Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный
университет имени П.М. Машерова»
Кафедра прикладного и системного программирования

В.В. Новый

ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ НА ОСНОВЕ TCP СОКЕТОВ

Методические рекомендации к выполнению лабораторных работ

Витебск
ВГУ имени П.М. Машерова
2023

УДК 004.4(076)
ББК 32.973.5я73
Н76

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 7 от 26.04.2023.

Автор: старший преподаватель кафедры прикладного и системного программирования ВГУ имени П.М. Машерова **В.В. Новый**

Р е ц е н з е н т :
заведующий кафедрой информационных систем и технологий УО «ВГТУ», кандидат технических наук, доцент *В.Е. Казаков*

Новый, В.В.

Н76 Программирование сетевых приложений на основе TCP сокетов : методические рекомендации к выполнению лабораторных работ / В.В. Новый. – Витебск : ВГУ имени П.М. Машерова, 2023. – 25 с.

В методических рекомендациях предлагаются задания для организации лабораторных работ по дисциплине «Программирование сетевых приложений», сопровождающиеся необходимыми теоретическими сведениями и пояснениями, помогающие ознакомиться с разработкой программ сетевого взаимодействия на базе TCP сокетов с использованием языка программирования Java и его библиотеки классов. Приведен ряд заданий для самостоятельного их выполнения студентами и более глубокого изучения указанной тематики.

Предназначается для обучающихся по специальности «Информационные системы и технологии (в здравоохранении)».

УДК 004.4(076)
ББК 32.973.5я73

ДЕРЖАНИЕ

ВВЕДЕНИЕ

Современное программное обеспечение тесно связано с компьютерными сетями. Обмен данными применяется как в сложных крупномасштабных информационных системах, так и в самых миниатюрных приложениях. Даже простейший текстовый редактор умеет проверять обновления на сайте автора и подгружать необходимые плагины из сети. Данные методические рекомендации преследуют цель познакомить студентов с основами разработки сетевых приложений на языке программирования Java с использованием TCP сокетов и позволяют повторить необходимый для успешного освоения программирования сетевых приложений материал из дисциплины «Компьютерные сети», в частности, вопросы, связанные с клиент-серверным подходом к построению сетевых приложений, логической адресацией на сетевом уровне, понятиями сокета и порта.

Данное учебное издание содержит обзор основных классов, применяемых при построении приложений сетевого взаимодействия на основе TCP сокетов в библиотеке языка программирования Java: *InetAddress, NetworkInterface, Socket, ServerSocket, InetSocketAddress*. Приведенные темы дополняют знания, полученные в курсе «Компьютерные сети», и могут служить отправной точкой для более глубокого освоения сетевого программирования на языках высокого уровня, а также выступать базой для изучения дисциплин «Сервис ориентированная архитектура», «Системный анализ и проектирование информационных систем», использоваться при подготовке курсовых и дипломных работ.

Приведенные задания лабораторных работ ориентированы на разработку сетевых приложений по официальным RFC-описаниям стандартных протоколов, что позволяет параллельно с изучением сетевого программирования на языке Java формировать навыки работы с документацией по прикладным протоколам.

Материал методических рекомендаций соответствует темам учебной программы дисциплины «Программирование сетевых приложений» специальности «Информационные системы и технологии (в здравоохранении)».

ЛАБОРАТОРНАЯ РАБОТА.
ОСНОВЫ ПРОГРАММИРОВАНИЯ КЛИЕНТ-СЕРВЕРНЫХ ПРОГРАММ НА БАЗЕ СОКЕТОВ

Цель работы: рассмотреть основы разработки простейших клиент-серверных программ, основные классы и интерфейсы реализации сетевого взаимодействия, методы организации клиент-серверного взаимодействия на базе сокетов TCP/IP для серверов и клиентов; сформировать умения разработки сетевых приложений на языке программирования Java.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Основные понятия и термины из компьютерных сетей

Наиболее распространенным способом построения сетевых приложений уже довольно долгое время является *клиент-серверный подход*.

Под **клиентом (client)** понимается модуль, предназначенный для формирования и передачи сообщений-запросов к ресурсам удаленного компьютера от разных приложений с последующим приёмом результатов из сети и передачей их соответствующим приложениям.

Сервер (server) – это модуль, который постоянно ожидает прихода из сети запросов от клиентов и, приняв запрос, пытается его обслужить, как правило, с участием локальной операционной системы; один сервер может обслуживать запросы сразу нескольких клиентов (поочередно или одновременно).

Говоря по-другому, сервер реализует некоторую службу (service), услугами которой пользуются клиентские приложения: клиентское приложение инициирует подключение, а серверное – ожидает запросов на которые должно отвечать. С использованием такого подхода построены системы электронной почты, World Wide Web, различные банковские и медицинские информационные системы и т.д.

Альтернативой клиент-серверному подходу в разработке сетевых приложений является одноранговый или пиринговый подход (peer-to-peer, P2P) при котором программы на компьютерах пользователей взаимодействуют друг с другом напрямую. Такой подход применяется в ряде сетевых чатов и систем обмена файлами.

При программировании сетевых приложений в настоящее время чаще всего используется интерфейс прикладного программирования на базе сокетов. Этот интерфейс поддерживается всеми распространенными операционными системами и может использоваться практически на любом языке программирования: от ассемблера до TypeScript.

