Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6 по курсу**

**«Операционные системы»**

**Сервера сообщений**

| Студент | Гришин Алексей Юрьевич |
| --- | --- |
| Группа | М8О-208Б-21 |
| Вариант | 43 |
| Преподаватель | Соколов Андрей Алексеевич |
| Оценка | 5 |
| Дата | 29.12.2022 |
| Подпись |  |

Москва, 2022.

## **Постановка задачи**

### **Цель работы**

Целью является приобретение практических навыков в:

* Управлении серверами сообщений
* Применение отложенных вычислений
* Интеграция программных систем друг с другом

### **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 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: create <id> [<parent>]
* Исполнение команды на вычислительном узле: exec <id> [<param>]
* Проверка доступности узла (формат указан в варианте)
* Удаление узла: remove <id>

### **Вариант 43**

#### **Топология**

Все вычислительные узлы находятся в идеально сбалансированном бинарном дереве.

#### **Набор команд**

Необходимо реализовать таймер. Формат команды сохранения значения:

exec id subcommand

где, subcommand – одна из трех команд:

1. start – запустить таймер
2. stop – остановить таймер
3. time – показать время локального таймера в миллисекундах

#### **Команда проверки**

Формат команды: pingall

Вывод всех недоступных узлов вывести разделенные через точку запятую.

## **Общие сведения о программе**

Основная команда компилируется из файла main.c. Использует следующие вспомогательные модули: worker\_tree, worker, queue, user\_commands, mq. Также, используются заголовочные файлы string.h, sys/wait.h, stdio.h.

Программа, представляющая исполнительный узел, компилируется из файла node.c и использует вспомогательные модули mq, worker, time. Также, используются заголовочные файлы stdlib, string, unistd.h, sys/wait.h, stdio.h.

### **Системные вызовы**

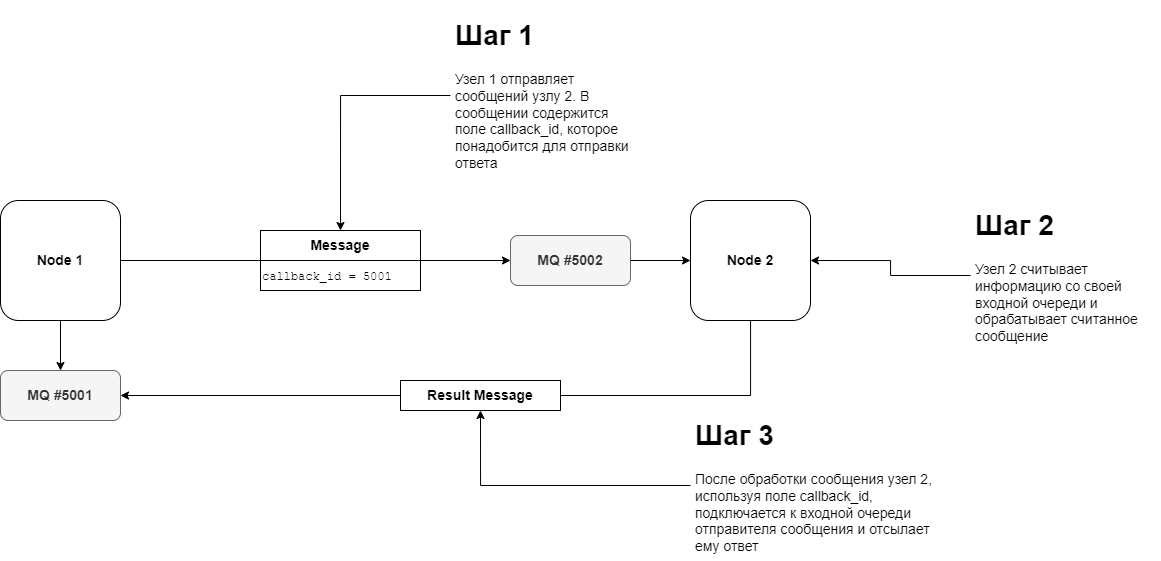
1. zmq\_ctx\_new() – создает новый ZeroMQ контекст. В нем мы можем запустить сокеты. Контекст является потокобезопасным.
2. zmq\_socket(ctx, type) – создает новый сокет. Первым параметром передается контекст, относительно которого создается сокет. Вторым параметром указывается тип паттерна соединения сокета: PUSH / PULL, SEND / RECV, PAIR.
3. zmq\_setsockopt(socket, name, value, length) – позволяет определить значение некоторого параметра сокета. Название сокета указывается в параметре name. Значение и размер значения указываются в параметрах value и length соответственно.
4. zmq\_bind(socket, endpoint) – подключает socket к endpoint, который представляет собой строку-адрес, состоящую из IP и PORT. Далее сокет является серверным. То есть, принимает входные соединения.
5. zmq\_connect(socket, endpoint) – подключает сокет к endpoint. Далее сокет играет роль клиентского.
6. zmq\_msg\_init(msg) – инициализирует объект сообщения, выделяя память под него.
7. zmq\_msg\_init\_size(msg, size) – выделяет необходимое количество ресурсов для сообщения.
8. zmq\_msg\_data(msg) – возвращает указатель на начало области памяти, где хранятся данные сообщения
9. zmq\_msg\_send(msg, socket, flag) – отправляет сообщение msg по сокету socket. Дополнительно можно указать опции в параметре flag. Например, флаг ZMQ\_DONTWAIT устанавливает операции неблокирующий режим.
10. zmq\_msg\_close(msg) – очищает память, занятую под сообщение
11. zmq\_msg\_recv(msg, socket, flags) – считывает входящее сообщение с сокета socket. Результат записывается в сообщение msg. Также, можно указать специальные опции при получении сообщения. Указываются они в параметре flags. Среди них есть флаг ZMQ\_DONTWAIT, который делает операцию чтения неблокирующей
12. zmq\_msg\_size(msg) – возвращает количество байтов, занимаемых содержанием сообщения
13. zmq\_ctx\_destroy(ctx) – удаляет ранее созданный контекст
14. zmq\_close(socket) – удаляет ранее созданный сокет.

### **Описание**

Библиотека ZeroMQ основана на сокетах. Сокеты – один из способов межпроцессорного взаимодействия, который позволяет общаться процессам как на одной ЭВМ, так и на разных. Главное – чтобы они были соединены сетью. Но, из названия следует, что библиотека должна быть связана с очередью сообщений, а мы взаимодействуем лишь с сокетами. Дело в том, что при установлении соединения между двумя сокетами создается очередь сообщений, в которую и записываются или считываются сообщения. Иными словами, сокеты являются лишь оберткой для работы с очередями сообщений.

