

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе

Дисциплина: Технологии компьютерных сетей

Тема: Программирование сокетов протоколов TCP и UDP

Выполнили студенты гр. 43501/3

Мальцев М.С.

Преподаватель

Зозуля А.В

«_____» _____ 2018 г.

Санкт-Петербург
2018

1 Цель работы.

Ознакомиться с принципами работы протоколам TCP и UDP.

2 Задание

Система распределенных математических расчетов

Разработать распределенную систему, состоящую из приложений клиента и сервера, для распределенного расчета простых чисел. Информационная система должна обеспечивать параллельную работу нескольких клиентов.

Серверное приложение должно реализовывать следующие функции:

1. Прослушивание определенного порта
2. Обработка запросов на подключение по этому порту клиентов
3. Поддержка одновременной работы нескольких клиентов с использованием механизма нитей и средств синхронизации доступа к разделяемым между нитями ресурсам.
4. Принудительное отключение конкретного клиента
5. Хранение рассчитанных клиентами простых чисел, а также текущей нижней границы диапазона для нового запроса на расчет
6. Выдача клиентам максимального рассчитанного простого числа
7. Выдача клиентам последних N рассчитанных простых чисел
8. Выдача клиентам необходимого диапазона расчета чисел
9. Сохранение состояния при выключении сервера

Клиентское приложение должно реализовывать следующие функции:

1. Возможность параллельной работы нескольких клиентов с одного или нескольких IP-адресов
2. Установление соединения с сервером (возможно, с регистрацией на сервере)
3. Разрыв соединения
4. Обработка ситуации отключения сервером
5. Получение от сервера и вывод максимально рассчитанного простого числа

6. Получение от сервера и вывод последних N рассчитанных простых чисел
7. Получение от сервера диапазона расчета простых чисел (нижнюю грань выдает сервер, количество проверяемых чисел - клиент)
8. Расчет простых чисел в требуемом диапазоне (имеет смысл проверять остатки от деления на все нечетные числа в пределах $\sqrt{N_{\max}}+1$)
9. Передача серверу набора рассчитанных простых чисел

3 Ход работы

3.1 Прикладной протокол

Для решения поставленной задачи был разработан прикладной протокол передачи данных. Планируемая длина пакета в протоколе равна 26 байт.

Запросы :

- MAX – Выдача максимального рассчитанного простого числа
[MAX]?;
- LAST [number] – Выдача последних N рассчитанных простых чисел
[LAST]?[number];
- FROM – Выдача диапазона для расчета чисел
[FROM]?;
- POST [number] – Загрузка на сервер рассчитанного числа
[POST]?[number];

Ответы :

- 200; - OK [data]
- 400; - Bad Request

3.2 Описание архитектуры

Архитектура серверного приложения изображена на UML диаграмме, рисунок 3.1.

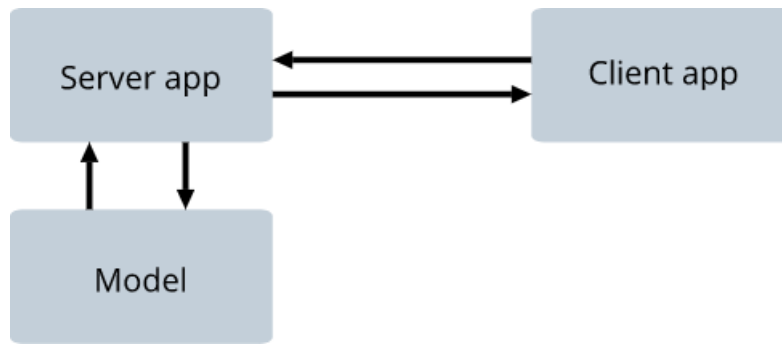


Рис. 3.1: UML-диаграмма компонентов приложения

Разработанная архитектура использовалась как для создания UDP, так и для создания TCP приложения. Причём, при реализации TCP и UDP приложений, компонент под названием *Model*, который отвечает за бизнес логику оставался неизменным. Также для обоих приложений общими стали файл, отвечающий за конфигурацию серверной части приложения и файл, содержащий небольшие функции, типа разделения строки по символу.

3.2.1 Бизнес логика

Основная задача этого компонента – это предоставление функций:

- хранения и выдача по запросу уже существующих чисел
- сохранение новых чисел
- проверка дублирования чисел
- выдача новых диапазонов для расчета простых чисел
- в случае, если диапазон был запрошен, но данные по нему не пришли в течении некоторого времени, то диапазон снова помещается в пул выдаваемых

Для этого компонента был разработан следующий интерфейс представленный на рисунке 3.2.

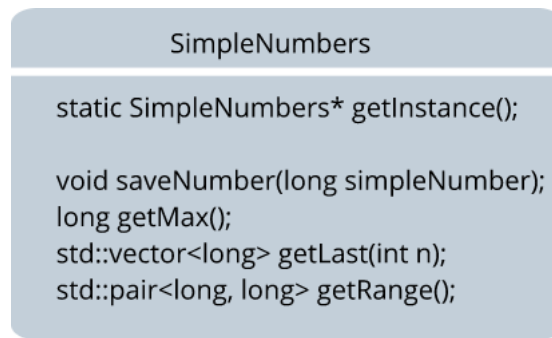


Рис. 3.2: Интерфейс компонента, отвечающего за бизнес логику

Был применен шаблон проектирования Singleton. Это было продиктовано тем, что подобный компонент в системе должен быть только один.

3.2.2 TCP сервер и клиент

Архитектура серверной части TCP приложения представлена на рисунке 3.3.

