Московский авиационный институт
(национальный исследовательский университет)
Институт № 8 «Информационные технологии и прикладная математика»

Лабораторная работа №6,7,8
по курсу «Операционные системы»

Студент: Минеева Светлана Алексеевна

Группа: М8О-210Б-21

Преподаватель: Миронов Е.С

Вариант №24
Оценка: ____
Дата: 19.12.2022
Подпись: _____

Москва, 2022

Содержание:

1. Цель работы
2. Задание
3. Вариант задания
4. Общие сведения о программе
5. Общий метод и алгоритм решения
6. Текст программы
7. Демонстрация работы программы
8. Вывод

1. Цель работы

Целью является приобретение практических навыков в:

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

2. Задание

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 Пример:
> create 10 5
Ok: 3128

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

Пример:

> remove 10

Ok

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
Пример:
Можно найти в описании конкретной команды, определенной вариантом задания.

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.

3. Вариант задания

Вариант №24:

Тип топологии:

Топология №2: Аналогично топологии 2, но узлы находятся в дереве общего вида.

Тип команд для вычислительных узлов:

Набор команд №4 (поиск подстроки в строке):

Формат команды:

> exec id
> text_string
> pattern_string
[result] – номера позиций, где найден образец, разделенный точкой с запятой

text_string — текст, в котором искать образец. Алфавит: [A-Za-z0-9]. Максимальная длина строки 10^8 символов.

pattern_string — образец

Пример:
> exec 10
> abracadabra

> abra
Ok:10:0;7
> exec 10
> abracadabra
> mmm
Ok:10:-1

Примечания: Выбор алгоритма поиска не важен.

Тип проверки доступности узлов:

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
Пример:
> ping 10
Ok: 1 // узел 10 доступен
> ping 17
Ok: 0 // узел 17 недоступен

Возможные сервера сообщений:

1. ZeroMQ
2. MSMQ
3. RabbitMQ
4. Nats

4. Общие сведения о программе

Программа состоит из шести файлов: control.cpp, calculation_node.cpp, search.cpp, search.hpp, topology.hpp, zmq_std.hpp.

Основные библиотечные вызовы в программе связаны с выбранной библиотекой для обмена сообщениями – ZeroMQ, которая была установлена на личный ноутбук для выполнения данной лабораторной работы.

5. Общий метод и алгоритм решения

Связь между вычислительными узлами будет поддерживаться с помощью ZMQ_PAIR. Я думаю, это наиболее подходящая связь для жесткого контроля небольшого количества узлов. При инициализации устанавливается время ожидания ZMQ_SNDTIMEO и ZMQ_RCVTIMEO, чтобы предусмотреть случай, когда дочерний процесс убит. Для обмена информацией будет использоваться специальная структура NodeToken,в которой есть перечислимое поле actions. Вычислительные узлы обрабатывают каждое сообщение: если идентификатор сообщения не совпадает с идентификатором узла, то он отправляет сообщение дальше и ждёт ответа снизу. Каждый вычислительный узел имеет отдельный поток для вычислений и свою очередь вычислений. Чтобы получить результат вычислений обратно, нужно запросить их от вычислительного узла. Такой подход необходим, потому что неизвестно, сколько нужно ждать результат от узла. Для поиска подстроки в строке будет использоваться алгоритм Кнута-Морриса-Пратта с препроцессингом через Z-функцию строки.

