**ЛАБОРАТОРНАЯ РАБОТА №3**

# РАЗРАБОТКА ПРИЛОЖЕНИЙ АРХИТЕКТУРЫ «КЛИЕНТ-СЕРВЕР», ВЗАИМОДЕЙСТВУЮЩИХ ПО ПРОТОКОЛУ TCP (ПАРАЛЛЕЛЬНЫЙ СЕРВЕР)

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

## Протокол TCP (Transmission Control Protocol)

Протокол TCP предоставляет транспортные услуги, отличающиеся от услуг UDP. Вместо ненадежной доставки датаграмм без установления соединений, он обеспечивает гарантированную доставку с установлением соединений в виде байтовых потоков.

Протокол TCP используется в тех случаях, когда требуется надежная [доставка](http://msaver.ru/goto/?durl=http://ad.admitad.com/goto/e1cd29760a0ec26fd9f6b2f3c7b52c/) сообщений. Он освобождает прикладные процессы от необходимости использовать таймауты и повторные передачи для обеспечения надежности.

Протокол TCP был спроектирован в качестве связующего протокола для обеспечения интерактивной работы между компьютерами. TCP обеспечивает надежность и достоверность обмена данными между процессами на компьютерах, входящих в общую сеть TCP, с одной стороны, взаимодействует с прикладным протоколом пользовательского приложения, а с другой, с протоколом, обеспечивающим "низкоуровневые" функции: маршрутизацию и адресацию пакетов, которые выполняет IP.

Протокол TCP требует, чтобы перед отправкой сообщения было открыто соединение. Серверное приложение должно выполнить так называемое **пассивное открытие (passive open),** чтобы создать соединение с известным номером порта, и, вместо того чтобы отправлять вызов в сеть, сервер переходит в ожидание поступления входящих запросов. Клиентское приложение должно выполнить**активное открытие (active open)**, отправив серверному приложению синхронизирующий порядковый номер (SYN), идентифицирующий соединение. Клиентское приложение может использовать динамический номер порта в качестве локального порта.

Сервер должен отправить клиенту подтверждение (ACK) вместе с порядковым номером (SYN) сервера. В свою очередь клиент отвечает АСК, и соединение устанавливается.

После этого может начаться процесс отправки и получения сообщений. При получении сообщения в ответ всегда отправляется сообщение АСК. Если до получения АСК отправителем истекает тайм-аут, сообщение помещается в очередь на повторную передачу.

TCP — это сложный, требующий больших затрат времени протокол, что объясняется его механизмом установления соединения, но он берет на себя заботу о гарантированной доставке пакетов, избавляя нас от необходимости включать эту функциональную возможность в прикладной протокол.

Протокол TCP имеет встроенную возможность надежной доставки. Если сообщение не отправлено корректно, мы получим сообщение об ошибке.

### Порты и установление TCP-соединений

В протоколе TCP также, как и в UDP, для связи с прикладными процессами используются порты. Номера портам присваиваются аналогичным образом: имеются стандартные, зарезервированные номера (например, номер 21 закреплен за сервисом FTP, 23 - за telnet), а менее известные приложения пользуются произвольно выбранными локальными номерами.

Однако в протоколе TCP порты используются несколько иным способом. Для организации надежной передачи данных предусматривается установление *логического соединения* между двумя прикладными процессами. В рамках соединения осуществляется обязательное подтверждение правильности приема для всех переданных сообщений, и при необходимости выполняется повторная передача. Соединение в TCP позволяет вести передачу данных одновременно в обе стороны, то есть полнодуплексную передачу.

Соединение в протоколе TCP идентифицируется парой полных адресов обоих взаимодействующих процессов (оконечных точек). Адрес каждой из оконечных точек включает IP-адрес (номер сети и номер компьютера) и номер порта. Одна оконечная точка может участвовать в нескольких соединениях.

Установление соединения выполняется в следующей последовательности:

* При установлении соединения одна из сторон является инициатором. Она посылает запрос к протоколу TCP на открытие порта для передачи (active open).
* После открытия порта протокол TCP на стороне процесса-инициатора посылает запрос процессу, с которым требуется установить соединение.
* Протокол TCP на приемной стороне открывает порт для приема данных (passive open) и возвращает квитанцию, подтверждающую прием запроса.
* Для того чтобы передача могла вестись в обе стороны, протокол на приемной стороне также открывает порт для передачи (active port) и также передает запрос к противоположной стороне.
* Сторона-инициатор открывает порт для приема и возвращает квитанцию. Соединение считается установленным. Далее происходит обмен данными в рамках данного соединения.

**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**

Сокет это программная абстракция, используемая для представления соединений между двумя машинами. Гипотетический «кабель» соединяющий две машины, каждый конец которого вставлен в сокет.

