Практическая работа №3
Программирование JAVA сокетов

Цель работы: знакомство студентов с одной из базовых технологии реализации конечной точки для передачи и приема данных по сети - сокетом.

Теоретические сведени

Клиент-серверная архитектура - это самая известная структура приложений в Интернете. В этой архитектуре клиенты (например, персональные компьютеры, устройства IoT и т. Д.) вначале запрашивают ресурсы с сервера, а затем сервер отправляет соответствующие ответы на запросы клиентов. Чтобы это произошло, как на клиентах, так и на серверах должен быть реализован какой-то механизм, поддерживающий эту сетевую транзакцию. Этот механизм называется коммуникацией через сокеты.

Стоит отметить, что существует два типа сокетов для TCP и UDP. Поскольку большинство сетевых приложений используют TCP, **в тексте работы речь пойдёт только о TCP-сокетах и их реализации.** Основное различие между ними заключается в том, что UDP не имеет соединения, то есть между клиентом и сервером нет сеанса, в то время как TCP ориентирован на соединение, то есть сначала должно быть установлено эксклюзивное соединение между клиентом и сервером для связи.

Сокет работает по типичной модели запрос / ответ, где в java-программе, называемой клиентом, вызывается другая программа, называемая сервером, работающая на другой JVM. Клиент должен отправить запрос, и сервер отвечает ответом. В этой модели обмен всегда инициируется клиентом; сервер не может отправлять какие-либо данные без предварительного запроса клиента. Стоит отметить, что данная модель хорошо работала во всемирной паутине, когда клиенты время от времени запрашивали документы, которые нечасто менялись, но ограничения этого подхода становятся все более актуальными, поскольку контент меняется быстро и пользователи ожидают более интерактивного взаимодействия в сети. Например, WebSocket устраняет эти ограничения, предоставляя полнодуплексный канал связи между клиентом и сервером. Протокол WebSocket (JSR 356) будет рассмотрен в следующей практической работе. Но на практике, сокеты активно используются при реализации клент-серверного взаимодействия.

Сокет - это одна из конечных точек двустороннего канала связи между двумя программами, работающими в сети. Сокет привязан к номеру порта, чтобы уровень TCP мог идентифицировать приложение, данные предназначенные для отправки.

Каждый сервер - это программа, которая работает в определенной системе и прослушивает определенный порт. Сокеты привязаны к номерам портов, и когда мы запускаем любой сервер, он просто слушает сокет и ждет клиентских запросов. Например, сервер tomcat, работающий на порту 8080, ожидает клиентских запросов и, получив любой клиентский запрос, отвечает на них.

Следовательно, для запуска любого сокета необходим запускаемый порт. Очевидно, что он размещен на машине, идентифицируемой именем хоста и уникальной адресованной IP.

В Java классы для поддержки программирования сокетов упакованы в пакет java.net. В соответствии с Java, java.net пакет поставляется с двумя классами Socket и ServerSocket для функций клиента и сервера соответственно.

Обобщённая схема работы сокетов

В общем виде, для связи через сокет требуется серверный сокет, привязанный к порту хоста. На приведенной выше диаграмме показано, что сервер запускает серверный сокет с портом и прослушивает входящий запрос. После поступления запроса для клиента создается уникальный сокет, а потоки ввода и вывода используются для взаимодействия с каналом, созданным для клиента. На стороне клиента есть еще один сокет для инициирования соединения с сервером и отправки запроса.

Рассмотрим более подробно процесс подключения с использованием сокетов JAVA, как на стороне клиента, так и на стороне сервера.

На стороне сервера:

Обычно сервер работает на определенном компьютере, и он создает «серверный сокет» (связанный с портом) и приостанавливает работу. Сервер регистрирует свою службу под номером порта. Затем сервер ожидает эту службу. Класс ServerSocket используется на стороне сервера: он просто ожидает вызовов от клиента (ов).

На стороне клиента:

Клиенты знают имя хоста сервера и номер порта, с которым сервер работает. Клиент может установить соединение с сервером, запросив создание Socket, предназначенного для сервера для порта, на котором была зарегистрирована служба. Клиент подключается к

сокету сервера; Затем создаются два сокета: «клиентский сокет» на стороне клиента и «сокет клиентской службы» на стороне сервера. Эти сокеты соединены друг с другом.