Для общения по сети сокеты используют стек протоколов TCP / IP, следуя которому каждому узлу сети соответствует пара из адреса и порта. **Адрес** – битовая структура, размер которой зависит от версии протокола. Так, для протокола IPv4 адрес имеет размер в 32 бита, а для IPv6 – 128 битов. **Порт** – целое число в диапазоне от . Порты созданы с целью того, чтобы при сгруппировать сообщения, посылаемые ЭВМ, на принадлежность каждому процессу, чтобы не возникло ситуации, что процесс A считал сообщение, предназначенное процессу B.

Для организации общения между узлами вычислительной системы я использовал паттерн обмена сообщениями Pipeline. При таком паттерне мы ограничиваем все сокеты, участвующие в обмене сообщения на PUSH и PULL сокеты. Задача PUSH сокетов заключается в отправке сообщений, а задача PULL сокетов – в чтении отправленных сообщений. По моему мнению, такой паттерн ближе всего относится к идее очереди сообщений. Процесс обмена сообщениями между двумя узлами приведен ниже.



При создании очереди сообщений ее идентификатор генерируется самостоятельно. Если посмотреть на реализацию создания очередей сообщений, то мы увидим, что на самом деле создается PULL сокет, порт которого подбирается автоматически. Идентификатор очереди на самом деле является портом, который прослушивает соответствующий сокет.

Чтобы вычислительная система следовала топологии, указанной в варианте задания, я выстроил соответствующую иерархии из вычислительных узлов. То есть, если в топологии указано, что узел 2 является дочерним по отношению к узлу 1, то это же отношение иерархии сохраняется и на их процессы (процесс, соответствующий узлу 1, является родительским по отношению к процессу, соответствующему узлу 2). Отправка сообщения осуществляется через broadcast. То есть, изначально корневой узел отправляет сообщение, полученное от пользователя, своим дочерним узлам. Далее, дочерние узлы, получив сообщение от родительского либо обрабатывают его и возвращают результат обратно родительскому узлу, либо передают сообщение своим дочерним узлам и ждут ответа от них. Если при получении ответов от дочерних узлов оба оказались отрицательными, то только в этом случае обратно возвращается отрицательный ответ.

Проверка на работоспособность узла осуществляется через timeout. Устанавливается определенное количество миллисекунд, после истечения которых узел считается неактивным.

Также, было создано “мнимое” представление топологии узлов в виде структуры данных бинарного дерева. Такое дерево будет играть роль виртуальной копии иерархии узлов, что позволит быстро определить, какому узлу необходимо послать запрос на создание дочернего узла, при этом поддерживая сбалансированность двоичного дерева.

### **Исходный код**

#### **mq/mq.c**

| #include "mq.h"  #include <zmq.h>  #include <stdlib.h>  #include <assert.h>  #include <string.h>  #include <error.h>  int listeners\_counter = 0;  static char \* \_\_mq\_server\_address = "0.0.0.0";  static int \_\_max\_endpoint\_size = 256;  #define HANDLE\_ERROR \  exit(errno)  int \_\_auto\_bind(void \* socket) {  char endpoint[\_\_max\_endpoint\_size];  char addr[\_\_max\_endpoint\_size];  size\_t endpoint\_size = \_\_max\_endpoint\_size;  int port;  sprintf(endpoint, "tcp://%s:\*", \_\_mq\_server\_address);  if (zmq\_bind(socket, endpoint) != 0)  return -1;  zmq\_getsockopt(socket, ZMQ\_LAST\_ENDPOINT, & endpoint, & endpoint\_size);  sscanf(endpoint, "tcp://0.0.0.0:%d", & port);  # // TODO: добавить зависимость от \_\_mq\_server\_address:  return port;  }  void mq\_create(MessageQueue \* mq) {  mq -> context = zmq\_ctx\_new();  mq -> socket = zmq\_socket(mq -> context, ZMQ\_PULL);  zmq\_setsockopt(mq -> socket, ZMQ\_IDENTITY, & listeners\_counter, sizeof(listeners\_counter));  listeners\_counter++;  mq -> id = \_\_auto\_bind(mq -> socket);  if (mq -> id == -1) {  HANDLE\_ERROR;  }  }  void mq\_connect(MessageQueue \* mq, int mq\_id) {  char endpoint[\_\_max\_endpoint\_size];  mq -> context = zmq\_ctx\_new();  mq -> socket = zmq\_socket(mq -> context, ZMQ\_PUSH);  mq -> id = mq\_id;  sprintf(endpoint, "tcp://%s:%d", \_\_mq\_server\_address, mq\_id);  int res = zmq\_connect(mq -> socket, endpoint);  assert(res == 0);  }  int mq\_id(MessageQueue \* mq) {  return mq -> id;  }  void mq\_send(MessageQueue \* mq, void \* data, size\_t size) {  zmq\_msg\_t message;  int res;  if (zmq\_msg\_init( & message) != 0) {  HANDLE\_ERROR;  }  if (zmq\_msg\_init\_size( & message, size) != 0) {  HANDLE\_ERROR;  }  memcpy(zmq\_msg\_data( & message), data, size);  if (zmq\_msg\_send( & message, mq -> socket, 0) == -1) {  HANDLE\_ERROR;  }  if (zmq\_msg\_close( & message) != 0) {  HANDLE\_ERROR;  }  }  void mq\_recv(MessageQueue \* mq, void \*\* buff, size\_t \* size) {  zmq\_msg\_t message;  int res;  if (zmq\_msg\_init( & message) != 0) {  HANDLE\_ERROR;  }  if (zmq\_msg\_recv( & message, mq -> socket, 0) == -1) {  if (errno == EAGAIN) {  \* buff = NULL;  \* size = -1;  return;  } else {  HANDLE\_ERROR;  }  }  \* size = zmq\_msg\_size( & message);  \* buff = malloc( \* size);  memcpy( \* buff, zmq\_msg\_data( & message), \* size);  if (zmq\_msg\_close( & message) != 0) {  HANDLE\_ERROR;  }  }  void mq\_set\_timeout(MessageQueue \* mq, int timeout) {  int res;  if (zmq\_setsockopt(mq -> socket, ZMQ\_RCVTIMEO, & timeout, sizeof(timeout))) {  HANDLE\_ERROR;  }  if (zmq\_setsockopt(mq -> socket, ZMQ\_SNDTIMEO, & timeout, sizeof(timeout))) {  HANDLE\_ERROR;  }  }  void mq\_close(MessageQueue \* mq) {  zmq\_close(mq -> socket);  zmq\_ctx\_destroy(mq -> context);  } |
| --- |

