Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»
Дисциплина «Операционные системы»

Лабораторные работы №6-8

Тема: Управление серверами сообщений,
применение отложенный вычислений,
интеграция программных систем друг с другом

Студент: Инютин М. А.
Группа: М8О-207Б-19
Преподаватель: Миронов. Е. С.
Дата:
Оценка:

Москва, 2020

1. Постановка задачи

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать два 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;
- Удаление существующего вычислительного узла;
- Исполнение команды на вычислительном узле;
- Проверка доступности вычислительного узла.

Вариант 11. Все вычислительные узлы находятся в списке. Есть только один управляющий узел.

Исполнение команды — поиск подстроки в строке.

Команда проверки — проверка доступности конкретного узла.

2. Описание программы

Связь между вычислительными узлами будем поддерживать с помощью *ZMQ_PAIR*. При инициализации установить время ожидания *ZMQ_SNDTIMEO* и *ZMQ_RECVTIMEO*, чтобы предусмотреть случай, когда дочерний процесс был убит. Для обмена информацией будем использовать специальную структуру *node_token_t*, 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 я использовал алгоритм Кнута-Морриса-Пратта с препроцессингом через Z-функцию строки.

3. Набор тестов

Программа обрабатывает команды пользователя до окончания ввода.

Тест 1

```
create 1 -1
ping 1
exec 1 a b
exec 1 a aaa
exec 1 abra abracadabra
back 1
ping 1
back 1
ping 1
back 1
ping 1
remove 1
```

Тест 2

```
create 1 -1
create 2 -1
create 3 -1
create 4 -1
create 5 -1
exec 4 aba abacababacaba
exec 1 aba abacababacaba
exec 5 aba abacababacaba
exec 1 aba abacababacaba
exec 3 aba abacababacaba
ping 1
ping 2
ping 3
ping 4
ping 5
back 1
back 1
back 3
back 4
back 5
remove 4
remove 1
remove 3
remove 2
remove 5
```

Tecm 3
create 1 -1
create 2 1
create 5 -1
create 3 2
create 6 5
create 7 5
create 4 3
create 8 5
create 9 8
create 0 1
remove 8
remove 4
remove 2
remove 6
remove 0
ping 1
ping 3
ping 5
ping 7
ping 9
remove 5
remove 7
remove 1
remove 9
remove 3

4. Результат выполнения тестов

Программа выводит на экран результат обработки каждой команды.

Тест 1

```
OK: 17493
OK: 1
OK
OK
OK
OK: 17493 : -1
OK: 1
OK: 17493 : 0, 1, 2
OK: 1
OK: 17493 : 0, 7
OK: 1
OK
```

Тест 2

```
OK: 17711
OK: 17717
OK: 17723
OK: 17729
OK: 17735
OK
OK
OK
OK
OK
OK: 1
OK: 1
OK: 1
OK: 1
OK: 1
OK: 17711 : 0, 4, 6, 10
OK: 17717 : 0, 4, 6, 10
OK: 17723 : 0, 4, 6, 10
OK: 17729 : 0, 4, 6, 10
OK: 17735 : 0, 4, 6, 10
OK
OK
OK
OK
OK
```

Тест 3
OK: 17032
OK: 17038
OK: 17044
OK: 17050
OK: 17056
OK: 17062
OK: 17070
OK: 17076
OK: 17084
OK: 17092
OK
OK
OK
OK
OK
OK: 1
OK: 1
OK: 1
OK: 1
OK: 1
OK
OK
OK
OK
OK

5. Листинг программы

Для удобства функции инициализации сокета, получения и отправки сообщения вынесены в отдельный файл zmq_std.hpp, топология в topology.hpp. В файле control.cpp расположен код для управляющего узла, а в calculation_node.cpp для вычислительного узла. Функции для поиска подстроки находятся в файлах search.hpp и search.cpp