В Java, Вы создаете сокет для установления соединения с другой машиной, затем Вы получаете InputStream и OutputStream (либо с помощью соответствующих преобразователей, Reader и Writer) из сокета, который соответствующим образом представляет соединение, как потоковый объект ввода вывода. Есть два класса сокетов, основанных на потоках: ServerSocket - используется сервером, чтобы “слушать” входящие соединения и Socket - используется клиентом для инициирования соединения.

Как только клиент создает соединение по сокету, ServerSocket возвращает (с помощью метода accept( ) ) соответствующий объект Socket по которому будет происходить связь на стороне сервера. Начиная с этого момента, у Вас появляется соединение Socket к Socket, и Вы считаете эти соединения одинаковыми, потому что они действительно одинаковые. В результате, Вы используете методы getInputStream( ) и getOutputStream( ) для создания соответствующих объектов InputStream и OutputStream из каждого Socket.

## Разработка сервера, взаимодействующего по протоколу TCP

Создание структуры сервера показано на следующей функциональной диаграмме:

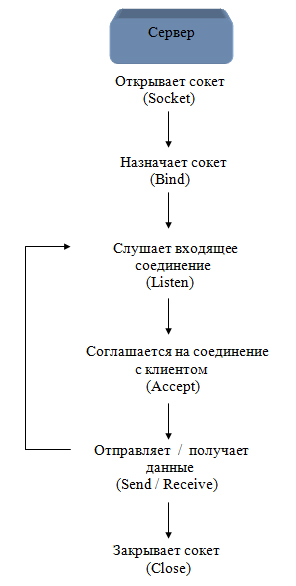


Рис.2.1. Создание структуры сервера

**Создание сокета сервера**

Для создания сокетов и управления ими в Java есть специальные классы java.net.Socket и java.net.ServerSocket. Первый для клиента, второй для сервера. Так же нам будут необходимы два класса из пакета java.io.\*: BufferedReader и PrintWriter для чтения/записи в сокет.

Для начала подключимся к порту хоста. Сделать это можно с помощью конструктора класса ServerSocket. Обратите внимание, что конструктор выбрасывает исключение типа IOException, т.е. нам понадобится блок try - catch:

Пример 2.4.

ServerSocket servers;

try {

servers = new ServerSocket(4444); *//создание сокета сервера*

} catch (IOException e) {

System.out.println("Couldn't listen to port 4444");

System.exit(-1);

}

После успешного подключения к порту сервер должен ждать подключения от клиента. Сделать это можно так:

Пример 2.5.

Socket fromclient;

try {

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

fromclient= servers.accept(); *//прослушивание*

*//(ожидание запроса на соединения)*

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

} catch (IOException e) {

System.out.println("Can't accept");

System.exit(-1);

}

Рассмотрим этот блок подробнее. Метод servers.accept() позволяет серверу следить за портом, или иначе говоря ждать подключения клиента. Как только клиент подключается - сокет для клиента сразу же создается. В противном случае выбрасывается исключение IOException.

Как только клиент подключился к серверу, сервер должен создать потоки ввода и вывода для связи с ним. Это можно сделать следующим образом:

Пример 2.6.

// получение входного и выходного потоков

BufferedReader in;

PrintWriter out;

in=new BufferedReader(new

InputStreamReader(fromclient.getInputStream()));

*// чтение байта из входного потока*

out = new PrintWriter(fromclient.getOutputStream(),true); *//запись байта*

*//в выходной поток*

И далее можно просто считывать данные из потока in и записывать данные в out.

Если конструктор ServerSocket завершается неуспешно, программа просто завершается (обратите внимание, что мы должны считать что конструктор ServerSocket не оставляет открытых сетевых сокетов если он завершается неудачно). В этом случает, main( ) выбрасывает исключение IOException и блок try не обязателен. Если конструктор ServerSocket завершается успешно, то остальные вызовы методов должны быть окружены блоками try-finally, чтобы убедиться, что независимо от того как блок завершит работу, ServerSocket будет корректно закрыт.

При написании сетевых программ Вы должны быть внимательными, при использовании автоматического сброса буферов. Каждый раз, когда Вы сбрасываете буфер, пакет должен создаться и отправиться. В этом случае, это именно то, что мы хотим, т.к. если пакет, содержащий строку не отослан то обмен информацией между сервером и клиентом остановится. С другой стороны, конец строки это конец сообщения. Но во многих случаях, сообщения не ограничиваются строками, так что более эффективно будет не использовать автоматический сброс, а позволить встроенной буферизации решать, когда необходимо создать и отослать пакеты. В этом случае, могут отсылаться большие пакеты и процесс пойдет быстрее.

