Министерство науки и высшего образования

Московский Авиационный Институт
(национальный исследовательский университет)

ЛАБОРАТОРНАЯ РАБОТЫ №6-8
по курсу операционные системы I семестр 2022/2023

Студент: Гришин Павел Федорович
Группа: М8О-201Б-21
Вариант: 48
Преподователь : Миронов Евгений Сергеевич
Оценка:
Дата:
Подпись:

Москва, 2023

Содержание

Постановка задачи

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом. 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.
Топология - бинарное дерево поиска.
Список основных поддерживаемых команд:

- create id - создать новый вычислительный узел с указанным id
- remove id - удалить вычислительный узел с указанным id и её дочерние узлы.
- exec id text pattern - поиск подстроки в строке вычислительным узлом с указанным id
- ping id - проверка доступности вычислительного узла с указанным id.

Цель работы

Приобритение практических навыков в:

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

Алгоритм решения

Имеем два исполняемых файла **server** и **client**, но работаем только с одним **server**, а другой используется внутри вызова execl для создания вычислительного узла. Вычислительный узел создается, когда пользователь вводит команду *create id*, если узел с указанным id существует, то он не создается, проверка существования узла производится с помошью поиска в бинарном дереве поиска. После этого есть доступный узел, к которому можно обратиться с помощью команд *exec id text pattern* и *ping id*. *ping id* - проверяет доступность узла, путем передачи сообщения, содержащего тип сообщения и id нужной ноды, когда сообщение дойдет до нужного узла, узел отправит обратно сообщение управляющей ноде, тем самым сообщив, что он доступен. *exec id text pattern*, выполняется подобным же образом, только теперь сообщение еще содержит строку и подстроку. В ответ отправляются начальные позиции совпадений. *remove id* работает следующем образом: ищется данный узел и его дочерние узлы, дальше всем этим нодам отправляется о сообщение об удалении, перед завершение работы, они закрывают сокеты, с помощью которых осуществляется передача сообщений, после этого завершает работу.

Код программы

Заголовчный файл message.h для сообщений

```
#include "zmq.hpp"
#include <string>
#include <vector>

#ifndef LAB6_8__MESSAGE_H_
#define LAB6_8__MESSAGE_H_


enum message_type_t {
  messagePing,
  messageCreate,
  messageRemove,
  messageExec,
  messageExecAnswer,
  messagePingAnswer,
  messageCreateAnswer,
};

enum MessageDirection{
  toClient,
  toServer,
};

struct Header {
  MessageDirection direction;
  message_type_t type;
  int to_id_node;
};

struct CreateBody {
  int child_id;
};

struct ExecBody {
  std::string text;
  std::string pattern;
};

struct pingBodyAnswer {
  int node_id;
};

struct exec_body_answer {
  std::vector<int> entries;
};

struct create_body_answer {
  int proccess_id;
  std::string error;
};
create_body_answer getMessageCreateAnswer(zmq::message_t& data);
zmq::message_t fillMessageExec(int endPoint, std::string& text, std::string& pattern);

zmq::message_t fillMessageExecAnswer(std::vector<int>& enrties);

zmq::message_t fillMessageCreate(int parent_id, int child_id);
```

```
57
58 zmq::message_t fillMessageCreateAnswer(int pid, std::string error);
59 zmq::message_t fillMessageRemove(int endPoint);
60 zmq::message_t fillMessagePing(int endPoint);
61
62 zmq::message_t fillMessagePingAnswer(int src_id);
63
64 Header* getMessageHeader(zmq::message_t& data);
65
66 CreateBody getMessageCreate(zmq::message_t& data);
67
68 ExecBody getMessageExec(zmq::message_t& data);
69
70 exec_body_answer getMessageExecAnswer(zmq::message_t& data);
71
72 pingBodyAnswer getMessagePingAnswer(zmq::message_t& data);
73
74 #endif //LAB6_8__MESSAGE_H_
```

Заголовчный файл ZMQUtils.h с настройками сокетов

```
1  #include "zmq.hpp"
2  #include <string>
3
4  #ifndef LAB6_8__ZMQUTILS_H_
5  #define LAB6_8__ZMQUTILS_H_
6
7  std::string createNameSocket(int node_id);
8
9  std::string createNameSocketToChildren(int node_id);
10
11 std::string createNameSocketToParent(int node_id);
12
13 #endif
```

