Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №5-7 по курсу

«Операционные системы»

Студент: Калиниченко Артём Андреевич
Группа: М8О–210Б–22
Вариант: 17
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023.

Постановка задачи

Цель работы

Целью является приобретение практических навыков в:

- Управление серверами сообщений №5
- Применение отложенных вычислений №6
- Интеграция программных систем друг с другом №7

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.

Вариант 17

Топология: все вычислительные узлы находятся в списке. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду: create id -1.

Команда для узлов: нахождение индексов вхождения подстроки в строку

Команда проверки: ping id

Общие сведения о программе

1. Реализуется топология в файле topology.hpp, необходимый набор функций для передачи сообщений “клиент-сервер” ZeroMQ my_zmq.hpp. Отдельно реализована функция для поиска подстроки в строке. Основной частью являются два файла control.cpp и calculation.cpp.

Основные файлы программы

my_zmq.hpp

```
#pragma once


#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
};

/*объявляем константы и структуры*/
struct node_token_t {
	actions_t action;
	long long parent_id, id;
};

namespace my_zmq {
	void init_pair_socket(void* & context, void* & socket) {
```

```
		/*инициализируем контекст и сокет для передачи сообщений в пару*/
		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) { /*шаблон для получения сообщения из сокета*/
		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) {/*функция отправляет сообщение (передаваемое как указатель на объект типа T)
	через указанный сокет*/
		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) {
        /*функция отправляет сообщение через ZeroMQ сокет, но не ждет ответа на него*/
        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;
    }
    /* Returns true if T was successfully queued on the socket */
```

```
template<class T>
bool recieve_msg_wait(T & reply_data, void* socket) { /*принимает сообщение типа T через сокет ZMQ_PAIR и
ждет, пока сообщение не будет получено*/
    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) { /*отправляет сообщение через zmq-сокет*/
    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
         */


        /* send_msg && receive_msg */
        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;
            }
        }
}
```

topology.hpp

```
#pragma once
```

```
#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:
		topology_t() noexcept : container(), container_size(0) {}
		~topology_t() {}

		bool erase(const T & elem) { /*функция удаляет элемент типа T из контейнера container*/
			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) {// в каком списке существует (или нет) элемент с идентификатором $id
			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) {
            /*Функция insert принимает на вход два аргумента: parent и elem.
            Она ищет элемент parent в контейнере и добавляет элемент elem после него в ту же подсписок,
            если parent был найден. Если parent не найден, функция возвращает false, иначе - true*/
                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) { /*добавляет элемент в структуру topology_t*/
                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> /*оператор вывода в поток*/
```

```
        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;
        }
};
```

search.hpp

```
#pragma once

#include <string>
#include <vector>

const int SIZE = 1024;

std::vector<int> search(const std::string& str, const std::string& ptrn) {
   std::vector<int> ans;
   int n = str.size();
   int m = ptrn.size();

   for (int i = 0; i <= n - m; ++i) {
      bool found = true;

      for (int j = 0; j < m; ++j) {
      if (str[i + j] != ptrn[j]) {
         found = false;
         break;
      }
      }

      if (found) {
         ans.push_back(i);
      }
   }
```

```
    return ans;
}
```

control.cpp

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

#include "topology.hpp"
#include "myzmq.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; /*для хранения
контекстов и сокетов для дочерних процессов*/

    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;
                my_zmq::init_pair_socket(new_context,
new_socket);
                rc = zmq_bind(new_socket, ("tcp://*:" +
std::to_string(PORT_BASE + id)).c_str());
                /*привязка сокета к TCP-адресу с
использованием порта,
                который вычисляется как сумма PORT_BASE и
id.*/
                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 = my_zmq::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 = my_zmq::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 (my_zmq::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 = my_zmq::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);
                                    my_zmq::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 (my_zmq::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 (my_zmq::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<int> calculated;
                                                while
(my_zmq::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 (!my_zmq::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.cpp

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

#include "search.hpp"
#include "myzmq.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<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<int> res = search(cur.first, cur.second);
                        std::list<int> res_list;
                        for (const 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()});
my_zmq::send_msg_dontwait(info_token, node_parent_socket);

std::list<int> cur_calculated;

bool has_child = false;
bool awake = true;
bool calc = true;
while (awake) {
	node_token_t token;
	my_zmq::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) {
				if (done_queue.empty()) {
					reply->action = exec;
				} else {
					cur_calculated = done_queue.front();
					done_queue.pop();
					reply->action = success;
					reply->id = getpid();
				}
				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 (my_zmq::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
                             */
                            my_zmq::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 (my_zmq::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);
                                }
                                my_zmq::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 = my_zmq::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 =
my_zmq::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 =
my_zmq::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 (my_zmq::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 (my_zmq::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 = my_zmq::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);
                                        my_zmq::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 (my_zmq::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 (my_zmq::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 {
                                        if (calc_queue.empty()) {
                                                pthread_cond_signal(&cond);
                                        }
                                        calc_queue.push({pattern, text});

                                        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 (my_zmq::send_recieve_wait(token_down,
reply_down, node_socket) and reply_down.action == success) {
                                *reply = reply_down;
                        }
                }
        }
```

```
                my_zmq::send_msg_dontwait(reply, node_parent_socket);
        }
}
```

Вывод

В ходе выполнения лабораторной работы я приобрел практические навыки в управлении серверами сообщений, применении отложенных вычислений и интеграции программных систем друг с другом, а также познакомился с технологией передачи сообщений с помощью библиотеки ZeroMQ. Эта технология позволяет удобным образом реализовывать межпроцессорное взаимодействие.