Понятие **сокета (socket)** используется в сетевом программировании для идентификации конечных точек сетевого взаимодействия (конкретных

приложений, обменивающихся данными по сети). Для этого необходимо решить две проблемы:

- Найти компьютер, на котором запущено необходимое приложение в сети;
- Найти сетевой процесс, с которым необходимо взаимодействовать на найденном компьютере.

Как Вы помните из курса компьютерных сетей, за определение сетевого интерфейса компьютера в сетях отвечает сетевой уровень. Для однозначной идентификации в сети на этом уровне каждому интерфейсу компьютера присваивается **IP-адрес (Internet Protocol address)**. В настоящее время используются две версии протокола IP и, соответственно, два формата IP-адресов: 4-байтовые IPv4 адреса и 16-байтовые IPv6 адреса. Стандартным форматом записи IPv4 адреса, который с точки зрения программного обеспечения является целым 32-битным числом, является **десятичная точечная нотация**, которая подразумевает, что адрес разбивается на четыре октета, каждый из которых записывается в виде числа в десятичной системе счисления и отделяется от остальных чисел точками. Например, **192.168.0.1** является примером такого формата записи IPv4 адреса. IPv6 адрес принято записывать группами шестнадцатеричных чисел, разделенных двоеточиями. Например, 2023:adb9:0000:0000:0001:cafe:15a0. Каждая группа представляет два байта адреса. Одна последовательная группа нулей может быть заменена двойным двоеточием (::), ведущие нули в группе могут быть пропущены, так что указанный выше адрес может быть записан как 2023:adb9::1:cafe:15a0.

Для удобства пользователей компьютерной сети предусмотрена возможность обращения к хостам по символьным именам, вместо числовых адресов. Однако сетевые протоколы работают только с числовыми адресами. Это требует дополнительной работы по **разрешению (resolve)** символьного имени в IP адрес. Разрешением имен в современном интернете занимаются система **DNS (Domain Name System)** и локальные конфигурационные файлы операционной системы.

За идентификацию приложения на заданном компьютере отвечает такая абстракция транспортного уровня как номер **порта (port)**. Номера портов представляют собой беззнаковые целые числа в диапазоне от 0 до 65535 (16-бит, номер порта 0 зарезервирован). Каждый номер порта может быть выделен одновременно ровно одному приложению. При этом номера портов от 1 до 1023 обычно соответствуют строго определенным сетевым службам и часто называются «*хорошо известными*». Например, порт 443 закреплен за службой WWW работающей поверх протоколов SSL/TLS, а порт 110 – за почтовым сервером POP3 без использования шифрования. В ряде операционных систем (например, GNU/Linux) их использование разрешено только приложениям, запущенным с root-правами (euid = 0). Любое сервер-

ное приложение, предоставляющее клиентам услуги, будет иметь как минимум один связанный с ним порт, на котором оно будет ожидать запросов клиентов. Когда клиент попытается связаться с таким приложением, он должен будет указать IP-адрес компьютера, на котором работает этот сервер и номер порта запрашиваемой услуги. Компьютер назначения выполнит проверку номера порта и передаст запрос клиента на подключение нужной серверной программе на обработку.

Как было упомянуто выше, в интерфейсе сокетов IP-адрес компьютера и номер порта, связанный с определенным приложением, объединяются такой абстракцией как сокет. Сокет однозначно идентифицирует одну из сторон сетевого взаимодействия. Когда клиент пытается подключиться к серверу, он создает сокет на своей стороне. Когда сервер получает от клиента запрос на подключение, он в свою очередь, создает сокет, связанный с подключившимся клиентом, на своей стороне. Таким образом реализуется идентификация различных клиентов при взаимодействии на стороне сервера.

В зависимости от класса транспортного обслуживания и поддержки соединения, сокеты можно разделить на **потоковые** и **дейтаграммные**. Потоковые сокеты в качестве транспортного протокола используют протокол **TCP (Transmission Control Protocol)**, призванный решать проблемы с нарушением последовательности передачи пакетов, их повреждением или потерей. Дейтаграммные сокеты в качестве транспортного используют протокол **UDP (User Datagram Protocol)**, который считается ненадежным, так как не гарантирует правильного порядка передаваемых пакетов и не гарантирует их доставки до получателя. Однако, протокол UDP имеет меньшие накладные расходы, связанные с меньшим размером заголовков и более быстрой отправкой данных, а также поддерживает широковещание. Этим объясняется его применение в приложениях потокового аудио и видео вещания.

Основные классы и интерфейсы реализации сетевого взаимодействия в Java

Библиотека языка программирования Java содержит большое количество классов, облегчающих разработку сетевых приложений. Далее будут рассмотрены некоторые из них, а также примеры их использования на практике.

Класс InetAddress

Как было отмечено выше, одной из первых задач сетевого взаимодействия является идентификация сетевого узла. Этой задаче служит класс *InetAddress* из пакета *java.net*. Для задания адреса назначения могут быть использованы или IP-адрес (версии 4 или 6), или символьное имя узла. Класс имеет два прямых подкласса: *Inet4Address* и *Inet6Address*, которые представляют соответственно IPv4 и IPv6 адреса. Экземпляры этого класса

являются неизменяемыми (immutable), т.е. после создания объекта они не могут быть изменены и ссылаются на тот же адрес, что и при создании. Экземпляры класса могут хранить как IP адрес, так и соответствующее ему DNS-имя (если экземпляр класса создавался на основе символьного имени хоста или выполнялось разрешение IP-адреса в символьное имя через запрос к реверс-зоне). Класс также выполняет кэширование для успешного или неудачного разрешения имен с настраиваемым TTL.