#### **mq/mq.h**

| #ifndef \_\_MESSAGE\_QUERY\_H\_\_  #define \_\_MESSAGE\_QUERY\_H\_\_  #include <stddef.h>  #define WORKER\_OK 0  #define WORKER\_ERROR\_TIMEOUT 1  #define WORKER\_ERROR\_INTERNAL 2  typedef struct {  int id;  void \* socket;  void \* context;  } MessageQueue;  void mq\_create(MessageQueue \* mq);  void mq\_connect(MessageQueue \* mq, int mq\_id);  int mq\_id(MessageQueue \* mq);  void mq\_send(MessageQueue \* mq, void \* data, size\_t size);  void mq\_recv(MessageQueue \* mq, void \*\* buff, size\_t \* size);  void mq\_set\_timeout(MessageQueue \* mq, int timeout);  void mq\_close(MessageQueue \* mq);  #endif |
| --- |

#### **queue/queue.c**

| #include "queue.h"  void queue\_init(Queue \* q) {  q -> head = NULL;  }  void queue\_push(Queue \* q, void \* value, size\_t value\_size) {  QElement \* element = (QElement \* ) malloc(sizeof(QElement));  element -> value = value;  element -> value\_size = value\_size;  element -> next = NULL;  QElement \* head = q -> head;  if (head == NULL) {  q -> head = element;  return;  }  while (head -> next != NULL) head = head -> next;  head -> next = element;  }  bool queue\_empty(Queue \* q) {  return q -> head == NULL;  }  void queue\_pop(Queue \* q, void \*\* value, size\_t \* value\_size) {  void \* res\_value = NULL;  size\_t res\_size = -1;  if (q -> head != NULL) {  QElement \* head = q -> head;  q -> head = q -> head -> next;  res\_value = head -> value;  res\_size = head -> value\_size;  free(head);  }  if (value != NULL) \* value = res\_value;  if (value\_size != NULL) \* value\_size = res\_size;  }  void queue\_delete(Queue \* q) {  while (!queue\_empty(q)) queue\_pop(q, NULL, NULL);  } |
| --- |

#### **queue/queue.h**

| #ifndef \_\_QUEUE\_H\_\_  #define \_\_QUEUE\_H\_\_  #include <stdlib.h>  #include <stdbool.h>  typedef struct \_\_QElement {  void \* value;  size\_t value\_size;  struct \_\_QElement \* next;  } QElement;  typedef struct {  QElement \* head;  } Queue;  void queue\_init(Queue \* q);  void queue\_push(Queue \* q, void \* value, size\_t value\_size);  void queue\_pop(Queue \* q, void \*\* value, size\_t \* value\_size);  bool queue\_empty(Queue \* q);  void queue\_delete(Queue \* q);  #endif |
| --- |

#### **timer/timer.c**

| #include "timer.h"  #include <sys/time.h>  #include <stddef.h>  Timemark \_\_get\_time() {  struct timeval timemark;  gettimeofday( & timemark, NULL);  return timemark.tv\_sec \* 1000 LL + timemark.tv\_usec / 1000;  }  void \_\_timer\_update(Timer \* timer) {  Timemark curr\_timemark = \_\_get\_time();  timer -> passed\_time += (curr\_timemark - timer -> last\_timemark);  timer -> last\_timemark = curr\_timemark;  }  void timer\_init(Timer \* timer) {  timer -> passed\_time = 0;  timer -> last\_timemark = \_\_get\_time();  timer -> is\_active = 0;  }  void timer\_start(Timer \* timer) {  timer -> last\_timemark = \_\_get\_time();  timer -> is\_active = 1;  }  void timer\_stop(Timer \* timer) {  \_\_timer\_update(timer);  timer -> is\_active = 0;  }  Milliseconds timer\_time(Timer \* timer) {  if (timer -> is\_active) \_\_timer\_update(timer);  return timer -> passed\_time;  } |
| --- |

#### **timer/timer.h**

| #ifndef \_\_TIMER\_H\_\_  #define \_\_TIMER\_H\_\_  typedef unsigned long Milliseconds;  typedef unsigned long Timemark;  typedef struct {  Milliseconds passed\_time;  Timemark last\_timemark;  int is\_active;  } Timer;  void timer\_init(Timer \* timer);  void timer\_start(Timer \* timer);  void timer\_stop(Timer \* timer);  Milliseconds timer\_time(Timer \* timer);  #endif |
| --- |

#### **worker/worker.c**

| #include "worker.h"  #include <zmq.h>  #include <unistd.h>  #include <stdio.h>  #include <stdlib.h>  #include <assert.h>  #include <string.h>  void \_\_send\_unavailable\_error(WorkerMessage \* message, MessageQueue \* callback) {  WorkerResult result = {  .message\_id = message -> message\_id,  .worker\_id = message -> receiver\_id,  .message\_type = message -> type,  .result\_type = RESULT\_UNAVAILABLE  };  mq\_send(callback, & result, sizeof(result));  }  pid\_t worker\_create(Worker \* worker, char \* execute\_path, int worker\_id) {  MessageQueue callback;  pid\_t pid;  mq\_create( & callback);  pid = fork();  if (pid == 0) {  char callback\_id\_str[256];  char worker\_id\_str[256];  sprintf(callback\_id\_str, "%d", mq\_id( & callback));  sprintf(worker\_id\_str, "%d", worker\_id);  char \* args[] = {  execute\_path,  worker\_id\_str,  callback\_id\_str,  NULL  };  execv(execute\_path, args);  exit(2);  }  if (pid > 0) {  CreateResult \* recv\_data;  size\_t recv\_data\_size;  pid\_t created\_worker\_pid;  mq\_recv( & callback, (void \*\* ) & recv\_data, & recv\_data\_size);  assert(recv\_data != NULL);  mq\_connect( & worker -> mq, recv\_data -> worker\_mq\_id);  worker -> id = worker\_id;  worker -> pid = recv\_data -> worker\_pid;  free(recv\_data);  return worker -> pid;  }  if (pid < 0) {  exit(2);  }  }  void worker\_send\_message(Worker \* worker, WorkerMessage \* message, size\_t message\_size) {  MessageQueue local\_callback, callback;  WorkerMessage \* message\_copy;  void \* recv\_data;  size\_t recv\_data\_size;  mq\_connect( & callback, message -> callback\_id);  mq\_create( & local\_callback);  mq\_set\_timeout( & local\_callback, 1000);  message\_copy = (WorkerMessage \* ) malloc(message\_size);  memcpy(message\_copy, message, message\_size);  message\_copy -> callback\_id = mq\_id( & local\_callback);  mq\_send( & worker -> mq, message\_copy, message\_size);  mq\_recv( & local\_callback, & recv\_data, & recv\_data\_size);  if (recv\_data == NULL)  \_\_send\_unavailable\_error(message, & callback);  else {  mq\_send( & callback, recv\_data, recv\_data\_size);  free(recv\_data);  }  free(message\_copy);  mq\_close( & local\_callback);  mq\_close( & callback);  }  void worker\_delete(Worker \* worker) {  mq\_close( & worker -> mq);  } |
| --- |