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_t {
     fail    = 0,
     success = 1,
     create  = 2,
     destroy = 3,
     bind    = 4,
     ping    = 5,
     exec    = 6,
     info    = 7,
     back    = 8
};

struct node_token_t {
     actions_t action;
     long long parent_id, id;
};

namespace zmq_std {
     void init_pair_socket(void* & context, void* & socket) {
          int rc;
          context = zmq_ctx_new();
          socket = zmq_socket(context, ZMQ_PAIR);
          rc  =  zmq_setsockopt(socket,  ZMQ_RCVTIMEO,  &WAIT_TIME,
sizeof(int));
          assert(rc == 0);
          rc  =  zmq_setsockopt(socket,  ZMQ_SNDTIMEO,  &WAIT_TIME,
sizeof(int));
          assert(rc == 0);
     }
```

```
	template<class T>
	void recieve_msg(T & reply_data, void* socket) {
		int rc = 0;
		zmq_msg_t reply;
		zmq_msg_init(&reply);
		rc = zmq_msg_recv(&reply, socket, 0);
		assert(rc == sizeof(T));
		reply_data = *(T*)zmq_msg_data(&reply);
		rc = zmq_msg_close(&reply);
		assert(rc == 0);
	}

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

	template<class T>
	bool send_msg_dontwait(T* token, void* socket) {
		int rc;
		zmq_msg_t message;
		zmq_msg_init(&message);
		rc = zmq_msg_init_size(&message, sizeof(T));
		assert(rc == 0);
		rc = zmq_msg_init_data(&message, token, sizeof(T), NULL,
NULL);
		assert(rc == 0);
		rc = zmq_msg_send(&message, socket, ZMQ_DONTWAIT);
		if (rc == -1) {
			zmq_msg_close(&message);
			return false;
		}
		assert(rc == sizeof(T));
		return true;
	}

	template<class T>
	bool recieve_msg_wait(T & reply_data, void* socket) {
		int rc = 0;
		zmq_msg_t reply;
		zmq_msg_init(&reply);
		rc = zmq_msg_recv(&reply, socket, 0);
```

```
            if (rc == -1) {
                zmq_msg_close(&reply);
                return false;
            }
            assert(rc == sizeof(T));
            reply_data = *(T*)zmq_msg_data(&reply);
            rc = zmq_msg_close(&reply);
            assert(rc == 0);
            return true;
        }

        /* Returns true if T was successfully queued on the socket */
        template<class T>
        bool send_msg_wait(T* token, void* socket) {
            int rc;
            zmq_msg_t message;
            zmq_msg_init(&message);
            rc = zmq_msg_init_size(&message, sizeof(T));
            assert(rc == 0);
            rc = zmq_msg_init_data(&message, token, sizeof(T), NULL,
NULL);
            assert(rc == 0);
            rc = zmq_msg_send(&message, socket, 0);
            if (rc == -1) {
                zmq_msg_close(&message);
                return false;
            }
            assert(rc == sizeof(T));
            return true;
        }

        /*
         * Returns true if socket successfully queued
         * message and recieved reply
         */
        template<class T>
        bool send_recieve_wait(T* token_send, T & token_reply, void*
socket) {
            if (send_msg_wait(token_send, socket)) {
                if (recieve_msg_wait(token_reply, socket)) {
                    return true;
                } else {
                    return false;
                }
            } else {
                return false;
            }
        }
}

#endif /* ZMQ_STD_HPP */
```

topology.hpp

```
#ifndef TOPOLOGY_HPP
#define TOPOLOGY_HPP

#include <iostream>
#include <list>

template<class T>
class topology_t {
private:
     using list_type = std::list< std::list<T> >;
     using iterator = typename std::list<T>::iterator;
     using list_iterator = typename list_type::iterator;

     list_type container;
     size_t container_size;
public:
     explicit     topology_t()     noexcept     :     container(),
container_size(0) {}
     ~topology_t() {}

     bool erase(const T & elem) {
          for  (list_iterator  it1  =  container.begin();  it1  !=
container.end(); ++it1) {
               for (iterator it2 = it1->begin(); it2 != it1->end();
++it2) {
                    if (*it2 == elem) {
                         if (it1->size() > 1) {
                              it1->erase(it2);
                         } else {
                              container.erase(it1);
                         }
                         --container_size;
                         return true;
                    }
               }
          }
          return false;
     }

     long long find(const T & elem) {
          long long ind = 0;
          for  (list_iterator  it1  =  container.begin();  it1  !=
container.end(); ++it1) {
               for (iterator it2 = it1->begin(); it2 != it1->end();
++it2) {
                    if (*it2 == elem) {
                         return ind;
                    }
               }
               ++ind;
```