Файл message.cpp с реализацией функций для создания сообщений

```
1  #include "message.h"
2  #include <cstring>
3  zmq::message_t fillMessageExec(int endPoint, std::string& text, std::string& pattern){
4    Header header {
5      toClient,
6      messageExec,
7      endPoint,
8    };
9
10   int size_of_message = sizeof(Header) + text.size() + pattern.size() + 2 * sizeof(int);
11   zmq::message_t data(size_of_message);
12   char* memory_pointer = (char*)data.data();
13
14   std::memcpy(memory_pointer, &header, sizeof(header));
15   memory_pointer += sizeof(header);
16
17   int text_size = text.size();
18   std::memcpy(memory_pointer, &text_size, sizeof(text_size));
19   memory_pointer += sizeof(text_size);
20
21   std::copy(text.begin(), text.end(), memory_pointer);
22   memory_pointer += text.size();
23
```

```
24    int pattern_size = pattern.size();
25    std::memcpy(memory_pointer, &pattern_size, sizeof(pattern_size));
26    memory_pointer += sizeof(pattern_size);
27
28    std::copy(pattern.begin(), pattern.end(), memory_pointer);
29    memory_pointer += pattern.size();
30
31    return data;
32  }
33
34  zmq::message_t fillMessageExecAnswer(std::vector<int>& enrties){
35    Header header {
36      toServer,
37      messageExecAnswer,
38      -1,
39    };
40
41    int size_of_message = sizeof(Header) + sizeof(int) * enrties.size() + sizeof(int);
42    zmq::message_t data(size_of_message);
43    char* memory_pointer = (char*)data.data();
44
45    std::memcpy(memory_pointer, &header, sizeof(header));
46    memory_pointer += sizeof(header);
47
48    int entries_size = enrties.size();
49    std::memcpy(memory_pointer, &entries_size, sizeof(entries_size));
50    memory_pointer += sizeof(entries_size);
51
52    std::copy(enrties.begin(), enrties.end(), (int*)memory_pointer);
53    memory_pointer += sizeof(int) * enrties.size();
54
55    return data;
56  }
57
58  zmq::message_t fillMessageCreate(int parent_id, int child_id){
59    Header header {
60      toClient,
61      messageCreate,
62      parent_id
63    };
64
65    CreateBody body {
66      child_id
67    };
68
69    int size_of_message = sizeof(Header) + sizeof(body);
70    zmq::message_t data(size_of_message);
71    char* memory_pointer = (char*)data.data();
72
73    std::memcpy(memory_pointer, &header, sizeof(header));
74    memory_pointer += sizeof(header);
75
76    std::memcpy(memory_pointer, &body, sizeof(body));
77    memory_pointer += sizeof(body);
78
79    return data;
80  }
81
82  zmq::message_t fillMessageCreateAnswer(int pid, std::string error){
83    Header header {
```

```
    toServer,
    messageCreateAnswer,
    -1
  };

  int size_of_message = sizeof(Header) + sizeof(pid) + sizeof(int) + error.size() * sizeof(error);
  zmq::message_t data(size_of_message);
  char* memory_pointer = (char*)data.data();

  std::memcpy(memory_pointer, &header, sizeof(header));
  memory_pointer += sizeof(header);

  std::memcpy(memory_pointer, &pid, sizeof(pid));
  memory_pointer += sizeof(pid);

  int error_size = error.size();
  std::memcpy(memory_pointer, &error_size, sizeof(error_size));
  memory_pointer += sizeof(error_size);

  std::copy(error.begin(), error.end(), memory_pointer);
  memory_pointer += error.size();

  return data;
}

zmq::message_t fillMessageRemove(int endPoint){
  Header header {
    toClient,
    messageRemove,
    endPoint
  };

  int size_of_message = sizeof(Header);
  zmq::message_t data(size_of_message);
  char* memory_pointer = (char*)data.data();

  std::memcpy(memory_pointer, &header, sizeof(header));
  memory_pointer += sizeof(header);

  return data;
}

zmq::message_t fillMessagePing(int endPoint){
  Header header {
    toClient,
    messagePing,
    endPoint
  };

  int size_of_message = sizeof(Header);
  zmq::message_t data(size_of_message);
  char* memory_pointer = (char*)data.data();

  std::memcpy(memory_pointer, &header, sizeof(header));
  memory_pointer += sizeof(header);

  return data;
}

zmq::message_t fillMessagePingAnswer(int src_id){
```

```
144   Header header {
145     toServer,
146     messagePingAnswer,
147     -1
148   };
149 
150   pingBodyAnswer body {
151     src_id
152   };
153 
154   int size_of_message = sizeof(Header) + sizeof(body);
155   zmq::message_t data(size_of_message);
156   char* memory_pointer = (char*)data.data();
157 
158   std::memcpy(memory_pointer, &header, sizeof(header));
159   memory_pointer += sizeof(header);
160 
161   std::memcpy(memory_pointer, &body, sizeof(body));
162   memory_pointer += sizeof(body);
163 
164   return data;
165 }
166 
167 Header* getMessageHeader(zmq::message_t& data){
168   return (Header*)data.data();
169 }
170 
171 CreateBody getMessageCreate(zmq::message_t& data){
172   CreateBody body;
173   std::memcpy(&body, (char *)data.data() + sizeof(Header), sizeof(body));
174   return body;
175 }
176 
177 create_body_answer getMessageCreateAnswer(zmq::message_t& data){
178   char* memory_pointer = (char*)data.data() + sizeof(Header);
179 
180   int process_id;
181   std::memcpy(&process_id, memory_pointer, sizeof(int));
182   memory_pointer += sizeof(process_id);
183 
184   int error_size;
185   std::memcpy(&error_size, memory_pointer, sizeof(int));
186   memory_pointer += sizeof(error_size);
187 
188   std::string error(memory_pointer, error_size);
189   memory_pointer += error_size;
190 
191   return {process_id, error};
192 }
193 
194 ExecBody getMessageExec(zmq::message_t& data){
195   char* memory_pointer = (char*)data.data() + sizeof(Header);
196 
197   int text_size;
198   std::memcpy(&text_size, memory_pointer, sizeof(int));
199   memory_pointer += sizeof(int);
200 
201   std::string text(memory_pointer, text_size);
202   memory_pointer += text_size;
203 
```

```
  int pattern_size;
  std::memcpy(&pattern_size, memory_pointer, sizeof(int));
  memory_pointer += sizeof(int);

  std::string pattern(memory_pointer, pattern_size);
  memory_pointer += pattern_size;

  return {text, pattern};
}

exec_body_answer getMessageExecAnswer(zmq::message_t& data){
  char* memory_pointer = (char*)data.data() + sizeof(Header);

  int entries_size;
  std::memcpy(&entries_size, memory_pointer, sizeof(int));
  memory_pointer += sizeof(int);

  std::vector<int> entries;
  int* integer_pointer = (int*)memory_pointer;
  std::copy(integer_pointer, integer_pointer + entries_size, std::back_inserter(entries));

  return {entries};
}

pingBodyAnswer getMessagePingAnswer(zmq::message_t& data){
  pingBodyAnswer body;
  std::memcpy(&body, (char *)data.data() + sizeof(Header), sizeof(body));
  return body;
}
```

