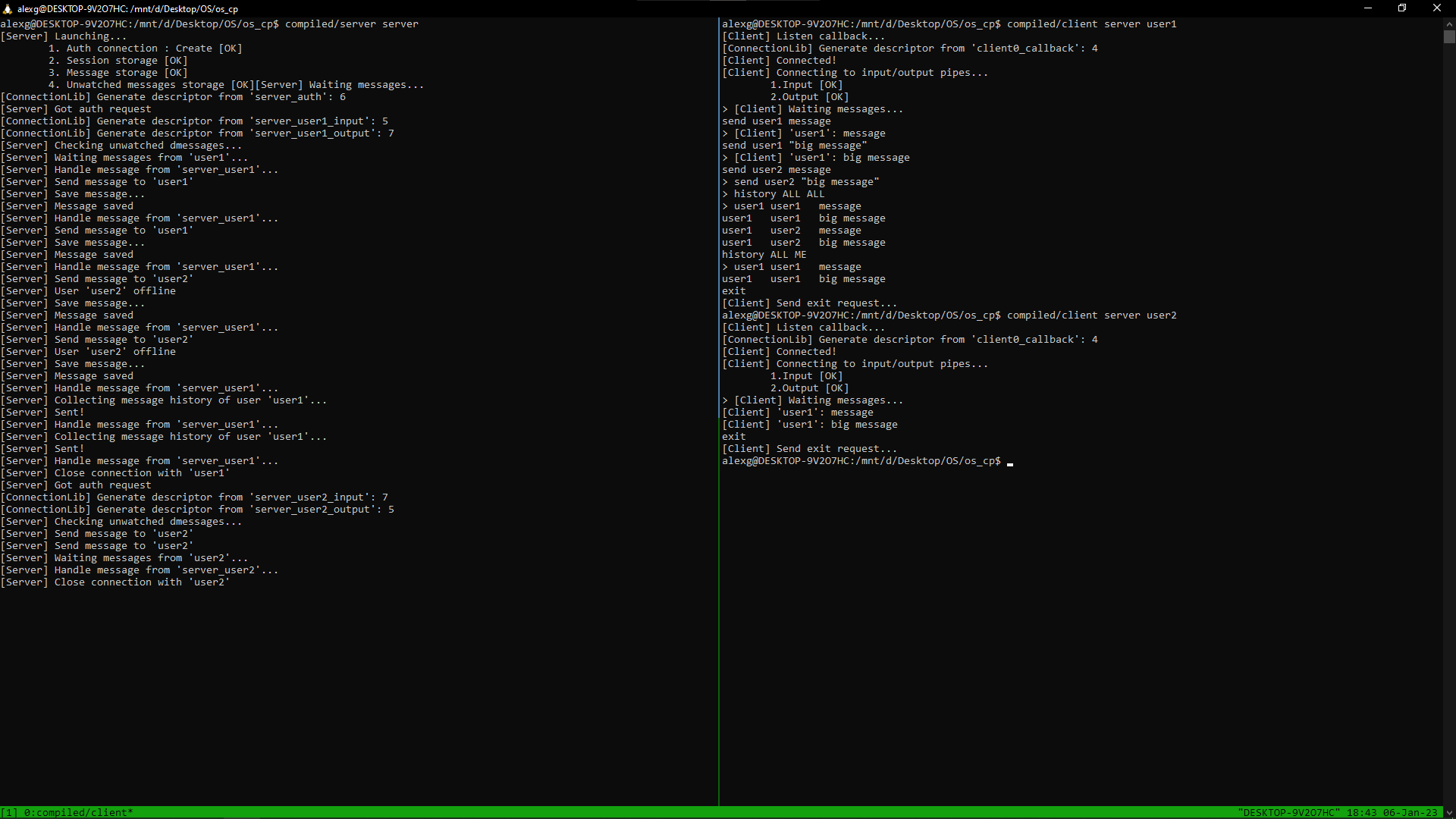
#### **client.c**

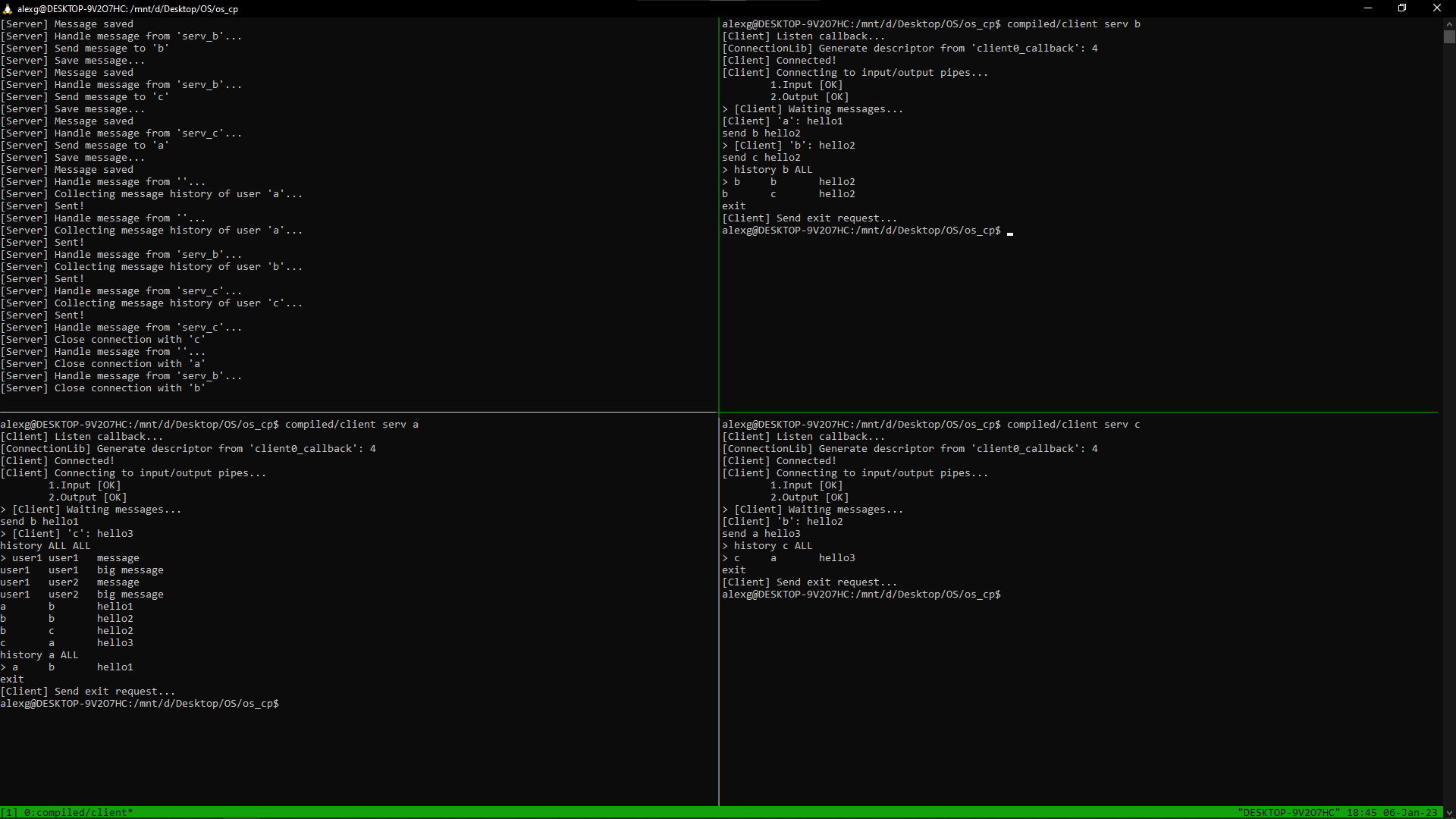
| #include "connection/connection.h"  #include "message/message.h"  #include "message/io/io.h"  #include "support/support.h"  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <errno.h>  #include <pthread.h>  #include <signal.h>  PipeConnection callback;  PipeConnection session\_input;  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.