Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №6-8 по курсу «Операционные системы»

Студент: О. В. Бабин
Преподаватель: А. А. Соколов
Группа: М8О-206Б-19
Дата: 28.12.2020
Оценка:
Подпись:

Москва, 2020

1 Постановка задачи

Цель работы:

Приобретение практических навыков в:

- Управлении серверами сообщений (№6)
- Применение отложенных вычислений (№7)
- Интеграция программных систем друг с другом (№8)

Задание (вариант 21):

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом. При убийстве («kill -9») любого вычислительного узла система должна пытаться максимально сохранять свою работоспособность, а именно все дочерние узлы убитого узла могут стать недоступными, но родительские узлы должны сохранить свою работоспособность.

Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд:

Создание нового вычислительного узла
Формат команды: create id [parent]
id – целочисленный идентификатор нового вычислительного узла
parent – целочисленный идентификатор родительского узла. Если топологией не предусмотрено введение данного параметра, то его необходимо игнорировать (если его ввели)
Формат вывода:
«Ok: pid», где pid – идентификатор процесса для созданного вычислительного узла
«Error: Already exists» - вычислительный узел с таким идентификатором уже существует
«Error: Parent not found» - нет такого родительского узла с таким идентификатором
«Error: Parent is unavailable» - родительский узел существует, но по каким-то причинам с ним не удается связаться «Error: [Custom error]» - любая другая обрабатываемая ошибка
Примечания: создание нового управляющего узла осуществляется пользователем программы при помощи запуска исполняемого файла. Id и pid — это разные идентификаторы.

Удаление существующего вычислительного узла
Формат команды: remove id
id – целочисленный идентификатор удаляемого вычислительного узла Формат вывода:
«Ok» - успешное удаление
«Error: Not found» - вычислительный узел с таким идентификатором не найден
«Error: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом
«Error: [Custom error]» - любая другая обрабатываемая ошибка
Примечание: при удалении узла из топологии его процесс должен быть завершен и работоспособность вычислительной сети не должна быть нарушена.

Исполнение команды на вычислительном узле
Формат команды: exec id [params]
id – целочисленный идентификатор вычислительного узла, на который отправляется команда
Формат вывода:
«Ok:id: [result]», где result – результат выполненной команды
«Error:id: Not found» - вычислительный узел с таким идентификатором не найден
«Error:id: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом
«Error:id: [Custom error]» - любая другая обрабатываемая ошибка
Примечание: выполнение команд должно быть асинхронным. Т.е. пока выполняется команда на одном из вычислительных узлов, то можно отправить следующую команду на другой вычислительный узел.

Вариант 21.
Топология 2
Все вычислительные узлы находятся в дереве общего вида. Есть только один управляющий узел.

Набор команд 3 (локальный таймер)
Формат команды сохранения значения: exec id subcommand
subcommand – одна из трех команд: start, stop, time.
start – запустить таймер
stop – остановить таймер
time – показать время локального таймера в миллисекундах
Пример:

```
>exec 10 time
```

```
Ok:10: 0
>exec 10 start
Ok:10
>exec 10 start
Ok:10
*прошло 10 секунд*
>exec 10 time
Ok:10: 10000
*прошло 2 секунды*
>exec 10 stop
Ok:10
*прошло 2 секунды*
>exec 10 time
Ok:10: 12000
```

Команда проверки 3

Формат команды: heartbit time

Каждый узел начинает сообщать раз в time миллисекунд о том, что он работоспособен. Если от узла нет сигнала в течении 4*time миллисекунд, то должна выводится пользователю строка: «Heartbit: node id is unavailable now», где id – идентификатор недоступного вычислительного узла.

Пример:

```
>heartbit 2000
Ok
```

2 Общий метод и алгоритм решения

Для работы с очередями используется ZMQ, программа собирается при помощи Makefile. Управляющий узел – server, вычислительные узлы – client. client и server - отдельные программы, причем новые вычислительные узлы создаются через fork() и execl(...). Для удобной работы с ZMQ создадим удобные обёртки над функциями библиотеки (заголовочный файл m_zmq.h). Сокеты будем создавать через ipc. В качестве типа сокетов возьмём Publish-Subscribe сокеты как наиболее подходящие. Также напишем классы Socket, Client и Server для работы на высоком уровне. Для хранения дерева общего вида будем использовать собственный класс IdTree.