Файл ZMQUtils.cpp с реализацикй функция для настройки сокетов

```
#include "ZMQUtils.h"

std::string createNameSocket(int node_id){
  std::string name = "ipc://sock" + std::to_string(node_id+5000);
  return name;
}

std::string createNameSocketToChildren(int node_id){
  std::string name = createNameSocket(node_id) +"_to_children";
  return name;
}

std::string createNameSocketToParent(int node_id){
  std::string name = createNameSocket(node_id) +"_to_parent";
  return name;
}
```

Файл Tree.h содержит реализацию бинарного дерева поиска

```
#include <iostream>
#include <vector>

using namespace std;

#ifndef LAB6_8__TREE_H_
#define LAB6_8__TREE_H_

typedef struct Node Node;
```

```
10
11 struct Node {
12   int value;
13   Node* left, * right;
14 };
15
16 Node* init_tree(int data) {
17   Node* root = (Node*)malloc(sizeof(Node));
18   root->left = root->right = NULL;
19   root->value = data;
20   return root;
21 }
22
23 Node* create_node(int data) {
24   Node* node = (Node*)malloc(sizeof(Node));
25   node->value = data;
26   node->left = node->right = NULL;
27   return node;
28 }
29
30 void free_tree(Node* root) {
31   Node* temp = root;
32   if (!temp)
33   return;
34   free_tree(temp->left);
35   free_tree(temp->right);
36   if (!temp->left && !temp->right) {
37   free(temp);
38   return;
39   }
40 }
41
42 void insert(Node* root, int value) {
43   if (!root->left && root->value > value) {
44   root->left = create_node(value);
45   return;
46   } else if (!root->right && root->value < value) {
47     root->right = create_node(value);
48   return;
49   }
50   if(root->value > value){
51   insert(root->left, value);
52   } else if (root->value < value) {
53   insert(root->right, value);
54   }
55 }
56 bool find(Node* root, int value) {
57   if (!root) {
58   return false;
59   }
60   if (root->value > value) {
61   return find(root->left, value);
62   } else if (root->value < value) {
63   return find(root->right, value);
64   } else {
65   return true;
66   }
67 }
68
69 Node* search(Node* root, int value){
```

```
  if (root->value == value) {
  return root;
  }
  if (root->value > value) {
  return search(root->left, value);
  } else {
  return search(root->right, value);
  }
}

void getChildren (Node* root, vector<int>& children){
  if(!root){
  return;
  }
  getChildren(root->left, children);
  getChildren(root->right, children);
  children.push_back(root->value);
}

vector<int> getChildren (Node* root, int value){
  Node* node = search(root, value);
  vector<int> childern;
  getChildren(node, childern);
  return childern;
}


#endif
```