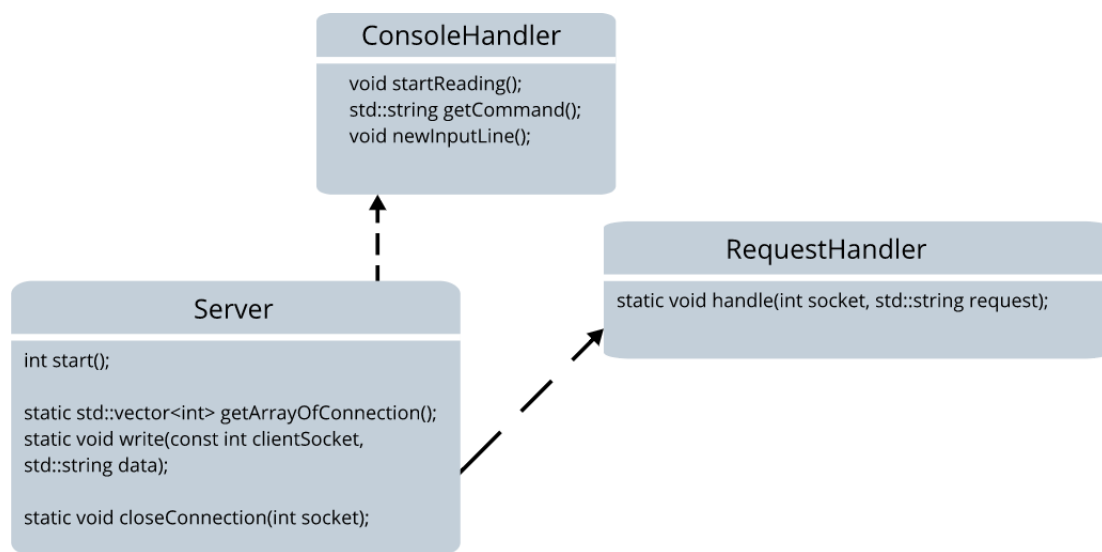


Рис. 3.3: UML-диаграмма классов TCP сервера

ConsoleHandler – класс, отвечающий за считывание и обработку консольных команд. Запускается в отдельном процессе.

RequestHandler – класс, отвечающий за обработку запросов и генерацию ответов.

Server – основной класс, задача которого принять и обработать входящее соединение и вызвать соответствующий обработчик. Также запускает **ConsoleHandler** в отдельном процессе.

Интерфейс клиентской части ТСП приложения представлена на рисунке 3.4.

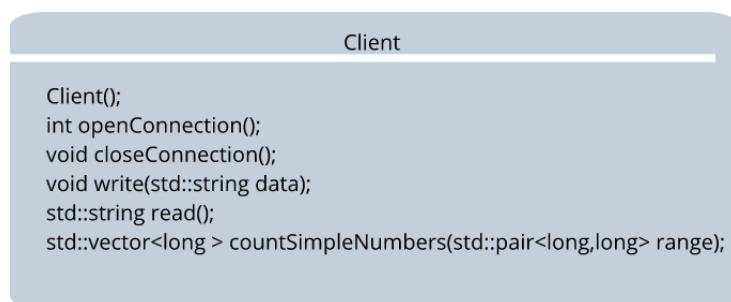


Рис. 3.4: Интерфейс ТСП клиента

Поверх этого класса, обеспечивающего работу с ТСП сервером было разработано консольное приложение, которое принимало и выполняло команды пользователя.

Описание интерфейса:

- *next* – получить диапазон от сервера, вычислить все простые числа на нем и отправить их на сервер
- *max* – запросить максимальное вычисленное простое число
- *last[number]* – запросить *number* последних простых чисел
- *exit* – закончить обмен и закрыть соединение

3.2.3 UDP сервер и клиент

Общая идея работы серверного и клиентского UDP приложения похожа на работу ТСП приложения. Основные отличия заключаются в том, что при использовании UDP структура пакета упрощается и вопросы надежной передачи данных решаются разработчиком на прикладном уровне.

В описаном задании требуется обеспечить надежность работы приложения путем разрешения трех проблемных моментов:

- Потеря пакетов
- Дублирование пакетов
- Перемешивание пакетов

В данной реализации эти вопросы решаются следующим способом. Проблема потери пакетов решается использованием подтверждающих пакетов, иначе называемых как Acknowledgement или АСК. Проблема дублирования

исчезает в связи с тем, что если на клиентскую часть приходит несколько ответов, то ничего плохого в этом нет, а если на серверную, то она способна грамотно обработать, если это действительно критично. Например, если приходит несколько запросов POST с одинаковым числом, то сервер запишет лишь одно, а если несколько запросов на рассчитываемый интервал, то он их выдаст, но когда через некоторое время поймет, что ему никто не отвечает на них, отзовет интервалы обратно. Случай с перемешиванием пакетов тоже не является критичным в связи с тем, что посылаемые запросы довольно просты и не требуют определенной последовательности. Они могут одинаково успешно обрабатываться в любой последовательности.

Также небольшие изменения в проекте вызвал и тот факт, что пришлось изменять сигнатуры функций, потому что для идентификация UDP клиента со стороны сервера используются структуры данных, вместо переменной типа `integer`, как это было в TCP.

3.3 Результаты тестирования

Приложения были протестированы и продемонстрированы преподавателю. Тестирование проводилось в следующих условиях:

- 10 клиентов одновременно отправляли всевозможные запросы на сервер, причем периодически старые завершали работу и запускались новые
- некоторые соединения намеренно обрывались, с целью проверить корректность обработки данных ситуаций
- при завершении работы сервера проверялось, что все клиентские приложения получили уведомления о корректном окончании передачи данных
- отдельно для UDP были промоделированы три проблемные ситуации (потеря, дублирование, перемешивание пакетов)

Разработанные приложения как TCP, так и UDP успешно справились с тестированием. Во всех перечисленных случаях система уверенно выполняла предписанные требования и справлялась со своей основной задачей – распределенным вычислением простых чисел.

3.3.1 Приложение на основе TCP

Для тестирования приложения был запущен сервер и три клиентских приложения. Клиентские приложения одновременно посылают различные запросы на сервер. Сервер успешно обеспечивал работу всех подключенных клиентов.

