**ЛАБОРАТОРНАЯ РАБОТА №3**

# РАЗРАБОТКА ПРИЛОЖЕНИЙ АРХИТЕКТУРЫ «КЛИЕНТ-СЕРВЕР»,ВЗАИМОДЕЙСТВУЮЩИЙ ПО ПРОТОКОЛУ ТСР(ПОСЛЕДОВАТЕЛЬНЫЙ СЕРВЕР)

Цель:Научиться создавать приложения архитектуры «клиент-сервер», взаимодействующих по протоколу TCP/IP, реализующих последовательное подключение клиентов к серверу.

Transmission Control Protocol/Internet Protocol (TCP/IP) — это промышленный стандарт стека протоколов, разработанный для глобальных сетей.

Лидирующая роль стека TCP/IP объясняется следующими его свойствами:

* Это наиболее завершенный стандартный и в то же время популярный стек сетевых протоколов, имеющий многолетнюю историю.
* Почти все большие сети передают основную часть своего трафика с помощью протокола TCP/IP.
* Это метод получения доступа к сети Internet.
* Этот стек служит основой для создания intranet — корпоративной сети, использующей транспортные услуги Internet и гипертекстовую технологию WWW, разработанную в Internet.
* Все современные операционные системы поддерживают стек TCP/IP.
* Это гибкая технология для соединения разнородных систем как на уровне транспортных подсистем, так и на уровне прикладных сервисов.
* Это устойчивая масштабируемая межплатформенная среда для приложений клиент-сервер.

## **Сервера и клиенты**

В контексте работы в сети используются такие термины, как клиент и сервер. Сервер – это все, что имеет некоторый разделяемый (коллективно используемый) ресурс. Существуют вычислительные серверы, которые обеспечивают вычислительную мощность; серверы печати, которые управляют совокупностью принтеров; дисковые серверы, которые предоставляют работающее в сети дисковое пространство, и *Web*-серверы, которые хранят *Web*-приложения. Клиент – любой другой объект, который хочет получить доступ к специфическому серверу. Сервер – это постоянно доступный ресурс, в то время как клиент может «отключиться» после того, как он был обслужен.

Различие между сервером и клиентом существенно только, когда клиент пытается подключиться к серверу. Как только они соединятся, происходит процесс двухстороннего общения, и не важно, что один является сервером, а другой – клиентом.

Работа сервера – слушать соединение, которое выполняется с помощью специального создаваемого серверного объекта (сокета), содержащего *IP*-адрес и номер порта. Работа клиента – попытаться создать соединение с сервером, которое выполняется с помощью клиентского сокета. Как только соединение установлено, соединение превращается в потоковый объект ввода/вывода. С этого момента можно рассматривать соединение как файл, который можно читать и в который можно записывать данные. Единственная особенность – файл может обладать определенным интеллектом и обрабатывать передаваемые команды.

Эти функции обеспечиваются расширением программы сетевой библиотеки *java.net.\*;*.

## **Сокеты**

Передача данных по сети – сложный процесс, включающий в себя определение пути доставки данных, организацию взаимодействия, алгоритмы синхронизации, обработки сбойных ситуаций и т.п. Программное обслуживание такого процесса сложное. Для упрощения введено понятие сокета (гнезда) как конечной точки коммуникации. Сокет (гнездо, разъем) – это программная абстракция, используемая для представления «терминалов» соединений между двумя машинами.

Каждый из сокетов определяется типом и ассоциированным с ним процессом. Реально для передачи организуются определенные дескрипторы *TCP*-соединения, так называемые гнезда (*socket*): гнездо сервера и гнездо клиента, которые в Internet-домене включают в себя *IP*-адреса сервера и клиента и номера портов, через которые они взаимодействуют. Сервер обычно имеет закрепленный и постоянный во взаимодействии номер порта, а клиенту, обращающемуся по этому номеру для связи к серверу, назначается некоторый другой (эфемерный) номер порта после установления соединения с сервером на сеанс их взаимодействия. Таким образом основной порт освобождается для установления последующих связей (номер порта выбирается сервером из числа не занятых в диапазоне от 1024 до 65 535). Эта комбинация (*IP*-адрес и номера портов) однозначно определяет отдельные сетевые процессы в сети Internet (номера портов до 1024, как правило, резервируются для широко известных приложений, например 80 – для связывания с серверами *Web* по протоколу *HTTP*).