У класса нет публично доступного конструктора. Вместо этого предоставляется несколько статических фабричных методов для создания экземпляров класса:

static InetAddress[] getAllByName(String host)
static InetAddress getByName(String host)
static InetAddress getLocalHost()

Статический метод *getByName* используется для разрешения символьного DNS имени в IP-адрес в виде объекта *InetAddress*. Также *getByName* может работать с текстовым представлением IP-адреса и в этом случае он выполняет проверку его корректности. В случае ошибки создает исключение *UnknownHostException*. Статический метод *getAllByName* используется для получения массива объектов *InetAddress*, содержащих все IP-адреса для заданного имени. Статический метод *getLocalHost* получает имя текущего хоста от операционной системы и затем пытается разрешить его в IP-адрес, который и возвращает в виде объекта *InetAddress*. Метод *getCanonicalHostName* возвращает строку с FQDN именем для заданного хоста, разрешая в том числе CNAME ресурсные записи DNS.

Для хранимого адреса может быть получено его текстовое представление методом *getHostAddress*. Также для вывода на экран может быть применен метод *println* (который вызовет для объекта метод *toString*) к объекту класса. В этом случае адресная информация будет возвращена в формате: *имя_хоста / текстовое представление IP адреса* (если имя хоста не задавалось при создании экземпляра класса и не выполнялось запроса к обратной зоне DNS, то *имя_хоста* будет возвращено в виде пустой строки).

Экземпляр класса также умеет выполнять проверку доступности сетевого хоста по заданному адресу при помощи перегруженных методов *isReachable*. Проверка при наличии привилегии выполняется на основе отправки хосту назначения echo-запросов ICMP или через попытку подключения по протоколу TCP на порт простой TCP-службы Echo при недостаточном уровне привилегии.

Еще одна возможность – определение типа адреса, представленного объектом InetAddress:

public boolean isAnyLocalAddress()
public boolean isLinkLocalAddress()

public boolean isLoopbackAddress()
public boolean isMCGlobal()
public boolean isMCLinkLocal()
public boolean isMCNodeLocal()
public boolean isMCOrgLocal()
public boolean isMCSiteLocal()
public boolean isMulticastAddress()
public boolean isSiteLocalAddress()

Более подробную информацию по методам класса можно получить из официальной документации, доступной по адресу: *https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/net/InetAddress.html*

Пример использования класса InetAddress 1

Рассмотрим применение класса *InetAddress* для получения IP-адреса заданного хоста (листинг 1.1):

Листинг 1.1

```
1 import java.net.InetAddress;
2 import java.net.UnknownHostException;
3
4 public class ResolveName {
5     public static void main (String[] args) {
6         try {
7           InetAddress addr = InetAddress.getByName("www.vsu.by");
8           System.out.println(addr);
9         } catch (UnknownHostException ex) {
10          System.out.println("Couldn't resolve www.vsu.by");
11        }
12    }
13 }
```

После сборки и запуска получим:

```
D:\repos\java\InetAddress\ex1>C:\jdk\bin\java.exe ResolveName
www.vsu.by/93.125.24.155
```

Обратите внимание, что если заменить строку 7 на следующую:

```
InetAddress addr = InetAddress.getByName("93.125.24.155");
```

То вывод программы измениться и DNS-имя хоста не будет отображаться:

```
D:\repos\java\InetAddress\ex1>C:\jdk\bin\java.exe ResolveName
/93.125.24.155
```

Это связано с тем, что при конструировании экземпляра класса не задействовалось разрешение имени и не вызывался метод разрешения IP-адреса в DNS-имя в явном виде, который бы заполнил соответствующее поле экземпляра класса.

Пример использования класса InetAddress 2

Как известно, отдельным именам хостов может соответствовать несколько IP-адресов. Для получения всех адресов, соответствующих некоторому имени удобно использовать метод *getAllByName*, возвращающий массив объектов *InetAddress* по одному на каждый адрес (листинг 1.2):

Листинг 1.2

```
1 import java.net.InetAddress;
2 import java.net.UnknownHostException;
3
4 public class ResolveName {
5
6     public static void main (String[] args) {
7         try {
8         InetAddress[] addrs = InetAddress.getAllByName("yandex.ru");
9           for (InetAddress addr : addrs) {
10            System.out.println(addr);
11          }
12        } catch (UnknownHostException ex) {
13          System.out.println("Couldn't resolve yandex.ru");
14        }
15    }
16}
```

Здесь после получения массива адресов в 8 строке листинга в 9–11 строках выводятся, соответствующие DNS имени «yandex.ru» IP-адреса. Вывод может выглядеть следующим образом:

```
D:\repos\java\InetAddress\ex2>C:\jdk\bin\java.exe ResolveName
yandex.ru/5.255.255.77
yandex.ru/77.88.55.88
yandex.ru/5.255.255.70
yandex.ru/77.88.55.60
```

Класс NetworkInterface

Класс *NetworkInterface* из пакета *java.net* предоставляет набор методов для получения информации о проводных и беспроводных интерфейсах сетевых устройств. С его помощью можно получить также имена сетевых интерфейсов компьютера, для выбора необходимого, особенно в случае одновременного подключения к нескольким сетям. Экземпляр класса *NetworkInterface* может представлять, как физическое устройство (например, сетевую карту компьютера), так и программный объект (например, адаптер замыкания на себя или loopback-адаптер) с назначенными ему IP-адресами.