Файл server.cpp

```
#include <iostream>
#include <string>
#include "zmq.hpp"
#include <unistd.h>
#include "message.h"
#include "ZMQUtils.h"
#include "Tree.h"


static zmq::context_t context;

static zmq::socket_t publish_socket(context, ZMQ_PUB);
static zmq::socket_t sub_sock(context, ZMQ_SUB);

static std::string socket_name;
static std::vector<std::string> sockets_of_children;

Node* tree = init_tree(0);

void server_init(){

  socket_name = createNameSocket(0);
  publish_socket.bind(socket_name);

  sub_sock.setsockopt(ZMQ_SUBSCRIBE, nullptr, 0);
  int timeout = 300;

  sub_sock.setsockopt(ZMQ_RCVTIMEO, timeout);

```

```
}


bool receive_msg(zmq::message_t& msg){
  zmq::recv_result_t res;
  res = sub_sock.recv(msg);

  return res.has_value();
}

bool is_exists(int id) {
  zmq::message_t data = fillMessagePing(id);
  publish_socket.send(data, 0);

  zmq::message_t receive_data;
  Header* header;

  if(!receive_msg(receive_data))
  return false;

  pingBodyAnswer ans = getMessagePingAnswer(receive_data);
  if(ans.node_id != id)
  return false;

  return true;
}

void create(){
  int new_node_pid = -1;
  int child_id;
  std::cin >> child_id;

  if(is_exists(child_id)){
  std::cout << "Error: Already exists" << std::endl;
  return;
  }

  int fork_pid = fork();
  if(fork_pid == -1) {
  std::cout << "Error:fork: " << strerror(errno) << std::endl;
  return;
  }

  if(fork_pid == 0) {
  execl("client", "server", to_string(child_id).c_str(), socket_name.c_str(), NULL);
  }

  std::string parent_pub_socket_name = createNameSocketToParent(child_id);
  sockets_of_children.push_back(parent_pub_socket_name);
  sub_sock.connect(parent_pub_socket_name);

  new_node_pid = fork_pid;

  insert(tree, child_id);
std::cout << "OK: " << new_node_pid << std::endl;
}

void exec() {
  std::string text, pattern;
  int node_id;
```

```
90
91    std::cin >> node_id;
92    std::cin >> text;
93    std::cin >> pattern;
94
95    if(!is_exists(node_id)) {
96    std::cout << "Error:" << node_id << ": Not found" << std::endl;
97    return;
98    }
99
100   zmq::message_t data = fillMessageExec(node_id, text, pattern);
101   publish_socket.send(data);
102
103   zmq::message_t receive_data;
104   Header* header;
105   receive_msg(receive_data);
106   exec_body_answer answer = getMessageExecAnswer(receive_data);
107
108   std::cout << "OK:" << node_id << ": [";
109   if(answer.entries.size() > 0)
110   std::cout << answer.entries[0];
111   for(int i = 1; i < answer.entries.size(); ++i) {
112   std::cout << "; " << answer.entries[i];
113   }
114   std::cout << "]" << std::endl;
115 }
116
117 void ping() {
118   int dest_id;
119   std::cin >> dest_id;
120   if(is_exists(dest_id))
121   std::cout << "OK: 1" << std::endl;
122   else
123   std::cout << "Error: Not found" << std::endl;
124 }
125
126 void removeNode(){
127   int node_id;
128   cin >> node_id;
129   if (!find(tree, node_id)) {
130   cout << "Error: Not found" << endl;
131   return;
132   }
133   if(!is_exists(node_id)){
134   cout << "Error: Node is unavailable" << endl;
135   return;
136   }
137
138   for(int node : getChildren(tree, node_id)){
139   zmq::message_t data = fillMessageRemove(node);
140   publish_socket.send(data);
141   }
142   free_tree(search(tree, node_id));
143   cout << "OK";
144 }
145
146 void server_run(){
147   std::string input_cmd;
148   while(std::cin >> input_cmd) {
149     if(input_cmd == "create")
```

```
150     create();
151     else if(input_cmd == "exec")
152     exec();
153     else if(input_cmd == "ping")
154     ping();
155     else if(input_cmd == "remove")
156     removeNode();
157     else {
158     std::cout << "Not correct" << std::endl;
159     }
160 
161   }
162 }
163 
164 
165 int main() {
166   server_init();
167   server_run();
168   sub_sock.close();
169   publish_socket.close();
170   context.close();
171 }
```

