Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №6-8 по курсу «Операционные сети»
Тема: Очередь сообщений

Студент: Д. С. Ляшун
Преподаватель: Е. С. Миронов
Группа: М8О-207Б
Дата:
Оценка:
Подпись:

Москва, 2021

1 Постановка задачи

Цель работы: приобретение практических навыков в:

1. Управлении серверами сообщений (№6).
2. Применение отложенных вычислений (№7).
3. Интеграция программных систем друг с другом (№8).

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:

1. **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).

2. **Удаление существующего вычислительного узла**

   Формат команды: remove id

   id – целочисленный идентификатор удаляемого вычислительного узла.

3. **Исполнение команды на вычислительном узле**

   Формат команды: exec id [params]

   id – целочисленный идентификатор вычислительного узла, на который отправляется команда.

Вариант №8.

Тип топологии: 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.

Набор команд: Локальный таймер.

Формат команды сохранения значения: exec id subcommand

subcommand – одна из трех команд: start, stop, time.

start – запустить таймер.

stop – остановить таймер.

time – показать время локального таймера в миллисекундах.

Тип проверки доступности узлов:

Формат команды: ping id

Команда проверяет доступность конкретного узла.

2 Алгоритм решения

Для решения задачи можно воспользоваться библиотекой обмена сообщениями ZeroMQ, которая является довольно удобным средством при создании различных сложных коммуникационных решений. ZeroMQ позволяет налаживать связь между высоконагруженными приложениями по сети, а также внутри приложения между процессами или потоками. Для передачи данных используется технология сокетов. Сокет – программный интерфейс, обеспечивающий обмен данными. Стоит отметить одну из главных возможностей использования очереди сообщений – проведение асинхронных вычислений, когда программа не приостанавливается в ожидании результата работы какого-то внутреннего процесса, а продолжает работать дальше, при этом результат она может попытаться получить в любой момент, если потребуется, для этого просто необходимо проверить содержимое используемой очереди сообщений.

Для обмена сообщениями между процессами будет использоваться транспорт ZeroMQ TCP – стандартная технология однонаправленной передачи данных с использованием TCP-протоколов. В свою очередь механизм передачи сообщений (тип обмена) PAIR – взаимодействие только между клиентом и сервером. Данный тип взаимодействия не предполагает маршрутизации сообщений и не содержит уведомлений о доставке.

При написании исходного кода программы с очередью сообщений будут использоваться следующие процедуры и функции:

1. void *zmq_ctx_new() – функция, создающая новый контекст и возвращающая его хендл. Контексты помогают управлять любыми созданными сокетами, а также количеством потоков, которые использует ZeroMQ.

2. void *zmq_socket (void *context, int type) – функция создающая новый сокет типа type (определяет вид связи) в контексте context. Возвращает дескриптор (указатель) на сокет, в случае ошибки возвращает NULL.

3. int zmq_connect (void *socket, const char *endpoint) – функция, создающая выходящее соединение из сокета socket в конечную точку с адресом endpoint. Возвращает 0 в случае успеха, иначе – -1.

4. int zmq_bind (void *socket, const char *endpoint) – функция, связывающая сокет socket с локальной конечной точкой с адресом endpoint для принятия сообщений от неё. Возвращает 0 в случае успеха, иначе – -1.

5. int zmq_setsockopt (void *socket, int option_name, const void *option_value, size_t option_len) – функция, устанавливающая параметры сокета socket – изменяет значение параметра option_name на option_value размера option_len байт. Возвращает 0 в случае успеха, иначе – -1.

6. int zmq_close (void *socket) – функция, закрывающая сокет socket. Возвращает 0 в случае успеха, иначе – -1.

7. int zmq_ctx_term (void *context) – функция, уничтожающая контекст context, после чего он не может быть использован. Возвращает 0 в случае успеха, иначе – -1.

8. int zmq_msg_init_data (zmq_msg_t *msg, void *data, size_t size, zmq_free_fn *ffn, void *hint) – функция, инициализирующая сообщение msg из предоставленного буфера data размера size байт (сообщение становится владельцем данных, при уничтожении сообщения будет вызвана функция ffn освобождения памяти). Возвращает 0 в случае успеха, иначе – -1.