```
            }
            return -1;
        }

        bool insert(const T & parent, const T & elem) {
            for (list_iterator it1 = container.begin(); it1 !=
container.end(); ++it1) {
                for (iterator it2 = it1->begin(); it2 != it1->end();
++it2) {
                    if (*it2 == parent) {
                        it1->insert(++it2, elem);
                        ++container_size;
                        return true;
                    }
                }
            }
            return false;
        }

        void insert(const T & elem) {
            std::list<T> new_list;
            new_list.push_back(elem);
            ++container_size;
            container.push_back(new_list);
        }

        size_t size() {
            return container_size;
        }

        template<class U>
        friend std::ostream & operator << (std::ostream & of, const
topology_t<U> & top) {
            for (auto it1 = top.container.begin(); it1 !=
top.container.end(); ++it1) {
                of << "{";
                for (auto it2 = it1->begin(); it2 != it1->end(); +
+it2) {
                    of << *it2 << " ";
                }
                of << "}" << std::endl;
            }
            return of;
        }
};

#endif /* TOPOLOGY_HPP */
```

control.cpp

```
#include <unistd.h>
#include <vector>

#include "topology.hpp"
#include "zmq_std.hpp"

using node_id_type = long long;

int main() {
     int rc;
     topology_t<node_id_type> control_node;
     std::vector< std::pair<void*, void*> > childs;

     std::string s;
     node_id_type id;
     while (std::cin >> s >> id) {
          if (s == "create") {
               node_id_type parent_id;
               std::cin >> parent_id;
               if (parent_id == -1) {
                    void* new_context = NULL;
                    void* new_socket = NULL;
                    zmq_std::init_pair_socket(new_context,
new_socket);
                    rc    =    zmq_bind(new_socket,    ("tcp://*:"    +
std::to_string(PORT_BASE + id)).c_str());
                    assert(rc == 0);

                    int fork_id = fork();
                    if (fork_id == 0) {
                         rc       =       execl(NODE_EXECUTABLE_NAME,
NODE_EXECUTABLE_NAME, std::to_string(id).c_str(), NULL);
                         assert(rc != -1);
                         return 0;
                    } else {
                         bool ok = true;
                         node_token_t reply_info({fail, id, id});
                         ok = zmq_std::recieve_msg_wait(reply_info,
new_socket);

                         node_token_t*       token       =       new
node_token_t({ping, id, id});
                         node_token_t reply({fail, id, id});
                         ok    =    zmq_std::send_recieve_wait(token,
reply, new_socket);
                         if (ok and reply.action == success) {

     childs.push_back(std::make_pair(new_context, new_socket));
                              control_node.insert(id);
```

```
                                std::cout << "OK: " << reply_info.id
<< std::endl;
                            } else {
                                rc = zmq_close(new_socket);
                                assert(rc == 0);
                                rc = zmq_ctx_term(new_context);
                                assert(rc == 0);
                            }
                        }
                    } else if (control_node.find(parent_id) == -1) {
                        std::cout << "Error: Not found" << std::endl;
                    } else {
                        if (control_node.find(id) != -1) {
                            std::cout << "Error: Already exists" <<
std::endl;
                        } else {
                            int ind = control_node.find(parent_id);
                            node_token_t*       token       =       new
node_token_t({create, parent_id, id});
                            node_token_t reply({fail, id, id});
                            if       (zmq_std::send_recieve_wait(token,
reply, childs[ind].second) and reply.action == success) {
                                std::cout << "OK: " << reply.id <<
std::endl;
                                control_node.insert(parent_id, id);
                            } else {
                                std::cout   <<   "Error:   Parent   is
unavailable" << std::endl;
                            }
                        }
                    }
                } else if (s == "remove") {
                    int ind = control_node.find(id);
                    if (ind != -1) {
                        node_token_t*         token         =         new
node_token_t({destroy, id, id});
                        node_token_t reply({fail, id, id});
                        bool   ok   =   zmq_std::send_recieve_wait(token,
reply, childs[ind].second);
                        if (reply.action == destroy and reply.parent_id
== id) {
                            rc = zmq_close(childs[ind].second);
                            assert(rc == 0);
                            rc = zmq_ctx_term(childs[ind].first);
                            assert(rc == 0);
                            std::vector<    std::pair<void*,    void*>
>::iterator it = childs.begin();
                            while (ind--) {
                                ++it;
                            }
                            childs.erase(it);
```