При запуске программы создаётся сервер (управляющий узел). Затем создаётся вычислительный узел - корень дерева. Сервер работает в двух потоках: первый - работа с пользователем и отправка сообщений, второй - приём сообщений. Endpoint сокета создаётся по pid процесса вычислительного узла. При добавлении и удалении узлов происходит работа с IdTree, это необходимо, чтобы быстро проверять существование узла. Получив сообщение, вычислительный узел может либо отправить его обратно с новыми данными, либо отправить вниз своим детям (это нужно для операции heartbit и корректного уничтожения дерева процессов).

3 Исходный код

m_zmq.h

```
1
2  #pragma once
3
4  #include <string>
5
6  void* create_zmq_context();
7  void destroy_zmq_context(void* context);
8
9  enum class SocketType {
10   PUBLISHER,
11   SUBSCRIBER
12 };
13 void* create_zmq_socket(void* context, SocketType type);
14 void close_zmq_socket(void* socket);
15
16 enum class EndpointType {
17   CHILD_PUB,
18   PARRENT_PUB,
19 };
20 std::string create_endpoint(EndpointType type, pid_t id);
21
22 void bind_zmq_socket(void* socket, std::string endpoint);
23 void unbind_zmq_socket(void* socket, std::string endpoint);
24 void connect_zmq_socket(void* socket, std::string endpoint);
25 void disconnect_zmq_socket(void* socket, std::string endpoint);
26
27 enum class CommandType {
28   ERROR,
29   RETURN,
30   CREATE_CHILD,
31   REMOVE_CHILD,
32   TIMER_TIME,
33   TIMER_START,
34   TIMER_STOP,
35   HEARTBIT
36 };
37
38 struct Message {
39   CommandType command = CommandType::ERROR;
40   int to_id;
41   int value;
42   bool go_up = false;
43   int uniq_num;
44
45   Message();
```

```
46    Message(CommandType command_a, int to_id_a, int value_a);
47  };
48
49  bool operator==(const Message& lhs, const Message& rhs);
50
51  void send_zmq_msg(void* socket, Message msg);
52  Message get_zmq_msg(void* socket);
```