9. int zmq_msg_send (zmq_msg_t *msg, void *socket, int flags) – функция поставки в очередь сообщения msg по сокету socket в режиме flags (указание на неблокирующий режим, а также что это сообщение отправляется по частям). Возвращает число байтов в сообщении при успешной постановке, иначе – -1.

10. int zmq_msg_close (zmq_msg_t *msg) – функция, информирующая ZeroMQ о том, что сообщение можно закрыть т.е. освободить все ресурсы, которые с ним связаны. Возвращает 0 в случае успеха, иначе – -1.

11. int zmq_msg_init (zmq_msg_t *msg) – функция, инициализирующая сообщение msg как пустое, для возможного последующего использования как место для записи полученного сообщения. Возвращает 0 в случае успеха, иначе – -1.

12. void *zmq_msg_data (zmq_msg_t *msg) – функция, возвращающая указатель на содержимое сообщения msg.

13. int zmq_msg_recv (zmq_msg_t *msg, void *socket, int flags) – функция, производящая попытку получения сообщения msg по сокету socket из очереди сообщений в режиме flags (указание на неблокирующий режим, а также что это сообщение отправляется по частям). Возвращает число байтов в полученном сообщении, иначе – -1.

3 Листинг программы

Исходный код topology.hpp:

```
#pragma once

#include <iostream>
#include <list>

class Topology {
    public:
        bool Insert(const int parent, const int node);
        bool Erase(const int node);
        int Find(const int node);
    private:
      using list_type = std::list< std::list<int> >;
      using iterator = typename std::list<int>::iterator;
      using list_iterator = typename list_type::iterator;
      list_type data;
};

bool Topology::Insert(const int parent, const int node) {
  if (parent == -1) {
      std::list<int> new_list;
    new_list.push_back(node);
    data.push_back(new_list);
    return true;
  }
  for (list_iterator i = data.begin(); i != data.end(); ++i) {
    for (iterator j = i->begin(); j != i->end(); ++j) {
      if (*j == parent) {
          ++j;
        i->insert(j, node);
        return true;
      }
    }
  }
  return false;
}
bool Topology::Erase(const int node) {
  for (list_iterator i = data.begin(); i != data.end(); ++i) {
    for (iterator j = i->begin(); j != i->end(); ++j) {
      if (*j == node) {
          i->erase(j);
          if (i->size() == 0) {
              data.erase(i);
          }
        return true;
      }
    }
```

```
47    }
48    return false;
49  }
50  int Topology::Find(const int node) {
51    int num_list = 0;
52    for (list_iterator i = data.begin(); i != data.end(); ++i) {
53      for (iterator j = i->begin(); j != i->end(); ++j) {
54        if (*j == node) {
55          return num_list;
56        }
57      }
58      ++num_list;
59    }
60    return -1;
61  }
```