## Разработка клиента, взаимодействующего по протоколу TCP

Функции, которые используются для создания приложения-клиента , более или менее напоминают серверное приложение. Как и для сервера, используются те же методы для определения конечной точки, создания экземпляра сокета, отправки и получения данных и закрытия сокета.

Создание структуры клиента показано на следующей функциональной диаграмме:

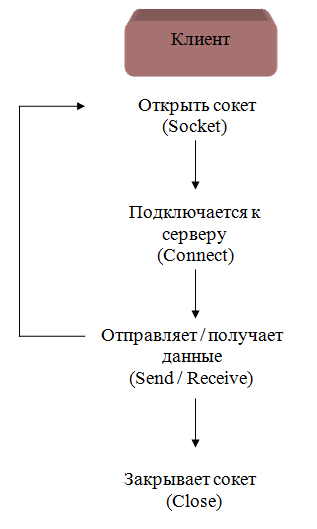


Рис.2.2. Создание структуры клиента

## Создание сокета клиента

Для создания клиента достаточно класса Socket и двух классов для ввода/вывода (см. Создаем сервер). Также необходимо знать имя компьютера (хоста), на котором запущен сервер и номер порта.

Конструктор сокета имеет два параметра: имя хоста и номер порта. Опять таки, конструктор выбрасывает исключение типа IOException.

Пример 2.7.

Socket fromserver = null;

fromserver = new Socket("localhost",4444);

Далее аналогичным образом, как для сервера, создаем потоки ввода вывода. И можно записывать/считывать данные.

**Реализация клиент-серверного приложения через сокеты**

Со стороны клиента программа работает следующим образом: клиент подсоединяется к серверу, отправляет данные, после чего получает данные от сервера и выводит их.

Со стороны сервера это выглядит следующим образом: сервер устанавливает сокет сервера на порт 3128, после чего ждёт входящих подключений. Приняв новое подключение, сервер передаёт его в отдельный вычислительный поток. В новом потоке сервер принимает от клиента данные, приписывает к ним порядковый номер подключения и отправляет данные обратно к клиенту.

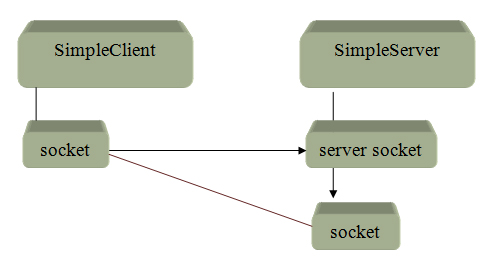


Рис.2.3. Логическая структура работы программ-примеров

#### Программа простого TCP/IP клиента

import java.lang.\*;

import java.io.\*;

import java.net.\*;

class Client {

public static void main(String args[]) {

try {

Socket skt = new Socket("localhost", 1234);

BufferedReader in = new BufferedReader(new

InputStreamReader(skt.getInputStream()));

System.out.print("Received string: '");

while (!in.ready()) {}

System.out.println(in.readLine()); // Read one line and output it

System.out.print("'\n");

in.close();

}

catch(Exception e) {

System.out.print("Whoops! It didn't work!\n");

}

}

}

#### Программа простого TCP/IP сервера

import java.lang.\*;

import java.io.\*;

import java.net.\*;

class Server {

public static void main(String args[]) {

String data = "Privet!!!";

try {

ServerSocket srvr = new ServerSocket(1234);

Socket skt = srvr.accept();

System.out.print("Server has connected!\n");

PrintWriter out = new PrintWriter(skt.getOutputStream(), true);

System.out.print("Sending string: '" + data + "'\n");

out.print(data);

out.close();

skt.close();

srvr.close();

}

catch(Exception e) {

System.out.print("Whoops! It didn't work!\n");

}

}

}

**Пример выполнения индивидуального задания**

Пример 2.8.

Данный пример демонстрирует работу двух приложений: сервера и клиента. Первым запускается сервер. 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.

**Приложение сервера**

package s2;

import java.net.\*;

import java.util.StringTokenizer;

import java.io.\*;

class ServerTCP{

private static class ClientHandler implements Runnable

{

// локальные используемые переменные

private int clientid=0;

private InputStream is;

private OutputStream os;

private int a=0,b=0,c=0;

private String sign="";

private Socket client;

public ClientHandler(int id,Socket s)

{

clientid=id;

client=s;

}

public void run()

{

try

{

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

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

boolean flag=true;

while(flag==true){//цикл для работы с одним клиентом

int k;

byte clientMessage[]=new byte[100];//байтовый массив, который будет использоваться при чтении из входного потока

try

{

k=is.read(clientMessage);//чтение иформации, посланной клиентом, из вхоного потока в массив clientMessage[] k - колличество считанных байт