m_zmq.cpp

```
1
2   #include "m_zmq.h"
3
4   #include <errno.h>
5   #include <string.h>
6   #include <unistd.h>
7   #include <zmq.h>
8
9   #include <iostream>
10  #include <tuple>
11
12  using namespace std;
13
14  void* create_zmq_context() {
15    void* context = zmq_ctx_new();
16    if (context == NULL) {
17      throw runtime_error("Can't create new context. pid:" + to_string(getpid()));
18    }
19    return context;
20  }
21
22  void destroy_zmq_context(void* context) {
23    if (zmq_ctx_destroy(context) != 0) {
24      throw runtime_error("Can't destroy context. pid:" + to_string(getpid()));
25    }
26  }
27
28  int get_zmq_socket_type(SocketType type) {
29    switch (type) {
30      case SocketType::PUBLISHER:
31        return ZMQ_PUB;
32      case SocketType::SUBSCRIBER:
33        return ZMQ_SUB;
34      default:
35        throw logic_error("Undefined socket type");
36    }
37  }
38
39  void* create_zmq_socket(void* context, SocketType type) {
40    int zmq_type = get_zmq_socket_type(type);
```

```
41    void* socket = zmq_socket(context, zmq_type);
42    if (socket == NULL) {
43      throw runtime_error("Can't create socket");
44    }
45    if (zmq_type == ZMQ_SUB) {
46      if (zmq_setsockopt(socket, ZMQ_SUBSCRIBE, 0, 0) == -1) {
47        throw runtime_error("Can't set ZMQ_SUBSCRIBE option");
48      }
49      int linger_period = 0;
50      if (zmq_setsockopt(socket, ZMQ_LINGER, &linger_period, sizeof(int)) == -1) {
51        throw runtime_error("Can't set ZMQ_LINGER option");
52      }
53    }
54    return socket;
55  }
56
57  void close_zmq_socket(void* socket) {
58    //cerr << "closing socket..." << endl;
59    sleep(1); // Don't comment it, because sometimes zmq_close blocks
60    if (zmq_close(socket) != 0) {
61      throw runtime_error("Can't close socket");
62    }
63  }
64
65  string create_endpoint(EndpointType type, pid_t id) {
66    switch (type) {
67      case EndpointType::PARRENT_PUB:
68        return "ipc://tmp/parrent_pub_"s + to_string(id);
69      case EndpointType::CHILD_PUB:
70        return "ipc://tmp/child_pub_"s + to_string(id);
71      default:
72        throw logic_error("Undefined endpoint type");
73    }
74  }
75
76  void bind_zmq_socket(void* socket, string endpoint) {
77    if (zmq_bind(socket, endpoint.data()) != 0) {
78      throw runtime_error("Can't bind socket");
79    }
80  }
81
82  void unbind_zmq_socket(void* socket, string endpoint) {
83    if (zmq_unbind(socket, endpoint.data()) != 0) {
84      throw runtime_error("Can't unbind socket");
85    }
86  }
87
88  void connect_zmq_socket(void* socket, string endpoint) {
89    if (zmq_connect(socket, endpoint.data()) != 0) {
```

```
90      throw runtime_error("Can't connect socket");
91    }
92  }
93
94  void disconnect_zmq_socket(void* socket, string endpoint) {
95    if (zmq_disconnect(socket, endpoint.data()) != 0) {
96      throw runtime_error("Can't disconnect socket");
97    }
98  }
99
100 int counter = 0;
101 Message::Message() {
102   uniq_num = counter++;
103 }
104
105 Message::Message(CommandType command_a, int to_id_a, int value_a)
106     : Message() {
107   command = command_a;
108   to_id = to_id_a;
109   value = value_a;
110 }
111
112 bool operator==(const Message& lhs, const Message& rhs) {
113   return tie(lhs.command, lhs.to_id, lhs.value, lhs.uniq_num) == tie(rhs.command, rhs.
        to_id, rhs.value, rhs.uniq_num);
114 }
115
116 void create_zmq_msg(zmq_msg_t* zmq_msg, Message msg) {
117   zmq_msg_init_size(zmq_msg, sizeof(Message));
118   memcpy(zmq_msg_data(zmq_msg), &msg, sizeof(Message));
119 }
120
121 void send_zmq_msg(void* socket, Message msg) {
122   zmq_msg_t zmq_msg;
123   create_zmq_msg(&zmq_msg, msg);
124   if (!zmq_msg_send(&zmq_msg, socket, 0)) {
125     throw runtime_error("Can't send message");
126   }
127   zmq_msg_close(&zmq_msg);
128 }
129
130 Message get_zmq_msg(void* socket) {
131   zmq_msg_t zmq_msg;
132   zmq_msg_init(&zmq_msg);
133   if (zmq_msg_recv(&zmq_msg, socket, 0) == -1) {
134     return Message{CommandType::ERROR, 0, 0};
135   }
136   Message msg;
137   memcpy(&msg, zmq_msg_data(&zmq_msg), sizeof(Message));
```

```
138   zmq_msg_close(&zmq_msg);
139   return msg;
140 }
```

socket.h

```
1
2  #pragma once
3
4  #include <unistd.h>
5
6  #include <memory>
7  #include <unordered_map>
8
9  #include "socket.h"
10 #include "tree.h"
11
12 class Server {
13  public:
14   Server();
15   ~Server();
16
17   pid_t pid() const;
18   Message last_message() const;
19
20   void send(Message message);
21   Message receive();
22
23   bool check(int id);
24   void create_child_cmd(int id, int parrent_id);
25   void remove_child_cmd(int id);
26   void exec_cmd(int id, CommandType type);
27   void heartbit_cmd(int time);
28   void print_tree();
29
30   friend void* second_thread(void* serv_arg);
31
32  private:
33   pid_t pid_;
34   void* context_ = nullptr;
35   std::unique_ptr<Socket> publiser_;
36   std::unique_ptr<Socket> subscriber_;
37
38   pthread_t receive_msg_loop_id;
39   bool terminated_ = false;
40
41   Message last_message_;
42   IdTree tree_;
43   std::unordered_map<int, bool> map_for_check_;
44 };
```