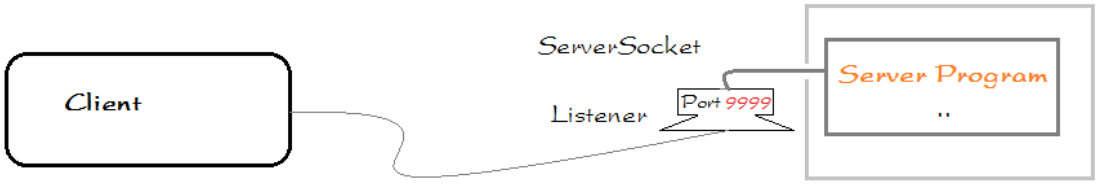


Рисунок 3.1 - Отправка запроса на подключение к серверу

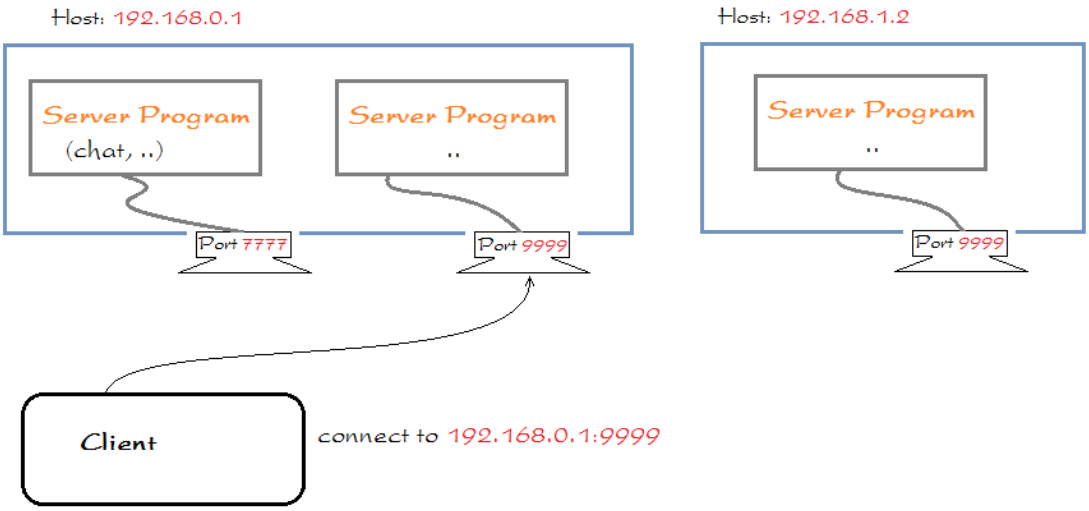
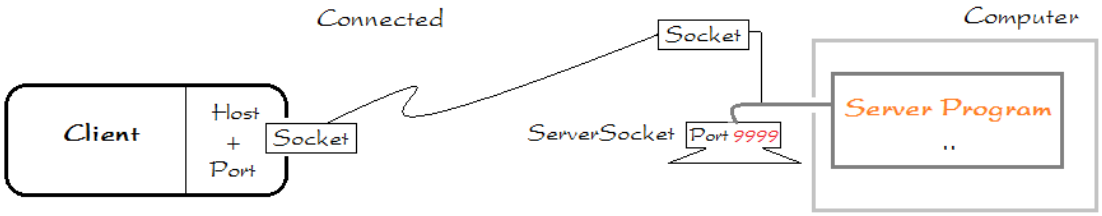


Рисунок 3.2 - Отправка запроса на подключение к серверу

Далее серверная программа принимает соединение от клиента. После принятия сервер получает новый сокет, который напрямую привязан к тому же локальному порту. Ему нужен новый сокет, чтобы он мог продолжать прослушивать исходный сокет (ServerSocket) для запросов на соединение. И все это при удовлетворении потребностей подключенного клиента. Вот как принять соединение от клиента :



Теперь клиент и сервер могут общаться, записывая или читая свои сокеты.