```
                    }   else   if   (reply.action   ==   bind   and
reply.parent_id == id) {
                         rc = zmq_close(childs[ind].second);
                         assert(rc == 0);
                         rc = zmq_ctx_term(childs[ind].first);
                         assert(rc == 0);

     zmq_std::init_pair_socket(childs[ind].first,
childs[ind].second);
                         rc = zmq_bind(childs[ind].second, ("tcp://
*:" + std::to_string(PORT_BASE + reply.id)).c_str());
                         assert(rc == 0);
                    }
                    if (ok) {
                         control_node.erase(id);
                         std::cout << "OK" << std::endl;
                    } else {
                         std::cout << "Error: Node is unavailable"
<< std::endl;
                    }
               } else {
                    std::cout << "Error: Not found" << std::endl;
               }
          } else if (s == "ping") {
               int ind = control_node.find(id);
               if (ind != -1) {
                    node_token_t* token = new node_token_t({ping,
id, id});
                    node_token_t reply({fail, id, id});
                    if   (zmq_std::send_recieve_wait(token,   reply,
childs[ind].second) and reply.action == success) {
                         std::cout << "OK: 1" << std::endl;
                    } else {
                         std::cout << "OK: 0" << std::endl;
                    }
               } else {
                    std::cout << "Error: Not found" << std::endl;
               }
          } else if (s == "back") {
               int ind = control_node.find(id);
               if (ind != -1) {
                    node_token_t* token = new node_token_t({back,
id, id});
                    node_token_t reply({fail, id, id});
                    if   (zmq_std::send_recieve_wait(token,   reply,
childs[ind].second)) {
                         if (reply.action == success) {
                              node_token_t*   token_back   =   new
node_token_t({back, id, id});
                              node_token_t   reply_back({fail,   id,
id});
```

```
                                std::vector<unsigned int> calculated;
                                while
(zmq_std::send_recieve_wait(token_back,                reply_back,
childs[ind].second) and reply_back.action == success) {

     calculated.push_back(reply_back.id);
                                }
                                if (calculated.empty()) {
                                     std::cout << "OK: " << reply.id
<< " : -1" << std::endl;
                                } else {
                                     std::cout << "OK: " << reply.id
<< " : ";
                                     for   (size_t   i   =   0;   i   <
calculated.size() - 1; ++i) {
                                          std::cout  <<  calculated[i]
<< ", ";
                                     }
                                     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 (s == "exec") {
                std::string pattern, text;
                std::cin >> pattern >> text;
                int ind = control_node.find(id);
                if (ind != -1) {
                     bool ok = true;
                     std::string text_pattern = pattern + SENTINEL +
text + SENTINEL;
                     for (size_t i = 0; i < text_pattern.size(); ++
+i) {
                          node_token_t*       token       =       new
node_token_t({exec, text_pattern[i], id});
                          node_token_t reply({fail, id, id});
                          if      (!zmq_std::send_recieve_wait(token,
reply, childs[ind].second) or reply.action != success) {
                               ok = false;
                               break;
                          }
                     }
```

```
                    if (ok) {
                        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 i = 0; i < childs.size(); ++i) {
            rc = zmq_close(childs[i].second);
            assert(rc == 0);
            rc = zmq_ctx_term(childs[i].first);
            assert(rc == 0);
        }
}
```

calculation_node.cpp

```
#include <list>
#include <pthread.h>
#include <queue>
#include <tuple>
#include <unistd.h>