client.h

```

#pragma once

#include <unistd.h>

#include <chrono>
#include <memory>
#include <string>

#include "socket.h"

class Client {
 public:
  Client(int id, std::string parrent_endpoint);
  ~Client();

  int id() const;
  pid_t pid() const;

  void send_up(Message message);
  void send_down(Message message);
  Message receive();

  void start_timer();
  void stop_timer();
  int get_time();
  void heartbit(int time);

  void
  add_child(int id);

 private:
  int id_;
  pid_t pid_;
  void* context_ = nullptr;
  std::unique_ptr<Socket> child_publiser_;
  std::unique_ptr<Socket> parrent_publiser_;
  std::unique_ptr<Socket> subscriber_;

  bool is_timer_started = false;
  std::chrono::steady_clock::time_point start_;
  std::chrono::steady_clock::time_point finish_;

  bool terminated_ = false;
};
```

client.cpp

```

```

```cpp
#include "client.h"

#include <unistd.h>

#include <iostream>

#include "m_zmq.h"

using namespace std;

const string CLIENT_EXE = "./client";
const double MESSAGE_WAITING_TIME = 1;

Client::Client(int id, std::string parrent_endpoint) {
  id_ = id;
  pid_ = getpid();
  cerr << to_string(pid_) + " Starting client..."s << endl;
  context_ = create_zmq_context();

  string endpoint = create_endpoint(EndpointType::CHILD_PUB, getpid());
  child_publiser_ = make_unique<Socket>(context_, SocketType::PUBLISHER, endpoint);
  endpoint = create_endpoint(EndpointType::PARRENT_PUB, getpid());
  parrent_publiser_ = make_unique<Socket>(context_, SocketType::PUBLISHER, endpoint);

  subscriber_ = make_unique<Socket>(context_, SocketType::SUBSCRIBER, parrent_endpoint
      );

  sleep(MESSAGE_WAITING_TIME);
  send_up(Message(CommandType::CREATE_CHILD, id, getpid()));
}

Client::~Client() {
  if (terminated_) {
    cerr << to_string(pid_) + " Client double termination"s << endl;
    return;
  }

  cerr << to_string(pid_) + " Destroying client..."s << endl;
  terminated_ = true;

  sleep(MESSAGE_WAITING_TIME);
  try {
    child_publiser_ = nullptr;
    parrent_publiser_ = nullptr;
    subscriber_ = nullptr;
    destroy_zmq_context(context_);
  } catch (exception& ex) {
    cerr << to_string(pid_) + " " + " Client wasn't destroyed: "s << ex.what() << endl;
  }
```

```
50 }
51
52 void Client::send_up(Message message) {
53   message.go_up = true;
54   parrent_publiser_->send(message);
55 }
56
57 void Client::send_down(Message message) {
58   message.go_up = false;
59   child_publiser_->send(message);
60 }
61
62 Message Client::receive() {
63   Message msg = subscriber_->receive();
64   return msg;
65 }
66
67 void Client::add_child(int id) {
68   pid_t child_pid = fork();
69   if (child_pid == -1) {
70     throw runtime_error("Can't fork");
71   }
72   if (child_pid == 0) {
73     execl(CLIENT_EXE.data(), CLIENT_EXE.data(), to_string(id).data(), child_publiser_->
         endpoint().data(), NULL);
74     throw runtime_error("Can't execl ");
75   }
76
77   string endpoint = create_endpoint(EndpointType::PARRENT_PUB, child_pid);
78   subscriber_->subscribe(endpoint);
79 }
80
81 void Client::start_timer() {
82   is_timer_started = true;
83   start_ = std::chrono::steady_clock::now();
84 }
85
86 void Client::stop_timer() {
87   is_timer_started = false;
88   finish_ = std::chrono::steady_clock::now();
89 }
90
91 int Client::get_time() {
92   if (is_timer_started) {
93     finish_ = std::chrono::steady_clock::now();
94   }
95   return std::chrono::duration_cast<std::chrono::milliseconds>(finish_ - start_).count
       ();
96 }
```

```
97
98  void Client::heartbit(int time) {
99    sleep((double)time / 1000); //s to ms
100   send_up(Message(CommandType::HEARTBIT, id_, 0));
101 }
102
103 int Client::id() const {
104   return id_;
105 }
106
107 pid_t Client::pid() const {
108   return pid_;
109 }
```