Сокеты для работы в сети можно создать двух типов:

1)  потоковые для *TCP*-соединения. *TCP* могут передавать даннные только между двумя приложениями, т. к. они предполагают наличие канала между этими приложениями;

2)  дейтаграммные. Для дейтаграмм не нужно создавать канал, данные посылаются приложению с использованием адреса, состоящего из сокета и номера порта (в дейтаграммах не гарантируются доставка и корректность последовательности передачи пакетов). Для передачи дейтаграмм не нужны ни механизмы подтверждения связи, ни механизмы управления потоком данных.

Для упрощения представления такого соединения, представим себе сокет, размещенный на некоторой машине, и виртуальный «кабель», соединяющий две машины, каждый конец которого вставлен в сокет. Для *TCP* -соединений в *Java* используется два класса сокетов: ServerSocket – класс, используемый сервером, чтобы «слушать» входящие соединения, и *Socket* – используемый клиентом для инициирования соединения.

TCP/IP-сокеты позволяют реализовать надежные двунаправленные, ориентированные на работу с потоками соединения точка-точка между удаленными узлами. В целом алгоритм работы системы клиент-сервер выглядит следующим образом:

1. Сервер подключается к порту на хосте и ждет соединения с клиентом

2. Клиент создает сокет и пытается соединить его с портом на хосте

3. При соединении клиента сервер получает свой сокет для взаимодействия с клиентом и переходит в режим ожидания команд от клиента

4. Клиент формирует команду и передает ее через сокет серверу, переходит в режим ожидания ответа

5. Сервер принимает команду через сокет, выполняет ее и пересылает ответ клиенту.

## **Сокеты TCP/IP серверов**

Как было указано выше, для создания сокетов серверов используется класс *ServerSocket*. Указанный класс используется для создания серверов, которые прослушивают либо локальные, либо удаленные программы клиента, чтобы соединяться с ними на опубликованных портах.

Конструкторы класса *ServerSocket*:

* *ServerSocket(int port)* – создает сокет сервера на указанном порте с длиной очереди по умолчанию 50.
* *ServerSocket(int port, int maxQueue)* – создает сокет сервера на указанном порте с максимальной длиной очереди *maxQueue*.
* *ServerSocket(int port, int maxQueue, InetAddress localAddress)* – создает сокет сервера на указанном порте с максимальной длиной очереди *maxQueue.*

Для взаимодействия по протоколу TCP/IP в языке Java можно использовать интерфейс *сокетов*. Термин "сокет" (socket) обозначает одновременно библиотеку сетевых интерфейсов и оконечное устройство канала связи (точку связи), через которое процесс может передавать или получать данные.

TCP/IP-сокеты позволяют реализовать надежные двунаправленные, ориентированные на работу с потоками соединения точка-точка между удаленными узлами.

В целом алгоритм работы системы клиент-сервер выглядит следующим образом:

1. Сервер подключается к порту на хосте и ждет соединения с клиентом
2. Клиент создает сокет и пытается соединить его с портом на хосте
3. При соединении клиента сервер получает свой сокет для взаимодействия с клиентом и переходит в режим ожидания команд от клиента
4. Клиент формирует команду и передает ее через сокет серверу, переходит в режим ожидания ответа
5. Сервер принимает команду через сокет, выполняет ее и пересылает ответ клиенту

Итак, на стороне сервера необходимо начать прослушивать определенный порт TCP/IP, для того чтобы удаленный клиент мог выполнять подключения. Это можно сделать при помощи класса java.net.ServerSocket.