6. Текст программы

control.cpp

```
#include <unistd.h>
#include <vector>
#include <iostream>
#include <pthread.h>
#include <time.h>
#include "topology.hpp"
#include "zmq_std.hpp"

using NodeIdType = long long;

int main() {
  int parameter;
  topology <NodeIdType, nodes> ControlNode;
  std::vector <std::pair<void*, void*> > childs;
  std::vector <NodeIdType> ids;
  std::string string;
  NodeIdType id;

  while(std::cin >> string >> id) {
    if(string == "create") {
      NodeIdType ParentId;
      std::cin >> ParentId;

      if (ParentId == -1) {
        void* NewContext = NULL;
```

```
        void* NewSocket = NULL;
        ZmqStandard::InitPairSocket(NewContext, NewSocket);
        parameter = zmq_bind(NewSocket, ("tcp://*:" + std::to_string(PORT_BASE +
id)).c_str());
        assert(parameter == 0);
        int ForkId = fork();

        if(ForkId == 0) {
          parameter = execl(NODE_EXECUTABLE_NAME, NODE_EXECUTABLE_NAME,
std::to_string(id).c_str(), NULL);
          assert(parameter != -1);
          return 0;
        }
        else {
          bool state = true;
          NodeToken ReplyInfo({fail, id, id});
          state = ZmqStandard::RecieveMessageWait(ReplyInfo, NewSocket);
          NodeToken* token = new NodeToken({ping, id, id});
          NodeToken reply({fail, id, id});
          state = ZmqStandard::SendRecieveWait(token, reply, NewSocket);

          if(state and reply.action == success) {
            childs.push_back(std::make_pair(NewContext, NewSocket));
            ControlNode.insert(id);
            std::cout << "OK: " << ReplyInfo.id << std::endl;
            ids.push_back(id);
          }
          else {
            parameter = zmq_close(NewSocket);
            assert(parameter == 0);
            parameter = zmq_ctx_term(NewContext);
            assert(parameter == 0);
          }
        }
      }
      else if(ControlNode.find(ParentId) == -1) {
        std::cout << "Error: Not found" << std::endl;
      }
      else {
        if(ControlNode.find(id) != -1) {
          std::cout << "Error: Already exists" << std::endl;
        }
        else {
          int position = ControlNode.find(ParentId);
          NodeToken* token = new NodeToken({create, ParentId, id});
          NodeToken reply({fail, id, id});

          if(ZmqStandard::SendRecieveWait(token, reply, childs[position].second) and
reply.action == success) {
            std::cout << "OK: " << reply.id << std::endl;
            ids.push_back(id);
            ControlNode.insert(ParentId, id);
          }
```

```
          else {
            std::cout << "Error: Parent is unavailable" << std::endl;
          }
        }
      }
    }
    else if(string == "remove") {
      int position = ControlNode.find(id);

      if(position != -1) {
        NodeToken* token = new NodeToken({destroy, id, id});
        NodeToken reply({fail, id, id});
        bool state = ZmqStandard::SendRecieveWait(token, reply,
childs[position].second);

        if(reply.action == destroy and reply.ParentId == id) {
          parameter = zmq_close(childs[position].second);
          assert(parameter == 0);
          parameter = zmq_ctx_term(childs[position].first);
          assert(parameter == 0);
          std::vector <std::pair <void*, void*>>::iterator IteratorChild =
childs.begin();

          while(position--) {
            ++IteratorChild;
          }
          childs.erase(IteratorChild);
        }
        else if(reply.action == bind and reply.ParentId == id) {
          parameter = zmq_close(childs[position].second);
          assert(parameter == 0);
          parameter = zmq_ctx_term(childs[position].first);
          assert(parameter == 0);
          ZmqStandard::InitPairSocket(childs[position].first,
childs[position].second);
          parameter = zmq_bind(childs[position].second, ("tcp://*:" +
std::to_string(PORT_BASE + reply.id)).c_str());
          assert(parameter == 0);
        }
        if(state) {
          ControlNode.erase(id);

          for(auto iterator = ids.begin(); iterator != ids.end(); ++iterator) {
            if (*iterator == id){
              ids.erase(iterator);
              break;
            }
          }

          std::cout << "OK" << std::endl;
        }
        else {
          std::cout << "Error: Node is unavailable" << std::endl;
```

```
        }
      }
      else {
        std::cout << "Error: Not found" << std::endl;
      }
    }
    else if(string == "heartbit") {
      int position = 0;
      int count = 0;
      double start = 0, end = 0;
      start = clock();

      for(;;) {
        for(auto iterator = ids.begin(); iterator != ids.end(); ++iterator) {
          position = ControlNode.find(*iterator);

          if (position == -1) {
            std::cout << "Error: Not found" << *iterator << std::endl;
          }
          else {
            NodeToken* token = new NodeToken({ping, *iterator, *iterator});
            NodeToken reply({fail, *iterator, *iterator});

            if(ZmqStandard::SendRecieveWait(token, reply, childs[position].second) and
reply.action == success) {
              std::cout << "OK:" << *iterator << std::endl;
            }
            else {
              std::cout << "Heartbit: node" << *iterator << "is unavailable now" <<
std::endl;
            }
          }

          ++count;
        }

        end = clock();

        if(start > 4*id or end > 4*id) {
          break;
        }
      }
    }
    else if(string == "back") {
      int position = ControlNode.find(id);

      if(position != -1) {
        NodeToken* token = new NodeToken({back, id, id});
        NodeToken reply({fail, id, id});

        if(ZmqStandard::SendRecieveWait(token, reply, childs[position].second)) {
          if(reply.action == success) {
            NodeToken* TokenBack = new NodeToken({back, id, id});
```

```
            NodeToken ReplyBack({fail, id, id});
            std::vector <unsigned int> calculated;

            while(ZmqStandard::SendRecieveWait(TokenBack, ReplyBack,
childs[position].second) and ReplyBack.action == success) {
              calculated.push_back(ReplyBack.id);
            }

            if(calculated.empty()) {
              std::cout << "OK: " << reply.id << " : -1" << std::endl;
            }
            else {
              std::cout << "OK: " << reply.id << " : ";

              for(size_t index = 0; index < calculated.size() - 1; ++index) {
                std::cout << calculated[index] << ", ";
              }

              std::cout << calculated.back() << std::endl;
            }
          }
          else {
            std::cout << "Error: No calculations to back" << std::endl;
          }
        }
        else {
          std::cout << "Error: Node is unavailable" << std::endl;
        }
      }
      else {
        std::cout << "Error: Not found" << std::endl;
      }
    }
    else if(string == "exec") {
      std::string pattern, text;
      std::cin >> pattern >> text;
      int position = ControlNode.find(id);

      if(position != -1) {
        bool state = true;
        std::string TextPattern = pattern + SENTINEL + text + SENTINEL;

        for(size_t index = 0; index < TextPattern.size(); ++index) {
          NodeToken* token = new NodeToken({exec, TextPattern[index], id});
          NodeToken reply({fail, id, id});

          if(!ZmqStandard::SendRecieveWait(token, reply, childs[position].second) or
reply.action != success) {
            state = false;
            break;
          }
        }
```

```
        if(state) {
          std::cout << "OK" << std::endl;
        }
        else {
          std::cout << "Error: Node is unavailable" << std::endl;
        }
      }
      else {
        std::cout << "Error: Not found" << std::endl;
      }
    }
  }

  for(size_t index = 0; index < childs.size(); ++index) {
    parameter = zmq_close(childs[index].second);
    assert(parameter == 0);
    parameter = zmq_ctx_term(childs[index].first);
    assert(parameter == 0);
  }
}
```