client_main.cpp

```
1
2   #include <signal.h>
3
4   #include <iostream>
5   #include <string>
6
7   #include "client.h"
8
9   using namespace std;
10
11  const int ERR_TERMINATED = 1;
12  const int UNIVERSAL_MESSAGE_ID = -256;
13
14  Client* client_ptr = nullptr;
15  void TerminateByUser(int) {
16    if (client_ptr != nullptr) {
17      client_ptr->~Client();
18    }
19    cerr << to_string(getpid()) + " Terminated by user"s << endl;
20    exit(0);
21  }
22
23  void process_msg(Client& client, const Message msg) {
24    cerr << to_string(getpid()) + " Client message: "s << static_cast<int>(msg.command)
          << " " << msg.to_id << " " << msg.value << endl;
25    switch (msg.command) {
26      case CommandType::ERROR:
27        throw runtime_error("Error message recieved");
28      case CommandType::RETURN:
29        client.send_up(msg);
30        break;
31      case CommandType::CREATE_CHILD:
32        client.add_child(msg.value);
33        break;
```

```
34      case CommandType::REMOVE_CHILD: {
35        if (msg.to_id != UNIVERSAL_MESSAGE_ID) {
36          client.send_up(msg);
37        }
38        Message tmp = msg;
39        tmp.to_id = UNIVERSAL_MESSAGE_ID;
40        client.send_down(tmp);
41        TerminateByUser(0);
42        break;
43      }
44      case CommandType::TIMER_START:
45        client.start_timer();
46        client.send_up(msg);
47        break;
48      case CommandType::TIMER_STOP:
49        client.stop_timer();
50        client.send_up(msg);
51        break;
52      case CommandType::TIMER_TIME: {
53        int val = client.get_time();
54        client.send_up(Message(CommandType::TIMER_TIME, client.id(), val));
55        break;
56      }
57      case CommandType::HEARTBIT:
58        client.send_down(msg);
59        client.heartbit(msg.value);
60        break;
61      default:
62        throw logic_error("Not implemented message command");
63    }
64  }
65
66  int main(int argc, char const* argv[]) {
67    if (argc != 3) {
68      cerr << argc;
69      cerr << "USAGE: " << argv[0] << " <id> <parrent_pub_endpoint" << endl;
70    }
71
72    try {
73      if (signal(SIGINT, TerminateByUser) == SIG_ERR) {
74        throw runtime_error("Can't set SIGINT signal");
75      }
76      if (signal(SIGSEGV, TerminateByUser) == SIG_ERR) {
77        throw runtime_error("Can't set SIGSEGV signal");
78      }
79
80      Client client(stoi(argv[1]), string(argv[2]));
81      client_ptr = &client;
82      cerr << to_string(getpid()) + " Client is started correctly"s << endl;
```

```
83
84      while (true) {
85        Message msg = client.receive();
86        if (msg.to_id != client.id() && msg.to_id != UNIVERSAL_MESSAGE_ID) {
87          if (msg.go_up) {
88            client.send_up(msg);
89          } else {
90            client.send_down(msg);
91          }
92          continue;
93        }
94        process_msg(client, msg);
95      }
96
97    } catch (exception& ex) {
98      cerr << to_string(getpid()) + " Client exception: "s << ex.what() << "\nTerminated
          by exception" << endl;
99      exit(ERR_TERMINATED);
100   }
101   cerr << to_string(getpid()) + " Client is finished correctly"s << endl;
102   return 0;
103 }
```