Исходный код interface.hpp:

```
1   #pragma once
2
3   #include <string.h>
4   #include <zmq.h>
5   #include <string>
6   #include <cstdlib>
7   #include <iostream>
8   #define check_ok(VALUE, OKVAL, MSG) if (VALUE != OKVAL) { std::cout << MSG << std::
        endl; exit(-1); }
9   #define check_wrong(VALUE, WRONGVAL, MSG) if (VALUE == WRONGVAL) { std::cout << MSG <<
         std::endl; exit(-1); }
10
11  const int VALUE_PORT = 8000;
12  const int WAIT_TIME = 1000;
13
14  enum Action {
15      create,
16      destroy,
17      fail,
18      success,
19      exec_start,
20      exec_stop,
21      exec_time,
22      bind,
23      ping,
24      info,
25  };
26
27  struct Token {
28      Action action;
29    int parent_id;
30    int id;
```

```
};

void CreateSocket(void* & context, void* & socket) {
  int res;
  context = zmq_ctx_new();
  check_wrong(context, NULL, "Error creating context");
  socket = zmq_socket(context, ZMQ_PAIR);
  check_wrong(socket, NULL, "Error creating socket");
  res = zmq_setsockopt(socket, ZMQ_RCVTIMEO, &WAIT_TIME, sizeof(int));
  check_ok(res, 0, "Error changing options of socket");
  res = zmq_setsockopt(socket, ZMQ_SNDTIMEO, &WAIT_TIME, sizeof(int));
  check_ok(res, 0, "Error changing options of socket");
}

void DeleteSocket(void* & context, void* & socket) {
    int res;
    res = zmq_close(socket);
    check_ok(res, 0, "Error when socket closed");
  res = zmq_ctx_term(context);
  check_ok(res, 0, "Error when context closed");
}

bool SendMessage(Token* token, void* socket, int type_work) { // ZMQ_DONTWAIT - dont
    wait, 0 - with waiting
  int res;
  zmq_msg_t message;
  res = zmq_msg_init_data(&message, token, sizeof(Token), NULL, NULL);
  check_ok(res, 0, "Error creating message");
  res = zmq_msg_send(&message, socket, type_work);
  if (res == -1) {
      std::cout << "Error sending message" << std::endl;
    zmq_msg_close(&message);
    return false;
  }
  check_ok(res, sizeof(Token), "Error getting wrong message");
  return true;
}

bool RecieveMessage(Token& reply_data, void* socket) {
  int res = 0;
  zmq_msg_t reply;
  zmq_msg_init(&reply);
  check_ok(res, 0, "Error creating message-reply");
  res = zmq_msg_recv(&reply, socket, 0);
  if (res == -1) {
      std::cout << "Error getting message" << std::endl;
    res = zmq_msg_close(&reply);
    check_ok(res, 0, "Error closing message");
    return false;
```

```
79    }
80    check_ok(res, sizeof(Token), "Error getting wrong message");
81    reply_data = *(Token*)zmq_msg_data(&reply);
82    res = zmq_msg_close(&reply);
83    check_ok(res, 0, "Error closing message");
84    return true;
85  }
86
87  bool DialogMessages(Token* send, Token& reply, void* socket) {
88    if (SendMessage(send, socket, 0) && RecieveMessage(reply, socket)) {
89        return true;
90    }
91    return false;
92  }
```