#### **worker/worker.h**

| #ifndef \_\_WORKER\_H\_\_  #define \_\_WORKER\_H\_\_  #include "../mq/mq.h"  #include "../queue/queue.h"  #include <stdbool.h>  #include <unistd.h>  typedef struct {  int id;  MessageQueue mq;  pid\_t pid;  } Worker;  typedef enum {  CREATE,  PING,  DELETE,  TIMER\_START,  TIMER\_STOP,  TIMER\_TIME  } MessageType;  typedef enum {  RESULT\_OK,  RESULT\_UNAVAILABLE,  RESULT\_PING,  NOT\_FOUND,  ALREADY\_EXISTS  } ResultType;  typedef struct {  int message\_id;  int receiver\_id;  int callback\_id;  MessageType type;  } WorkerMessage;  typedef struct {  int message\_id;  int receiver\_id;  int callback\_id;  MessageType type;  int created\_worker\_id;  } CreateMessage;  typedef struct {  int message\_id;  int receiver\_id;  int callback\_id;  MessageType type;  } PingMessage;  typedef struct {  int message\_id;  int receiver\_id;  int callback\_id;  MessageType type;  } DeleteMessage;  typedef struct {  int message\_id;  int receiver\_id;  int callback\_id;  MessageType type;  } StartTimerMessage;  typedef struct {  int message\_id;  int receiver\_id;  int callback\_id;  MessageType type;  } StopTimerMessage;  typedef struct {  int message\_id;  int receiver\_id;  int callback\_id;  MessageType type;  } GetTimeMessage;  typedef struct {  int message\_id;  int worker\_id;  MessageType message\_type;  ResultType result\_type;  } WorkerResult;  typedef struct {  int message\_id;  int worker\_id;  MessageType message\_type;  ResultType result\_type;  pid\_t worker\_pid;  int worker\_mq\_id;  } CreateResult;  typedef struct {  int message\_id;  int worker\_id;  MessageType message\_type;  ResultType result\_type;  unsigned long time;  } TimeResult;  typedef struct {  int message\_id;  int worker\_id;  MessageType message\_type;  ResultType result\_type;  Queue unavailable\_nodes;  } PingResult;  pid\_t worker\_create(Worker \* worker, char \* execute\_path, int worker\_id);  void worker\_send\_message(Worker \* worker, WorkerMessage \* message, size\_t message\_size);  void worker\_delete(Worker \* worker);  #endif |
| --- |

#### **worker\_tree/worker\_tree.c**

| #include "worker\_tree.h"  #include <stdlib.h>  #include <stdio.h>  void node\_init(Node \* node, WorkerId id) {  node -> id = id;  node -> left = NULL;  node -> right = NULL;  }  bool node\_full(Node \* node) {  return node -> left != NULL && node -> right != NULL;  }  void node\_add(Node \* parent, Node \* child) {  if (parent -> left == NULL) parent -> left = child;  else if (parent -> right == NULL) parent -> right = child;  }  void tree\_init(WorkerTree \* tree) {  tree -> root = NULL;  }  Result tree\_create\_node(WorkerTree \* tree, WorkerId worker\_id) {  Result res = {  .parent = -1,  .child = worker\_id  };  if (tree -> root == NULL) {  tree -> root = (Node \* ) malloc(sizeof(Node));  node\_init(tree -> root, worker\_id);  return res;  }  Queue q;  queue\_init( & q);  queue\_push( & q, (void \* ) tree -> root, sizeof(tree -> root));  while (!queue\_empty( & q)) {  Node \* node;  queue\_pop( & q, (void \*\* ) & node, NULL);  if (!node\_full(node)) {  Node \* c = (Node \* ) malloc(sizeof(Node));  node\_init(c, worker\_id);  node\_add(node, c);  queue\_delete( & q);  res.parent = node -> id;  return res;  }  queue\_push( & q, node -> left, sizeof(node -> left));  queue\_push( & q, node -> right, sizeof(node -> right));  }  queue\_delete( & q);  }  bool \_\_find\_node(Node \* node, WorkerId worker\_id) {  if (node == NULL) return false;  if (node -> id == worker\_id) return true;  return \_\_find\_node(node -> left, worker\_id) || \_\_find\_node(node -> right, worker\_id);  }  bool tree\_exists(WorkerTree \* tree, WorkerId worker\_id) {  return \_\_find\_node(tree -> root, worker\_id);  }  void \_\_print\_node(Node \* node, int layer) {  if (node == NULL) return;  \_\_print\_node(node -> right, layer + 1);  for (int i = 0; i < layer; i++) printf("\t");  printf("[Worker #%d]\n", node -> id);  \_\_print\_node(node -> left, layer + 1);  }  void print\_tree(WorkerTree \* tree) {  \_\_print\_node(tree -> root, 0);  }  void \_\_delete\_node(Node \* node) {  if (node == NULL) return;  \_\_delete\_node(node -> left);  \_\_delete\_node(node -> right);  free(node);  }  void tree\_delete(WorkerTree \* tree) {  \_\_delete\_node(tree -> root);  }  Node \* \_\_remove\_node(Node \* node, WorkerId worker\_id) {  if (node == NULL) return NULL;  if (node -> id == worker\_id) {  \_\_delete\_node(node);  return NULL;  }  node -> left = \_\_remove\_node(node -> left, worker\_id);  node -> right = \_\_remove\_node(node -> right, worker\_id);  return node;  }  void tree\_remove\_node(WorkerTree \* tree, WorkerId worker\_id) {  tree -> root = \_\_remove\_node(tree -> root, worker\_id);  }  void tree\_get\_nodes(WorkerTree \* tree, Queue \* nodes) {  Queue q;  queue\_init( & q);  queue\_push( & q, (void \* ) tree -> root, sizeof(tree -> root));  while (!queue\_empty( & q)) {  Node \* node;  queue\_pop( & q, (void \*\* ) & node, NULL);  if (node == NULL) continue;  queue\_push(nodes, (void \* ) & node -> id, sizeof(node -> id));  queue\_push( & q, (void \* ) node -> left, sizeof(node -> left));  queue\_push( & q, (void \* ) node -> right, sizeof(node -> right));  }  queue\_delete( & q);  } |
| --- |