Первоначально, необходимо создать экземпляр класса ServerSocket и указать в конструкторе порт, который будет прослушиваться. Если заданный порт занят, будет сгенерировано исключение IOException.

Пример 2.1. Создание экземпляр класса ServerSocket

**ServerSocket** server=null;

try

{

server = new **ServerSocket**(12345);

}

catch(IOException e) {}

Далее, необходимо перевести сервер в режим ожидания подключений. Делается это при помощи метода accept() класса ServerSocket. Метод accept выполняет ожидание соединения со стороны клиента и возвращает объект класса java.net.Socket в случае успешной работы. В случае ошибки, метод генерирует исключение IOException.

Пример 2.2. Ожидание соединения со стороны клиента

**Socket** sock=null;

try

{

System.out.println("Waiting for a client...");

sock = server.**accept**();

System.out.println("Client connected");

}

catch(IOException e) {}

При необходимости можно установить максимальный интервал ожидания клиентских подключений. Делается это при помощи метода setSoTimeout класса ServerSocket.

Полученный объект java.net.Socket далее будет использоваться для приема и передачи данных клиенту.

На стороне клиента необходимо осуществить подключение к серверу. Для этого используется класс java.net.Socket. На клиенте необходимо явно создать объект класса Socket и передать в конструктор параметры подключения к серверу: IP-адрес сервера и порт. В случае ошибки, конструктор сгенерирует исключение IOException.

Пример 2.3. Создание объекта класса Socket

**Socket** sock =null;try

{

System.out.println("Connecting to server..."); sock = new **Socket**("localhost",12345); System.out.println("Connected");

}

catch(IOException e) {}

Входные и выходные потоки, реализуются с помощью классов InputStream и OutputStream:

Пример 2.4. Получение входного и выходного потоков

InputStream inputstream = reg.getInputStream();

OutputStream outputstream = reg.getOutputStream();

Получив объекты, реализующие потоки, можно воспользоваться предоставляемыми ими методами, чтобы организовать взаимодействие по сети. Например, организовать чтение байта из входного потока можно при помощи метода *read*, а запись байта в выходной поток – с использованием метода *write*:

Пример 2.5. Чтение и запись байтов

int c= inputstream.read(); **//** чтение байта из входного потока

outputstream. write(с); //запись байта в выходной поток

## **Прием и передача двоичных данных**

Для приема/передачи двоичных данных необходимо сконструировать потоки DataInputStream и DataOutputStream на основе потоков ввода-вывода сокета (getInputStream() и getOutputStream() ).

Пример 2.6. Передача с клиента на сервер массив целых чисел

int N = 3;

int A[] = new int[3];

for (int i=0; i<N; i++)

A[i] = 10\*i;

// Передаю массив на сервер

try

{

DataOutputStream out =

new DataOutputStream(sock.getOutputStream());

// Передаю количество элементов

out.writeInt(N);

// Передаю элементы

for (int i=0; i<N; i++)

out.writeInt(A[i]);

} catch (IOException e) {}

Пример 2.7. Серверный код, который получит данные числа и выведет

их на экран.

try

{

DataInputStream in = new DataInputStream(sock.getInputStream());

// Получаю количество чисел

int N = in.readInt();

// Получаю числа и вывожу на экран

for (int i=0; i<N; i++)

{

int val = in.readInt();

System.out.println(val);

}

} catch (IOException e) {}

## **Сокеты TCP/IP клиентов**

Для создания сокета клиента используется конструктор:

* Socket(String hostname, int port) *–* создает сокет, соединяющий локальную хост-машину с именованной хост-машиной и портом; может выбрасывать исключение UnknownHostException или IOException.
* Socket(InetAddress ipAddress, int port)– создает сокет, аналогичный предыдущему, но используется уже существующий объект класса InetAddres и порт; может выбрасывать исключение IOException.

Сокет может в любое время просматривать связанную с ним адресную и портовую информацию при помощи методов, представленных в табл. 2. 1.

