Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работ №6 по курсу**

**«Операционные системы»**

**Сервера сообщений**

Студент: Ефимов Александр Владимирович

Группа: М80 – 201Б-18

Вариант: 6

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2019

1. **Постановка задачи**

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.

**Топология:** дерево общего вида

**Тип команд:** поиск подстроки в строке

**Тип проверки доступности узлов:** *ping id*

1. **Описание программы**

**Controller.cpp** содержит исходный код для управляющего узла. После его запуска и создания портов для общения с вычислительными узлами, программа входит в цикл, который продолжает работать до получения команды *“exit”*. Он также принимает следующие команды:

* *“add [id] [parent id]”*

Создает вычислительный узел и привязывает его к *[parent id]*. Если в поле родительского id написан -1, то привязывает к управляющему узлу.

После попытки создания возвращает результат (успех или неуспех).

* *“remove [id]”*

Удаляет *[id]*. Также возвращает успех или неуспех.

* *“ping [id]”*

Отправляет вычислительному узлу запрос, после чего ожидает заданное количество времени. Если ответ пришел, то возвращает успех, иначе предупреждает об истечении времени.

* *“exec [id] [text] [pattern]"*

Отправляет вычислительному узлу задание на поиск в тексте.

Узел будет возвращать первые числа каждого вхождения (начиная с 0) или -1, если ничего не найдено.

* *“check”*

Во избежание столкновения ввода и вывода, а также для отдельного получения результатов в случае, если задание занимает много времени, проверка происходит либо после выполнения любой команды, либо после ввода *“check”* (что технически является пустой командой, позволяющей программе продолжить до функции проверки результатов *“exec”*).

**Worker.cpp** – исходный код для вычислительного узла. Сразу после его создания он отправляет родителю сообщение об успехе и переходит в режим ожидания, когда один из входящих сокетов не получит сообщение, после чего узел просматривает адресата и принимает решение – если id адресата и собственный не совпадают, то отправить ниже или выше по дереву (в зависимости из какого сокета сообщение), иначе выполнить действие, описанное в сообщении.

**Lexicon.hpp** – заголовок, который содержит в себе константы (такие как время истечения ожидания), общие структуры (структура сообщения) и типы.

1. **Набор тестов**

|  |
| --- |
| create 10 -1  create 11 10  create 12 10  create 7 11  ping 10  ping 11  ping 12  ping 7  exec 7 abracadabra abra  exec 10 aabaabcaxaabaabcy aab  remove 11  ping 7  ping 11  ping 12  remove 10  ping 12  ping 10  exit |

Тест, который проверяет работоспособность узлов, акцентируя внимания на узлах на второй глубине (т.е. детей узлов, привязанных к управляющему).

1. **Результаты выполнения тестов.**

|  |
| --- |
| ► create 10 -1  ID 10 reports: success  ► create 11 10  ID 11 reports: success  ► create 12 10  ID 12 reports: success  ► create 7 11  ID 7 reports: success  ► ping 10  ID 10 reports: success  ► ping 11  ID 11 reports: success  ► ping 12  ID 12 reports: success  ► ping 7  ID 7 reports: success  ► exec 7  abracadabra  abra  MAILBOX: ID 7 reports: 0;7  ► exec 10  aabaabcaxaabaabcy  aab  MAILBOX: ID 10 reports: 0;3;9;12  ► remove 11  ID 11 reports: success  Subtree deleted  ► ping 7  ERROR: ID not found  ► ping 11  ERROR: ID not found  ► ping 12  ID 12 reports: success  ► remove 10  ID 10 reports: success  Subtree deleted  ► ping 12  ERROR: ID not found  ► ping 10  ERROR: ID not found  ► exit  CONTROLLER: Ending service (EXEC mailbox will be checked once more)► create 10 -1  ID 10 reports: success  ► create 11 10  ID 11 reports: success  ► create 12 10  ID 12 reports: success  ► create 7 11  ID 7 reports: success  ► ping 10  ID 10 reports: success  ► ping 11  ID 11 reports: success  ► ping 12  ID 12 reports: success  ► ping 7  ID 7 reports: success  ► exec 7  abracadabra  abra  MAILBOX: ID 7 reports: 0;7  ► exec 10  aabaabcaxaabaabcy  aab  MAILBOX: ID 10 reports: 0;3;9;12  ► remove 11  ID 11 reports: success  Subtree deleted  ► ping 7  ERROR: ID not found  ► ping 11  ERROR: ID not found  ► ping 12  ID 12 reports: success  ► remove 10  ID 10 reports: success  Subtree deleted  ► ping 12  ERROR: ID not found  ► ping 10  ERROR: ID not found  ► exit  CONTROLLER: Ending service (EXEC mailbox will be checked once more) |