#### **worker\_tree/worker\_tree.h**

| #ifndef \_\_WORKER\_TREE\_H\_\_  #define \_\_WORKER\_TREE\_H\_\_  #include "../queue/queue.h"  #include <stdbool.h>  #include <stdlib.h>  typedef int WorkerId;  typedef struct \_\_Node {  WorkerId id;  struct \_\_Node \* left;  struct \_\_Node \* right;  } Node;  typedef struct {  Node \* root;  } WorkerTree;  typedef struct {  WorkerId parent;  WorkerId child;  } Result;  void tree\_init(WorkerTree \* tree);  Result tree\_create\_node(WorkerTree \* tree, WorkerId worker\_id);  void tree\_remove\_node(WorkerTree \* tree, WorkerId worker\_id);  bool tree\_exists(WorkerTree \* tree, WorkerId worker\_id);  void tree\_get\_nodes(WorkerTree \* tree, Queue \* nodes);  void print\_tree(WorkerTree \* tree);  void tree\_delete(WorkerTree \* tree);  #endif |
| --- |

#### **main.c**

| #include "./worker\_tree/worker\_tree.h"  #include "./worker/worker.h"  #include "./queue/queue.h"  #include "user\_commands.h"  #include "./mq/mq.h"  #include <string.h>  #include <sys/wait.h>  #include <stdio.h>  MessageQueue callback;  Worker root\_worker;  WorkerTree tree;  int command\_counter = 1;  WorkerResult base\_result(UserCommand cmd, MessageType message\_type, ResultType result\_type) {  WorkerResult result = {  .message\_id = cmd.command\_id,  .worker\_id = cmd.worker\_id,  .message\_type = message\_type,  .result\_type = result\_type  };  return result;  }  void send\_to\_output(WorkerResult \* result, size\_t result\_size) {  MessageQueue output;  mq\_connect( & output, mq\_id( & callback));  mq\_send( & output, result, result\_size);  mq\_close( & output);  }  void send\_not\_found\_result(UserCommand cmd, MessageType type) {  WorkerResult result = base\_result(cmd, type, NOT\_FOUND);  send\_to\_output( & result, sizeof(result));  }  void send\_already\_exists\_result(UserCommand cmd, MessageType type) {  WorkerResult result = base\_result(cmd, type, ALREADY\_EXISTS);  send\_to\_output( & result, sizeof(result));  }  int get\_command\_id() {  return command\_counter++;  }  UserCommand scan\_user\_command() {  char command\_type[256];  int command\_id = get\_command\_id();  UserCommand command;  printf("(%d) > ", command\_id);  scanf("%s", command\_type);  if (!strcmp(command\_type, "create")) {  command.type = USER\_CREATE;  scanf("%d", & command.worker\_id);  }  if (!strcmp(command\_type, "remove")) {  command.type = USER\_REMOVE;  scanf("%d", & command.worker\_id);  }  if (!strcmp(command\_type, "ping")) {  command.type = USER\_PING;  command.worker\_id = -1;  }  if (!strcmp(command\_type, "exec")) {  scanf("%d", & command.worker\_id);  char action[256];  scanf("%s", action);  if (!strcmp(action, "start")) command.type = USER\_EXEC\_START;  if (!strcmp(action, "stop")) command.type = USER\_EXEC\_STOP;  if (!strcmp(action, "time")) command.type = USER\_EXEC\_TIME;  }  if (!strcmp(command\_type, "close")) {  command.type = USER\_CLOSE;  command.worker\_id = -1;  }  command.command\_id = command\_id;  return command;  }  void result\_type\_repr(char \* str, ResultType type) {  switch (type) {  case RESULT\_OK:  strcpy(str, " : ok");  break;  case RESULT\_UNAVAILABLE:  strcpy(str, " : error : node is unavailable");  break;  case NOT\_FOUND:  strcpy(str, " : error : not found");  break;  case ALREADY\_EXISTS:  strcpy(str, " : error : already exists");  break;  case RESULT\_PING:  strcpy(str, " : ok");  break;  }  }  void handle\_result\_messages() {  WorkerResult \* result;  size\_t result\_size;  char result\_type\_str[256];  mq\_recv( & callback, (void \*\* ) & result, & result\_size);  result\_type\_repr(result\_type\_str, result -> result\_type);  printf("[Main] <<< (%d)", result -> message\_id);  if (result -> message\_type == TIMER\_TIME || result -> message\_type == TIMER\_START || result -> message\_type == TIMER\_STOP) {  printf(" : %d", result -> worker\_id);  }  printf("%s", result\_type\_str);  if (result -> result\_type == NOT\_FOUND || result -> result\_type == ALREADY\_EXISTS || result -> result\_type == RESULT\_UNAVAILABLE) {  printf("\n");  return;  }  if (result -> result\_type == RESULT\_PING) {  PingResult \* ping\_result = (PingResult \* ) result;  printf(": ");  while (!queue\_empty( & ping\_result -> unavailable\_nodes)) {  int \* unavailable\_node\_id;  queue\_pop( & ping\_result -> unavailable\_nodes, (void \*\* ) & unavailable\_node\_id, NULL);  printf("%d; ", \* unavailable\_node\_id);  }  }  if (result -> message\_type == CREATE) {  CreateResult \* create\_result = (CreateResult \* ) result;  printf(" : %d", create\_result -> worker\_pid);  }  if (result -> message\_type == TIMER\_TIME) {  TimeResult \* time\_result = (TimeResult \* ) result;  printf(" : %lu", time\_result -> time);  }  printf("\n");  }  void handle\_create\_command(UserCommand cmd) {  Result add\_node\_result;  CreateMessage message;  if (tree\_exists( & tree, cmd.worker\_id)) {  send\_already\_exists\_result(cmd, CREATE);  return;  }  add\_node\_result = tree\_create\_node( & tree, cmd.worker\_id);  message.message\_id = cmd.command\_id;  message.receiver\_id = add\_node\_result.parent;  message.callback\_id = mq\_id( & callback);  message.type = CREATE;  message.created\_worker\_id = add\_node\_result.child;  worker\_send\_message( & root\_worker, (WorkerMessage \* ) & message, sizeof(message));  }  void handle\_remove\_command(UserCommand cmd) {  Result remove\_node\_result;  DeleteMessage message;  if (!tree\_exists( & tree, cmd.worker\_id)) {  send\_not\_found\_result(cmd, DELETE);  return;  }  tree\_remove\_node( & tree, cmd.worker\_id);  message.message\_id = cmd.command\_id;  message.receiver\_id = cmd.worker\_id;  message.callback\_id = mq\_id( & callback);  message.type = DELETE,  worker\_send\_message( & root\_worker, (WorkerMessage \* ) & message, sizeof(message));  }  void handle\_exec\_start\_command(UserCommand cmd) {  StartTimerMessage message = {  .message\_id = cmd.command\_id,  .receiver\_id = cmd.worker\_id,  .callback\_id = mq\_id( & callback),  .type = TIMER\_START  };  if (!tree\_exists( & tree, cmd.worker\_id)) {  send\_not\_found\_result(cmd, TIMER\_START);  return;  }  worker\_send\_message( & root\_worker, (WorkerMessage \* ) & message, sizeof(message));  }  void handle\_exec\_stop\_command(UserCommand cmd) {  StopTimerMessage message = {  .message\_id = cmd.command\_id,  .receiver\_id = cmd.worker\_id,  .callback\_id = mq\_id( & callback),  .type = TIMER\_STOP  };  if (!tree\_exists( & tree, cmd.worker\_id)) {  send\_not\_found\_result(cmd, TIMER\_STOP);  return;  }  worker\_send\_message( & root\_worker, (WorkerMessage \* ) & message, sizeof(message));  }  void handle\_exec\_time\_command(UserCommand cmd) {  GetTimeMessage message = {  .message\_id = cmd.command\_id,  .receiver\_id = cmd.worker\_id,  .callback\_id = mq\_id( & callback),  .type = TIMER\_TIME  };  if (!tree\_exists( & tree, cmd.worker\_id)) {  send\_not\_found\_result(cmd, TIMER\_TIME);  return;  }  worker\_send\_message( & root\_worker, (WorkerMessage \* ) & message, sizeof(message));  }  void handle\_ping\_command(UserCommand cmd) {  Queue nodes, unavailable\_nodes;  MessageQueue local\_callback, output;  PingResult result;  queue\_init( & nodes);  queue\_init( & unavailable\_nodes);  mq\_create( & local\_callback);  mq\_connect( & output, mq\_id( & callback));  tree\_get\_nodes( & tree, & nodes);  while (!queue\_empty( & nodes)) {  int \* worker\_id;  queue\_pop( & nodes, (void \*\* ) & worker\_id, NULL);  PingMessage ping\_message = {  .message\_id = -1,  .receiver\_id = \* worker\_id,  .callback\_id = mq\_id( & local\_callback),  .type = PING  };  WorkerResult \* ping\_result;  size\_t ping\_result\_size;  worker\_send\_message( & root\_worker, (WorkerMessage \* ) & ping\_message, sizeof(ping\_message));  mq\_recv( & local\_callback, (void \*\* ) & ping\_result, & ping\_result\_size);  if (ping\_result -> result\_type == RESULT\_UNAVAILABLE)  queue\_push( & unavailable\_nodes, (void \* ) worker\_id, sizeof( \* worker\_id));  }  result.message\_id = cmd.command\_id;  result.worker\_id = -1;  result.message\_type = PING;  result.result\_type = RESULT\_PING;  memcpy( & result.unavailable\_nodes, & unavailable\_nodes, sizeof(unavailable\_nodes));  mq\_send( & output, & result, sizeof(result));  }  void handle\_close\_command(UserCommand cmd) {  MessageQueue local\_callback;  mq\_create( & local\_callback);  DeleteMessage message = {  .message\_id = cmd.command\_id,  .receiver\_id = root\_worker.id,  .callback\_id = mq\_id( & local\_callback),  .type = DELETE  };  worker\_send\_message( & root\_worker, (WorkerMessage \* ) & message, sizeof(message));  waitpid(root\_worker.pid, NULL, 0);  exit(0);  }  int main(void) {  mq\_create( & callback);  tree\_init( & tree);  worker\_create( & root\_worker, "worker", -1);  tree\_create\_node( & tree, root\_worker.id);  while (1) {  UserCommand cmd = scan\_user\_command();  switch (cmd.type) {  case USER\_CREATE:  handle\_create\_command(cmd);  print\_tree( & tree);  break;  case USER\_PING:  handle\_ping\_command(cmd);  break;  case USER\_REMOVE:  handle\_remove\_command(cmd);  print\_tree( & tree);  break;  case USER\_EXEC\_START:  handle\_exec\_start\_command(cmd);  break;  case USER\_EXEC\_STOP:  handle\_exec\_stop\_command(cmd);  break;  case USER\_EXEC\_TIME:  handle\_exec\_time\_command(cmd);  break;  case USER\_CLOSE:  handle\_close\_command(cmd);  break;  }  handle\_result\_messages();  }  } |
| --- |