Таблица 2.1. Методы просмотра адресной и портовой информации

|  |  |
| --- | --- |
| Метод | Описание |
| InetAddres getInetAddress() | Возвращает *InetAddress*-объект, связанный с *Soket*-объектом |
| Int getPotr() | Возвращает удаленный порт, с которым соединен данный *Soket*-объект |
| Int getLocalPort() | Возвращает локальны порт, с которым соединен данный *Soket*-объект |

## Пример 2.4. Реализация взаимодействия клиента и сервера на базе протокола TCP/IP (последовательный сервер)

Данный пример демонстрирует работу двух приложений: сервера и клиента. Первым запускается сервер. К нему может быть последовательно подключено несколько клиентов. Приложение клиента имеет GUI-интерфейс .Клиент посылает сообщения на сервер, сервер их перенаправляет назад клиенту. Как только клиент закрывает окно, сервер разрывает с ним соединение. Приложение сервера в консольной реализации.

Клиент посылает серверу произвольные числа и получает назад количество чисел, кратных трем.

**Приложение сервера.**

package servertcp;

import java.net.\*;

import java.io.\*;

class ServerTCP {

static int countclients = 0;//счетчик подключившихся клиентов

public static void main(String args[]) throws IOException {

ServerSocket sock = null;

InputStream is = null;

OutputStream os = null;

try {

sock = new ServerSocket(1024); //создаем серверный сокет работающий локально по порту 1024

while (true) { //бесконечный цикл для возможности подключения последовательно нескольних клиентов

Socket client = sock.accept(); //сработает, когда клиент подключится,

// для него выделится отдельный сокет client

countclients++; //количество подключившихся клиентов увеличивается на 1

System.out.println("=======================================");

System.out.println("Client " + countclients + " connected");

is = client.getInputStream(); //получили входной поток для чтения данных

os = client.getOutputStream();//получили выходной поток для записи данных

boolean flag = true;

while (flag) {

byte[] bytes = new byte[1024];

is.read(bytes); //чтение иформации, посланной клиентом, из вхоного потока в массив bytes[]

String str = new String(bytes, "UTF-8"); // переводим тип byte в тип String

String[] numbers = str.split(" "); // разбиваем строку на подстроки пробелами

String m = ""; //переменнная,в которую будут записываться числа делящиеся на 3

bytes = new byte[1024];

for (int i = 0; i < numbers.length - 1; i++) {

System.out.println("клиент прислал число " + numbers[i]);

if (Integer.parseInt(numbers[i]) % 3 == 0) {

m += numbers[i] + " "; // записываютмя числа,которые деляется на 3

}

}

bytes = m.getBytes();

os.write(bytes); // отправляем клиенту информацию

}

}

} catch (Exception e) {

System.out.println("Error " + e.toString());

} finally {

is.close();//закрытие входного потока

os.close();//закрытие входного потока

sock.close();//закрытие сокета, выделенного для работы с подключившимся клиентом

System.out.println("Client " + countclients + " disconnected");

}

}}

**Приложение клиента.**

package clienttcp;

import java.awt.Button;

import java.awt.Frame;

import java.awt.Label;

import java.awt.TextArea;

import java.awt.TextField;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.WindowEvent;

import java.awt.event.WindowListener;

import java.io.\*;

import java.net.\*;

class ClientTCP extends Frame implements ActionListener, WindowListener {

TextField tf, tf1, tf2, tf3, tf4, tf5, tf6;

TextArea ta;

Label la, la1, la2, la3, la4;

Socket sock = null;

InputStream is = null;

OutputStream os = null;

public static void main(String args[]) {

ClientTCP c = new ClientTCP();

c.GUI();

}

private void GUI() {

// super("Клиент");

setTitle("КЛИЕНТ");

tf = new TextField("127.0.0.1");//ip adress клиента

tf1 = new TextField("1024");// port клиента

tf2 = new TextField();

tf3 = new TextField();

tf4 = new TextField();

tf5 = new TextField();

tf6 = new TextField();

ta = new TextArea();

la = new Label("IP ADRESS");

la1 = new Label("port");

la2 = new Label("sending date");

la3 = new Label("result ");

la4 = new Label(" ");