#include "search.hpp"
#include "zmq_std.hpp"

const std::string SENTINEL_STR = "$";

long long node_id;
pthread_mutex_t mutex;
pthread_cond_t cond;
std::queue< std::pair<std::string, std::string> > calc_queue;
std::queue< std::list<unsigned int> > done_queue;

void* thread_func(void*) {
          while (1) {
               pthread_mutex_lock(&mutex);
               while (calc_queue.empty()) {
                    pthread_cond_wait(&cond, &mutex);
               }
               std::pair<std::string, std::string> cur =
calc_queue.front();
               calc_queue.pop();
               pthread_mutex_unlock(&mutex);
               if (cur.first == SENTINEL_STR and cur.second ==
SENTINEL_STR) {
                    break;
               } else {
                    std::vector<unsigned int> res =
KMPStrong(cur.first, cur.second);
                    std::list<unsigned int> res_list;
                    for (const unsigned int & elem : res) {
                         res_list.push_back(elem);
                    }
                    pthread_mutex_lock(&mutex);
                    done_queue.push(res_list);
                    pthread_mutex_unlock(&mutex);
               }
          }
          return NULL;
     }

int main(int argc, char** argv) {
     int rc;
     assert(argc == 2);
     node_id = std::stoll(std::string(argv[1]));
```

```
     void* node_parent_context = zmq_ctx_new();
     void*  node_parent_socket  =  zmq_socket(node_parent_context,
ZMQ_PAIR);
     rc  =  zmq_connect(node_parent_socket,  ("tcp://localhost:"  +
std::to_string(PORT_BASE + node_id)).c_str());
     assert(rc == 0);
     long long child_id = -1;
     void* node_context = NULL;
     void* node_socket = NULL;
          pthread_t calculation_thread;
     rc = pthread_mutex_init(&mutex, NULL);
     assert(rc == 0);
     rc = pthread_cond_init(&cond, NULL);
     assert(rc == 0);
     rc  =  pthread_create(&calculation_thread,  NULL,  thread_func,
NULL);
     assert(rc == 0);
     std::string pattern, text;
     bool flag_sentinel = true;
     node_token_t* info_token = new node_token_t({info, getpid(),
getpid()});
     zmq_std::send_msg_dontwait(info_token, node_parent_socket);
     std::list<unsigned int> cur_calculated;
     bool has_child = false;
     bool awake = true;
     bool calc = true;
     while (awake) {
          node_token_t token;
          zmq_std::recieve_msg(token, node_parent_socket);
          node_token_t*  reply  =  new  node_token_t({fail,  node_id,
node_id});

          if (token.action == back) {
               if (token.id == node_id) {
                    if (calc) {
                         pthread_mutex_lock(&mutex);
                         if (done_queue.empty()) {
                              reply->action = exec;
                         } else {
                              cur_calculated = done_queue.front();
                              done_queue.pop();
                              reply->action = success;
                              reply->id = getpid();
                         }
                         pthread_mutex_unlock(&mutex);
                         calc = false;
                    } else {
                         if (cur_calculated.size() > 0) {
                              reply->action = success;
                              reply->id = cur_calculated.front();
                              cur_calculated.pop_front();
```

```
                        } else {
                            reply->action = exec;
                            calc = true;
                        }
                    }
                } else {
                    node_token_t*       token_down       =       new
node_token_t(token);
                    node_token_t reply_down(token);
                    reply_down.action = fail;
                    if      (zmq_std::send_recieve_wait(token_down,
reply_down, node_socket) and reply_down.action == success) {
                        *reply = reply_down;
                    }
                }
            } else if (token.action == bind and token.parent_id ==
node_id) {
                /*
                 * Bind could be recieved when parent created node
                 * and this node should bind to parent's child
                 */
                zmq_std::init_pair_socket(node_context,
node_socket);
                rc    =    zmq_bind(node_socket,    ("tcp://*:"    +
std::to_string(PORT_BASE + token.id)).c_str());
                assert(rc == 0);
                has_child = true;
                child_id = token.id;
                node_token_t* token_ping = new node_token_t({ping,
child_id, child_id});
                node_token_t reply_ping({fail, child_id, child_id});
                if           (zmq_std::send_recieve_wait(token_ping,
reply_ping, node_socket) and reply_ping.action == success) {
                    reply->action = success;
                }
            } else if (token.action == create) {
                if (token.parent_id == node_id) {
                    if (has_child) {
                        rc = zmq_close(node_socket);
                        assert(rc == 0);
                        rc = zmq_ctx_term(node_context);
                        assert(rc == 0);
                    }
                    zmq_std::init_pair_socket(node_context,
node_socket);
                    rc   =   zmq_bind(node_socket,   ("tcp://*:"   +
std::to_string(PORT_BASE + token.id)).c_str());
                    assert(rc == 0);