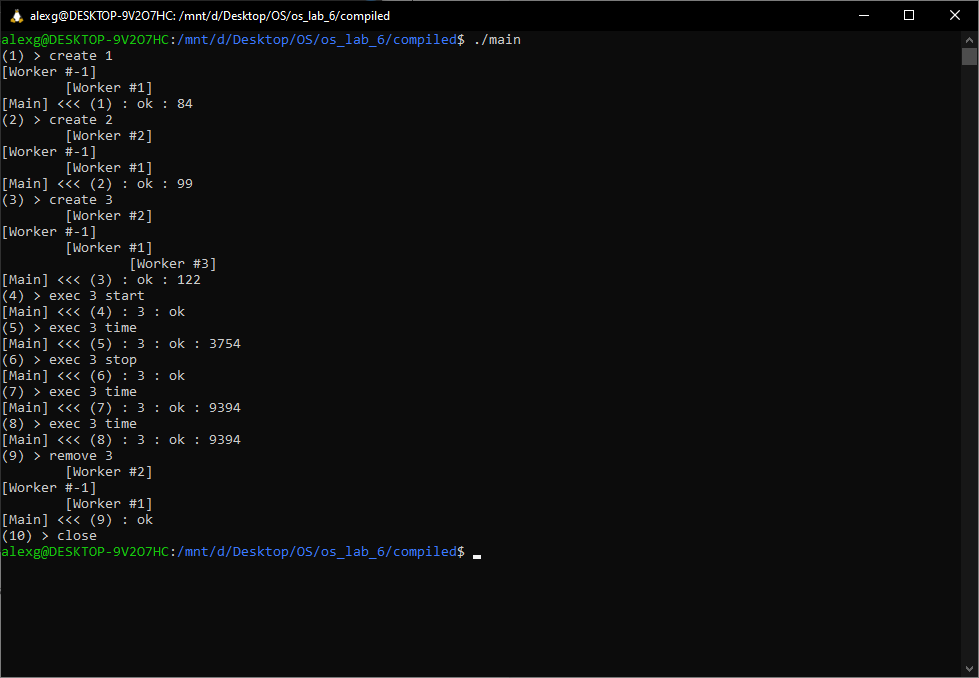
#### **node.c**

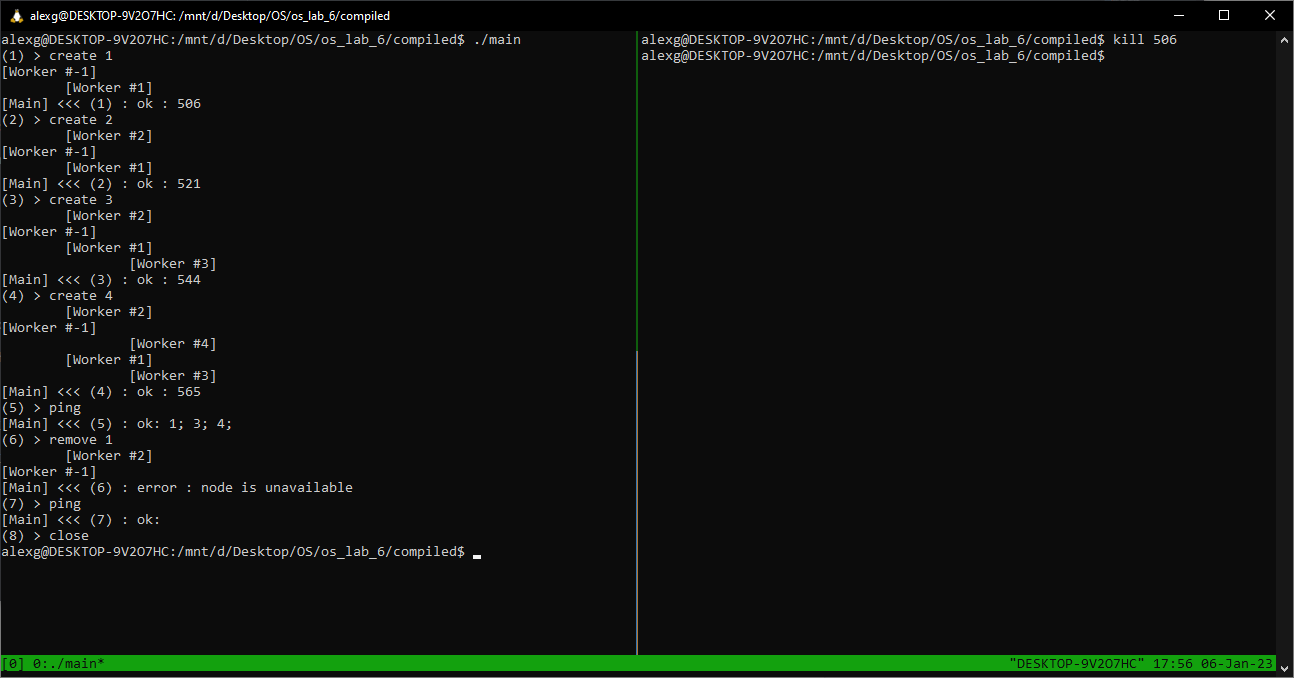
| #include "./mq/mq.h"  #include "./worker/worker.h"  #include "./timer/timer.h"  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <sys/wait.h>  #include <stdio.h>  int id;  MessageQueue mq;  Timer timer;  #define WORKERS\_AMOUNT 2  int created\_workers = 0;  Worker child\_workers[WORKERS\_AMOUNT];  void delete\_child\_worker(int worker\_id) {  int i;  int found = 0;  for (i = 0; i < created\_workers; i++) {  if (child\_workers[i].id == worker\_id) {  found = 1;  break;  }  }  for (int j = i; j < created\_workers - 1; j++) {  memcpy( & child\_workers[j], & child\_workers[j + 1], sizeof(child\_workers[j]));  }  if (found) created\_workers--;  if (created\_workers < 0) created\_workers = 0;  }  void init\_worker(int worker\_id, int callback\_id) {  MessageQueue callback;  int worker\_mq\_id;  mq\_create( & mq);  id = worker\_id;  mq\_connect( & callback, callback\_id);  CreateResult result = {  .worker\_id = id,  .message\_type = CREATE,  .result\_type = RESULT\_OK,  .worker\_pid = getpid(),  .worker\_mq\_id = mq\_id( & mq)  };  mq\_send( & callback, & result, sizeof(result));  timer\_init( & timer);  }  void handle\_ping\_message(PingMessage \* message) {  MessageQueue callback;  WorkerResult result = {  .message\_id = message -> message\_id,  .worker\_id = id,  .message\_type = message -> type,  .result\_type = RESULT\_OK  };  mq\_connect( & callback, message -> callback\_id);  mq\_send( & callback, & result, sizeof(result));  mq\_close( & callback);  }  void handle\_create\_message(CreateMessage \* message) {  MessageQueue callback;  CreateResult result = {  .message\_id = message -> message\_id,  .worker\_id = id,  .message\_type = message -> type,  .result\_type = RESULT\_OK,  .worker\_pid = -1,  .worker\_mq\_id = -1  };  mq\_connect( & callback, message -> callback\_id);  if (created\_workers >= WORKERS\_AMOUNT) {  mq\_close( & callback);  exit(2);  }  result.worker\_pid = worker\_create( & child\_workers[created\_workers], "worker", message -> created\_worker\_id);  result.worker\_mq\_id = mq\_id( & child\_workers[created\_workers].mq);  created\_workers++;  mq\_send( & callback, & result, sizeof(result));  mq\_close( & callback);  }  void handle\_delete\_message(DeleteMessage \* message) {  DeleteMessage local\_message;  MessageQueue local\_callback, callback;  WorkerResult result = {  .message\_id = message -> message\_id,  .worker\_id = id,  .message\_type = message -> type,  .result\_type = RESULT\_OK  };  mq\_create( & local\_callback);  mq\_connect( & callback, message -> callback\_id);  memcpy( & local\_message, message, sizeof(local\_message));  local\_message.callback\_id = mq\_id( & local\_callback);  for (int i = 0; i < created\_workers; i++) {  local\_message.receiver\_id = child\_workers[i].id;  mq\_send( & child\_workers[i].mq, & local\_message, sizeof(local\_message));  }  mq\_send( & callback, & result, sizeof(result));  mq\_close( & local\_callback);  mq\_close( & callback);  exit(0);  }  void handle\_timer\_start\_message(StartTimerMessage \* message) {  MessageQueue callback;  WorkerResult