Button btn = new Button("connect ");

Button btn1 = new Button("send ");

tf.setBounds(200, 50, 70, 25);

tf1.setBounds(330, 50, 70, 25);

tf2.setBounds(150, 200, 50, 25);

tf3.setBounds(210, 200, 50, 25);

tf4.setBounds(270, 200, 50, 25);

tf5.setBounds(330, 200, 50, 25);

tf6.setBounds(390, 200, 50, 25);

ta.setBounds(150, 300, 150, 100);

btn.setBounds(50, 50, 70, 25);

btn1.setBounds(50, 200, 70, 25);

la.setBounds(130, 50, 150, 25);

la1.setBounds(280, 50, 150, 25);

la2.setBounds(150, 150, 150, 25);

la3.setBounds(160, 250, 150, 25);

la4.setBounds(600, 10, 150, 25);

add(tf);

add(tf1);

add(tf2);

add(tf3);

add(tf4);

add(tf5);

add(tf6);

add(btn);

add(btn1);

add(ta);

add(la);

add(la1);

add(la2);

add(la3);

add(la4);

setSize(600, 600);

setVisible(true);

addWindowListener(this);

btn.addActionListener(al);

btn1.addActionListener(this);

tf2.getText();

tf3.getText();

tf4.getText();

tf5.getText();

tf6.getText();

}

public void windowClosing(WindowEvent we) {

if (sock != null && !sock.isClosed()) { // если сокет не пустой и сокет открыт

try {

sock.close(); // сокет закрывается

} catch (IOException e) {

}

}

this.dispose();

}

public void windowActivated(WindowEvent we) {} ;

public void windowClosed(WindowEvent we) {};

public void windowDeactivated(WindowEvent we) {};

public void windowDeiconified(WindowEvent we) {} ;

public void windowIconified(WindowEvent we) {};

public void windowOpened(WindowEvent we) { } ;

public void actionPerformed(ActionEvent e) {

if (sock == null) {

return;

}

try {

is = sock.getInputStream(); // входной поток для чтения данных

os = sock.getOutputStream();// выходной поток для записи данных

String numbers = ""; //перменная,в которую записываются введенные числа

numbers += tf2.getText() + " ";

numbers += tf3.getText() + " ";

numbers += tf4.getText() + " ";

numbers += tf5.getText() + " ";

numbers += tf6.getText() + " ";

os.write(numbers.getBytes()); // отправляем введенные данные. Тип string переводим в byte

byte[] bytes = new byte[1024];

is.read(bytes); //получаем назад информацию,которую послал сервер

String str = new String(bytes, "UTF-8"); // переводим тип byte в String

String[] n = str.split(" "); // разбиваем строку на подстроки пробелами

for (int i = 0; i < n.length - 1; i++) {

ta.append(n[i] + "\n"); // в text area записываем полученные данные

}

} catch (Exception ex) {

ex.printStackTrace();

} finally {

try {

os.close();//закрытие выходного потока

is.close();//закрытие входного потока

sock.close();//закрытие сокета, выделенного для работы с сервером

} catch (IOException e1) {

e1.printStackTrace();

}

}

}

public void actionPerformed2(ActionEvent e) {}

ActionListener al = new ActionListener() { //событие на нажатие кнопки

@Override

public void actionPerformed(ActionEvent arg0) {

try {

sock = new Socket(InetAddress.getByName(tf.getText()), Integer.parseInt(tf1.getText()));