Каждый из созданных узлов успешно выполняет команды. Кроме того, управляющий узел правильно ведет счет об удаленных и добавленных узлах.

1. **Листинг программы**

*lexicon.hpp*

|  |
| --- |
| #pragma once  #include <string>  #include <zmq.hpp>  //----------//  // Settings //  //----------//  const int TIMEOUT\_MS = 5000;  const int EXEC\_TIMEOUT\_MS = 200;  const size\_t MAX\_STRING\_LENGTH = 256;  const char WORKER\_NAME[] = "worker";  //----------------------//  // Types and structures //  //----------------------//  using TNodeID = int;  enum CommType { EXEC, PING, ADD, DEL };  enum Signal { START, SUCC, FAIL, ERR };  struct TLetter {  // Addressing <id>  TNodeID id;  CommType commType;  Signal sig;  };  //-----------//  // Functions //  //-----------//  const size\_t PORT\_SIZE = 10;  std::string ParsePort(zmq::socket\_t& socket) {  char address[1024];  char port[10] = {};  size\_t size = sizeof(address);  socket.getsockopt(ZMQ\_LAST\_ENDPOINT, &address, &size);  int i = 0, j = 0;  while (address[i - 1] != ':') ++i;  ++i;  while (address[i - 1] != ':') ++i;  for (; address[i] != '\0' && i < 1024 && j < PORT\_SIZE; ++j, ++i) {  port[j] = address[i];  }  return std::string (port);  }  void CommandRoute(zmq::socket\_t& socket, const int id, const TLetter command) {  std::string idStr (std::to\_string(id));  // Found out that you can't write directly to message  // Total hours wasted here: ~20  // That one was my fault. We get it.  socket.send(idStr.c\_str(), idStr.size(), ZMQ\_SNDMORE);  socket.send((void\*)&command, sizeof(TLetter), 0);  }  // In file search.o  std::string FindPos(const std::string pattern, const std::string text);  // Result - indexes of match ups, divided by ";"  // If none found, empty string |