result = {  .message\_id = message -> message\_id,  .worker\_id = id,  .message\_type = message -> type,  .result\_type = RESULT\_OK  };  mq\_connect( & callback, message -> callback\_id);  timer\_start( & timer);  mq\_send( & callback, & result, sizeof(result));  mq\_close( & callback);  }  void handle\_timer\_stop\_message(StopTimerMessage \* message) {  MessageQueue callback;  WorkerResult result = {  .message\_id = message -> message\_id,  .worker\_id = id,  .message\_type = message -> type,  .result\_type = RESULT\_OK  };  mq\_connect( & callback, message -> callback\_id);  timer\_stop( & timer);  mq\_send( & callback, & result, sizeof(result));  mq\_close( & callback);  }  void handle\_timer\_time\_message(GetTimeMessage \* message) {  MessageQueue callback;  TimeResult result = {  .message\_id = message -> message\_id,  .worker\_id = id,  .message\_type = message -> type,  .result\_type = RESULT\_OK,  .time = 0  };  mq\_connect( & callback, message -> callback\_id);  result.time = timer\_time( & timer);  mq\_send( & callback, & result, sizeof(result));  mq\_close( & callback);  }  void delegate\_message(WorkerMessage \* message, size\_t message\_size) {  MessageQueue callback, local\_callback;  mq\_connect( & callback, message -> callback\_id);  mq\_create( & local\_callback);  mq\_set\_timeout( & local\_callback, 100);  message -> callback\_id = mq\_id( & local\_callback);  for (int i = 0; i < created\_workers; i++)  mq\_send( & child\_workers[i].mq, message, message\_size);  int res = 0;  for (int i = 0; i < created\_workers; i++) {  WorkerResult \* recv\_data;  size\_t recv\_data\_size;  mq\_recv( & local\_callback, (void \*\* ) & recv\_data, & recv\_data\_size);  if (recv\_data == NULL)  continue;  if (recv\_data -> result\_type != RESULT\_UNAVAILABLE) {  mq\_send( & callback, recv\_data, recv\_data\_size);  mq\_close( & local\_callback);  free(recv\_data);  res = 1;  break;  }  free(recv\_data);  }  if (message -> type == DELETE) {  delete\_child\_worker(message -> receiver\_id);  }  if (res) return;  WorkerResult result = {  .message\_id = message -> message\_id,  .worker\_id = message -> receiver\_id,  .message\_type = message -> type,  .result\_type = RESULT\_UNAVAILABLE  };  mq\_send( & callback, & result, sizeof(result));  }  int main(int argc, char \* argv[]) {  WorkerMessage \* message;  size\_t message\_size;  init\_worker(atoi(argv[1]), atoi(argv[2]));  while (1) {  mq\_recv( & mq, (void \*\* ) & message, & message\_size);  if (message -> receiver\_id != id) {  delegate\_message(message, message\_size);  free(message);  continue;  }  switch (message -> type) {  case PING:  handle\_ping\_message((PingMessage \* ) message);  break;  case CREATE:  handle\_create\_message((CreateMessage \* ) message);  break;  case DELETE:  handle\_delete\_message((DeleteMessage \* ) message);  break;  case TIMER\_START:  handle\_timer\_start\_message((StartTimerMessage \* ) message);  break;  case TIMER\_STOP:  handle\_timer\_stop\_message((StopTimerMessage \* ) message);  break;  case TIMER\_TIME:  handle\_timer\_time\_message((GetTimeMessage \* ) message);  break;  }  free(message);  }  return 0;  } |
| --- |