Исходный код control.cpp:

```
1   #include "topology.hpp"
2   #include "interface.hpp"
3
4   #include <unistd.h>
5   #include <vector>
6   #include <string>
7   #include <cstdlib>
8   #include <iostream>
9   #define check_ok(VALUE, OKVAL, MSG) if (VALUE != OKVAL) { std::cout << MSG << std::
        endl; exit(-1); }
10  #define check_wrong(VALUE, WRONGVAL, MSG) if (VALUE == WRONGVAL) { std::cout << MSG <<
         std::endl; exit(-1); }
11
12  char* const CALCULATE_NAME = "calculate";
13
14  auto main() -> int {
15      int res;
16    Topology nodes;
17    std::vector< std::pair<void*, void*> > nodes_info;
18    std::string oper;
19    int id;
20    while (std::cin >> oper >> id) {
21      if (oper == "create") {
22        int par_id;
23        std::cin >> par_id;
24        if (nodes.Find(id) != -1) {
25            std::cout << "Error: Already exist" << std::endl;
26            continue;
27        }
28        if (par_id == -1) {
29          void* new_context = NULL;
30          void* new_socket = NULL;
31          CreateSocket(new_context, new_socket);
```

```
32        res = zmq_bind(new_socket, ("tcp://*:" + std::to_string(VALUE_PORT + id)).c_str
              ());
33        check_ok(res, 0, "Error when bind socket with ....");
34
35        int fork_id = fork();
36        check_wrong(fork_id, -1, "Error when creating new process with fork()");
37        if (fork_id == 0) {
38          res = execl(CALCULATE_NAME, CALCULATE_NAME, std::to_string(id).c_str(), NULL)
                ;
39          check_wrong(res, -1, "Error when changing execution child process");
40          return 0;
41        }
42        else {
43          bool success = true;
44          Token reply_info({Action::fail, id, id});
45          success = RecieveMessage(reply_info, new_socket);
46          if (success && reply_info.action == Action::info) {
47            nodes_info.push_back(std::make_pair(new_context, new_socket));
48            nodes.Insert(par_id, id);
49            std::cout << "OK: " << reply_info.id << std::endl;
50          }
51          else {
52            DeleteSocket(new_context, new_socket);
53          }
54        }
55      }
56      else {
57          int ind = nodes.Find(par_id);
58          if (ind == -1) {
59              std::cout << "Error: Parent not found" << std::endl;
60              continue;
61          }
62        Token* request_create = new Token({Action::create, par_id, id});
63        Token reply_create({Action::fail, id, id});
64        if (DialogMessages(request_create, reply_create, nodes_info[ind].second) &&
              reply_create.action == Action::success) {
65          std::cout << "OK: " << reply_create.id << std::endl;
66          nodes.Insert(par_id, id);
67        }
68        else {
69          std::cout << "Error: Parent is unavailable" << std::endl;
70        }
71      }
72    }
73    else if (oper == "remove") {
74      int ind = nodes.Find(id);
75      if (ind == -1) {
76          std::cout << "Error: Not found" << std::endl;
77          continue;
```

```
78      }
79      Token* request_destroy = new Token({Action::destroy, id, id});
80      Token reply_destroy({Action::fail, id, id});
81      bool result = DialogMessages(request_destroy, reply_destroy, nodes_info[ind].
            second);
82      if (!result) {
83        std::cout << "Error: Node is unavailable" << std::endl;
84      }
85      else if (reply_destroy.action == Action::fail) {
86        std::cout << "Error: Erase was failed" << std::endl;
87      }
88      else if (reply_destroy.action == Action::success) {
89        if (reply_destroy.parent_id == id) {
90          DeleteSocket(nodes_info[ind].first, nodes_info[ind].second);
91          nodes_info.erase(nodes_info.begin() + ind);
92        }
93        nodes.Erase(id);
94        std::cout << "OK" << std::endl;
95      }
96      else if (reply_destroy.action == Action::bind && reply_destroy.parent_id == id) {
            // ???
97        DeleteSocket(nodes_info[ind].first, nodes_info[ind].second);
98        CreateSocket(nodes_info[ind].first, nodes_info[ind].second);
99        res = zmq_bind(nodes_info[ind].second, ("tcp://*:" + std::to_string(VALUE_PORT
              + reply_destroy.id)).c_str());
100       check_ok(res, 0, "Error when bind socket with ....");
101       nodes.Erase(id);
102       std::cout << "OK" << std::endl;
103     }
104   }
105   else if (oper == "ping") {
106     int ind = nodes.Find(id);
107     if (ind == -1) {
108         std::cout << "Error: Not found" << std::endl;
109         continue;
110     }
111     Token* request_ping = new Token({Action::ping, id, id});
112     Token reply_ping({Action::fail, id, id});
113     if (DialogMessages(request_ping, reply_ping, nodes_info[ind].second) &&
            reply_ping.action == Action::success) {
114       std::cout << "OK: 1" << std::endl;
115     }
116     else {
117       std::cout << "OK: 0" << std::endl;
118     }
119   }
120   else if (oper == "exec") {
121     std::string subcom;
122     std::cin >> subcom;
```

```
123       int ind = nodes.Find(id);
124       if (ind == -1) {
125           std::cout << "Error: Not found" << std::endl;
126           continue;
127       }
128       if (subcom == "start") {
129         Token* request_start = new Token({Action::exec_start, id, id});
130         Token reply_start({Action::fail, id, id});
131         if (DialogMessages(request_start, reply_start, nodes_info[ind].second) &&
              reply_start.action == Action::success) {
132           std::cout << "OK: " << reply_start.id << std::endl;
133         }
134         else {
135           std::cout << "Error starting timer in " << id << std::endl;
136         }
137       }
138       else if (subcom == "stop") {
139         Token* request_stop = new Token({Action::exec_stop, id, id});
140         Token reply_stop({Action::fail, id, id});
141         if (DialogMessages(request_stop, reply_stop, nodes_info[ind].second) &&
              reply_stop.action == Action::success) {
142           std::cout << "OK: " << reply_stop.id << std::endl;
143         }
144         else {
145           std::cout << "Error stoping timer in " << id << std::endl;
146         }
147       }
148       else if (subcom == "time") {
149         Token* request_time = new Token({Action::exec_time, id, id});
150         Token reply_time({Action::fail, id, id});
151         if (DialogMessages(request_time, reply_time, nodes_info[ind].second) &&
              reply_time.action == Action::success) {
152           std::cout << "OK: " << reply_time.parent_id << ": " << reply_time.id << std::
                endl;
153         }
154         else {
155           std::cout << "Error getting time in " << id << std::endl;
156         }
157       }
158       else {
159           std::cout << "Error: Wrong command of timer" << std::endl;
160       }
161     }
162     else {
163         std::cout << "Error: Wrong command" << std::endl;
164     }
165   }
166   for (size_t i = 0; i < nodes_info.size(); ++i) {
167     DeleteSocket(nodes_info[i].first, nodes_info[i].second);
```

```
168   }
169 }
```