*worker.cpp*

|  |
| --- |
| #include <zmq.hpp>  #include <iostream>  #include <map>  #include <string>  #include <sstream>  #include <thread>  #include <unistd.h>  #include "lexicon.hpp"  #include "zhelpers.hpp"  // #define DEBUG  void ConnectToPort(const std::string port, zmq::socket\_t& socket) {  std::string address = "tcp://localhost:" + port;  socket.connect(address.c\_str());  }  zmq::message\_t Confirmation(const TNodeID id) {  TLetter res {id, ADD, SUCC};  zmq::message\_t msg((void\*) &res, sizeof(TLetter));  return msg;  }  void WorkerRoutine(zmq::context\_t& context, const std::string& port, const TNodeID id, bool& work) {  zmq::socket\_t execSend(context, ZMQ\_PUSH);  ConnectToPort(port, execSend);  zmq::socket\_t threadRecieve(context, ZMQ\_PULL);  threadRecieve.connect("inproc://silkroad");  zmq::message\_t inMsg;  while(work) {  if (threadRecieve.recv(&inMsg, ZMQ\_NOBLOCK)) {  std::string all((char\*) inMsg.data(), inMsg.size()),  pattern, text;  std::stringstream allStream (all);  allStream >> pattern >> text;  std::string result( FindPos(pattern, text) );    TLetter outLetter = {id, EXEC, SUCC};  execSend.send((void\*)&outLetter, sizeof(TLetter), ZMQ\_SNDMORE);    zmq::message\_t outMsg;  if (result.size() == 0) {  outMsg.rebuild("-1", 2);  } else {  outMsg.rebuild((void\*)result.c\_str(), result.size());  }  execSend.send(outMsg, 0);  }  }    return;  }  int main(int argc, char\* argv[]) {  //Arguments are: ID, parent "from" port, parent "to" port, controller exec port  if (argc < 5) {  #ifdef DEBUG  std::cerr << "INVALID\_ARG\_NUM" << std::endl;  #endif  return -1;  }  TNodeID id = std::atoi(argv[1]);  std::string idStr(argv[1]),  parentRcvPort(argv[2]),  parentSndPort(argv[3]),  controllerPort(argv[4]),  childrenSendPort,  childrenRecievePort;  //-------------------//  // Establish sockets //  //-------------------//  zmq::context\_t context(1);  // Upstream: 2 sockets and exec socket  zmq::socket\_t parentRecieve(context, ZMQ\_DEALER);  zmq::socket\_t parentSend (context, ZMQ\_PUSH);  parentRecieve.setsockopt(ZMQ\_IDENTITY, idStr.c\_str(), idStr.size()); //Discards '\0'  ConnectToPort(parentRcvPort, parentRecieve);  ConnectToPort(parentSndPort, parentSend);  // Downstream: 2 sockets  zmq::socket\_t childrenSend (context, ZMQ\_ROUTER);  zmq::socket\_t childrenRecieve(context, ZMQ\_PULL);  childrenRecieve.setsockopt(ZMQ\_RCVTIMEO, TIMEOUT\_MS);  childrenSend.bind("tcp://127.0.0.1:\*");  childrenRecieve.bind("tcp://127.0.0.1:\*");  childrenSendPort = ParsePort(childrenSend);  childrenRecievePort = ParsePort(childrenRecieve);  // In process: thread  zmq::socket\_t threadSend(context, ZMQ\_PUSH);  threadSend.bind("inproc://silkroad");  bool work = true;  std::thread execThread(WorkerRoutine, std::ref(context), std::cref(controllerPort), id, std::ref(work));  //---------------------//  // Send "Okay message" //  //---------------------//  zmq::message\_t msg = Confirmation(id);  