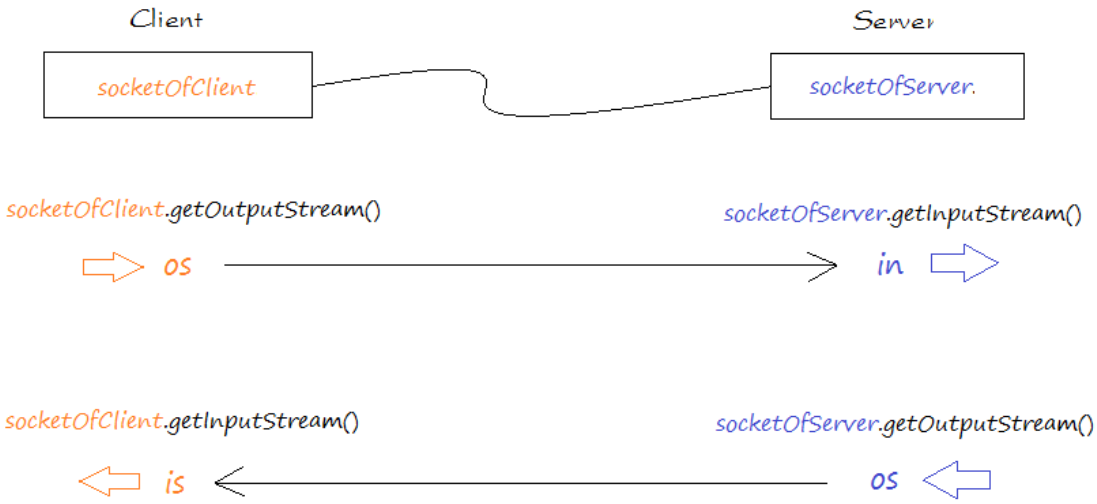


Рисунок 3.4 - Процесс передачи данных от клиента к серверу и обратно

Данные, записанные в выходной поток Socket клиента, будут получены через входной поток Socket of Server. И наоборот, данные, записанные в выходной поток в Socket of Server, будут использоваться во входном сокете клиента.

Исходя из вышеизложенного, можно отметить, что связь TCP-сокета между клиентом и сервером состоит из нескольких этапов, которые показаны на рисунке ниже.

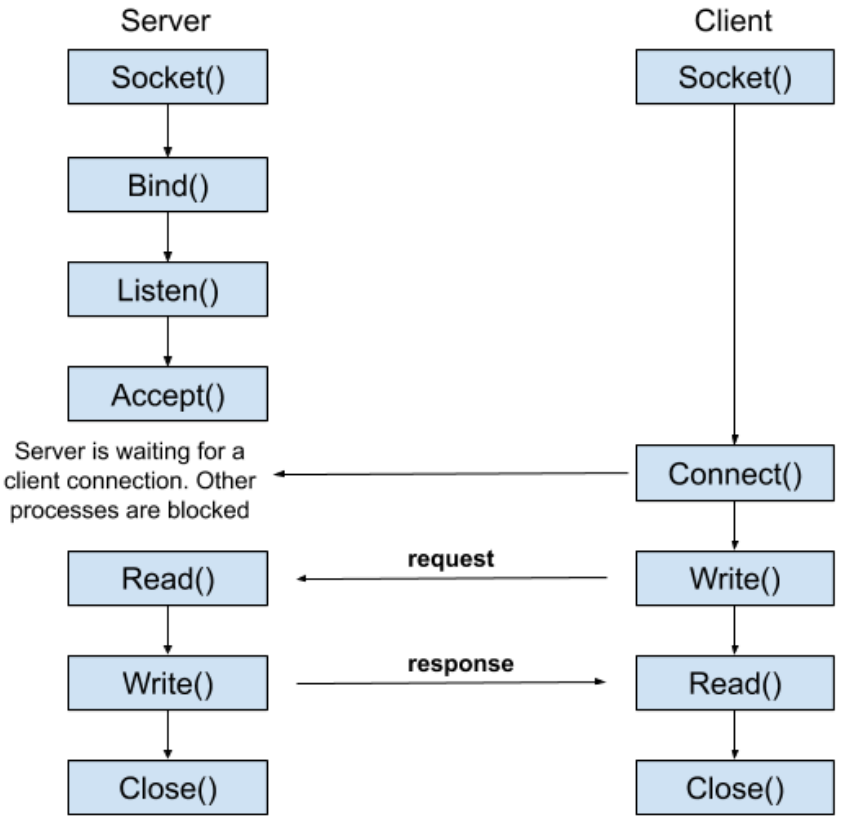


Рисунок 3.5 - Схема обмена данными через сокет TCP

- **Socket ()** - на сервере создается конечная точка для связи.
- **Bind ()** - присвоение уникального номера сокету и резервирование уникальной комбинации IP-адреса и порта для созданного сокета.
- **Listen ()** - после создания сокета сервер ожидает подключения клиента.
- **Accept ()** - сервер получает запрос на подключение от клиентского сокета.
- **Connect ()** - клиент и сервер связаны друг с другом.
- **Send () / Recieve ()** - Обмен данными между клиентом и сервером
- **Close ()** - после обмена данными сервер и клиент разрывают соединение.