Исходный код calculate.cpp:

```
1  #include "interface.hpp"
2
3  #include <unistd.h>
4  #include <cstdlib>
5  #include <iostream>
6  #include <cmath>
7  #include <ctime>
8  #include <chrono>
9  #define check_ok(VALUE, OKVAL, MSG) if (VALUE != OKVAL) { std::cout << MSG << std::
       endl; exit(-1); }
10 #define check_wrong(VALUE, WRONGVAL, MSG) if (VALUE == WRONGVAL) { std::cout << MSG <<
        std::endl; exit(-1); }
11
12 using std::chrono::duration_cast;
13 using std::chrono::milliseconds;
14 using std::chrono::seconds;
15 using std::chrono::system_clock;
16 char* const CALCULATE_NAME = "calculate";
17 long long GetTime() {
18   long long millisec = duration_cast<milliseconds>(system_clock::now().
         time_since_epoch()).count();
19   return millisec;
20 }
21
22 int main(int argc, char* argv[]) {
23   int res;
24   check_ok(argc, 2, "Error: Wrong count arguments in calculate process");
25   int node_id = std::stoll(std::string(argv[1]));
26   int child_id = -1;
27
28   void* my_context = zmq_ctx_new();
29   void* my_socket = zmq_socket(my_context, ZMQ_PAIR);
30   void* child_context = NULL;
31   void* child_socket = NULL;
32
33   res = zmq_connect(my_socket, ("tcp://localhost:" + std::to_string(VALUE_PORT +
         node_id)).c_str());
34
35   check_ok(res, 0, "Error when connecting to socket in calculate process");
36
37   long long start = -1, finish = -1, time_ans = 0;
38
39   Token* info_token = new Token({Action::info, getpid(), getpid()});
40   SendMessage(info_token, my_socket, ZMQ_DONTWAIT);
41   bool is_parent = false;
```

```
42  bool work = true;
43  while (work) {
44    Token token;
45    RecieveMessage(token, my_socket);
46    Token* reply = new Token({Action::fail, node_id, node_id});
47        if (token.action == Action::bind && token.parent_id == node_id) {
48      CreateSocket(child_context, child_socket);
49      res = zmq_bind(child_socket, ("tcp://*:" + std::to_string(VALUE_PORT + token.id))
            .c_str());
50      check_ok(res, 0, "Error bind to socket in calculate process");
51      is_parent = true;
52      child_id = token.id;
53      reply->action = Action::success;
54    }
55    else if (token.action == Action::create) {
56      if (token.parent_id == node_id) {
57        if (is_parent) {
58            DeleteSocket(child_context, child_socket);
59        }
60        CreateSocket(child_context, child_socket);
61        res = zmq_bind(child_socket, ("tcp://*:" + std::to_string(VALUE_PORT + token.id
              )).c_str());
62        check_ok(res, 0, "Error when bind with child socket");
63        int fork_id = fork();
64        check_wrong(fork_id, -1, "Error creating calculating process using fork");
65        if (fork_id == 0) {
66          res = execl(CALCULATE_NAME, CALCULATE_NAME, std::to_string(token.id).c_str(),
                NULL);
67          check_wrong(res, -1, "Error when changing execution in calculate process");
68          return 0;
69        }
70        else {
71          bool result = true;
72          Token reply_info({Action::fail, token.id, token.id});
73          result = RecieveMessage(reply_info, child_socket);
74          if (!result) {
75            DeleteSocket(child_context, child_socket);
76          }
77          else {
78            if (reply_info.action == Action::info) {
79              reply->id = reply_info.id;
80              reply->parent_id = reply_info.parent_id;
81            }
82            if (is_parent) {
83              Token* request_bind = new Token({Action::bind, token.id, child_id});
84              Token reply_bind({Action::fail, token.id, token.id});
85              result = DialogMessages(request_bind, reply_bind, child_socket);
86              if (result && reply_bind.action == Action::success) {
87                child_id = token.id;
```

```
88                reply->action = Action::success;
89              }
90              else {
91                DeleteSocket(child_context, child_socket);
92              }
93            }
94            else {
95              reply->action = Action::success;
96              child_id = token.id;
97              is_parent = true;
98            }
99          }
100       }
101     }
102     else if (is_parent) {
103       Token* request_create = new Token(token);
104         Token reply_create(token);
105       reply_create.action = Action::fail;
106       if (DialogMessages(request_create, reply_create, child_socket) && reply_create.
              action == Action::success) {
107         *reply = reply_create;
108       }
109     }
110   }
111   else if (token.action == Action::ping) {
112     if (token.id == node_id) {
113       reply->action = Action::success;
114     }
115     else if (is_parent) {
116       Token* request_ping = new Token(token);