calculation_node.cpp

```
#include <pthread.h>
#include <queue>
#include <tuple>
#include <list>
#include <unistd.h>
#include "search.hpp"
#include "zmq_std.hpp"

const std::string SENTINEL_STR = "$";

long long NodeId;
pthread_mutex_t mutex;
pthread_cond_t cond;
std::queue <std::pair <std::string, std::string>> CalculationQueue;
std::queue <std::list <unsigned int>> DoneQueue;

void* ThreadFunction(void*) {
  while(1) {
    pthread_mutex_lock(&mutex);

    while(CalculationQueue.empty()) {
      pthread_cond_wait(&cond, &mutex);
    }

    std::pair <std::string, std::string> current = CalculationQueue.front();
    CalculationQueue.pop();
    pthread_mutex_unlock(&mutex);

    if(current.first == SENTINEL_STR and current.second == SENTINEL_STR) {
      break;
```

```
    }
    else {
      std::vector <unsigned int> result = KMPFunction(current.first, current.second);
      std::list <unsigned int> ResultList;

      for (const unsigned int &element : result) {
        ResultList.push_back(element);
      }

      pthread_mutex_lock(&mutex);
      DoneQueue.push(ResultList);
      pthread_mutex_unlock(&mutex);
    }
  }

  return NULL;
}

int main(int ArgumentCount, char** ArgumentVector) {
  int parameter;
  assert(ArgumentCount == 2);
  NodeId = std::stoll(std::string(ArgumentVector[1]));

  void* NodeParentContext = zmq_ctx_new();
  void* NodeParentSocket = zmq_socket(NodeParentContext, ZMQ_PAIR);
  parameter = zmq_connect(NodeParentSocket, ("tcp://localhost:" +
std::to_string(PORT_BASE + NodeId)).c_str());
  assert(parameter == 0);
  long long ChildId = -1;
  void* NodeContext = NULL;
  void* NodeSocket = NULL;

  pthread_t CalculationThread;
  parameter = pthread_mutex_init(&mutex, NULL);
  assert(parameter == 0);
  parameter = pthread_cond_init(&cond, NULL);
  assert(parameter == 0);
  parameter = pthread_create(&CalculationThread, NULL, ThreadFunction, NULL);
  assert(parameter == 0);

  std::string pattern, text;
  bool FlagSentinel = true;

  NodeToken* InformationToken = new NodeToken({info, getpid(), getpid()});
  ZmqStandard::send_msg_dontwait(InformationToken, NodeParentSocket);

  std::list<unsigned int> CurrentCalculated;

  bool HavingChild = false;
  bool awake = true;
  bool calculate = true;

  while(awake) {
```

```
    NodeToken token;
    ZmqStandard::RecieveMessage(token, NodeParentSocket);
    NodeToken* reply = new NodeToken({fail, NodeId, NodeId});

    if(token.action == back) {
      if(token.id == NodeId) {
        if(calculate) {
          pthread_mutex_lock(&mutex);

          if(DoneQueue.empty()) {
            reply->action = exec;
          }
          else {
            CurrentCalculated = DoneQueue.front();
            DoneQueue.pop();
            reply->action = success;
            reply->id = getpid();
          }

          pthread_mutex_unlock(&mutex);
          calculate = false;
        }
        else {
          if(CurrentCalculated.size() > 0) {
            reply->action = success;
            reply->id = CurrentCalculated.front();
            CurrentCalculated.pop_front();
          }
          else {
            reply->action = exec;
            calculate = true;
          }
        }
      }
      else {
        NodeToken* TokenDown = new NodeToken(token);
        NodeToken ReplyDown(token);
        ReplyDown.action = fail;

        if(ZmqStandard::SendRecieveWait(TokenDown, ReplyDown, NodeSocket) and
ReplyDown.action == success) {
          *reply = ReplyDown;
        }
      }
    }
    else if(token.action == bind and token.ParentId == NodeId) {
      ZmqStandard::InitPairSocket(NodeContext, NodeSocket);
      parameter = zmq_bind(NodeSocket, ("tcp://*:" + std::to_string(PORT_BASE +
token.id)).c_str());
      assert(parameter == 0);
      HavingChild = true;
      ChildId = token.id;
      NodeToken* TokenPing = new NodeToken({ping, ChildId, ChildId});
```

```
      NodeToken ReplayPing({fail, ChildId, ChildId});

      if(ZmqStandard::SendRecieveWait(TokenPing, ReplayPing, NodeSocket) and
ReplayPing.action == success) {
        reply->action = success;
      }
    }
    else if(token.action == create) {
      if(token.ParentId == NodeId) {
        if(HavingChild) {
          parameter = zmq_close(NodeSocket);
          assert(parameter == 0);
          parameter = zmq_ctx_term(NodeContext);
          assert(parameter == 0);
        }

        ZmqStandard::InitPairSocket(NodeContext, NodeSocket);
        parameter = zmq_bind(NodeSocket, ("tcp://*:" + std::to_string(PORT_BASE +
token.id)).c_str());
        assert(parameter == 0);
        int ForkId = fork();

        if(ForkId == 0) {
          parameter = execl(NODE_EXECUTABLE_NAME, NODE_EXECUTABLE_NAME,
std::to_string(token.id).c_str(), NULL);
          assert(parameter != -1);
          return 0;
        }
        else {
          bool state = true;
          NodeToken ReplyInfo({fail, token.id, token.id});
          state = ZmqStandard::RecieveMessageWait(ReplyInfo, NodeSocket);

          if(ReplyInfo.action != fail) {
            reply->id = ReplyInfo.id;
            reply->ParentId = ReplyInfo.ParentId;
          }

          if(HavingChild) {
            NodeToken* TokenBind = new NodeToken({bind, token.id, ChildId});
            NodeToken ReplayBind({fail, token.id, token.id});
            state = ZmqStandard::SendRecieveWait(TokenBind, ReplayBind, NodeSocket);
            state = state and (ReplayBind.action == success);
          }

          if(state) {
            NodeToken* TokenPing = new NodeToken({ping, token.id, token.id});