Рассмотрим пример относительно простой программы (простой последовательный сервер) на JAVA для демонстрации работы сокетов. Здесь мы напишем две программы на Java. Одна будет программой, работающей на сервере, а другая - клиентской программой, которая будет взаимодействовать с сервером.

В этом клиент-серверном приложении есть два класса - SocketServer и SocketClient. SocketServer класс выполняет следующие действия для прослушивания запросов:

- Создает серверный сокет, используя порт с помощью конструктора.
- Прослушивает соединения, используя метод serverSocket.accept(). Это блокирующий вызов и ожидание поступления запроса.
- Как только поступает запрос, он переходит к следующему набору инструкций.
- Кроме того, он использует OutputStream объект сокета для записи на выход.
- Он использует InputStream объект сокета для чтения ввода.
- Сервер читает из входного потока, преобразует его в String и затем возвращает с ответом.

Соединительный сокет создается в блоке try-with-resources, поэтому он автоматически закрывается в конце блока. Только после обслуживания даты и времени и закрытия соединения сервер вернется в режим ожидания следующего клиента.

Реализуем серверную часть, как показано в листинге ниже.

```
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
```

```
import java.net.Socket;

public class SocketServer {

  public static final int SERVER_PORT = 50001;

  public static void main (String[] args){

    try {
      ServerSocket server = new ServerSocket(SERVER_PORT);
      Socket clientConn = server.accept();
                    DataOutputStream   serverOutput   =   new
DataOutputStream(clientConn.getOutputStream());
      serverOutput.writeBytes("JAVA\n");
      clientConn.close();
    }
    catch (IOException e) {
      e.printStackTrace();
    }

  }

}
```

В приведенной выше программе сервер открывает сокет из порта 50001 на сервере и ожидает клиента server.accept(). После подключения клиента создается выходной поток. Это можно использовать для отправки данных с сервера подключенному клиенту. Это именно то, что serverOutput.writeBytes()делает. После отправки данных соединение с клиентом закрывается.

Теперь создадим клиента для взаимодействия с сервером сокетов, созданным выше. Реализация Socket-клиента показана в листинге ниже.

```
      import java.io.BufferedReader;
      import java.io.InputStream;
      import java.io.InputStreamReader;
      import java.net.Socket;

      public class SocketClient {
        public static void main(String[] args){
          try {
                        Socket   clientSocket   =   new   Socket
("localhost",50001);
            InputStream is = clientSocket.getInputStream();
                  BufferedReader  br  =  new  BufferedReader(new
InputStreamReader(is));
            String receivedData = br.readLine();
```

```
                    System.out.println("Полученные данные:
"+receivedData);
          }
          catch (Exception e) {
            e.printStackTrace();
          }
        }
      }
```

Вышеупомянутая программа действует как клиент, создавая соединение с серверным сокетом. После подключения он получает данные, отправленные сервером. Входной поток подключается к буферу с помощью BufferedReader, для хранения полученных данных, поскольку мы не можем гарантировать, что данные используются сразу после получения. Затем данные читаются из буфера и выводятся в консоль.

Для того, чтобы продемонстрировать работу клиент-серверного приложения, запустить первоначально запустите серверную Java-программу, а затем клиентскую Java-программу выполнив команды (если в консоли) javac `SocketServer`.java && java SocketServer (для компиляции и запуска серверной части) и javac SocketClient.java && java SocketClient 127.0.0.1 (для компиляции и запуска клиентской части). Протестировать работу сервера можно с помощью команд netstat -an | grep 50001 , где 50001 тот порт, который указан в SERVER_PORT и команды NetCat nc localhost 59090. Последняя должна вернуть соответствующий текст.

На стороне клиента Socket конструктор принимает IP-адрес (localhost) и порт (50001) на сервере. Если запрос на соединение принят, мы получаем объект сокета для связи. Данное приложение настолько простое, что клиент никогда не пишет на сервер, а только читает.

Стоит обратить внимание, что данные пример обладает рядом существенных недостатков. Так, каждый клиент должен дождаться, пока предыдущий клиент не будет полностью обслужен, прежде чем он даже будет принят, что приводит к определённым сложностям при приёме несколько запросов. Немаловажным фактом, является то, что связь через сокеты всегда буферизуется. Это означает, что ничего не отправляется и не принимается, пока буферы не заполнятся или вы явно не очистите буфер.