                    int fork_id = fork();
                    if (fork_id == 0) {
```

```
                        rc      =      execl(NODE_EXECUTABLE_NAME,
NODE_EXECUTABLE_NAME, std::to_string(token.id).c_str(), NULL);
                        assert(rc != -1);
                        return 0;
                    } else {
                        bool ok = true;
                        node_token_t  reply_info({fail,  token.id,
token.id});
                        ok = zmq_std::recieve_msg_wait(reply_info,
node_socket);
                        if (reply_info.action != fail) {
                            reply->id = reply_info.id;
                            reply->parent_id                    =
reply_info.parent_id;
                        }
                        if (has_child) {
                            node_token_t*    token_bind    =    new
node_token_t({bind, token.id, child_id});
                            node_token_t         reply_bind({fail,
token.id, token.id});
                            ok                                  =
zmq_std::send_recieve_wait(token_bind, reply_bind, node_socket);
                            ok  =  ok  and  (reply_bind.action  ==
success);
                        }
                        if (ok) {
                            /*  We  should  check  if  child  has
connected to this node */
                            node_token_t*    token_ping    =    new
node_token_t({ping, token.id, token.id});
                            node_token_t         reply_ping({fail,
token.id, token.id});
                            ok                                  =
zmq_std::send_recieve_wait(token_ping, reply_ping, node_socket);
                            ok  =  ok  and  (reply_ping.action  ==
success);
                            if (ok) {
                                reply->action = success;
                                child_id = token.id;
                                has_child = true;
                            } else {
                                rc = zmq_close(node_socket);
                                assert(rc == 0);
                                rc = zmq_ctx_term(node_context);
                                assert(rc == 0);
                            }
                        }
                    }
                } else if (has_child) {
                    node_token_t*       token_down       =       new
node_token_t(token);
```

```
                    node_token_t reply_down(token);
                    reply_down.action = fail;
                    if      (zmq_std::send_recieve_wait(token_down,
reply_down, node_socket) and reply_down.action == success) {
                        *reply = reply_down;
                    }
                }
            } else if (token.action == ping) {
                if (token.id == node_id) {
                    reply->action = success;
                } else if (has_child) {
                    node_token_t*       token_down       =       new
node_token_t(token);
                    node_token_t reply_down(token);
                    reply_down.action = fail;
                    if      (zmq_std::send_recieve_wait(token_down,
reply_down, node_socket) and reply_down.action == success) {
                        *reply = reply_down;
                    }
                }
            } else if (token.action == destroy) {
                if (has_child) {
                    if (token.id == child_id) {
                        bool ok = true;
                        node_token_t*     token_down     =     new
node_token_t({destroy, node_id, child_id});
                        node_token_t  reply_down({fail,  child_id,
child_id});
                        ok                                          =
zmq_std::send_recieve_wait(token_down, reply_down, node_socket);
                        /* We should get special reply from child
*/
                        if   (reply_down.action   ==   destroy   and
reply_down.parent_id == child_id) {
                            rc = zmq_close(node_socket);
                            assert(rc == 0);
                            rc = zmq_ctx_term(node_context);
                            assert(rc == 0);
                            has_child = false;
                            child_id = -1;
                        } else if (reply_down.action == bind and
reply_down.parent_id == node_id) {
                            rc = zmq_close(node_socket);
                            assert(rc == 0);
                            rc = zmq_ctx_term(node_context);
                            assert(rc == 0);

    zmq_std::init_pair_socket(node_context, node_socket);
                            rc      =      zmq_bind(node_socket,
("tcp://*:" + std::to_string(PORT_BASE + reply_down.id)).c_str());
                            assert(rc == 0);
```