            NodeToken ReplayPing({fail, token.id, token.id});
            state = ZmqStandard::SendRecieveWait(TokenPing, ReplayPing, NodeSocket);
            state = state and (ReplayPing.action == success);

            if(state) {
              reply->action = success;
```

```
              ChildId = token.id;
              HavingChild = true;
            }
            else {
              parameter = zmq_close(NodeSocket);
              assert(parameter == 0);
              parameter = zmq_ctx_term(NodeContext);
              assert(parameter == 0);
            }
          }
        }
      }
      else if(HavingChild) {
        NodeToken* TokenDown = new NodeToken(token);
        NodeToken ReplyDown(token);
        ReplyDown.action = fail;

        if (ZmqStandard::SendRecieveWait(TokenDown, ReplyDown, NodeSocket) and
ReplyDown.action == success) {
          *reply = ReplyDown;
        }
      }
    }
    else if(token.action == ping) {
      if(token.id == NodeId) {
        reply->action = success;
      }
      else if(HavingChild) {
        NodeToken* TokenDown = new NodeToken(token);
        NodeToken ReplyDown(token);
        ReplyDown.action = fail;

        if (ZmqStandard::SendRecieveWait(TokenDown, ReplyDown, NodeSocket) and
ReplyDown.action == success) {
          *reply = ReplyDown;
        }
      }
    }
    else if(token.action == destroy) {
      if(HavingChild) {
        if(token.id == ChildId) {
          bool state = true;
          NodeToken* TokenDown = new NodeToken({destroy, NodeId, ChildId});
          NodeToken ReplyDown({fail, ChildId, ChildId});
          state = ZmqStandard::SendRecieveWait(TokenDown, ReplyDown, NodeSocket);

          if(ReplyDown.action == destroy and ReplyDown.ParentId == ChildId) {
            parameter = zmq_close(NodeSocket);
            assert(parameter == 0);
            parameter = zmq_ctx_term(NodeContext);
            assert(parameter == 0);
            HavingChild = false;
            ChildId = -1;
```

```
          }
          else if(ReplyDown.action == bind and ReplyDown.ParentId == NodeId) {
            parameter = zmq_close(NodeSocket);
            assert(parameter == 0);
            parameter = zmq_ctx_term(NodeContext);
            assert(parameter == 0);
            ZmqStandard::InitPairSocket(NodeContext, NodeSocket);
            parameter = zmq_bind(NodeSocket, ("tcp://*:" + std::to_string(PORT_BASE +
ReplyDown.id)).c_str());
            assert(parameter == 0);
            ChildId = ReplyDown.id;
            NodeToken* TokenPing = new NodeToken({ping, ChildId, ChildId});
            NodeToken ReplayPing({fail, ChildId, ChildId});

            if(ZmqStandard::SendRecieveWait(TokenPing, ReplayPing, NodeSocket) and
ReplayPing.action == success) {
              state = true;
            }
          }
          if(state) {
            reply->action = success;
          }
        }
        else if(token.id == NodeId) {
          parameter = zmq_close(NodeSocket);
          assert(parameter == 0);
          parameter = zmq_ctx_term(NodeContext);
          assert(parameter == 0);
          HavingChild = false;
          reply->action = bind;
          reply->id = ChildId;
          reply->ParentId = token.ParentId;
          awake = false;
        }
        else {
          NodeToken* TokenDown = new NodeToken(token);
          NodeToken ReplyDown(token);
          ReplyDown.action = fail;

          if(ZmqStandard::SendRecieveWait(TokenDown, ReplyDown, NodeSocket) and
ReplyDown.action == success) {
            *reply = ReplyDown;
          }
        }
      }
      else if(token.id == NodeId) {
        reply->action = destroy;
        reply->ParentId = NodeId;
        reply->id = NodeId;
        awake = false;
      }
    }
    else if(token.action == exec) {
```

```
      if(token.id == NodeId) {
        char symbol = token.ParentId;

        if(symbol == SENTINEL) {
          if(FlagSentinel) {
            std::swap(text, pattern);
          }
          else {
            pthread_mutex_lock(&mutex);

            if(CalculationQueue.empty()) {
              pthread_cond_signal(&cond);
            }

            CalculationQueue.push({pattern, text});
            pthread_mutex_unlock(&mutex);
            text.clear();
            pattern.clear();
          }

          FlagSentinel = FlagSentinel ^ 1;
        }
        else {
          text = text + symbol;
        }

        reply->action = success;
      }
      else if(HavingChild) {
        NodeToken* TokenDown = new NodeToken(token);
        NodeToken ReplyDown(token);
        ReplyDown.action = fail;

        if(ZmqStandard::SendRecieveWait(TokenDown, ReplyDown, NodeSocket) and
ReplyDown.action == success) {
          *reply = ReplyDown;
        }
      }
    }

    ZmqStandard::send_msg_dontwait(reply, NodeParentSocket);
  }

  if(HavingChild) {
    parameter = zmq_close(NodeSocket);
    assert(parameter == 0);
    parameter = zmq_ctx_term(NodeContext);
    assert(parameter == 0);
  }

  parameter = zmq_close(NodeParentSocket);
  assert(parameter == 0);
  parameter = zmq_ctx_term(NodeParentContext);
```

```
  assert(parameter == 0);
  pthread_mutex_lock(&mutex);

  if(CalculationQueue.empty()) {
    pthread_cond_signal(&cond);
  }

  CalculationQueue.push({SENTINEL_STR, SENTINEL_STR});
  pthread_mutex_unlock(&mutex);

  parameter = pthread_join(CalculationThread, NULL);
  assert(parameter == 0);

  parameter = pthread_cond_destroy(&cond);
  assert(parameter == 0);
  parameter = pthread_mutex_destroy(&mutex);
  assert(parameter == 0);
}
```