Клиент 1:

```
Client connect to 127.0.0.1 7501
>_max
serverAnswer : 200| 7
>_next
serverAnswer : 200| 10,20
>_next
serverAnswer : 200| 20,30
>_max
serverAnswer : 200| 37
>_next
serverAnswer : 200| 30,40
>_max
serverAnswer : 200| 37
>_max
serverAnswer : 200| 37
>_next
serverAnswer : 200| 40,50
>_next
serverAnswer : 200| 50,60
>_max
serverAnswer : 200| 79
>_last 7
serverAnswer : 200| 53,59,61,67,
71,73,79,
```

Клиент 2:

```
Client connect to 127.0.0.1 7501
>_max
serverAnswer : 200| 37
>_max
serverAnswer : 200| 59
>_next
serverAnswer : 200| 60,70
>_max
serverAnswer : 200| 67
>_max
serverAnswer : 200| 79
```

Клиент 3:

```
Client connect to 127.0.0.1 7501
>_max
serverAnswer : 200| 37
>_max
serverAnswer : 200| 67
>_next
serverAnswer : 200| 70,80
>_max
serverAnswer : 200| 79
```

Сервер:

```
Server binding to port : 7501
server_> i
Number of connection : 3
1. client socket: 4
2. client socket: 5
3. client socket: 6
server_>
```

В данном тесте демонстрируется то, как три клиента взаимодействуя с сервером, вычисляют ряд простых чисел. Поочередно клиенты запрашивают диапазон для вычисления, считают числа и отправляют на сервер. После каждой отправки данных проверялось, что данные изменились. Проверка проводилась с помощью команды *max*, на которую сервер возвращает максимальное рассчитанное просто число. Результат соответствует ожиданиям.

3.3.2 Приложение на основе UDP

Тестирование приложения на основе UDP проводилось аналогичным способом. Также была рассмотрена критическая ситуация, когда сервер становится недоступным, а клиент продолжает отправку данных.

Клиент 1:

```
ack received
>_ max
ack received
serverAnswer : 200| 37
>_ last 4
ack received
serverAnswer : 200| 23,29,31,37,
>_ max
ack received
serverAnswer : 200| 47
>_ next
ack received
serverAnswer : 200| 50,60
>_ next
ack received
serverAnswer : 200| 60,70
>_ next
ack received
serverAnswer : 200| 70,80
>_ max
ack received
serverAnswer : 200| 79
>_ max
timed out waiting for ack
```

Клиент 2:

```
ack received
>_ max
ack received
serverAnswer : 200| 37
>_ last 4
ack received

serverAnswer : 200| 23,29,31,37,
>_ next
ack received
```

```
serverAnswer : 200| 40,50
>_ max
ack received
serverAnswer : 200| 47
>_ max
ack received
serverAnswer : 200| 79
>_ max
timed out waiting for ack
```

Клиент 3:

```
ack received
>_ max
ack received
serverAnswer : 200| 37
>_ last 4
ack received
serverAnswer : 200| 23,29,31,37,
>_ max
ack received
serverAnswer : 200| 47
>_ max
ack received
serverAnswer : 200| 79
>_ last 8
ack received
serverAnswer : 200| 47,53,59,61,67,71,73,79,
>_ next
ack received
serverAnswer : 200| 80,90
>_ last 4
ack received
serverAnswer : 200| 73,79,83,89,
>_ max
ack received
serverAnswer : 200| 89
>_ max
timed out waiting for ack
```

Все запросы на сервере логировались так, что было понятно с какого адреса и какое действие запрошено.

Как видно из вывода, вычисления простых чисел проходило успешно. Также когда сервер был выключен, то все клиенты поняли это и прекратили работу тоже.

4 Вывод

В результате выполнения работы было разработано два приложения отличающиеся протоколом передачи данных, который они использовали. Оба приложения успешно выполняли свою основную задачу – распределенное вычисление простых чисел. Было улучшено понимание принципов работы протоколов TCP и UDP.

К особенностям TCP можно отнести то, что соединение между клиентом и сервером является надёжным. Гарантируется доставка и правильная очередность пакетов. Что облегчает разработку приложения, к минусам же можно отнести меньшую скорость работы по сравнению с UDP.

К особенностям UDP можно отнести предельную простоту передаваемого пакета, что обеспечивает высокую скорость передачи данных. Но в тоже время появляются проблемы надежности передачи. Пакет может не дойти до получателя, может быть отправлен несколько раз, также пакеты могут прийти в случайной последовательности. Все это осложняет разработку UDP приложения. Также к особенностям можно отнести то, что, в отличие от TCP, при использовании UDP четкого разделения между серверной частью приложения и клиентской нет. Это можно понять лишь анализируя трафик.

Исходя из полученного опыта, можно сделать вывод, что в случае, когда есть возможность использовать TCP, надо это делать. Но бывает, что очень критична скорость передачи и не так критична надежность, к примеру при передаче видео, или устройству не хватает вычислительности мощности для своевременной обработки структуры пакета TCP, в этом случае стоит использовать UDP.

Исходный код разработанного приложения размещен в git-репозитории https://github.com/mikle9997/networks_and_telecommunications

5 Приложение 1. Листинги исходного кода.

```
1 #ifndef SERVER_CONFIG_H
2 #define SERVER_CONFIG_H
3
4 class Config {
5
6 public:
7     static const int NUMBER_OF_READ_SYMBOLS = 26;
8     static const int NUMBER_OF_CLIENTS = 10;
9     static const int PORT = 7501;
10    static const int HOP = 10;
11    static const char DELIMITER = ',';
```

```

12     static const char *FILE_PATH;
13     static const char *INET_ADDR;
14 };
15
16 #endif //SERVER_CONFIG_H

```

```

1 #include "Config.h"
2
3 const char *Config::FILE_PATH = "../server/";
4 const char *Config::INET_ADDR = "127.0.0.1";

```

```

1 #ifndef TCPIP_UTILITY_H
2 #define TCPIP_UTILITY_H
3
4
5 #include <string>
6 #include <vector>
7 #include <sstream>
8 #include <sys/socket.h>
9 #include "Config.h"
10
11
12 class Utility {
13
14 public:
15     static void split(std::string str, std::vector<std::string>& cont, char
        divider);
16     static int read_n(int s1, char *result);
17     static std::string read_delimiter(int s1);
18 };
19
20
21 #endif //TCPIP_UTILITY_H