parentSend.send(msg);  //------------------//  // Setup poll items //  //------------------//  zmq::pollitem\_t items[] = {  {(void\*)parentRecieve, 0, ZMQ\_POLLIN, 0},  {(void\*)childrenRecieve, 0, ZMQ\_POLLIN, 0}  };  #ifdef DEBUG  std::cout << "OUT: " << childrenSendPort << " | IN: " << childrenRecievePort << std::endl;  #endif  //----------------//  // Main loop vars //  //----------------//  std::map<TNodeID, TNodeID> subtreeID;  while (work) {  zmq::message\_t message;  zmq::poll(items, 2, -1);  if(items[0].revents & ZMQ\_POLLIN) {  parentRecieve.recv(&message, ZMQ\_DONTWAIT);  TLetter ioLetter = \*(static\_cast<TLetter\*> (message.data()));    if (ioLetter.id == id) {  // Initialize data to avoid doing that in GOTO statements  pid\_t pid;  zmq::message\_t idMessage,  returnMessage;  std::string childID, strOfID; // childID for one ID, strOfID for string of IDs in delete  switch (ioLetter.commType) {  case ADD:  parentRecieve.recv(&idMessage, ZMQ\_DONTWAIT);  childID = std::string((char\*)idMessage.data(), idMessage.size());  #ifdef DEBUG  pid = 1;  #else  pid = fork();  #endif  if (pid < -1) {  ioLetter.sig = ERR;  parentSend.send(&ioLetter, sizeof(TLetter), 0);  break;    } else if (pid == 0) {  if ( execl(WORKER\_NAME,  WORKER\_NAME,  childID.c\_str(),  childrenSendPort.c\_str(),  childrenRecievePort.c\_str(),  controllerPort.c\_str(),NULL) < 0 )  {  #ifdef DEBUG  std::cerr << "ERROR: Failed to create new process" << std::endl;  #endif  return -1;  }  } else if (pid == 1) {  #ifdef DEBUG  std::cout << "MANUAL\_SERVER\_CREATION\_MODE" << std::endl;  #endif  }  if (childrenRecieve.recv(&returnMessage)) {  // Send the return of a child, containing its id  parentSend.send(returnMessage, ZMQ\_SNDMORE);  parentSend.send(idStr.c\_str(), idStr.size(), 0);  // Add id of this node to map in a parent nodes  TNodeID childIdInt = std::atoi(childID.c\_str());  subtreeID.insert(std::pair<TNodeID, TNodeID> (childIdInt, childIdInt));  } else {  ioLetter.id = id; // Replace with id of this node  ioLetter.sig = FAIL; // Notify of failure  returnMessage.rebuild((void\*)&ioLetter, sizeof(TLetter));  parentSend.send(returnMessage);  }  break;  case DEL:  // Send delete command to kids to trigger recursion;  for (std::pair<const TNodeID, TNodeID>& e : subtreeID) {  // Adjacent node are reachable from themselves  if (e.first == e.second) {  std::string childStr(std::to\_string(e.second));  TLetter end {e.second, DEL, START};  zmq::message\_t endNote((void\*)&end, sizeof(TLetter), 0);  childrenSend.send(childStr.c\_str(), childStr.size(), ZMQ\_SNDMORE);  childrenSend.send(endNote, 0);  if (!childrenRecieve.recv(&message, 0)) {  // Apparently, messages need to synchnorise as well  // Jesus fucking christ  TLetter childLetter = \*( static\_cast<TLetter\*> (message.data()) );  parentSend.send((void\*)&childLetter, sizeof(TLetter), 0);  continue;  }  }  }    // Build a string containing all IDs of this subtree  for (auto