search.cpp

```
#include "search.hpp"

std::vector <unsigned int> ZFunction(const std::string &string) {
  unsigned int length = string.size();
  std::vector <unsigned int> ZFunctionVector(length);
  unsigned int ThirdIndex = 0, FourthIndex = 0;

  for(unsigned int index = 1; index < length; ++index) {
    if(index <= FourthIndex) {
      ZFunctionVector[index] = std::min(ZFunctionVector[index - ThirdIndex],
FourthIndex - index);
    }

    while(index + ZFunctionVector[index] < length and string[index +
ZFunctionVector[index]] == string[ZFunctionVector[index]]) {
      ++ZFunctionVector[index];
    }

    if(index + ZFunctionVector[index] > FourthIndex) {
      ThirdIndex = index;
      FourthIndex = index + ZFunctionVector[index];
    }
  }

  return ZFunctionVector;
}

std::vector <unsigned int> PrefixFunction(const std::string &string) {
  std::vector <unsigned int> ZFunctionVector = ZFunction(string);
  unsigned int length = string.size();
  std::vector <unsigned int> PrefixVector(length);
```

```
  for(unsigned int index = length - 1; index > 0; --index) {
    PrefixVector[index + ZFunctionVector[index] - 1] = ZFunctionVector[index];
  }

  return PrefixVector;
}

std::vector <unsigned int> KMPFunction(const std::string &pattern, const std::string
&text) {
  std::vector <unsigned int> vector = PrefixFunction(pattern);
  unsigned int PatternSize = pattern.size();
  unsigned int length = text.size();
  unsigned int index = 0;
  std::vector <unsigned int> carrier;

  if(PatternSize > length) {
    return carrier;
  }

  while(index < length - PatternSize + 1) {
    unsigned int SecondIndex = 0;

    while(SecondIndex < PatternSize and pattern[SecondIndex] == text[index +
SecondIndex]) {
      ++SecondIndex;
    }

    if(SecondIndex == PatternSize) {
      carrier.push_back(index);
    }
    else {
      if(SecondIndex > 0 and SecondIndex > vector[SecondIndex - 1]) {
        index = index + SecondIndex - vector[SecondIndex - 1] - 1;
      }
    }

    ++index;
  }

  return carrier;
}
```