```

```

1
2 #include <iostream>
3 #include "Utility.h"
4
5 void Utility::split(const std::string str, std::vector<std::string> &cont, const
    char divider) {
6     std::stringstream ss(str);
7     std::string token;
8     while (std::getline(ss, token, divider)) {
9         cont.push_back(token);
10    }
11 }
12
13 int Utility::read_n(int s1, char *result) {
14     char buf[1];
15     int number_of_entered = 0;
16     ssize_t rc = -1;
17
18     while (number_of_entered != Config::NUMBER_OF_READ_SYMBOLS) {
19         rc = recv(s1, buf, 1, 0);
20         if (rc == -1) {
21             break;
22         }
23         result[number_of_entered] = buf[0];
24         number_of_entered += rc;
25     }

```

```

26     return (int) rc;
27 }
28
29
30 std::string Utility::read_delimiter(int s1) {
31     char buf[1];
32     int number_of_entered = 0;
33     std::string outData;
34     ssize_t rc = -1;
35
36     while (true) {
37         rc = recv(s1, buf, 1, 0);
38         if (buf[0] == 0) {
39             continue;
40         }
41         if (rc == -1 || buf[0] == Config::DELIMITER) {
42             break;
43         }
44         outData.push_back(buf[0]);
45         number_of_entered += rc;
46     }
47     return outData;
48 }

```

```

1  #include "ConsoleHandler.h"
2  #include "Server.h"
3
4  void ConsoleHandler::startReading() {
5      while (true) {
6          newInputLine();
7          std::cin >> command;
8
9          auto connections = Server::getArrayOfConnection();
10
11         if (command == "info" or command == "i") {
12             std::cout << "Number of connection : " << connections.size() << std
13             ::endl;
14             for (int i = 0; i < connections.size(); i++) {
15                 std::cout << i + 1 << ". client socket: " << connections[i] <<
16                 std::endl;
17             }
18         }
19         else if (command == "exit" or command == "e") {
20             std::cout << "Shutdown server." << std::endl;
21             break;
22         }
23         else if (command == "help" or command == "h") {
24             std::cout << "i - info" << std::endl;
25             std::cout << "h - help" << std::endl;
26             std::cout << "c - close" << std::endl;
27             std::cout << "clear" << std::endl;
28             std::cout << "e - exit" << std::endl;
29         }
30         else if (command == "clear") {
31             for (int i = 0; i < 10; ++i) {
32                 std::cout << std::endl;
33             }
34         }
35         else if (command == "close" or command == "c") {
36             if (connections.empty()) {

```

```

35         std::cout << "No open connections" << std::endl;
36     } else {
37         std::cout << "Close connection." << std::endl;
38         for (int i = 0; i < connections.size(); i++) {
39             std::cout << "          " << i + 1 << ". client socket: " <<
connections[i] << std::endl;
40         }
41         std::cout << "Enter number of close connection : ";
42
43         int client_number = -1;
44         std::cin >> client_number;
45
46         Server::closeConnection(connections[client_number - 1]);
47
48         std::cout << "Socket " << connections[client_number - 1] << "
closed" << std::endl;
49     }
50     } else {
51         std::cout << command << ": command not found" << std::endl;
52     }
53 }
54 }
55
56 void ConsoleHandler::newInputLine() {
57     std::cout << "server_> ";
58 }
59
60 std::string ConsoleHandler::getCommand() {
61     return command;
62 }

```

```

1  #ifndef SERVER_CONSOLEHANDLER_H
2  #define SERVER_CONSOLEHANDLER_H
3
4  #include <iostream>
5
6  class ConsoleHandler {
7
8  public:
9      void startReading();
10     std::string getCommand();
11     void newInputLine();
12
13 private:
14     std::string command;
15 };
16
17
18 #endif //SERVER_CONSOLEHANDLER_H

```

```

1  #include "../Config.h"
2  #include "Server.h"
3
4  #include "model/SimpleNumbers.h"
5
6  int main()
7  {
8      auto server = new Server();
9      server->start();
10

```

```

11     return 0;
12 }

```

```

1  #ifndef TCPIP_REQUESTHANDLER_H
2  #define TCPIP_REQUESTHANDLER_H
3
4
5  #include <iostream>
6  #include <string>
7  #include "../Utility.h"
8  #include "model/SimpleNumbers.h"
9
10 class RequestHandler {
11
12 public:
13     static void handle(int socket, std::string request);
14
15 private:
16     static SimpleNumbers* model;
17 };
18
19
20 #endif //TCPIP_REQUESTHANDLER_H

```

```

1
2  #include "RequestHandler.h"
3  #include "Server.h"
4
5  SimpleNumbers* RequestHandler::model = SimpleNumbers::getInstance();
6
7  void RequestHandler::handle(int socket, std::string request) {
8      std::vector<std::string> partOfRequest;
9      Utility::split(request, partOfRequest, '?');
10
11      if (partOfRequest.size() < 2) {
12          Server::write(socket, "400|");
13          return;
14      }
15
16      std::string command = partOfRequest[0];
17      if (command == "MAX") {
18          Server::write(socket, "200| " + std::to_string(model->getMax()));
19
20      } else if (command == "LAST") {
21          std::stringstream ss;
22          auto last = model->getLast(std::stoi(partOfRequest[1]));
23          ss << "200| ";
24          for (long num : last) {
25              ss << num << ",";
26          }
27          Server::write(socket, ss.str());
28      } else if (command == "FROM") {
29          auto range = model->getRange();
30          Server::write(socket, "200| " + std::to_string(range.first) + "," + std
::to_string(range.second));
31      } else if (command == "POST") {
32          model->saveNumber(std::stoi(partOfRequest[1]));
33          Server::write(socket, "200| ");
34      } else {
35          Server::write(socket, "400| ");