e : subtreeID) {  strOfID += std::to\_string(e.first) + " ";  }  strOfID += idStr; // ID of this node  ioLetter.sig = SUCC;  returnMessage.rebuild(strOfID.c\_str(), strOfID.size());  parentSend.send((void\*)&ioLetter, sizeof(TLetter), ZMQ\_SNDMORE);  parentSend.send(returnMessage, 0);      parentSend.disconnect(std::string ("tcp://localhost:") + parentSndPort);  work = 0;    #ifdef DEBUG  std::cout << "ENDING\_SERVICE" << std::endl;  #endif    break;  case EXEC:  parentRecieve.recv(&message, 0);  threadSend.send(message, 0);  break;  case PING:  TLetter res;  res = { id, PING, SUCC };  message.rebuild(&res, sizeof(TLetter));  parentSend.send(message);  break;  }  //---------------------------//  // Redirect message to child //  //---------------------------//  } else {  auto node = subtreeID.find(ioLetter.id);  if (node == subtreeID.end()) {  ioLetter.id = id;  ioLetter.sig = FAIL;  parentSend.send(&ioLetter, sizeof(TLetter), 0);  continue;  }  std::string childID(std::to\_string(node->second));  zmq::message\_t msg(childID.c\_str(), childID.size());  childrenSend.send(msg, ZMQ\_SNDMORE);  msg.rebuild((void\*)&ioLetter, sizeof(TLetter));  if (ioLetter.commType == ADD || ioLetter.commType == EXEC) {  // In ADD and EXEC signal we will send additional frame,  // containing child id or two char arrays respectivly  childrenSend.send(msg, ZMQ\_SNDMORE);  parentRecieve.recv(&msg);  childrenSend.send(msg, 0);    } else {  // Otherwise it's just one frame;  childrenSend.send(msg, 0);  }  }  //-----------------------------//  // Message recieved from child //  //-----------------------------//  } else if (items[1].revents & ZMQ\_POLLIN) {  childrenRecieve.recv(&message, ZMQ\_DONTWAIT);  TLetter ioLetter = \*(static\_cast<TLetter\*> (message.data()) );  if (ioLetter.commType == ADD && ioLetter.sig == SUCC) {  // Recieve the id of the child, adjacent to this node  zmq::message\_t idMsg;  childrenRecieve.recv(&idMsg);  std::string childIdStr((char\*)idMsg.data(), idMsg.size());  // Map created id and childs id  subtreeID.insert( std::pair<TNodeID, TNodeID> ( ioLetter.id, std::atoi(childIdStr.c\_str()) ) );    //Resend signal and id of this node to its parent  idMsg.rebuild(idStr.c\_str(), idStr.size());  parentSend.send(message, ZMQ\_SNDMORE);  parentSend.send(idMsg, 0);    } else if (ioLetter.commType == DEL && ioLetter.sig == SUCC) {  zmq::message\_t idStringMsg;  childrenRecieve.recv(&idStringMsg);  std::string strOfID((char\*)idStringMsg.data(), idStringMsg.size());  std::stringstream IDstream(strOfID);  // Delete all ids in the map of this node  TNodeID ID;  while (IDstream >> ID) {  subtreeID.erase(ID);  }  // Continue sending  parentSend.send(message, ZMQ\_SNDMORE);  parentSend.send(idStringMsg, 0);  } else {  // There should never be EXEC signal,  // DEL or ADD with FAIL, PING shouldn't countain  // additional messages. Simply resend.  parentSend.send(message, 0);  }  }  }  execThread.join();  } |