search.hpp

```
#ifndef SEARCH_HPP
#define SEARCH_HPP

#include <string>
#include <vector>

std::vector <unsigned int> ZFunction(const std::string &string);
std::vector <unsigned int> PrefixFunction(const std::string &string);
```

```
std::vector <unsigned int> KMPFunction(const std::string &pattern, const std::string
&text);

#endif
```

topology.hpp

```
#ifndef TOPOLOGY_HPP
#define TOPOLOGY_HPP

#include <iostream>
#include <list>
#include <memory>
#include <utility>

class nodes {
  private:
    std::pair <long long, std::list <nodes>> IdNode;
    using iterator = typename std::list<nodes>::iterator;

  public:
    explicit nodes() noexcept : IdNode() {}

    nodes(const long long &element) {
      IdNode.first = element;
    }

    bool insert(const long long &parent, const long long &element) {
      for (iterator iterators = IdNode.second.begin(); iterators !=
IdNode.second.end(); ++iterators) {
        if ((*iterators).GetId() == parent) {
          nodes NewNode(element);
          (*iterators).IdNode.second.push_back(NewNode);
          return true;
        }
        else if(!(*iterators).EmptyNode()) {
            if((*iterators).insert(parent,element)) {
              return true;
            }
        }
      }

      return false;
    }

    bool insert(const long long &element) {
      this->IdNode.second.push_back(element);
      return true;
    }

    bool erase(const long long &element) {
      for (iterator iterators = this->IdNode.second.begin(); iterators != this-
>IdNode.second.end(); ++iterators) {
```

```
        if ((*iterators).GetId() == element) {
          if (!((*iterators).EmptyNode())) {
            long long id1 = (*iterators).LastElementDelete();
            (*iterators).SetId(id1);
            return true;
          }
          else {
            this->IdNode.second.erase(iterators);
            return true;
          }
        }
        else if(not((*iterators).EmptyNode())) {
          if ((*iterators).erase(element)) {
            return true;
          }
        }
      }
      return false;
    }

    long long GetId() {
      return this->IdNode.first;
    }

    void SetId(const long long &element) {
      this->IdNode.first = element;
    }

    long long LastElementDelete() {
      if(this->IdNode.second.front().EmptyNode()) {
        long long id1 = this->IdNode.second.front().GetId();
        this->IdNode.second.pop_front();
        return id1;
      }
      else {
        return this->IdNode.second.front().LastElementDelete();
      }
    }

    long long find(const long long &element) {
      long long position = 0;

      for(iterator iterators = this->IdNode.second.begin(); iterators != this-
>IdNode.second.end(); ++iterators) {
        if ((*iterators).GetId() == element) {
          return position;
        }
        else if(!(*iterators).EmptyNode()) {
          if ((*iterators).find(element) == 0) {
            return position;
          }
        }
      }
```

```
      return -1;
    }

    bool EmptyNode() {
      return this->IdNode.second.empty();
    }

    ~nodes(){}
};

template<class L, class N>
class topology {
  private:
    std::shared_ptr <N> root;
    size_t ContainerSize;

  public:
    explicit topology() noexcept : root(nullptr), ContainerSize(0) {}
    ~topology() {}

    bool insert(const L &parent, const L &element) {
      if(this->root != nullptr) {
        if((*root).GetId() == parent) {
          return (*root).insert(element);
        }
        else {
          return (*root).insert(parent, element);
        }
      }

      return false;
    }

    void insert(const L &element) {
      if(root == nullptr) {
        N NewNode(element);
        root = std::make_shared<N>(NewNode);
      }
      else {
        std::cout << "ERROR: Root exists" << std::endl;
      }
    }

    long long find(const L &element) {
      long long position = 0;

      if (this->root != nullptr) {
        if((*root).GetId() == element) {
          return position;
        }
        else {
          return (*root).find(element);
```

```
        }
      }

      return -1;
    }

    bool erase(const L &element) {
      if (this->root != nullptr) {
        if(((*root).GetId() == element) and (not((*root).EmptyNode()))) {
          long long id1 = (*root).LastElementDelete();
          (*root).SetId(id1);
          return true;
        }
        else if(((*root).GetId() == element) and ((*root).EmptyNode())) {
          this->root = nullptr;
          return true;
        }
        else if(((*root).GetId() != element) and (not((*root).EmptyNode()))) {
          return (*root).erase(element);
        }
      }
      return false;
    }

};

#endif
```