```

```

36         return ;
37     }
38 }

1
2 #include <unistd.h>
3 #include "Server.h"
4
5
6 int Server::acceptSocket ;
7 std::mutex Server::mtx;
8 std::vector<int> Server::arrayOfConnection ;
9 ConsoleHandler Server::consoleH ;
10
11
12 Server::Server() {
13     struct sockaddr_in local {
14         AF_INET,
15         htons(Config::PORT),
16         htonl(INADDR_ANY)
17     };
18     acceptSocket = socket(AF_INET, SOCK_STREAM, 0);
19     bind(acceptSocket, (struct sockaddr*)&local, sizeof(local));
20     listen(acceptSocket, 5);
21
22     std::cout << " Server binding to port : " << Config::PORT << std::endl;
23 }
24
25 int Server::start() {
26     std::thread acceptThr(acceptThread);
27     consoleH.startReading();
28
29     shutdown(acceptSocket, SHUT_RDWR);
30     close(acceptSocket);
31     acceptThr.join();
32
33     return 0;
34 }
35
36 void Server::acceptThread() {
37     int client_socket;
38     while (true) {
39         if (consoleH.getCommand() == "exit" or consoleH.getCommand() == "e") {
40             break;
41         }
42         if (arrayOfConnection.size() < Config::NUMBER_OF_CLIENTS) {
43             client_socket = accept(acceptSocket, nullptr, nullptr);
44             if (client_socket < 0) {
45                 break;
46             }
47             new std::thread(threadFunc, &client_socket);
48
49             mtx.lock();
50             arrayOfConnection.push_back(client_socket);
51             mtx.unlock();
52         }
53     }
54     for (int connection : arrayOfConnection) {
55         shutdown(connection, SHUT_RDWR);
56         close(connection);

```



```

57     }
58 }
59
60 void Server::threadFunc(int* data) {
61     int s1 = *data;
62     char result[Config::NUMBER_OF_READ_SYMBOLS];
63     int rc = -1;
64
65     while (true) {
66         rc = Utility::read_n(s1, result);
67         if (rc == -1) {
68             break;
69         }
70         auto res = std::string(result);
71         if (res == "EXIT?") {
72             closeConnection(s1);
73             break;
74         }
75         RequestHandler::handle(s1, res);
76     }
77     mtx.lock();
78     arrayOfConnection.erase(std::remove(arrayOfConnection.begin(),
79     arrayOfConnection.end(), s1), arrayOfConnection.end());
80     mtx.unlock();
81 }
82
83 std::vector<int> Server::getArrayOfConnection() {
84     return arrayOfConnection;
85 }
86
87 void Server::write(const int clientSocket, std::string data) {
88     data.push_back(Config::DELIMITER);
89     send(clientSocket, data.c_str(), data.size() + 1, 0);
90 }
91
92 void Server::closeConnection(int socket) {
93     shutdown(socket, SHUT_RDWR);
94     close(socket);
95 }

```

```

1 #ifndef SERVER_SERVER_H
2 #define SERVER_SERVER_H
3
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <thread>
8 #include <mutex>
9 #include <vector>
10 // #include <zconf.h>
11 #include <algorithm>
12 #include <iostream>
13 #include "../Config.h"
14 #include "ConsoleHandler.h"
15 #include "RequestHandler.h"
16
17 class Server {
18
19 public:
20     explicit Server();

```

```

21     ~Server() = default;
22
23     int start();
24
25     static std::vector<int> getArrayOfConnection();
26     static void write(const int clientSocket, std::string data);
27
28     static void closeConnection(int socket);
29
30 private:
31     static int acceptSocket;
32
33     static ConsoleHandler consoleH;
34     static std::mutex mtx;
35     static std::vector<int> arrayOfConnection;
36
37     static void acceptThread();
38     static void threadFunc(int* data);
39 };
40
41 #endif //SERVER_SERVER_H

```

```

1  #include "FileStorage.h"
2
3
4  FileStorage::FileStorage(const std::string fileName) {
5      this->fileName = new std::string(fileName);
6  }
7
8  FileStorage::~FileStorage() {
9      delete this->fileName;
10 }
11
12 void FileStorage::write(const std::string inputData) const {
13     std::ofstream out(*fileName);
14     if (!out.is_open()) {
15         std::cout << "Cannot open file." << std::endl;
16         return;
17     }
18     out << inputData;
19     out.close ();
20 }
21
22 std::string FileStorage::read() const {
23     std::ifstream in(*fileName);
24     std::string fileData;
25     if (!in.is_open()) {
26         std::cout << "Cannot open file." << std::endl;
27         return fileData;
28     }
29     in >> fileData;
30     in.close ();
31
32     return fileData;
33 }

```

```

1  #ifndef SERVER_FILESTORAGE_H
2  #define SERVER_FILESTORAGE_H
3
4  #include <iostream>

```

```

5 #include <fstream>
6
7 class FileStorage {
8
9 public:
10     explicit FileStorage(std::string fileName);
11     ~FileStorage();
12
13     void write(std::string inputData) const;
14     std::string read() const;
15
16 private:
17     const std::string* fileName;
18 };
19
20
21 #endif //SERVER_FILESTORAGE_H

```

```

1 #ifndef TCPIP_SIMPLENUMBERS_H
2 #define TCPIP_SIMPLENUMBERS_H
3
4 #include <vector>
5 #include <sstream>
6 #include <queue>
7 #include <algorithm>
8 #include <mutex>
9 #include "FileStorage.h"
10 #include "../.. /Config.h"
11 #include "../.. /Utility.h"
12
13 class SimpleNumbers {
14
15 public:
16     ~SimpleNumbers();
17
18     static SimpleNumbers* getInstance();
19
20     void saveNumber(long simpleNumber);
21     long getMax();
22     std::vector<long> getLast(int n);
23     std::pair<long, long> getRange();
24
25 private:
26     SimpleNumbers();
27
28     static std::mutex mtx;
29     static SimpleNumbers* instance;
30     const FileStorage* numberStorage;
31     const FileStorage* hopStorage;
32 };
33
34
35 #endif //TCPIP_MODEL_H

```

```

1 #include "SimpleNumbers.h"
2
3 SimpleNumbers* SimpleNumbers::instance;
4 std::mutex SimpleNumbers::mtx;
5
6 SimpleNumbers::SimpleNumbers() {