```
                                child_id = reply_down.id;
                                node_token_t*    token_ping    =    new
node_token_t({ping, child_id, child_id});
                                node_token_t         reply_ping({fail,
child_id, child_id});
                                if
(zmq_std::send_recieve_wait(token_ping,   reply_ping,   node_socket)
and reply_ping.action == success) {
                                     ok = true;
                                }
                           }
                           if (ok) {
                                reply->action = success;
                           }
                      } else if (token.id == node_id) {
                           rc = zmq_close(node_socket);
                           assert(rc == 0);
                           rc = zmq_ctx_term(node_context);
                           assert(rc == 0);
                           has_child = false;
                           reply->action = bind;
                           reply->id = child_id;
                           reply->parent_id = token.parent_id;
                           awake = false;
                      } else {
                           node_token_t*     token_down     =     new
node_token_t(token);
                           node_token_t reply_down(token);
                           reply_down.action = fail;
                           if (zmq_std::send_recieve_wait(token_down,
reply_down, node_socket) and reply_down.action == success) {
                                *reply = reply_down;
                           }
                      }
                 } else if (token.id == node_id) {
                      /* Special message to parent */
                      reply->action = destroy;
                      reply->parent_id = node_id;
                      reply->id = node_id;
                      awake = false;
                 }
            } else if (token.action == exec) {
                 if (token.id == node_id) {
                      char c = token.parent_id;
                      if (c == SENTINEL) {
                           if (flag_sentinel) {
                                std::swap(text, pattern);
                           } else {
                                pthread_mutex_lock(&mutex);
                                if (calc_queue.empty()) {
                                     pthread_cond_signal(&cond);
```

```
                                    }
                                    calc_queue.push({pattern, text});
                                    pthread_mutex_unlock(&mutex);
                                    text.clear();
                                    pattern.clear();
                               }
                               flag_sentinel = flag_sentinel ^ 1;
                          } else {
                               text = text + c;
                          }
                          reply->action = success;
                     } else if (has_child) {
                          node_token_t*       token_down       =       new
node_token_t(token);
                          node_token_t reply_down(token);
                          reply_down.action = fail;
                          if       (zmq_std::send_recieve_wait(token_down,
reply_down, node_socket) and reply_down.action == success) {
                               *reply = reply_down;
                          }
                     }
               }
               zmq_std::send_msg_dontwait(reply, node_parent_socket);
          }
          if (has_child) {
               rc = zmq_close(node_socket);
               assert(rc == 0);
               rc = zmq_ctx_term(node_context);
               assert(rc == 0);
          }
          rc = zmq_close(node_parent_socket);
          assert(rc == 0);
          rc = zmq_ctx_term(node_parent_context);
          assert(rc == 0);
          pthread_mutex_lock(&mutex);
          if (calc_queue.empty()) {
               pthread_cond_signal(&cond);
          }
          calc_queue.push({SENTINEL_STR, SENTINEL_STR});
          pthread_mutex_unlock(&mutex);
          rc = pthread_join(calculation_thread, NULL);
          assert(rc == 0);
          rc = pthread_cond_destroy(&cond);
          assert(rc == 0);
          rc = pthread_mutex_destroy(&mutex);
          assert(rc == 0);
}
```

search.hpp

```
#ifndef SEARCH_HPP
#define SEARCH_HPP

#include <string>
#include <vector>

std::vector<unsigned int> PrefixFunction(const std::string & s);
std::vector<unsigned int> KMPWeak(const std::string & pattern,
const std::string & text);

std::vector<unsigned int> ZFunction(const std::string & s);
std::vector<unsigned int> StrongPrefixFunction(const std::string &
s);
std::vector<unsigned int> KMPStrong(const std::string & pattern,
const std::string & text);

#endif /* SEARCH_HPP */
```

search.cpp

```
#include "search.hpp"

std::vector<unsigned int> PrefixFunction(const std::string & s) {
	unsigned int n = s.size();
	std::vector<unsigned int> p(n);
	for (unsigned int i = 1; i < n; ++i) {
		p[i] = p[i - 1];
		while (p[i] > 0 and s[i] != s[p[i]]) {
			p[i] = p[p[i] - 1];
		}
		if (s[i] == s[p[i]]) {
			++p[i];
		}
	}
	return p;
}

std::vector<unsigned int> KMPWeak(const std::string & pattern,
const std::string & text) {
	std::vector<unsigned int> p = PrefixFunction(pattern);
	unsigned int m = pattern.size();
	unsigned int n = text.size();
	unsigned int i = 0;
	std::vector<unsigned int> ans;
	if (m > n) {
		return ans;
	}
	while (i < n - m + 1) {
		unsigned int j = 0;
		while (j < m and pattern[j] == text[i + j]) {
			++j;
		}
		if (j == m) {
			ans.push_back(i);
		} else {
			if (j > 0 and j > p[j - 1]) {
				i = i + j - p[j - 1] - 1;
			}
		}
		++i;
	}
	return ans;
}

std::vector<unsigned int> ZFunction(const std::string & s) {
	unsigned int n = s.size();
	std::vector<unsigned int> z(n);
	unsigned int l = 0, r = 0;
	for (unsigned int i = 1; i < n; ++i) {
		if (i <= r) {
```