zmq_std.hpp

```
#ifndef ZMQ_STD_HPP
#define ZMQ_STD_HPP

#include <assert.h>
#include <errno.h>
#include <string.h>
#include <string>
#include "zmq.h"

const char* NODE_EXECUTABLE_NAME = "calculation";
const char SENTINEL = '$';
const int PORT_BASE = 8000;
const int WAIT_TIME = 1000;

enum actions {
  fail = 0,
  success = 1,
  create = 2,
  destroy = 3,
  bind = 4,
  ping = 5,
  exec = 6,
  info = 7,
```

```
  back = 8
};

struct NodeToken {
  actions action;
  long long ParentId, id;
};

namespace ZmqStandard {
  void InitPairSocket(void* &context, void* &socket) {
    int parameter;
    context = zmq_ctx_new();
    socket = zmq_socket(context, ZMQ_PAIR);
    parameter = zmq_setsockopt(socket, ZMQ_RCVTIMEO, &WAIT_TIME, sizeof(int));
    assert(parameter == 0);
    parameter = zmq_setsockopt(socket, ZMQ_SNDTIMEO, &WAIT_TIME, sizeof(int));
    assert(parameter == 0);
  }

  template <class Class>
  void RecieveMessage(Class &ReplyData, void* socket) {
    int parameter = 0;
    zmq_msg_t reply;
    zmq_msg_init(&reply);
    parameter = zmq_msg_recv(&reply, socket, 0);
    assert(parameter == sizeof(Class));
    ReplyData = *(Class*)zmq_msg_data(&reply);
    parameter = zmq_msg_close(&reply);
    assert(parameter == 0);
  }

  template <class Class>
  void SendMessage(Class* token, void* socket) {
    int parameter = 0;
    zmq_msg_t message;
    zmq_msg_init(&message);
    parameter = zmq_msg_init_size(&message, sizeof(Class));
    assert(parameter == 0);
    parameter = zmq_msg_init_data(&message, token, sizeof(Class), NULL, NULL);
    assert(parameter == 0);
    parameter = zmq_msg_send(&message, socket, 0);
    assert(parameter == sizeof(Class));
  }

  template <class Class>
  bool send_msg_dontwait(Class* token, void* socket) {
    int parameter;
    zmq_msg_t message;
    zmq_msg_init(&message);
    parameter = zmq_msg_init_size(&message, sizeof(Class));
    assert(parameter == 0);
    parameter = zmq_msg_init_data(&message, token, sizeof(Class), NULL, NULL);
    assert(parameter == 0);
```

```
  parameter = zmq_msg_send(&message, socket, ZMQ_DONTWAIT);

  if (parameter == -1) {
    zmq_msg_close(&message);
    return false;
  }

  assert(parameter == sizeof(Class));
  return true;
}

template <class Class>
bool RecieveMessageWait(Class &ReplyData, void* socket) {
  int parameter = 0;
  zmq_msg_t reply;
  zmq_msg_init(&reply);
  parameter = zmq_msg_recv(&reply, socket, 0);

  if (parameter == -1) {
    zmq_msg_close(&reply);
    return false;
  }

  assert(parameter == sizeof(Class));
  ReplyData = *(Class*)zmq_msg_data(&reply);
  parameter = zmq_msg_close(&reply);
  assert(parameter == 0);
  return true;
}

template <class Class>
bool SendMessageWait(Class* token, void* socket) {
  int parameter;
  zmq_msg_t message;
  zmq_msg_init(&message);
  parameter = zmq_msg_init_size(&message, sizeof(Class));
  assert(parameter == 0);
  parameter = zmq_msg_init_data(&message, token, sizeof(Class), NULL, NULL);
  assert(parameter == 0);
  parameter = zmq_msg_send(&message, socket, 0);

  if (parameter == -1) {
    zmq_msg_close(&message);
    return false;
  }

  assert(parameter == sizeof(Class));
  return true;
}

template <class Class>
bool SendRecieveWait(Class* TokenSend, Class & TokenReply, void* socket) {
  if (SendMessageWait(TokenSend, socket)) {
```

```
      if (RecieveMessageWait(TokenReply, socket)) {
        return true;
      }
      else {
        return false;
      }
    }
    else {
      return false;
    }
  }
}

#endif
```