```

```

7     numberStorage = new FileStorage(std::string(Config::FILE_PATH) + "data");
8     hopStorage = new FileStorage(std::string(Config::FILE_PATH) + "hop");
9 }
10
11 SimpleNumbers::~SimpleNumbers() {
12     delete numberStorage;
13     delete hopStorage;
14     delete instance;
15 }
16
17 SimpleNumbers* SimpleNumbers::getInstance() {
18     if (instance == 0) {
19         instance = new SimpleNumbers();
20     }
21     return instance;
22 }
23
24 void SimpleNumbers::saveNumber(const long simpleNumber) {
25     auto newRecord = std::to_string(simpleNumber) + "|";
26
27     mtx.lock();
28     auto fileData = numberStorage->read();
29     mtx.unlock();
30
31     std::vector<std::string> formattedData;
32     Utility::split(fileData, formattedData, '|');
33
34     std::vector<long> outputData;
35
36     for (std::string strNum : formattedData) {
37         outputData.push_back(std::stol(strNum));
38     }
39
40     outputData.push_back(simpleNumber);
41
42     std::sort(outputData.begin(), outputData.end());
43
44     std::stringstream ss;
45     for (long longNum : outputData) {
46         ss << longNum << "|";
47     }
48
49     mtx.lock();
50     numberStorage->write(ss.str());
51     mtx.unlock();
52 }
53
54 long SimpleNumbers::getMax() {
55     auto simpNums = getLast(1);
56     if (!simpNums.empty()) {
57         return simpNums[0];
58     } else {
59         return 0;
60     }
61 }
62
63 std::vector<long> SimpleNumbers::getLast(int n) {
64     mtx.lock();
65     auto fileData = numberStorage->read();
66     mtx.unlock();

```

```

67
68     std::vector<std::string> formattedData;
69     Utility::split(fileData, formattedData, '|');
70
71     std::vector<long> outputData;
72
73     int size = (int) formattedData.size();
74     if (n < 0) {
75         n = 0;
76     } else if (n > size) {
77         n = size;
78     }
79
80     for (int i = size - n; i < size; i++) {
81         outputData.push_back(std::stol(formattedData[i]));
82     }
83     return outputData;
84 }
85
86 std::pair<long, long> SimpleNumbers::getRange() {
87     mtx.lock();
88     auto nextHop = std::stoi(hopStorage->read());
89     std::pair<long, long> range(Config::HOP * nextHop, Config::HOP * (nextHop +
90 1));
91     nextHop++;
92     hopStorage->write(std::to_string(nextHop));
93     mtx.unlock();
94
95     return range;
96 }

```

```

1  #ifndef SERVER_CONFIG_H
2  #define SERVER_CONFIG_H
3
4  class Config {
5
6  public:
7      static const int NUMBER_OF_READ_SYMBOLS = 50;
8      static const int ACKNOWLEDGE_LEN = 5;
9      static const int PORT = 7500;
10     static const int HOP = 10;
11     static const char DELIMITER = ',';
12     static const char *FILE_PATH;
13     static const char *INET_ADDR;
14     static const int APPROXIMATE_NUMBER_OF_CLIENTS = 5;
15 };
16
17 #endif //SERVER_CONFIG_H

```

```

1  #include "Config.h"
2
3  const char *Config::FILE_PATH = "../server/";
4  const char *Config::INET_ADDR = "127.0.0.1";

```

```

1  #ifndef TCPIP_UTILITY_H
2  #define TCPIP_UTILITY_H
3
4
5  #include <string>
6  #include <vector>

```

```

7  #include <sstream>
8  #include <sys/socket.h>
9  #include <netinet/in.h>
10 #include "Config.h"
11 #include <sys/ioctl.h>
12
13
14 class Utility {
15
16 public:
17     static void split(const std::string &str, std::vector<std::string> &cont,
18         char divider);
19 };
20
21 #endif //TCPIP_UTILITY_H

```

```

1
2 #include <iostream>
3 #include "Utility.h"
4
5 void Utility::split(const std::string &str, std::vector<std::string> &cont,
6     const char divider) {
7     std::stringstream ss(str);
8     std::string token;
9     while (std::getline(ss, token, divider)) {
10         cont.push_back(token);
11     }
12 }

```

```

1
2 #include "Server.h"
3
4 int main()
5 {
6     auto server = new Server();
7     server->start();
8
9     return 0;
10 }

```

```

1 #ifndef TCPIP_REQUESTHANDLER_H
2 #define TCPIP_REQUESTHANDLER_H
3
4
5 #include <iostream>
6 #include <string>
7 #include "../Utility.h"
8 #include "model/SimpleNumbers.h"
9 #include <sys/time.h>
10 #include <sys/types.h>
11 #include <unistd.h>
12
13 class RequestHandler {
14
15 public:
16     static void handle(struct sockaddr_in si_other, std::string request);
17
18 private:
19     static SimpleNumbers* model;

```

```

20 };
21
22
23 #endif //TCPIP_REQUESTHANDLER_H

```

```

1
2 #include "RequestHandler.h"
3 #include "Server.h"
4
5 SimpleNumbers* RequestHandler::model = SimpleNumbers::getInstance();
6
7 void RequestHandler::handle(struct sockaddr_in si_other, std::string request) {
8     std::vector<std::string> partOfRequest;
9     Utility::split(request, partOfRequest, '?');
10
11     if (partOfRequest.size() < 2) {
12         Server::write(si_other, "400|");
13     } else {
14         std::string command = partOfRequest[0];
15         if (command == "TEST") {
16             Server::write(si_other, "TEST_C");
17
18         } else if (command == "MAX") {
19             Server::write(si_other, "200| " + std::to_string(model->getMax()));
20
21         } else if (command == "LAST") {
22             std::stringstream ss;
23             auto last = model->getLast(std::stoi(partOfRequest[1]));
24             ss << "200| ";
25             for (long num : last) {
26                 ss << num << ",";
27             }
28             Server::write(si_other, ss.str());
29         } else if (command == "FROM") {
30             auto range = model->getRange();
31             Server::write(si_other, "200| " + std::to_string(range.first) + ","
32 + std::to_string(range.second));
33         } else if (command == "POST") {
34             model->saveNumber(std::stoi(partOfRequest[1]));
35             Server::write(si_other, "200| ");
36         } else {
37             Server::write(si_other, "400| ");
38         }
39 }

```

```

1
2 #include "Server.h"
3
4
5 int Server::serverSocket;
6 std::string Server::command;
7
8
9 Server::Server() {
10     struct sockaddr_in si_me;
11
12     memset((char *) &si_me, 0, sizeof(si_me));
13
14     si_me.sin_family = AF_INET;