//создается сокет по ip адрессу и порту

} catch (NumberFormatException e) {

} catch (UnknownHostException e) {

} catch (IOException e) {

}} };}

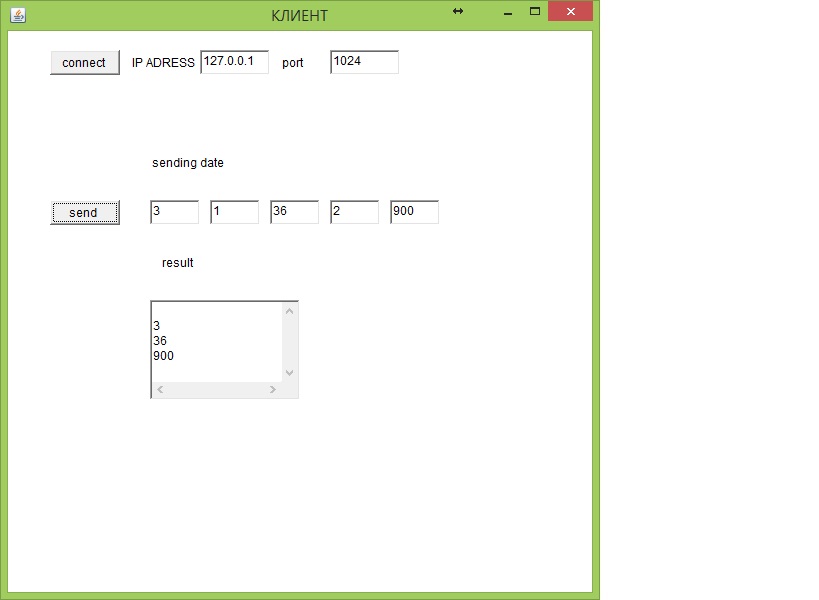


Рисунок 2.1. Интерфейс Клиента

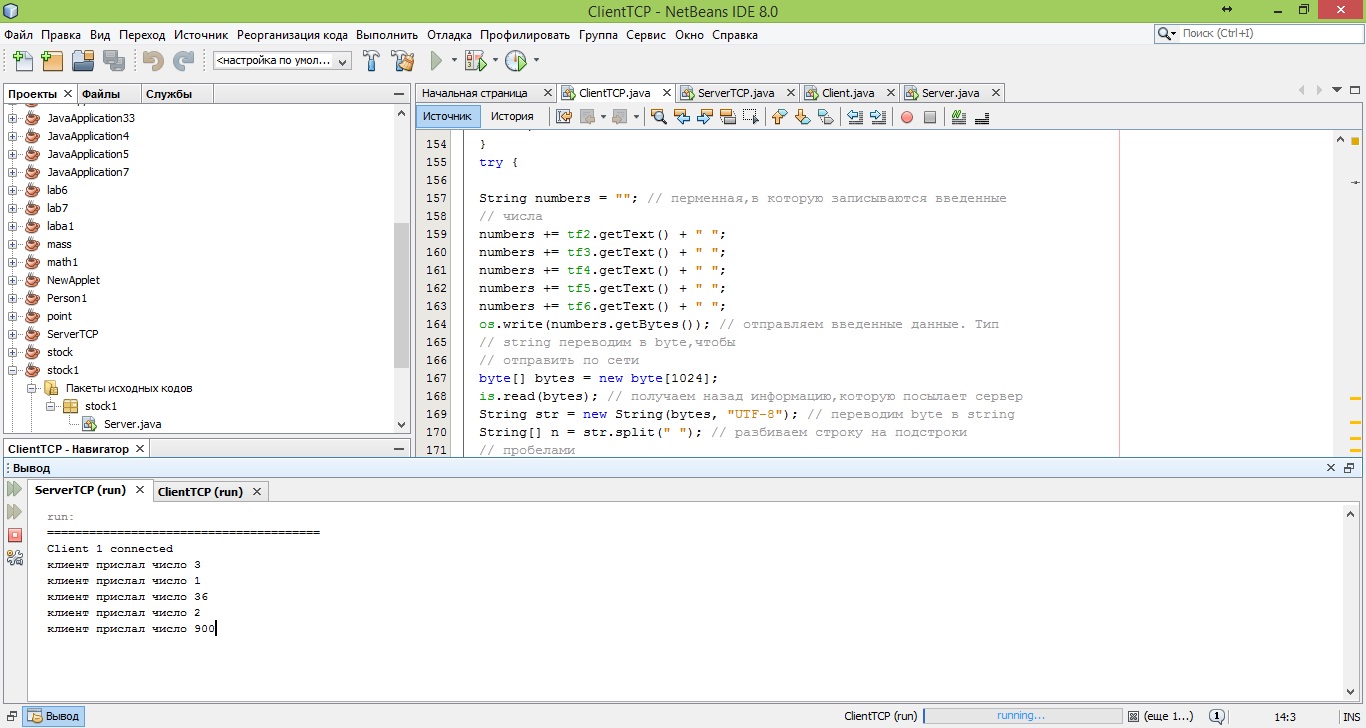


Рисунок 2.2. Консоль Сервера

## 

## **Задания для самостоятельного выполнения**

На базе приведенного выше примера разработать приложение, позволяющее осуществлять взаимодействие клиента и сервера по совместному решению задач обработки информации (смотреть варианты индивидуальных заданий). Протокол взаимодействия – *TCP.* Приложение клиента должно иметь GUI-интерфейс, показанный на рис. 2.3. Приложение сервера можно оставить в консольной реализации. Сервер должен предоставлять возможность последовательного установления соединения с клиентами (как это реализовано в примере).

**Клиент**

TextField

Label

Frame

**IP address**

127.0.0.1

Button

TextArea



**send**

**port**

1230

Hello

World

**Sending data (посылаемые данные)**

**Result (полученный от сервера результат)**

hELLO wORLD

Рис. 2.3. Интерфейс Клиента

**Варианты индивидуальных заданий (взаимодействие клиента и сервера по совместному решению задач обработки информации)**

1. Клиент посылает два числа серверу и одну из математических операций: «\*», «/», «+», «–» ,– сервер соответственно умножает, делит, складывает либо вычитает эти два числа и посылает ответ назад клиенту.
2. Клиент посылает слово серверу, сервер возвращает назад в обратном порядке следования букв это слово клиенту.
3. Клиент посылает два числа серверу m и n, сервер возвращает m!+n! этих чисел назад клиенту.