*controller.cpp*

|  |
| --- |
| #include <iostream>  #include <string>  #include <map>  #include <memory>  #include <unistd.h>  #include <sstream>  #include "lexicon.hpp"  // #define DEBUG  void CheckMailbox(zmq::socket\_t& execRecieve) {  zmq::message\_t msg;  while (execRecieve.recv(&msg)) {  TLetter inLetter = \*( static\_cast<TLetter\*> (msg.data()) );  if (inLetter.commType == EXEC){  if (inLetter.sig == SUCC) {  execRecieve.recv(&msg);  std::string report((char\*)msg.data(), msg.size());  std::cout << "MAILBOX: ID " << inLetter.id << " reports: " << report << std::endl;  } else {  std::cout << "ID " << inLetter.id << " reports: " << (inLetter.sig == FAIL ? "failure" : "error") << std::endl;  }  } else {  std::cout << "MAILBOX: Broken IO message" << std::endl;  }  }  }  int main() {  zmq::context\_t context(1);  //---------------//  // Port creation //  //---------------//  zmq::socket\_t treeSend (context, ZMQ\_ROUTER);  treeSend.setsockopt(ZMQ\_ROUTER\_HANDOVER, 1);  zmq::socket\_t treeRecieve(context, ZMQ\_PULL);  treeRecieve.setsockopt(ZMQ\_RCVTIMEO, TIMEOUT\_MS);    zmq::socket\_t execRecieve(context, ZMQ\_PULL);  execRecieve.setsockopt(ZMQ\_RCVTIMEO, EXEC\_TIMEOUT\_MS);  std::string outPort, inPort, execPort;  try {  #ifdef DEBUG  treeSend.bind("tcp://127.0.0.1:44050");  treeRecieve.bind("tcp://127.0.0.1:44051");  execRecieve.bind("tcp://127.0.0.1:44052");  #else  treeSend.bind("tcp://127.0.0.1:\*");  treeRecieve.bind("tcp://127.0.0.1:\*");  execRecieve.bind("tcp://127.0.0.1:\*");  #endif  } catch (zmq::error\_t& err) {  std::cerr << "FATAL ERROR: Could not create one or multiple sockets" << std::endl;  return err.num();  }  outPort = ParsePort(treeSend);  inPort = ParsePort(treeRecieve);  execPort = ParsePort(execRecieve);  //----------------//  // Main loop vars //  //----------------//  int id, parentID;  std::string command;  std::map<TNodeID, TNodeID> subtreeID;  #ifdef DEBUG  std::cout << "OUT: " << outPort << " | IN: " << inPort << " | EXEC: " << execPort << std::endl;  #endif    bool work = true;  while (work) {  std::cout << "► ";  std::cin >> command;  if (command == "create") {  std::cin >> id >> parentID;  if (id < 0) {  std::cout << "New ID must be positive" << std::endl;  continue;  } else if (parentID < -1) {  std::cout << "ERROR: Invalid parent ID" << std::endl;  continue;  } else {  auto node = subtreeID.find(id);  if (node != subtreeID.end()) {  std::cout << "ERROR: ID exists" << std::endl;  continue;  }  }  if (parentID == -1) {  #ifdef DEBUG  pid\_t pid = 1;  #else  pid\_t pid = fork();  #endif  if (pid < 0) {  std::cerr << "ERROR: Failed to fork" << std::endl;  continue;  } else if (pid == 0) {  std::string idStr( std::to\_string(id) );  // WHICH FUCKING DUMBCUNT DECIDED "oh let's not make the name implicit like fucking shell" FUCK  // Total hours wasted here: 7  if ( execl(WORKER\_NAME, WORKER\_NAME, idStr.c\_str(), outPort.c\_str(), inPort.c\_str(), execPort.c\_str(), NULL) < 0 ) {  std::cerr << "ERROR: Failed to create new process" << std::endl;  continue;  }  } else if (pid == 1) {  std::cout << "CONTROLLER: Manual server creation mode enabled" << std::endl;  }  } else {  // Auto type : std::map<TNodeID, TNodeID>::iterator  auto node = subtreeID.find(parentID);  if (node == subtreeID.end()) {  std::cout << "ERROR: ID not found" << std::endl;  continue;  }  // Tell parent to work  TLetter outLetter = {parentID, ADD, START};  zmq::message\_t msg(&outLetter, sizeof(TLetter));    std::string childIdStr(std::to\_string(node->second)); // Child of controller that has the parentID  treeSend.send(childIdStr.c\_str(), childIdStr.size(), ZMQ\_SNDMORE);  treeSend.send(msg, ZMQ\_SNDMORE);  // Add new child ID  childIdStr = std::to\_string(id);  treeSend.send(childIdStr.c\_str(), childIdStr.size(), 0);  }  #ifdef DEBUG  std::string waitInput;  std::cout << "ARBITRARY\_STRING\_INPUT\_WAIT: ";  std::cin >> waitInput;  #endif  zmq::message\_t msg;  if ( treeRecieve.recv(&msg, 0) ) {  TLetter res = \*(static\_cast<TLetter\*> (msg.data()));  if (parentID != -1) {  // Expect another ID message from node  treeRecieve.recv(&msg, 0);  }  if ( res.id == id && res.sig == SUCC) {  std::cout << "ID " << res.id << " reports: success" << std::endl;  if (parentID == -1) {  subtreeID.insert(std::pair<TNodeID, TNodeID>(id, id));  } else {  auto node = subtreeID.find(parentID);  