NetworkInterface не имеет публичных конструкторов. Для получения экземпляра этого класса используются три статических метода:

static NetworkInterface getByInetAddress(InetAddress addr)

static NetworkInterface getByName(String name)

static Enumeration<NetworkInterfaces> getNetworkInterfaces()

Первый из статических методов используется, если известен IP-адрес, назначенный интерфейсу устройства. Второй – если известно назначенное интерфейсу имя. Третий метод используется для перечисления всех доступных интерфейсов устройства.

После получения экземпляра класса *NetworkInterface* можно получить связанные с ним IP-адреса при помощи метода:

Enumeration <InetAddress> getInetAddresses()

Такая информация может быть необходима при запуске сервера для вывода IP-адресов, на которые могут в дальнейшем подключаться клиенты.

Помимо указанных выше, класс содержит большое количество других полезных методов. Например, *getHardwareAddress* позволяет получить массив байтов, содержащий аппаратный адрес интерфейса (например, MAC адрес сетевого адаптера). Метод *getMTU* позволяет узнать значение Maximum Transmission Unit для этого интерфейса.

В случае возникновения ошибок, методы класса *NetworkInterface* создают исключение *SocketException*.

Более подробную информацию по методам класса можно получить из официальной документации, доступной по адресу: *https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/net/NetworkInterface.html*

Пример использования класса NetworkInterface

Рассмотрим задачу перечисления интерфейсов компьютера и IP-адресов, которые настроены на этих интерфейсах (смотри листинг 1.3).

Листинг 1.3

```
1 import java.net.NetworkInterface;
2 import java.net.InetAddress;
3 import java.net.SocketException;
4 import java.util.Enumeration;
5
6 public class InterfaceEnum {
7
8     public static void main(String[] args) throws SocketException {
9        Enumeration<NetworkInterface> ifaces = NetworkInterface.get-
NetworkInterfaces();
10      if (ifaces == null) {
11        System.out.println("No interfaces found");
12      } else {
13        while (ifaces.hasMoreElements()) {
14          NetworkInterface iface = ifaces.nextElement();
15           System.out.println(iface.getName()+" "+iface.getDisplay-
Name()+":");
16         Enumeration<InetAddress> addrs = iface.getInetAddresses();
17          if(!addrs.hasMoreElements()){
18            System.out.println("\tNo addresses in list");
19          }
```

```
20          while(addrs.hasMoreElements()) {
21            InetAddress addr = addrs.nextElement();
22           System.out.println("\taddress: "+addr.getHostAddress());
23          }
24        }
25      }
26   }
27}
```

В этом листинге вначале перечисляются все сетевые интерфейсы методом *getNetworkInterfaces* (строка 9 листинга). Затем выполняется проверка на возможное отсутствие сетевых интерфейсов (строки 10–12). После этого для каждого интерфейса в цикле печатается его имя в операционной системе и его полное отображаемое имя (строка 15). Затем методом *getInetAddresses* перечисляются все адреса для выбранного интерфейса (строка 16), выполняется проверка на отсутствие сконфигурированных адресов (строки 17–19), после чего в строках 20–23 печатаются все адреса для этого интерфейса (IPv4 и IPv6).

TCP сокеты. Класс Socket

Прежде чем рассмотреть применение класса Socket, следует отметить, что клиентская и серверная стороны взаимодействия по протоколу TCP достаточно сильно различаются. Для их реализации в составе библиотеки языка Java существуют два отдельных класса – *Socket* и *ServerSocket*. Архитектура клиентского приложения предусматривает использование класса *Socket* для установки *соединения (connection)* с сервером. Соединение в этом случае представляет собой двусторонний канал взаимодействия между клиентом и сервером (*full-duplex* в терминологии компьютерных сетей). Перед использованием TCP соединения для обмена данными оно должно быть установлено: TCP клиент должен отправить запрос на установку соединения TCP серверу, который должен иметь открытый сокет в режиме прослушивания (listening) входящих подключений и создавать новый экземпляр класса *Socket* для каждого входящего запроса на подключение от клиента. Таким образом, TCP клиент использует один тип сокетов – *Socket*, а TCP сервер использует экземпляры двух классов: *ServerSocket* и *Socket*.

Обычный TCP клиент использует три основных этапа работы:

- Создание экземпляра класса *Socket* (при этом конструктор этого класса устанавливает подключение к TCP серверу на заданный порт);
- Обмен данными с сервером используя потоки ввода-вывода (подключенный сокет предоставляет *InputStream* и *OutputStream* для сетевого взаимодействия);
- Закрытие соединения с сервером (для экземпляра *Socket* вызывается метод *close*).