117       Token reply_ping(token);
118       reply_ping.action = Action::fail;
119       if (DialogMessages(request_ping, reply_ping, child_socket) && reply_ping.action
              == Action::success) {
120         *reply = reply_ping;
121       }
122     }
123   }
124   else if (token.action == Action::destroy) {
125     if (is_parent) {
126       if (token.id == child_id) {
127         Token* request_destroy = new Token({Action::destroy, node_id, child_id});
128         Token reply_destroy({Action::fail, child_id, child_id});
129         bool result = DialogMessages(request_destroy, reply_destroy, child_socket);
130         if (reply_destroy.action == Action::success && reply_destroy.parent_id ==
              child_id) {
131           DeleteSocket(child_context, child_socket);
132         reply->action = Action::success;
133         reply->id = child_id;
```

```
134             reply->parent_id = node_id;
135             child_id = -1;
136             is_parent = false;
137           }
138           else if (reply_destroy.action == Action::bind && reply_destroy.parent_id ==
                  node_id) {
139             DeleteSocket(child_context, child_socket);
140             CreateSocket(child_context, child_socket);
141             res = zmq_bind(child_socket, ("tcp://*:" + std::to_string(VALUE_PORT +
                    reply_destroy.id)).c_str());
142             check_ok(res, 0, "Error binding with calculate process");
143             reply->action = Action::success;
144             reply->id = child_id;
145             reply->parent_id = node_id;
146             child_id = reply_destroy.id;
147           }
148         }
149         else if (token.id == node_id) {
150           DeleteSocket(child_context, child_socket);
151           is_parent = false;
152           reply->action = Action::bind;
153           reply->parent_id = token.parent_id;
154           reply->id = child_id;
155           work = false;
156         }
157         else {
158           Token* request_destroy = new Token(token);
159           Token reply_destroy(token);
160           reply_destroy.action = fail;
161           if (DialogMessages(request_destroy, reply_destroy, child_socket) &&
                  reply_destroy.action == Action::success) {
162             *reply = reply_destroy;
163           }
164         }
165       }
166       else if (token.id == node_id) {
167         reply->action = Action::success;
168         reply->parent_id = node_id;
169         reply->id = node_id;
170         work = false;
171       }
172     }
173     else if (token.action == Action::exec_start) {
174       if (token.id == node_id) {
175         time_ans = 0;
176         start = GetTime();
177         reply->action = Action::success;
178       }
179       else if (is_parent) {
```

```
180         Token* request_start = new Token(token);
181         Token reply_start(token);
182         reply_start.action = Action::fail;
183         if (DialogMessages(request_start, reply_start, child_socket) && reply_start.
                action == Action::success) {
184           *reply = reply_start;
185         }
186       }
187     }
188     else if (token.action == Action::exec_stop) {
189       if (token.id == node_id) {
190         if (start != -1) {
191           finish = GetTime();
192           time_ans += finish - start;
193           finish = -1;
194           start = -1;
195           reply->action = Action::success;
196         }
197       }
198       else if (is_parent) {
199         Token* request_stop = new Token(token);
200         Token reply_stop(token);
201         reply_stop.action = Action::fail;
202         if (DialogMessages(request_stop, reply_stop, child_socket) && reply_stop.action
                == Action::success) {
203           *reply = reply_stop;
204         }
205       }
206     }
207     else if (token.action == Action::exec_time) {
208       if (token.id == node_id) {
209         if (start != -1) {
210           finish = GetTime();
211           time_ans += finish - start;
212           start = finish;
213         }
214         reply->action = Action::success;
215         reply->id = (int) time_ans;
216       }
217       else if (is_parent) {
218         Token* request_time = new Token(token);
219         Token reply_time(token);
220         reply_time.action = Action::fail;
221         if (DialogMessages(request_time, reply_time, child_socket) && reply_time.action
                == Action::success) {
222           *reply = reply_time;
223         }
224       }
225     }
```

```
226     SendMessage(reply, my_socket, ZMQ_DONTWAIT);
227   }
228   if (is_parent) {
229       DeleteSocket(child_context, child_socket);
230   }
231   DeleteSocket(my_context, my_socket);
232 }
```