#### **user\_commands.h**

| #ifndef \_\_USER\_COMMANDS\_H\_\_  #define \_\_USER\_COMMANDS\_H\_\_  typedef enum {  USER\_CREATE,  USER\_PING,  USER\_REMOVE,  USER\_EXEC\_START,  USER\_EXEC\_STOP,  USER\_EXEC\_TIME,  USER\_CLOSE  } UserCommandType;  typedef struct {  int command\_id;  UserCommandType type;  int worker\_id;  } UserCommand;  #endif |
| --- |

### **Пример работы**





## **Вывод**

В ходе выполнения лабораторной работы я получил опыт работы с библиотекой ZeroMQ, ознакомился с основными концептами идеи сервера очереди сообщений. Также, я изучил основные классификации очередей сообщений: синхронное или асинхронное взаимодействие, сохранение на диск, пересылка по сети, использование брокера.

Использование очереди сообщений в вычислительных сетях заметно повышает ее эффективность, однако, добавляет не меньше трудностей, так как теперь возникает необходимость в мониторинге системы в реальном времени, чтобы отслеживать моменты, когда какие-либо узлы стали неработоспособными. В вычислительных сетях основная трудность заключается в организации обмена сообщениями, так как при отправке мы не можем быть уверенными, что ответ нам придет, или что сообщение вообще пришло получателю.

По своей идее архитектура вычислительных сетей очень похожа на микросервисную архитектуру. Отличие заключается лишь в назначении узлов. В микросервисной архитектуре каждый узел имеет более широкое применение, так как выполнять они могут разные задачи, в то время как в вычислительных сетях все узлы по своей сути являются простыми репликами.