```

```

15     si_me.sin_port = htons(Config::PORT);
16     si_me.sin_addr.s_addr = htonl(INADDR_ANY);
17
18     serverSocket = socket(AF_INET, SOCK_DGRAM, 0);
19     if (serverSocket < 0) {
20         error_message("socket");
21     }
22
23     int b = bind(serverSocket, (struct sockaddr *) &si_me, sizeof(si_me));
24     if (b < 0) {
25         error_message("bind");
26     }
27
28     std::cout << " UDP server binding to port : " << Config::PORT << std::endl;
29 }
30
31 int Server::start() {
32     std::thread acceptThr(mainThread);
33
34     while (true) {
35         std::string in;
36         std::cin >> in;
37
38         if (in == "exit" or in == "e") {
39             command = in;
40             break;
41         }
42         else {
43             std::cout << "command not found" << std::endl;
44         }
45     }
46     closeConnection(serverSocket);
47     acceptThr.join();
48
49     return 0;
50 }
51
52 void Server::mainThread() {
53     struct sockaddr_in si_other;
54
55     char buffer[Config::NUMBER_OF_READ_SYMBOLS];
56     socklen_t slen = sizeof(si_other);
57     ssize_t recv_len;
58
59     while (true) {
60         if (command == "exit" or command == "e") {
61             break;
62         }
63         std::cout << "Waiting for data...";
64         fflush(stdout);
65
66         recv_len = recvfrom(serverSocket, buffer, Config::NUMBER_OF_READ_SYMBOLS
, MSG_CONFIRM, (struct sockaddr *) &si_other, &slen);
67         if (recv_len == -1) {
68             error_message("recvfrom()");
69         }
70
71         new std::thread(threadFunc, si_other, buffer);
72     }
73 }

```



```

74
75 void Server::threadFunc(struct sockaddr_in si_other, std::string data) {
76     std::cout << "Received packet from " << inet_ntoa(si_other.sin_addr) << ":" <<
    << ntohs(si_other.sin_port) << std::endl;
77     std::cout << "Data: " << data << std::endl;
78
79     RequestHandler::handle(si_other, data);
80 }
81
82 void Server::write(const sockaddr_in si_other, std::string data) {
83     data.push_back(Config::DELIMITER);
84
85     socklen_t slen = sizeof(si_other);
86     if (sendto(serverSocket, data.c_str(), data.size(), MSG_CONFIRM, (struct
    sockaddr*) &si_other, slen) == -1) {
87         error_message("sendto()");
88     }
89 }
90
91 void Server::closeConnection(int socket) {
92     shutdown(socket, SHUT_RDWR);
93     close(socket);
94 }
95
96 void Server::error_message(char* s) {
97     perror(s);
98     exit(EXIT_FAILURE);
99 }
100
101 int Server::getServerSocket() {
102     return serverSocket;
103 }

```

```

1 #ifndef SERVER_SERVER_H
2 #define SERVER_SERVER_H
3
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <arpa/inet.h>
7 #include <netinet/in.h>
8 #include <thread>
9 #include <mutex>
10 #include <vector>
11 #include <unistd.h>
12 #include <algorithm>
13 #include <iostream>
14 #include "../Config.h"
15 #include "RequestHandler.h"
16 #include <stdlib.h>
17 #include <string.h>
18
19 class Server {
20
21 public:
22     explicit Server();
23     ~Server() = default;
24
25     int start();
26
27     static void write(struct sockaddr_in si_other, std::string data);

```

```

28     static void closeConnection(int socket);
29     static int getServerSocket();
30
31 private:
32     static int serverSocket;
33     static std::string command;
34
35     static void mainThread();
36     static void threadFunc(struct sockaddr_in si_other, std::string data);
37
38     static void error_message(char* s);
39 };
40
41 #endif //SERVER_SERVER_H

```

```

1 #include "FileStorage.h"
2
3
4 FileStorage::FileStorage(const std::string fileName) {
5     this->fileName = new std::string(fileName);
6 }
7
8 FileStorage::~FileStorage() {
9     delete this->fileName;
10 }
11
12 void FileStorage::write(const std::string inputData) const {
13     std::ofstream out(*fileName);
14     if (!out.is_open()) {
15         std::cout << "Cannot open file." << std::endl;
16         return;
17     }
18     out << inputData;
19     out.close ();
20 }
21
22 std::string FileStorage::read() const {
23     std::ifstream in(*fileName);
24     std::string fileData;
25     if (!in.is_open()) {
26         std::cout << "Cannot open file." << std::endl;
27         return fileData;
28     }
29     in >> fileData;
30     in.close ();
31
32     return fileData;
33 }

```

```

1 #ifndef SERVER_FILESTORAGE_H
2 #define SERVER_FILESTORAGE_H
3
4 #include <iostream>
5 #include <fstream>
6
7 class FileStorage {
8
9 public:
10     explicit FileStorage(std::string fileName);
11     ~FileStorage();

```

```

12
13     void write(std::string inputData) const;
14     std::string read() const;
15
16 private:
17     const std::string* fileName;
18 };
19
20
21 #endif //SERVER_FILESTORAGE_H

```

```

1 #ifndef TCPIP_SIMPLENUMBERS_H
2 #define TCPIP_SIMPLENUMBERS_H
3
4 #include <vector>
5 #include <sstream>
6 #include <mutex>
7 #include "FileStorage.h"
8 #include "../.. /Config.h"
9 #include "../.. /Utility.h"
10 #include <set>
11 #include <algorithm>
12
13 class SimpleNumbers {
14
15 public:
16     ~SimpleNumbers();
17
18     static SimpleNumbers* getInstance();
19
20     void saveNumber(long simpleNumber);
21     long getMax();
22     std::vector<long> getLast(int n);
23     std::pair<long, long> getRange();
24
25 private:
26     SimpleNumbers();
27
28     static int getMinMissingNumber(const std::vector<int> &arr);
29
30     static std::mutex mtx;
31     static SimpleNumbers* instance;
32     const FileStorage* numberStorage;
33     const FileStorage* hopStorage;
34 };
35
36
37 #endif //TCPIP_MODEL_H