Исходный код Makefile

```
1 all: calculate.cpp control.cpp
2   g++ calculate.cpp -o calculate -lzmq
3   g++ control.cpp -o control -lzmq
```

4 Тесты и протокол работы

```
dmitry@dmitry-VirtualBox:~/Work_place/OS_labs/Lab6$ make
g++ calculate.cpp -o calculate -lzmq
g++ control.cpp -o control -lzmq
dmitry@dmitry-VirtualBox:~/Work_place/OS_labs/Lab6$ ./control
create -1 1
Error: Parent not found
create 1 -1
OK: 3791
create 2 1
OK: 3797
create 3 1
OK: 3802
create 4 1
OK: 3809
create 5 1
OK: 3816
ping 1
OK: 1
ping 2
OK: 1
ping 3
OK: 1
ping 4
OK: 1
ping 5
OK: 1
exec 5 time
OK: 5: 0
exec 5 start
OK: 5
remove 1
OK
remove 2
OK
remove 3
OK
remove 4
OK
exec 5 time
```

```
OK: 5: 18188
exec 5 stop
OK: 5
exec 5 time
OK: 5: 22655
exec 5 time
OK: 5: 22655
^C
```

5 Strace

В ходе выполнения программы видно, что получение сообщений из сокета осуществляется с помощью системного вызова recvfrom, а отправка – с помощью вызова sendto. Также при ожидании отправки/получении сообщений происходит вызов poll с установленным временем блокировки.

```
....
[pid  6922] rt_sigprocmask(SIG_BLOCK,~[RTMIN RT_1], <unfinished ...>
[pid  6921] <... fcntl resumed>)        = 0x802 (flags O_RDWR|O_NONBLOCK)
strace: Process 6923 attached
[pid  6920] <... epoll_wait resumed>[EPOLLIN,u32=1879088064,u64=140687827835840],
256,-1) = 1
[pid  6923] epoll_wait(7, <unfinished ...>
[pid  6920] recvfrom(10, <unfinished ...>
[pid  6923] <... epoll_wait resumed>[EPOLLOUT,u32=1610618672,u64=140451336165168],
256,-1) = 1
[pid  6920] <... recvfrom resumed>"^~~3~3",8192,0,NULL,NULL)
= 14
[pid  6923] epoll_ctl(7,EPOLL_CTL_MOD,9,EPOLLIN,u32=1610618672,u64=140451336165168
<unfinished ...>
[pid  6920] write(8,"",8 <unfinished ...>
[pid  6918] <... poll resumed>)         = 1 ([fd=8,revents=POLLIN])
[pid  6923] <... epoll_ctl resumed>)    = 0
[pid  6920] <... write resumed>)        = 8
[pid  6918] read(8, <unfinished ...>
[pid  6923] epoll_wait(7, <unfinished ...>
[pid  6920] epoll_wait(7, <unfinished ...>
[pid  6918] <... read resumed>"",8) = 8
[pid  6918] poll([fd=8,events=POLLIN],1,0) = 0 (Timeout)
[pid  6918] fstat(1,st_mode=S_IFCHR|0620,st_rdev=makedev(0x88,0),...) = 0
[pid  6918] write(1,"OK: 6921",9OK: 6921
)   = 9
[pid  6918] read(0,ping 1
"ping 1",1024)   = 7
[pid  6918] poll([fd=8,events=POLLIN],1,0) = 0 (Timeout)
[pid  6918] write(6,"",8) = 8
[pid  6920] <... epoll_wait resumed>[EPOLLIN,u32=4025681696,u64=93874830903072],
256,-1) = 1
[pid  6918] poll([fd=8,events=POLLIN],1,1000 <unfinished ...>
[pid  6920] poll([fd=6,events=POLLIN],1,0) = 1 ([fd=6,revents=POLLIN])
```

```
[pid  6920] read(6,"",8) = 8