server.h

```
1
2  #pragma once
3
4  #include <unistd.h>
5
6  #include <memory>
7  #include <unordered_map>
8
9  #include "socket.h"
10 #include "tree.h"
11
12 class Server {
13  public:
14   Server();
15   ~Server();
16
17   pid_t pid() const;
18   Message last_message() const;
19
20   void send(Message message);
21   Message receive();
22
23   bool check(int id);
24   void create_child_cmd(int id, int parrent_id);
25   void remove_child_cmd(int id);
```

```
  void exec_cmd(int id, CommandType type);
  void heartbit_cmd(int time);
  void print_tree();

  friend void* second_thread(void* serv_arg);

 private:
  pid_t pid_;
  void* context_ = nullptr;
  std::unique_ptr<Socket> publiser_;
  std::unique_ptr<Socket> subscriber_;

  pthread_t receive_msg_loop_id;
  bool terminated_ = false;

  Message last_message_;
  IdTree tree_;
  std::unordered_map<int, bool> map_for_check_;
};
```

server.cpp

```

#include "server.h"

#include <pthread.h>
#include <signal.h>
#include <unistd.h>

#include <iostream>

#include "m_zmq.h"

using namespace std;

const int ERR_LOOP = 2;
const int ERR_EXEC = 3;
const string CLIENT_EXE = "./client";
const double MESSAGE_WAITING_TIME = 1;
const int UNIVERSAL_MESSAGE_ID = -256;

void* second_thread(void* serv_arg) {
  Server* server_ptr = (Server*)serv_arg;
  pid_t server_pid = server_ptr->pid();
  try {
    pid_t child_pid = fork();
    if (child_pid == -1) {
      throw runtime_error("Can't fork");
    }
    if (child_pid == 0) {
```

```
29      execl(CLIENT_EXE.data(), CLIENT_EXE.data(), "0", server_ptr->publiser_->endpoint
            ().data(), NULL);
30      cerr << "Can't execl "s + CLIENT_EXE << endl;
31      kill(server_pid, SIGINT);
32      exit(ERR_EXEC);
33    }
34
35    string endpoint = create_endpoint(EndpointType::PARRENT_PUB, child_pid);
36    server_ptr->subscriber_ = make_unique<Socket>(server_ptr->context_, SocketType::
          SUBSCRIBER, endpoint);
37    server_ptr->tree_.add_to(0, {0, child_pid});
38
39    while (true) {
40      Message msg = server_ptr->subscriber_->receive();
41      if (msg.command == CommandType::ERROR) {
42        if (server_ptr->terminated_) {
43          return NULL;
44        } else {
45          cerr << "This bom" << endl;
46          throw runtime_error("Can't receive message");
47        }
48      }
49      server_ptr->last_message_ = msg;
50      cerr << "Message on server: " << static_cast<int>(msg.command) << " " << msg.
            to_id << " " << msg.value << endl;
51
52      switch (msg.command) {
53        case CommandType::CREATE_CHILD: {
54          auto& pa = server_ptr->tree_.get(msg.to_id);
55          pa.second = msg.value;
56          cout << "OK: " << server_ptr->last_message_.value << endl;
57          break;
58        }
59        case CommandType::REMOVE_CHILD:
60          server_ptr->tree_.remove(msg.to_id);
61          cout << "OK" << endl;
62          break;
63        case CommandType::TIMER_START:
64          cout << "OK:" << msg.to_id << endl;
65          break;
66        case CommandType::TIMER_STOP:
67          cout << "OK:" << msg.to_id << endl;
68          break;
69        case CommandType::TIMER_TIME: {
70          cout << "OK:" << msg.to_id << ": " << msg.value << endl;
71          break;
72        }
73        case CommandType::HEARTBIT:
74          server_ptr->map_for_check_[msg.to_id] = true;
```

```
75            break;
76          default:
77            break;
78        }
79      }
80
81    } catch (exception& ex) {
82      cerr << "Server exctption: " << ex.what() << "\nTerminated by exception on server