}

catch(Exception e) //если клиент просто остановил свою работу не отправив disconnect

{

break;//завершение цикла обработки данного клиента

}

String tempString=new String(clientMessage,0,k);// формирование строки из того что считанно

tempString=tempString.trim();// убираем пробелы в конце и в начале

if(tempString.compareTo("disconnect")==0) {

System.out.println("message \"disconnect\" recieved from client");

flag=false;// приведет к прекращению повторения цикла

}

else {

//подразумеваем что строка передана в виде a знак b

try

{

StringTokenizer tok=new StringTokenizer(tempString);// StringTokenizer - класс для разбивки строки на отдельные слова

a=Integer.parseInt(tok.nextToken());//выделяем a

sign=tok.nextToken();//выделяем знак

b=Integer.parseInt(tok.nextToken());//выделяем b

c=count(a,b,sign);// вызываем функцию

String answer=tempString+" = "+c;

os.write(answer.getBytes());//запись в выходной поток вычисленного ответа

System.out.println(answer);//вывод сообщения в консоль на сервере

} catch(Exception e)// если не получилось что то преобразовать или посчитать

{

os.write(("Wrong message. "+e.toString()).getBytes());// отправляем клиенту сообщение об ошибке состоящее из wrong message и описания ошибки

}}}

is.close();//закрытие входного потока

os.close();//закрытие выходного потока

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

}catch(Exception e)

{

}

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

}}

static int countclients=0;//счетчик подключившихся клиентов

private static int count(int a,int b,String sign) throws Exception

{

switch(sign)

{

case "-": return a-b;

case "+": return a+b;

case "/": return a/b;

case "\*": return a\*b;

default: throw new Exception("wrong sign");

}

}

public static void main(String args[]){

ServerSocket sock;

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("Client "+countclients+" connected");//вывод сообщения

new Thread(new ClientHandler(countclients++,client)).start();// стартуем новый поток обработки сообщений нового клиента

// сама обработка находится в методе run класса ClientHandler

}}

catch(Exception e){// в случае возникновения других не предвиденных ошибок

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

}}}

**Приложение Клиента**

package k2;

import java.io.\*;

import java.net.\*;

import javax.swing.\*;

public class ClientTCP{

static final String options[]={"send","disconnect"};

public static void main(String args[]){

int a=0,b=0;

char c=0;

String sign="";

Socket sock;

InputStream is;

OutputStream os;

try{

sock=new Socket(InetAddress.getByName("localhost"),1024);//создаем сокет клиента и соединяемся с сервером который находится на порту 1024 на этом же компьютере

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

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

boolean cont=true; // если станет = false цикл прервется

while(cont){//бесконечный цикл для принятия и отсылки сообщений серверу

String stringmessage=JOptionPane.showInputDialog("Введите операций в формате a \"знак операции\" "

+ "b или disconnect для остановки работы клиента");//считываем полылаемое сообщение для сервера

if(stringmessage==null)

{

JOptionPane.showMessageDialog(null, "Отключение от сервара...");

break;}

if(stringmessage.compareTo("disconnect")==0) {

byte bytemessage1[]=stringmessage.getBytes();//преобразование сообщения из типа String в тип byte[]

os.write(bytemessage1);//запись в выходной поток преобразованного сообщения

JOptionPane.showMessageDialog(null, "Отключение от сервара...");

break; //выход из цила while

}

else {

byte bytemessage2[]=stringmessage.getBytes();//преобразование сообщения из типа String в тип byte[]

os.write(bytemessage2);//запись в выходной поток преобразованного сообщения

byte readmessage[]=new byte[100];//создаем массив байт для чтения информации от сервера

int k=is.read(readmessage);//считываем сообщение посланное от сервера k - количетсво считанных символов

JOptionPane.showMessageDialog(null, "Полученное сообщение от сервера: \n"+new String(readmessage,0,k));//выводим полученное сообщение

}}

is.close();//закрываем входной поток

os.close();//закрываем выходной поток

sock.close();//закрываем сокет клиента

}

catch(Exception e){// если возникла непредвиденная ошибка

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

}}}

# Задания для самостоятельного выполнения

Для каждого варианта необходимо реализовать параллельное подключение клиентов к серверу.

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.

**Список использованных источников**

1. <http://professorweb.ru/my/csharp/web/level3/3_2.php>
2. <http://citforum.ru/nets/ip/glava_6.shtml>
3. <http://www.javaportal.ru/java/articles/ClientServer.html>
4. <http://www.quizful.net/post/java-socket-programming>
5. <http://www.codenet.ru/webmast/java/sockets.php>
6. <http://skif.bas-net.by/bsuir/base/node350.html>
7. <http://professorweb.ru/my/csharp/web/level1/1_4.php>