Для создания экземпляра класса Socket предусмотрено несколько конструкторов:

Socket()
Socket(InetAddress remoteAddr, int remotePort)
Socket(String remoteHost, int remotePort)
Socket(InetAddress remoteAddr, int remotePort, InetAddress localAddr, int localPort)
Socket(String remoteHost, int remotePort, InetAddress localAddr, int localPort)

Первый конструктор создает неподключенные сокет, который необходимо до обмена данными подключить вызовом одного из перегруженных методов *connect*:

void connect(SocketAddress destination)
void connect(SocketAddress destination, int timeout)

Параметр destination в этих вызовах представляет собой абстрактный класс *SocketAddress* – некий общий адрес для сокетов, который в стеке TCP/IP реализуется его подклассом *InetSocketAddress*. Класс InetSocketAddress в свою очередь инкапсулирует *InetAddress* и номер порта.

Следующие четыре конструктора создают TCP сокет и выполняют соединение с удаленным хостом на указанный порт до возврата из конструктора. Для первых двух из них не указываются локальный адрес (определяющий сетевой интерфейс компьютера для подключения) и локальный порт. В этом случае на локальном компьютере выбирается локальный адрес по умолчанию и некоторый из незанятых портов. Два других конструктора предусматривают выбор определенного локального интерфейса для организации соединения и позволяют указать определенный локальный порт для подключения. Для параметра *remoteHost* типа *String* возможны такие же значения, как и для методов создания экземпляра *InetAddress*: DNS имя удаленного хоста или строковое представление IP адреса.

Для реализации обмена данными необходимо получить один или оба потока ввода-вывода для подключенного сокета:

InputStream getInputStream()
OutputStream getOutputStream()

Это стандартные потоки ввода-вывода Java, которые могут быть обернуты в буферизирующие и форматирующие классы-оболочки, как и другие объекты потоков ввода-вывода.

Для закрытия сокета и связанных с ним потоков ввода-вывода доступны методы:

void close()
void shutdownInput()
void shutdownOutput()

Метод *close* закрывает сокет и связанные потоки ввода-вывода, предотвращая дальнейшие операции с сокетом. Метод *shutdownInput* закрывает входной поток TCP сокета, при этом любые непрочитанные входящие данные отбрасываются, и попытка чтения из сокета после этого приведет к исключению. Метод *shutdownOutput* выполняет закрытие выходного потока TCP сокета, однако данные, которые были уже записаны в выходной поток сокета доставляются удаленной стороне.

Для получения информации о соединении могут быть использованы следующие методы класса *Socket*:

InetAddress getInetAddress()
int getPort()
InetAddress getLocalAddress()
int getLocalPort()
SocketAddress getRemoteSocketAddress()
SocketAddress getLocalSocketAddress()

Эти методы позволяют получить для TCP соединения информацию о локальной и удаленной стороне соединения: IP-адреса участников соединения (*getInetAddress* и *getLocalAddress*), номера портов (*getPort* и *getLocalPort*) или эти же данные упакованные в подкласс *InetSocketAddress* абстрактного класса *SocketAddress*. Более подробно о классе *InetSocketAddress* будет рассказано ниже, в пункте, посвященном классу *ServerSocket*.

Кроме этих данных класс *Socket* имеет большое количество других атрибутов, часто называемых *опциями сокета (socket options)*. Более подробную информацию о них можно посмотреть в официальной документации компании Oracle: *https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/net/Socket.html*

Пример использования класса Socket

Рассмотрим пример использования класса Socket (листинг 1.4) – простейший TCP-клиент службы Echo. Служба Echo относится к простым сервисам TCP, используется для тестирования сетевого программного обеспечения и описывается в RFC 862 (смотри https://www.rfc-editor.org/rfc/rfc862). По умолчанию Echo сервер слушает 7 порт TCP и после установления соединения с клиентом получает от него данные и возвращает назад, повторяя это до тех пор, пока клиент не закроет соединение. Соответственно, задачей клиента является установить соединение с сервером на заданный порт (7 по умолчанию), передать серверу тестовую строку, получить ее назад и вывести на экран.

Листинг 1.4

```
1 import java.net.Socket;
2 import java.net.SocketException;
3 import java.io.IOException;
```

```
4 import java.io.InputStream;
5 import java.io.OutputStream;
6
7 public class EchoTCPclient {
8
9     public static void main(String[] args) throws IOException {
10      // Проверяем корректность командной строки
11      if ((args.length <2) || (args.length>3))
12       throw new IllegalArgumentException("Command line parameters:
<Server> <Word> [Port]");
13     String server = args[0]; // IP или имя сервера для подключения
14      byte[] data = args[1].getBytes(); // данные для отправки
15        // если аргумента 3, то args[2] – порт, иначе порт 7 по
умолчанию
16      int serverPort = (args.length == 3)?Integer.parseInt(args[2])
: 7;
17      Socket sock = new Socket(server, serverPort);
18      System.out.println("Connected to server. Sending echo");
19      InputStream in = sock.getInputStream();
20      OutputStream out = sock.getOutputStream();
21      out.write(data); // отправка данных серверу
22      // Получение данных от сервера назад
23      int totalBytesRecv = 0;
24      int bytesRecv;
25      while(totalBytesRecv < data.length) {
26        if ((bytesRecv = in.read(data, totalBytesRecv, data.length-
totalBytesRecv)) == -1)
27          throw new SocketException("Connection terminated");
28        totalBytesRecv += bytesRecv;
29      }
30      System.out.println("Echo received: "+new String(data));
31      sock.close();
32    }
33}
```