[pid  6920] epoll_ctl(7,EPOLL_CTL_MOD,10,EPOLLIN|EPOLLOUT,u32=1879088064,
u64=140687827835840) = 0
[pid  6920] sendto(10,"^0",14,0,NULL,0 <unfinished ...>
[pid  6923] <... epoll_wait resumed>[EPOLLIN,u32=1610618672,u64=140451336165168],
256,-1) = 1
[pid  6920] <... sendto resumed>)       = 14
[pid  6923] recvfrom(9, <unfinished ...>
[pid  6920] poll([fd=6,events=POLLIN],1,0 <unfinished ...>
...
[pid  6923] epoll_ctl(7,EPOLL_CTL_MOD,9,EPOLLIN|EPOLLOUT,u32=1610618672,
u64=140451336165168) = 0
[pid  6923] sendto(9,"^",14,0,NULL,0) = 14
[pid  6920] <... epoll_wait resumed>[EPOLLIN,u32=1879088064,u64=140687827835840],
256,-1) = 1
[pid  6923] poll([fd=6,events=POLLIN],1,0 <unfinished ...>
[pid  6920] recvfrom(10, <unfinished ...>
[pid  6923] <... poll resumed>)         = 0 (Timeout)
[pid  6920] <... recvfrom resumed>"^",8192,0,NULL,
NULL) = 14
[pid  6923] epoll_wait(7, <unfinished ...>
[pid  6920] write(8,"",8 <unfinished ...>
[pid  6918] <... poll resumed>)         = 1 ([fd=8,revents=POLLIN])
[pid  6923] <... epoll_wait resumed>[EPOLLOUT,u32=1610618672,u64=140451336165168],
256,-1) = 1
[pid  6920] <... write resumed>)        = 8
[pid  6918] read(8, <unfinished ...>
[pid  6923] epoll_ctl(7,EPOLL_CTL_MOD,9,EPOLLIN,u32=1610618672,u64=140451336165168
<unfinished ...>
[pid  6920] epoll_wait(7, <unfinished ...>
[pid  6918] <... read resumed>"",8) = 8
[pid  6923] <... epoll_ctl resumed>)    = 0
[pid  6918] poll([fd=8,events=POLLIN],1,0 <unfinished ...>
[pid  6923] epoll_wait(7, <unfinished ...>
[pid  6918] <... poll resumed>)         = 0 (Timeout)
[pid  6918] write(1,"OK: 1",6OK: 1
)      = 6
```

6 Выводы

В результате выполнения данной лабораторной работы я познакомился с использованием очереди сообщений ZeroMQ при написании программ в ОС Linux. Основную трудность для меня составило понять, как устроены системные вызовы для работы с этой технологией, на это мне пришлось потратить несколько часов чтения мануалов по ZeroMQ (которые написаны исключительно на английском). Стоит сказать, что очередь сообщений является довольно мощным средством для организации многопроцессорных приложений, позволяющая выполняться им согласованно даже на отдельных удаленных компьютерах. Также ключевым преимуществом этой технологии является возможность проведения асинхронных вычислений, например, когда процесс-клиент посылает процессу-серверу задачу на выполнение, при этом не блокируется на ожидание ответа, а продолжает дальше работать, обращаясь за результатом только при реальной необходимости.

Список литературы

[1] *ZeroMQ API*
URL: `http://api.zeromq.org/` (дата обращения: 20.12.2020).