Файл client.cpp

```
1  #include <iostream>
2  #include <string>
3  #include "zmq.hpp"
4  #include "message.h"
5  #include "ZMQUtils.h"
6  
7  
8  static zmq::context_t context;
9  static zmq::socket_t pub_sock_to_children(context, ZMQ_PUB);
10 static zmq::socket_t pub_sock_to_parent(context, ZMQ_PUB);
11 static zmq::socket_t sub_sock(context, ZMQ_SUB);
12 
13 static std::string pub_socket_to_children_name;
14 static std::string pub_socket_to_server_name;
15 
16 static std::string parent_socket_name;
17 static std::vector<std::string> children_sockets_name;
18 
19 static int node_id;
20 static bool run;
21 
22 void client_init(int argc, char* argv[]){
23   node_id = atoi(argv[1]);
24 
25   parent_socket_name = argv[2];
26 
27   pub_socket_to_children_name = createNameSocketToChildren(node_id);
28   pub_socket_to_server_name = createNameSocketToParent(node_id);
29 
30   pub_sock_to_children.bind(pub_socket_to_children_name);
31   pub_sock_to_parent.bind(pub_socket_to_server_name);
32 
33   sub_sock.connect(parent_socket_name);
34   sub_sock.setsockopt(ZMQ_SUBSCRIBE, 0, 0);
35 
```

```
36    run = true;
37  }
38
39  void exec(zmq::message_t& data) {
40    ExecBody body = getMessageExec(data);
41    std::vector<int> entries;
42
43    std::string::size_type position = 0;
44    while(std::string::npos != (position = body.text.find(body.pattern, position))){
45    entries.push_back(position);
46    ++position;
47    }
48
49    zmq::message_t answer = fillMessageExecAnswer(entries);
50    pub_sock_to_parent.send(answer);
51  }
52
53  void ping(zmq::message_t& data) {
54    zmq::message_t answer = fillMessagePingAnswer(node_id);
55    pub_sock_to_parent.send(answer);
56  }
57  void removeNode(Header* header) {
58    pub_sock_to_parent.close();
59    sub_sock.close();
60    context.close();
61    exit(0);
62  }
63
64
65  void task(zmq::message_t& data){
66    Header* header = getMessageHeader(data);;
67    if(header->type == messageExec)
68    exec(data);
69    else if(header->type == messagePing)
70    ping(data);
71    else if(header->type == messageRemove)
72    removeNode(header);
73  }
74
75  void client_run(){
76    while(run){
77      zmq::message_t data;
78      sub_sock.recv(data);
79      Header* header = getMessageHeader(data);
80      if(header->to_id_node == node_id)
81      task(data);
82      if(header->to_id_node != node_id){
83      if(header->direction == toServer)
84        pub_sock_to_parent.send(data);
85      else
86        pub_sock_to_children.send(data);
87      }
88    }
89  }
90
91
92  int main(int argc, char* argv[]) {
93    client_init(argc, argv);
94    client_run();
95  }
```

Тест кейсы

```
create 3
OK: 12963
create 5
OK: 12969
create 7
OK: 12972
ping 5
OK: 1
ping 3
OK: 1
ping 13
Error: Not found
exec 5 abbaba ab
OK:5: [0; 3]
exec 5 ccc c
OK:5: [0; 1; 2]
remove 5
OK
ping 5
Error: Not found
```

Вывод

Данная лабораторная работа была направлена на изучение технологии очереди сообщений, на основое которой необходимо было построить сеть с заданной топологией. В ходе выполнения лабороторной работе, оказалось, что работать с очередью сообщений **ZMQ** так же удобно как с **pipe** и **shared memory**. Навыки, приобретенные в данной лабораторной работе, являются очень полезными, так как во многих проектах используется очередь сообщений.