          receive loop" << endl;
83      exit(ERR_LOOP);
84    }
85    return NULL;
86  }
87
88  Server::Server() {
89    pid_ = getpid();
90    cerr << to_string(pid_) + " Starting server..."s << endl;
91    context_ = create_zmq_context();
92
93    string endpoint = create_endpoint(EndpointType::CHILD_PUB, getpid());
94    publiser_ = make_unique<Socket>(context_, SocketType::PUBLISHER, endpoint);
95
96    if (pthread_create(&receive_msg_loop_id, 0, second_thread, this) != 0) {
97      throw runtime_error("Can't run second thread");
98    }
99  }
100
101 Server::~Server() {
102   if (terminated_) {
103     cerr << to_string(pid_) + " Server double termination" << endl;
104     return;
105   }
106
107   cerr << to_string(pid_) + " Destroying server..."s << endl;
108   terminated_ = true;
109
110   for (pid_t pid : tree_.get_all_second()) {
111     kill(pid, SIGINT);
112   }
113
114   try {
115     publiser_ = nullptr;
116     subscriber_ = nullptr;
117     destroy_zmq_context(context_);
118   } catch (exception& ex) {
119     cerr << "Server wasn't destroyed: " << ex.what() << endl;
120   }
121 }
122
```

```
123 void Server::send(Message message) {
124   message.go_up = false;
125   publiser_->send(message);
126 }
127
128 Message Server::receive() {
129   return subscriber_->receive();
130 }
131
132 pid_t Server::pid() const {
133   return pid_;
134 }
135
136 Message Server::last_message() const {
137   return last_message_;
138 }
139
140 bool Server::check(int id) {
141   Message msg(CommandType::RETURN, id, 0);
142   send(msg);
143   sleep(MESSAGE_WAITING_TIME);
144   if (last_message_ == msg) {
145     return true;
146   } else {
147     return false;
148   }
149 }
150
151 void Server::create_child_cmd(int id, int parrent_id) {
152   if (tree_.find(id)) {
153     cout << "Error: Already exists" << endl;
154     return;
155   }
156   if (!tree_.find(parrent_id)) {
157     cout << "Error: Parent not found" << endl;
158     return;
159   }
160   if (!check(parrent_id)) {
161     cout << "Error: Parent is unavailable" << endl;
162     return;
163   }
164   send(Message(CommandType::CREATE_CHILD, parrent_id, id));
165   tree_.add_to(parrent_id, {id, 0});
166 }
167
168 void Server::remove_child_cmd(int id) {
169   if (id == 0) {
170     cout << "Can't remove zero child" << endl;
171     return;
```

```
172   }
173   if (!tree_.find(id)) {
174     cout << "Error: Not found" << endl;
175     return;
176   }
177   if (!check(id)) {
178     cout << "Error: Node is unavailable" << endl;
179     return;
180   }
181   send(Message(CommandType::REMOVE_CHILD, id, 0));
182 }
183 
184 void Server::exec_cmd(int id, CommandType type) {
185   if (!tree_.find(id)) {
186     cout << "Error: Not found" << endl;
187     return;
188   }
189   if (!check(id)) {
190     cout << "Error: Node is unavailable" << endl;
191     return;
192   }
193   send(Message(type, id, 0));
194 }
195 
196 void Server::heartbit_cmd(int time) {
197   if (time < 1000) {
198     cout << "Too low time for heartbit" << endl;
199   }
200   send(Message(CommandType::HEARTBIT, UNIVERSAL_MESSAGE_ID, time));
201   auto uset = tree_.get_all_first();
202   for (int id : uset) {
203     map_for_check_[id] = false;
204   }
205   sleep(4 * (double)time / 1000);
206   for (auto& [id, bit] : map_for_check_) {
207     if (!bit) {
208       cout << "Heartbit: node " << id << " is unavailable now" << endl;
209     }
210     bit = false;
211   }
212   cout << "OK" << endl;
213 }
214 
215 void Server::print_tree() {
216   tree_.print();
217 }
```