subtreeID.insert(std::pair<TNodeID, TNodeID>(id, node->second));  }    } else if (res.sig == FAIL || res.sig == ERR) {  std::cout << "ID " << res.id << " reports: " << (res.sig == FAIL ? "failure" : "error") << std::endl;  } else {  std::cout << "CONTROLLER: Broken IO message" << std::endl;  }  } else {  std::cout << "Node is not responding" << std::endl;  }  } else if (command == "remove") {  std::cin >> id;  auto node = subtreeID.find(id);  if (node == subtreeID.end()) {  std::cout << "ERROR: ID not found" << std::endl;  continue;  }  TLetter ioLetter = {id, DEL, START};  zmq::message\_t msg ((void\*)&ioLetter, sizeof(TLetter));  std::string childID(std::to\_string(node->second));  treeSend.send(childID.c\_str(), childID.size(), ZMQ\_SNDMORE);  treeSend.send(msg, 0);  if ( treeRecieve.recv(&msg, 0) ) {  ioLetter = \*(static\_cast<TLetter\*> (msg.data()));  if ( ioLetter.id == id && ioLetter.sig == SUCC) {  std::cout << "ID " << ioLetter.id << " reports: success" << std::endl;  zmq::message\_t idStringMsg;  treeRecieve.recv(&idStringMsg);  std::string strOfID((char\*)idStringMsg.data(), idStringMsg.size());  std::stringstream IDstream(strOfID);  TNodeID ID;  while (IDstream >> ID) {  subtreeID.erase(ID);  }  std::cout << "Subtree deleted" << std::endl;  } else if (ioLetter.sig == FAIL || ioLetter.sig == ERR) {  std::cout << "ID " << ioLetter.id << " reports: " << (ioLetter.sig == FAIL ? "failure" : "error") << std::endl;  } else {  std::cout << "CONTROLLER: Broken IO message" << std::endl;  }  } else {  std::cout << "Node is not responding" << std::endl;  }  } else if (command == "exec") {  std::cin >> id;  if (id < 0) {  std::cout << "ERROR: Incorrect ID" << std::endl;  continue;  }  auto node = subtreeID.find(id);  if (node == subtreeID.end()) {  std::cout << "ERROR: ID not found" << std::endl;  continue;  }    TLetter ioLetter = {id, EXEC, START};  zmq::message\_t msg ((void\*)&ioLetter, sizeof(TLetter));  std::string childID(std::to\_string(node->second));  treeSend.send(childID.c\_str(), childID.size(), ZMQ\_SNDMORE);  treeSend.send(msg, ZMQ\_SNDMORE);  std::string pattern, text;  std::cin >> text >> pattern;  treeSend.send(std::string(pattern + " " + text).c\_str(), pattern.size() + 1 + text.size());  CheckMailbox(execRecieve);  } else if (command == "ping") {  std::cin >> id;  if (id < 0) {  std::cout << "ERROR: Incorrect ID" << std::endl;  continue;  }  auto node = subtreeID.find(id);  if (node == subtreeID.end()) {  std::cout << "ERROR: ID not found" << std::endl;  continue;  }  TLetter ioLetter = {id, PING, START};  CommandRoute(treeSend, node->second, ioLetter);  zmq::message\_t msg;  if (treeRecieve.recv(&msg, 0)) {  ioLetter = \*(static\_cast<TLetter\*> (msg.data()));  if (ioLetter.id == id && ioLetter.sig == SUCC) {  std::cout << "ID " << ioLetter.id << " reports: success" << std::endl;    } else if (ioLetter.sig == FAIL || ioLetter.sig == ERR) {  std::cout << "ID " << ioLetter.id << " reports: " << (ioLetter.sig == FAIL ? "failure" : "error") << std::endl;  } else {  std::cout << "CONTROLLER: Broken IO message" << std::endl;  }  } else {  std::cout << "CONTROLLER: No response from node\n"  << "Delete node from the map? (y/n): "<< std::endl;  char ch;  do {  std::cin >> ch;  if (ch == 'y') {  subtreeID.erase(id);  }  else if (ch == 'n') { ; }  else  {  std::cout << "(y/n): ";  }  } while (ch != 'y' && ch != 'n');  }  } else if (command == "check") {  // Do nothing, check the exec mailbox  } else if (command == "exit") {  for (std::pair<const TNodeID, TNodeID>& e : subtreeID) {  if (e.first == e.second) {  std::string idStr(std::to\_string(e.second));  TLetter end {e.second, DEL, START};  zmq::message\_t endNote((void\*)&end, sizeof(TLetter), 0);  treeSend.send(idStr.c\_str(), idStr.size(), ZMQ\_SNDMORE);  treeSend.send(endNote, 0);  }  }  std::cout << "CONTROLLER: Ending service (EXEC mailbox will be checked once more)" << std::endl;  work = 0;  } else {  std::cout << "CONTROLLER: Not a command" << std::endl;  std::cin.ignore(256, '\n');  }  //--------------------//  // EXEC mailbox check //  //--------------------//  CheckMailbox(execRecieve);  }  } |

1. **Вывод**

Библиотека ZeroMQ упрощает создание собственного сервера, абстрагируя такие действия на нижнем уровне, как создание сокетов или очередей к ним, но оставляет за программистом руководство над топологиями и поддерживание рабочего состояние серверов.