Далее рассмотрим модернизацию нашего приложения, создав простой многопоточный сервер, который получает строки текста от клиента и отправляет обратно строки в верхнем регистре. Клиент подключается, сервер порождает поток, посвященный только этому клиенту, для чтения, прописных букв и ответа. Сервер может

одновременно прослушивать и обслуживать других клиентов, поэтому у нас есть настоящий параллелизм.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;
import java.util.concurrent.Executors;

/**
  * Серверная программа, которая принимает запросы от
клиентов на обработку строк с заглавных букв.
  * Когда клиент подключается, то запускается новый поток
  * для его обработки. Получение клиентских данных, их
  * использование и отправка ответа - все это делается в
  * потоке, что обеспечивает гораздо большую пропускную
  *способность, поскольку одновременно может
  * обрабатываться больше клиентов.
 */
public class CapitalizeServer {

    /**
     * Запускается сервер. Когда клиент подключается,
     * сервер создает новый поток для обслуживания и
     * немедленно возвращается к прослушиванию.
     * Приложение ограничивает количество потоков через
     * пул потоков (в противном случае миллионы клиентов
     * могут привести к исчерпанию ресурсов сервера из-за
     * выделения слишком большого количества потоков).
     */
        public static void main(String[] args) throws
Exception {
        try (var listener = new ServerSocket(59898)) {
            System.out.println("Сервер запущен...");
            var pool = Executors.newFixedThreadPool(20);
            while (true) {
                                                pool.execute(new
Capitalizer(listener.accept()));
            }
        }
    }

     private static class Capitalizer implements Runnable
{
        private Socket socket;

        Capitalizer(Socket socket) {
            this.socket = socket;
```

```
            }

            @Override
            public void run() {
                System.out.println("Подключение: " + socket);
                try {
                                                    var   in   =   new
Scanner(socket.getInputStream());
                                                   var   out   =   new
PrintWriter(socket.getOutputStream(), true);
                    while (in.hasNextLine()) {
                        out.println(in.nextLine().toUpperCase
());
                    }
                } catch (Exception e) {
                    System.out.println("Ошибка:" + socket);
                } finally {
                    try {
                        socket.close();
                    } catch (IOException e) {
                    }
                    System.out.println("Closed: " + socket);
                }
            }
        }
    }
```

Стоит отметить основные нюансы данной реализации многопоточного сервера. Во-первых, серверный сокет, приняв соединение, не делает ничего, кроме запуска потока. Во-вторых, в Java никогда не следует создавать потоки напрямую; вместо этого необходимо использовать пул потоков и службу исполнителя для управления потоками. Ограничение размера пула потоков защитить распределённое приложение от заваления миллионами клиентов. Finally блок закрывает сокет. Здесь не используется блок try-with-resources, потому что сокет был создан в основном потоке. То, что выполняется в потоках, называется задачами; они реализуют Runnable интерфейс; они делают свою работу своим run методом. В конструкторе задачи никогда не должно быть слишком много работы! Конструктор запускается в основном потоке. Необходимо помещать всю работу (кроме захвата аргументов конструктора) в run метод.

Далее необходимо реализовать клиента, но первоначально необходимо протестировать работоспособность сервера, через утилиту NetCat, выполнив команду nc 127.0.0.1 59898

```
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

public class CapitalizeClient {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
              System.err.println("Передайте IP-адрес сервера в
качестве единственного аргумента командной строки");
            return;
        }
        try (var socket = new Socket(args[0], 59898)) {
              System.out.println("Введите строки текста, затем
Ctrl + D или Ctrl + C, чтобы выйти");
            var scanner = new Scanner(System.in);
            var in = new Scanner(socket.getInputStream());
                                        var     out     =     new
PrintWriter(socket.getOutputStream(), true);
            while (scanner.hasNextLine()) {
                out.println(scanner.nextLine());
                System.out.println(in.nextLine());
            }
        }
    }
}
```

Этот клиент многократно считывает строки из стандартного ввода, отправляет их на сервер и записывает ответы сервера. Его можно использовать, как в интерактивном режиме, передавая через консоль данные, так и передавая файл.

Задание на практическую работу

Необходимо создать клиент-серверное приложение на языке JAVA с использованием socket, для широковещательного общения пользователей. Приложение может быть как консольным, так и оснащённым полноценным GUI. Клиентское приложение считывает данные из стандартного ввода и отсылает сообщение серверу (с помощью TCP/IP). Сервер, в свою очередь, накапливает сообщения и раз в 5 секунд осуществляет массовую рассылку всем клиентам. Если сообщений за указанный период не поступило, то рассылка не производится. Клиент, получивший сообщение, отображает на

экране текст данного сообщения. Структуру и поведение данного клиент-серверного приложения, в том числе, **например**, в части регистрации конкретного клиента и формата широковещательного сообщения, студент определяет самостоятельно.