server_main.cpp

```
1 
```

```
2  #include <signal.h>
3
4  #include <iostream>
5  #include <string>
6
7  #include "server.h"
8
9  using namespace std;
10
11 const int ERR_TERMINATED = 1;
12
13 Server* server_ptr = nullptr;
14 void TerminateByUser(int) {
15   if (server_ptr != nullptr) {
16     server_ptr->~Server();
17   }
18   cerr << to_string(getpid()) + " Terminated by user"s << endl;
19   exit(0);
20 }
21
22 void process_cmd(Server& server, string cmd) {
23   if (cmd == "check") {
24     int id;
25     cin >> id;
26     if (server.check(id)) {
27       cout << "OK" << endl;
28     } else {
29       cout << "NOT_OK" << endl;
30     }
31   } else if (cmd == "create") {
32     int id, parrent_id;
33     cin >> id >> parrent_id;
34     server.create_child_cmd(id, parrent_id);
35   } else if (cmd == "remove") {
36     int id;
37     cin >> id;
38     server.remove_child_cmd(id);
39   } else if (cmd == "exec") {
40     int id;
41     string sub_cmd;
42     cin >> id >> sub_cmd;
43     CommandType type;
44     if (sub_cmd == "time") {
45       type = CommandType::TIMER_TIME;
46     } else if (sub_cmd == "start") {
47       type = CommandType::TIMER_START;
48     } else if (sub_cmd == "stop") {
49       type = CommandType::TIMER_STOP;
50     } else {
```

```
51        cout << "Incorrect subcommand" << endl;
52        return;
53      }
54      server.exec_cmd(id, type);
55    } else if (cmd == "heartbit") {
56      int time;
57      cin >> time;
58      server.heartbit_cmd(time);
59    } else if (cmd == "print_tree") {
60      server.print_tree();
61    } else {
62      cout << "It's not a command" << endl;
63    }
64  }
65
66  int main() {
67    try {
68      if (signal(SIGINT, TerminateByUser) == SIG_ERR) {
69        throw runtime_error("Can't set SIGINT signal");
70      }
71      if (signal(SIGSEGV, TerminateByUser) == SIG_ERR) {
72        throw runtime_error("Can't set SIGSEGV signal");
73      }
74
75      Server server;
76      server_ptr = &server;
77      cerr << to_string(getpid()) + " Server is started correctly"s << endl;
78
79      string cmd;
80      while (cin >> cmd) {
81        process_cmd(server, cmd);
82      }
83
84    } catch (exception& ex) {
85      cerr << to_string(getpid()) + " Server exception: "s << ex.what() << "\nTerminated
          by exception" << endl;
86      exit(ERR_TERMINATED);
87    }
88    cerr << to_string(getpid()) + " Server is finished correctly"s << endl;
89    return 0;
90  }
```

4 Пример работы

Продемонстрирую процесс сборки программы и её работу:

```
windicor@Lina-HP:~$ make
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c server_main.cpp
-o server_main.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c server.cpp -o
server.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c m_zmq.cpp -o
m_zmq.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c socket.cpp -o
socket.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c tree.cpp -o
tree.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable server_main.o server.o
m_zmq.o socket.o tree.o -o server -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c client_main.cpp
-o client_main.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c client.cpp -o
client.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable client_main.o client.o
m_zmq.o socket.o tree.o -o client -lzmq -lpthread
windicor@Lina-HP:~$ ./server 2>log
OK: 166
create 0 0
Error: Already exists
create 1 0
OK: 169
create 2 1
OK: 172
create 3 1
OK: 175
create 4 2
OK: 178
remove 2
OK
exec 4 start
Error: Not found
exec 2 start
```

```
Error: Not found
exec 1 start
OK:1
exec 1 time
OK:1: 20912
exec 1 stop
OK:1
exec 1 time
OK:1: 25960
exec 1 time
OK:1: 25960
heartbit 1000
OK
heartbit 1000 //сделали kill -9 для узла 3
Heartbit: node 3 is unavailable now
OK
```

5 Вывод

В процессе написания лабораторной я научился основам работы с серверами сообщений на примере ZMQ, изучил возможные подходы к созданию приложений на основе серверов сообщений. Технология действительно удобна для решения ряда задач, а конкретно ZMQ обладает достаточно высокой скоростью передачи данных. Конечно, у ZMQ есть ряд недостатков, но особых трудностей при написании лабораторной это не вызвало. Использование C++ оказалось оправданным в этой работе, так как это позволило быстро разработать ряд классов-абстракций и не держать в голове архитектуру целиком. Также удобными оказаль механизм исключений и наличие деструкторов классов (не требоволась рассматривать все случаи, в которых, например, надо закрывать сокет и т. п.) Задача показалась достаточной интересной и необычной, поэтому решать её было удовольствием.