```
                z[i] = std::min(z[i - l], r - i);
            }
            while (i + z[i] < n and s[i + z[i]] == s[z[i]]) {
                ++z[i];
            }
            if (i + z[i] > r) {
                l = i;
                r = i + z[i];
            }
        }
        return z;
}

std::vector<unsigned int> StrongPrefixFunction(const std::string &
s) {
        std::vector<unsigned int> z = ZFunction(s);
        unsigned int n = s.size();
        std::vector<unsigned int> sp(n);
        for (unsigned int i = n - 1; i > 0; --i) {
            sp[i + z[i] - 1] = z[i];
        }
        return sp;
}

std::vector<unsigned int> KMPStrong(const std::string & pattern,
const std::string & text) {
        std::vector<unsigned int> p = StrongPrefixFunction(pattern);
        unsigned int m = pattern.size();
        unsigned int n = text.size();
        unsigned int i = 0;
        std::vector<unsigned int> ans;
        if (m > n) {
            return ans;
        }
        while (i < n - m + 1) {
            unsigned int j = 0;
            while (j < m and pattern[j] == text[i + j]) {
                ++j;
            }
            if (j == m) {
                ans.push_back(i);
            } else {
                if (j > 0 and j > p[j - 1]) {
                    i = i + j - p[j - 1] - 1;
                }
            }
            ++i;
        }
        return ans;
}
```

6. Выводы

В ходе выполнения лабораторной работы я изучил основы работы с очередями сообщений *ZeroMQ* и реализовал программу с использованием этой библиотеки. Для достижения отказоустойчивости я пробовал разные способы связи, больше всего подошёл *ZMQ_PAIR*. Самым сложным в работе оказались удаление узла из сети и вставка узла между другими узлами. При таких операциях нужно было переподключать сокеты на вычислительных узлах.

Когда параллельных вычислений становится мало, на помощь приходят распределённые вычисления (распределение вычислений осуществляется уже не между потоками процессора, а между отдельными ЭВМ). Очереди сообщений используются для взаимодействия нескольких машин в одной большой сети. Опыт работы с *ZeroMQ* пригодится мне при настройке собственной системы распределённых вычислений.

Список литературы

1. zmq_socket(3) — 0MQ Api — ZeroMQ API
   URL: http://api.zeromq.org/2-1:zmq-socket (дата обращения 08.12.2020)
2. zmq_bind(3) — 0MQ Api — ZeroMQ API
   URL: http://api.zeromq.org/2-1:zmq-bind (дата обращения 09.12.2020)
3. zmq_connect(3) — 0MQ Api — ZeroMQ API
   URL: http://api.zeromq.org/2-1:zmq-connect (дата обращения 09.12.2020)
4. Sockets and Patterns | ØMQ — The Guide — ZeroMQ Guide
   URL: https://zguide.zeromq.org/docs/chapter2/ (дата обращения 08.12.2020)
5. Socket API — ZeroMQ
   URL: https://zeromq.org/socket-api/ (дата обращения 08.12.2020)
6. zmq_setsockopt(3) — 0MQ Api — ZeroMQ API
   URL: http://api.zeromq.org/3-2:zmq-setsockopt (дата обращения 09.12.2020)
7. Messages — ZeroMQ
   URL: https://zeromq.org/messages/ (дата обращения 10.12.2020)
8. zmq_msg_send(3) — 0MQ Api — ZeroMQ API
   URL: http://api.zeromq.org/3-2:zmq-msg-send (дата обращения 10.12.2020)
9. zmq_msg_recv(3) — 0MQ Api — ZeroMQ API
   URL: http://api.zeromq.org/master:zmq-msg-recv (дата обращения 10.12.2020)