```

```

1 #include "SimpleNumbers.h"
2
3 SimpleNumbers* SimpleNumbers::instance;
4 std::mutex SimpleNumbers::mtx;
5
6 struct classcomp {
7     bool operator() (const long& lhs, const long& rhs) const
8     {return lhs<rhs;}
9 };
10
11 SimpleNumbers::SimpleNumbers() {

```

```

12     numberStorage = new FileStorage(std::string(Config::FILE_PATH) + "data");
13     hopStorage = new FileStorage(std::string(Config::FILE_PATH) + "hop");
14 }
15
16 SimpleNumbers::~SimpleNumbers() {
17     delete numberStorage;
18     delete hopStorage;
19     delete instance;
20 }
21
22 SimpleNumbers* SimpleNumbers::getInstance() {
23     if (instance == 0) {
24         instance = new SimpleNumbers();
25     }
26     return instance;
27 }
28
29 void SimpleNumbers::saveNumber(const long simpleNumber) {
30     auto newRecord = std::to_string(simpleNumber) + "|";
31
32     mtx.lock();
33     auto fileData = numberStorage->read();
34     mtx.unlock();
35
36     std::vector<std::string> formattedData;
37     Utility::split(fileData, formattedData, '|');
38
39     std::set<long, classcomp> outputData;
40
41     for (std::string strNum : formattedData) {
42         outputData.insert(std::stol(strNum));
43     }
44
45     outputData.insert(simpleNumber);
46
47
48     std::stringstream ss;
49     for (long longNum : outputData) {
50         ss << longNum << "|";
51     }
52
53     mtx.lock();
54     numberStorage->write(ss.str()); // записали число
55
56     auto hopStorageData = hopStorage->read();
57     mtx.unlock();
58     std::vector<std::string> formattedHopStorageData;
59     Utility::split(hopStorageData, formattedHopStorageData, '|');
60
61     std::string substr1;
62     std::string substr2;
63     if (!formattedHopStorageData.empty()) {
64         substr1 = formattedHopStorageData[0];
65         if (formattedHopStorageData.size() > 1) {
66             substr2 = formattedHopStorageData[1];
67         }
68     }
69
70     std::vector<std::string> wasCounted;
71     Utility::split(substr1, wasCounted, ','); // записали, что

```

диапазон отработан

```
72
73     std::set<long, classcomp> wasCountedData;
74
75     for (std::string strNum : wasCounted) {
76         wasCountedData.insert(std::stoi(strNum));
77     }
78     long range = simpleNumber / Config::HOP;
79     wasCountedData.insert(range);
80
81     std::stringstream ssHop;
82     for (auto longNum : wasCountedData) {
83         ssHop << longNum << ", ";
84     }
85     ssHop << "|" << substr2;
86
87     mtx.lock();
88     hopStorage->write(ssHop.str());
89     mtx.unlock();
90 }
91
92 long SimpleNumbers::getMax() {
93     auto simpNums = getLast(1);
94     if (!simpNums.empty()) {
95         return simpNums[0];
96     } else {
97         return 0;
98     }
99 }
100
101 std::vector<long> SimpleNumbers::getLast(int n) {
102     mtx.lock();
103     auto fileData = numberStorage->read();
104     mtx.unlock();
105
106     std::vector<std::string> formattedData;
107     Utility::split(fileData, formattedData, '|');
108
109     std::vector<long> outputData;
110
111     int size = (int) formattedData.size();
112     if (n < 0) {
113         n = 0;
114     } else if (n > size) {
115         n = size;
116     }
117
118     for (int i = size - n; i < size; i++) {
119         outputData.push_back(std::stoi(formattedData[i]));
120     }
121     return outputData;
122 }
123
124 std::pair<long, long> SimpleNumbers::getRange() {
125     mtx.lock();
126     auto fileData = hopStorage->read();
127     mtx.unlock();
128
129     std::vector<std::string> formattedData;
130     Utility::split(fileData, formattedData, '|');
```

```

131
132     std::string substr1;
133     std::string substr2;
134     if (!formattedData.empty()) {
135         substr1 = formattedData[0];
136         if (formattedData.size() > 1) {
137             substr2 = formattedData[1];
138         }
139     }
140
141     std::vector<std::string> wasCounted;
142     std::vector<std::string> inCounting;
143
144     Utility::split(substr1, wasCounted, ',');
145     Utility::split(substr2, inCounting, ',');
146
147     int firstVal = 0;
148     int lastVal = 0;
149     if (!inCounting.empty()) {
150         firstVal = std::stoi(inCounting[0]);
151         lastVal = std::stoi(inCounting[inCounting.size() - 1]);
152     }
153     if (lastVal - firstVal > Config::APPROXIMATE_NUMBER_OF_CLIENTS + 3) {
154         inCounting.erase(inCounting.begin());
155     }
156
157     std::set<int, classcomp> outputData;
158
159     for (std::string strNum : wasCounted) {
160         outputData.insert(std::stoi(strNum));
161     }
162     for (std::string strNum : inCounting) {
163         outputData.insert(std::stoi(strNum));
164     }
165
166     std::vector<int> dataVector(outputData.size());
167     std::copy(outputData.begin(), outputData.end(), dataVector.begin());
168
169     auto nextHop = getMinMissingNumber(dataVector);
170     std::pair<long, long> range(Config::HOP * nextHop, Config::HOP * (nextHop +
171 1));
172
173     std::stringstream ss;
174     for (auto longNum : wasCounted) {
175         ss << longNum << ",";
176         outputData.erase(std::stoi(longNum));
177     }
178     ss << "|";
179     for (auto longNum : outputData) {
180         ss << longNum << ",";
181     }
182     ss << nextHop << ",";
183
184     mtx.lock();
185     hopStorage->write(ss.str());
186     mtx.unlock();
187
188     return range;
189 }

```

```

190 int SimpleNumbers::getMinMissingNumber(const std::vector<int> &arr) {
191     for (int j = 0; j < arr.size(); ++j) {
192         if (std::find(arr.begin(), arr.end(), j) == arr.end()) {
193             return j;
194         }
195     }
196     if (!arr.empty()) {
197         return arr[arr.size() - 1] + 1;
198     } else {
199         return 0;
200     }
201 }

```