4. Клиент посылает два слова серверу, сервер их сравнивает и возвращает «истина», если они одинаковы по количеству и порядку следования в них букв, и «ложь»– при невыполнении хотя бы одного из этих условий.
5. Клиент посылает произвольный набор латинских букв серверу и получает их назад упорядоченными по алфавиту.
6. Клиент посылает серверу произвольный набор символов, сервер замещает каждый четвертый символ на «%».
7. Сервер генерирует прогноз погоды на неделю. Клиент посылает день недели и получает соответствующий прогноз.
8. Клиент посылает серверу произвольные числа и получает назад количество чисел, кратных трем.
9. Клиент посылает серверу символьную строку, содержащую пробелы, и получает назад ту же строку, но в ней между словами должен находиться только один пробел.
10. Клиент посылает серверу слово. Сервер определяет, является ли это слово палиндромом (палиндром – слово, читающееся одинаково как слева направо и справа налево).
11. Клиент посылает серверу два числа и получает назад НОД (наибольший общий делитель) этих чисел.
12. Клиент посылает серверу число от 0 до 10 и получает назад название этого числа прописью.
13. Клиент посылает серверу координаты точки Х и У в декартовой системе координат. Сервер определяет, в какой координатной четверти находится данная точка и посылает результат назад клиенту.
14. Клиент посылает серверу координаты прямоугольной области и точки в декартовой системе координат. Сервер определяет, лежит ли данная точка в прямоугольной области, и посылает результат назад клиенту.
15. Клиент посылает серверу шестизначный номер билета. Сервер определяет, является ли этот билет «счастливым». «Счастливым» называется такой билет, у которого сумма первых трех цифр равна сумме последних трех. Сервер посылает результат назад клиенту.