7. Демонстрация работы программы

MacBook-Pro-MacBook:~ macbookpro$ g++ -g -O2 -pedantic -pthread -std=c++17 -Wall -Werror -Wextra -c search.cpp
MacBook-Pro-MacBook:~ macbookpro$ g++ -g -O2 -pedantic -pthread -std=c++17 -Wall -Werror -Wextra -c calculation_node.cpp
MacBook-Pro-MacBook:~ macbookpro$ g++ -g -O2 -pedantic -pthread -std=c++17 -Wall -Werror -Wextra control.cpp -lzmq -o control
MacBook-Pro-MacBook:~ macbookpro$ g++ -g -O2 -pedantic -pthread -std=c++17 -Wall -Werror -Wextra calculation_node.o search.o -lzmq -o calculation
MacBook-Pro-MacBook:~ macbookpro$ ./control |cat>protocol
create 1 -1
create 2 1
create 3 2
create 4 2
exec 4 aba abaabacaba
exec 2 ab abaabacaba
exec 1 a abaabacaba
exec 3 abaabacaba abaabacaba
ping 1
ping 2
ping 3
ping 4
back 1
back 2
back 3
back 4
remove 4
remove 1
remove 3
remove 2
MacBook-Pro-MacBook:~ macbookpro$ cat protocol
OK: 2400
OK: 2401
OK: 2402

OK: 2403
OK
OK
OK
OK
OK: 2400 : 0, 2, 3, 5, 7, 9
OK: 2401 : 0, 3, 7
OK: 2402 : 0
OK: 2403 : 0, 3, 7
OK
OK
OK
OK
MacBook-Pro-MacBook:~ macbookpro$ ./control |cat>protocol
create 1 -1
create 5 1
create 7 5
create 3 1
create 6 5
create 8 7
create 2 1
create 4 3
create 9 8
heartbit 100
remove 5
remove 2
remove 3
remove 8
remove 7
remove 1
remove 4
remove 6
remove 9
MacBook-Pro-MacBook:~ macbookpro$ cat protocol
OK: 2349
OK: 2350
OK: 2351
OK: 2352
OK: 2353
OK: 2354
OK: 2355
OK: 2356
OK: 2357
OK:1
OK:5
OK:7
OK:3
OK:6
OK:8
OK:2
OK:4
OK:9

OK
OK
OK
OK
OK
OK
OK
OK
OK
MacBook-Pro-MacBook:~ macbookpro$

8. Вывод

В данной лабораторной работе я изучила основы работы с очередями сообщений ZeroMQ и реализовала программу с использованием этой библиотеки. Также я приобрела практические навыки в управлении серверами сообщений, применении отложенных вычислений и интеграции программных систем друг с другом. Для достижения отказоустойчивости был выбран и изучен ZMQ_PAIR, так как он лучше всего подходит для данной работы. ZMQ обладает высокой производительностью и пропускной способностью, но не поддерживает сохранение сообщений, от чего, в общем, и имеет большую эффективность по времени работы в сравнении с другими серверами сообщений. Данная лабораторная работа является крайне полезной, ведь очереди сообщений используются для взаимодействия нескольких машин в одной большой сети и опыт работы с ZeroMQ может пригодиться мне в будущем при настройке собственной системы распределённых вычислений.