В приведенном примере предполагается запуск программы с параметрами командной строки, задающими адрес или имя сервера для подключения, текст для передачи серверу Echo и необязательный номер порта сервера. После разбора входных параметров в строках 11–16 программы выполняется создание экземпляра класса *Socket* с одновременным подключением к удаленному хосту (строка 17). Затем у сокета забираются потоки ввода-вывода (строки 19–20), после чего байтовый массив со входной строкой отправляется серверу методом *write*. В последующих строках (23–29) выполняется приём от сервера возвращенной строки текста. Так как объем ответа известен заранее, то это сводится к циклическому повторению операции чтения из потока в массив *data*, пока не будет получено необходимое количество байт (метод *read* получает вторым параметром смещение в массиве байтов *data* куда должен записываться первый из получаемых байтов, а третьим параметром – максимальное количество байтов, которые можно

записать в массив). Если метод вернет -1, это будет означать разрыв соединения на другой стороне. Такая техника получения ответа объясняется тем, что TCP является потоковым протоколом и не сохраняет границы сообщений, используемых методами *read()* и *write()*: если мы отправляем серверу данные одним вызовом *write*, сервер может прочитать их несколькими порциями. Аналогично, если сервер даже получит данные одной порцией, то ответ сервера может быть разбит протоколом TCP на несколько частей. Одной из наиболее общих ошибок начинающих программистов является предположение, что данные, отправленные одним вызовом *write()* всегда будут получены за один вызов *read()*.

После печати ответа на экран (строка 30) выполняется закрытие соединения с сервером вызовом метода *close*.

TCP сокеты. Класс ServerSocket

В отличие от клиента, TCP сервер создает экземпляр класса *ServerSocket* указывая адрес локального сетевого интерфейса и локальный порт. Этот порт переводится сервером в режим прослушивания входящих подключений от клиентов. Для каждого подключающегося клиента повторяются следующие действия:

- Вызывается метод *accept()* для получения очередного клиентского подключения. Метод *accept()* блокируется до тех пор пока клиент не сделает попытку подключения, после чего создает экземпляр класса *Socket*, через который будет проводится обмен данными с подключившимся клиентом и возвращает его в качестве результата.
- Реализуется обмен данными с клиентом через потоки *InputStream* и *OutputStream*, полученные от *Socket*, в соответствии с протоколом.
- Когда обмен данными завершается, клиент, сервер или они оба вызывают метод *close()* класса *Socket* и закрывают соединение. После этого сервер возвращается к первому шагу.

Для создания экземпляра класса *ServerSocket* определены следующие конструкторы:

ServerSocket()
ServerSocket(int localPort)
ServerSocket(int localPort, int queueLimit)
ServerSocket(int localPort, int queueLimit, InetAddress localAddr)

Первый из конструкторов создает не связанный ни с одним локальным портом серверный сокет. Для использования такого сокета необходимо предварительно вызвать метод *bind()* для привязки конечной точки TCP к локальному порту. Следующие три конструктора используют параметр *localPort* для указания прослушиваемого сервером порта. Как было указано ранее, номер порта может принимать значения от 0 до 65535, значение 0 указывает на то, что операционная система должна выбрать неиспользуемый номер

порта сама. Параметр *queueLimit* задает размер очереди входящих подключений из которой *ServerSocket* выбирает следующее вызовом метода *accept*. В случае если очередь заполнена, клиенту будет отказано в подключении. При наличии адресного параметра *localAddr* он должен указывать на один из сетевых интерфейсов компьютера на котором сервер будет ожидать входящих запросов на подключение. Это может быть использовано на компьютерах с несколькими сетевыми интерфейсами.

Основными сетевыми операциями для *ServerSocket* являются:

void bind(SocketAddress endpoint)
void bind(SocketAddress endpoint, int queueLimit)
Socket accept()
void close()

Метод *bind* используется для связывания сокета с локальным интерфейсом и портом. *ServerSocket* может быть привязан только к одному порту. В случае, если указанный порт используется или экземпляр *ServerSocket* уже был связан с другим портом генерируется исключение *IOException*.

Метод *accept* возвращает подключенный экземпляр класса *Socket* для нового входящего соединения к серверному сокету. Если нет устанавливаемых соединений, то вызов *accept* блокируется пока не появится устанавливаемое соединение или не истечет таймаут.

Метод *close* закрывает сокет. В результате входящие соединения от клиентов будут отклонены.

Для получения информации о соединении могут быть использованы следующие методы:

InetAddress getInetAddress()
int getLocalPort()
SocketAddress getLocalSocketAddress()

Эти методы возвращают локальный адрес (адрес сетевого интерфейса на котором выполняется прослушивание порта), локальный порт или объект *InetSocketAddress* – подкласса *SocketAddress*, содержащий ту же информацию.

Обратите внимание, в отличие от класса *Socket*, класс *ServerSocket* не имеет ассоциированных с ним потоков ввода-вывода.

Помимо рассмотренных методов класс *ServerSocket* содержит большое количество методов для работы с атрибутами сокета (т.н. опциями сокета). Подробнее об этих методах можно узнать из официальной документации компании Oracle, доступной по адресу: *https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/net/ServerSocket.html*

Пример использования класса ServerSocket

В качестве примера использования класса *ServerSocket* рассмотрим TCP сервер для службы Echo – простого сервиса TCP, используемого для тестирования сетевого программного обеспечения и описываемого

в RFC 862 (смотри https://www.rfc-editor.org/rfc/rfc862). Echo сервер слушает указанный порт (по умолчанию 7) TCP, после соединения с клиентом получает от него некоторые данные и возвращает их назад, повторяя это до тех пор, пока клиент не закроет соединение (смотри Листинг 1.5).

Листинг 1.5

```
1 import java.net.Socket;
2 import java.net.ServerSocket;
3 import java.net.InetAddress;
4 import java.net.SocketAddress;
5 import java.io.IOException;
6 import java.io.InputStream;
7 import java.io.OutputStream;
8
9 public class EchoTCPserver {
10
11    private static final int BUFSIZE = 256; // размер буфера приёма
12
13    public static void main(String[] args) throws IOException {
14
15     if (args.length != 1) // Проверка корректности командной строки
16       throw new IllegalArgumentException("Command line parameters:
<Port>");
17
18      int serverPort = Integer.parseInt(args[0]);
19
20      // create server socket
21      ServerSocket serverSock = new ServerSocket(serverPort);
22
23      int recvMsgSize;
24      byte[] receiveBuf = new byte[BUFSIZE]; // receive buffer
25
26      while(true) {
27        Socket clientSock = serverSock.accept();
28
29         SocketAddress clientAddress = clientSock.getRemoteSocket-
Address();
30        System.out.println("Accepted client: "+clientAddress);
31
32        InputStream in = clientSock.getInputStream();
33        OutputStream out = clientSock.getOutputStream();
34
35        // receive data until client closes connection (-1)
36        while ((recvMsgSize = in.read(receiveBuf)) != -1) {
37          out.write(receiveBuf, 0, recvMsgSize);
38        }
39        clientSock.close();
40      }
41     }
42}
```

В начале программы (строки 15–18) выполняется обработка некорректного числа аргументов (программа должна получать один параметр – номер локального порта сервера). После этого в 21 строке создается экземпляр класса *ServerSocket* и указанный порт открывается в режиме прослушивания (это можно проверить при помощи команды netstat, изученной в курсе компьютерных сетей). После выделения буфера для приёма данных от клиента (строка 24) запускается бесконечный цикл, содержащий действия по обработке данных клиента. Вначале при помощи метода *accept* принимается запрос на соединение и создается клиентский сокет для взаимодействия (строка 27). Затем в 29–30 строках протоколируется подключение клиента: метод *getRemoteSocketAddress* возвращает экземпляр класса *InetSocketAddress*, который затем выводится на экран в формате /адрес_хоста:номер_порта. После этого для созданного для взаимодействия клиентского сокета получаются потоки ввода-вывода и запускается цикл считывания данных от клиента и отправки эха обратно (строки 36-38). Чтение продолжается до тех пор, пока *read* не вернет -1, что означает, что соединение было закрыто клиентом. В строке 39 вызывается метод *close* для клиентского сокета, который освобождает связанные с ним ресурсы. Строки 41–42 при таком построении программы никогда не будут достигнуты.

Результаты тестирования созданных программ представлены на рисунке.

Рисунок – Результаты запуска TCP-сервера и TCP-клиента

ЗАДАНИЯ И КОММЕНТАРИИ К ИХ ВЫПОЛНЕНИЮ

Задание 1. Наберите исходный код приведенных выше примеров и проверьте их работоспособность.

Задание 2. Разработка TCP-клиента и TCP-сервера. По условию Вашего варианта разработайте клиентское и серверное приложения для указанного сетевого протокола или службы.

Вариант 1. Протокол QotD (Quote of the Day) (RFC 865).

Вариант 2. Протокол Active Users (RFC 866).

Вариант 3. Протокол DayTime (RFC 867).

Вариант 4. Протокол Time (RFC 868).

Вариант 5. Dictionary Server Protocol (RFC 2229). Подмножество DEFINE/MATCH.

Вариант 6. Протокол PWDGEN (RFC 972).

Вариант 7. Протокол CharGen (RFC 864).

Вариант 8. Протокол Message Send (RFC 1312).

Вариант 9. Протокол Finger (RFC 742).

Вариант 10. RLogin (RFC 1282).

Вариант 11. Протокол WhoIs (RFC 3912).

Вариант 12. Протокол Telnet (RFC 854).

Указания.

- Разработка приложений ведется в группах по два человека: один из студентов разрабатывает клиентскую часть, второй – серверную.
- Каждая группа выполняет 2 варианта: для первого участника и для второго. При этом, части приложения между вариантами меняются: если для первого варианта Вы разрабатывали серверную часть, то для второго должны разрабатывать клиентскую и наоборот.
- Тестирование и сдача программ предполагает их работу по сети – между различными компьютерами.

Задание 3. Согласно указанному варианту разработайте серверное приложение.

Вариант 1. Разработайте приложение – простейший HTTP-сервер: приложение должно обрабатывать GET-запросы от клиента (браузера поль-

зователя) и возвращать ответ с нужным кодом (200, 404 и т.д.) и содержимым (файлы из заранее заданного каталога). При работе сервера входящие запросы и заголовки ответов на них должны протоколироваться в файл.

Использовать для разработки приложения специализированные компоненты или классы для реализации функции протокола HTTP - запрещено.

Вариант 2. Разработайте приложение – простейший почтовый POP3-сервер: приложение должно корректно обрабатывать команды: USER, PASS, RETR, DELE, LIST, STAT, TOP, QUIT. Приложение должно корректно информировать клиента о статусе выполнения команды (+OK, -ERR). Сообщения пользователей должны храниться в заранее заданном каталоге и соответствовать принятой структуре сообщения электронной почты.

Пояснения, примеры и список ссылок на стандарты, например, https://ru.wikipedia.org/wiki/POP3

Вариант 3. Разработайте сетевое приложение – простейший почтовый SMTP-сервер. Приложение должно реализовать поддержку команд MAIL FROM, RCPT TO, QUIT и DATA. На все получаемые команды сервер должен возвращать соответствующие коды ответа. По возможности сервер должен реализовать ограничение доступа к серверу по местоположению. Приложение может реализовать функции SMTP-транслятора: вместо отправки сообщений почтовому серверу получателя передавать сообщение следующему почтовому серверу.

Вариант 4. Разработайте сетевое приложение – сервер для игры «Крестики-нолики». Сервер должен предусматривать подключение 2 клиентов. Если подключен только 1 клиент, то он должен ожидать подключения второго пользователя. После подключения клиентов сервер должен контролировать очередность ходов клиентов и правильность каждого хода (запрещать ставить «крестик» или «нолик» в занятую клетку и т.д.). Для тестирования приложения можно использовать сетевой терминал PuTTY.

Вариант 5. Разработайте сетевое приложение – простейший чат-сервер. Приложение должно прослушивать определенный порт, принимать от клиентов подключения и реализовать для каждого подключенного клиента набор команд, включающий: задание имени пользователя, получение списка пользователей на сервере, отправку сообщения всем пользователям, отправку сообщения определенному пользователю (т.н. приватное сообщение), установку темы чата, получение темы чата.

Пример команд и их формат можно получить в документации по IRC, например, по ссылкам с https://ru.wikipedia.org/wiki/IRC.

Вариант 6. Разработайте сетевое приложение – простейший сервер сетевых викторин. Идея проекта: сервер содержит базу вопросов и ответов.

По команде пользователя запускается очередной тур викторины. Сервер выбирает случайным образом вопрос из базы и задаёт его всем подключенным игрокам. Первый, кто даёт верный ответ считается победителем и ему начисляются призовые очки. Приложение должно прослушивать определенный порт, принимать подключения от клиентов и реализовать для каждого подключенного клиента набор команд, включающий: задание имени пользователя, запуск/перезапуск викторины (если вопросов было задано не менее некоторого порогового числа, например, 10), предъявление вопроса пользователям, приём ответа на вопрос от пользователя, вывод рейтинга игроков в зависимости от количества правильных ответов.

Вариант 7. Разработайте сетевое приложение – простейший сервер информационного бота. Приложение должно реализовать следующие команды: вывод текущей погоды, вывод текущего времени и даты, вывод текущего курса валют, вывод текущей телепрограммы для указанного пользователем телеканала. В ответ на запрос с соответствующей командой должен отправляться ответ с требуемой информацией. Для простоты выводимая информация может храниться в файле на сервере и считываться по мере надобности, а не собираться в Интернете на профильных сайтах.

Вариант 8. Разработайте сетевое приложение – простейший сервер для текстовой ролевой компьютерной игры (многопользовательские версии таких приложений называются MUD). Приложение должно прослушивать определенный порт, принимать подключение от пользователя и реализовать возможность одновременного присутствия на некоторой виртуальной карте нескольких персонажей. Для пользователей должно быть доступно не менее 10 команд (например, пройти вперед, повернуть влево, повернуть вправо, сказать что-либо и др.).

Более подробное описание, примеры сообщений и ссылки на существующие разработки можно найти, например, на http://ru.wikipedia.org/wiki/MUD

Вариант 9. Разработайте сетевое приложение – простейший сервер для пошаговой игры жанра «файтинг». Сервер должен предусматривать подключение 2 клиентов. Если подключен только 1 клиент, то он должен ожидать подключения второго пользователя. После подключения второго пользователя, сервер предоставляет каждому из игроков возможность ввода команд вида: «хук левой», «блок левой», «джеб правой», «уклон вправо» и т.д. в течение нескольких раундов. После получения команды от обоих игроков, сервер сравнивает их действия и решает исход очередного раунда схватки. В зависимости от результата игрокам начисляются очки (уменьшаются заранее заданные). Игра идёт до получения одним из игроков требуемого количества очков, после чего объявляется победитель и сервер переходит в режим приёма подключений от следующих игроков.

Вариант 10. Разработайте сетевое приложение – простейший сервер Telnet. Приложение должно прослушивать стандартный для Telnet порт, принимать входящие подключения, в рамках подключения принимать команды и исполнять их на локальном компьютере. Вывод исполняемых команд должен отправляться подключенному клиенту. Предусмотрите простейшую аутентификацию пользователя (например, имя пользователя и пароль в открытом виде).

Краткое описание на русском языке: https://ru.wikipedia.org/wiki/Telnet

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Задание 1. Добавьте в первый пример для класса *InetAddress* метод, который инициализирует экземпляр класса IP-адресом. Определите и вызовите перед выводом ответа метод для разрешения IP-адреса в DNS-имя через обратную зону DNS. Почему выведенные имена могут не совпасть? Каким из методов *InetAddress* можно проверить, что речь идет об одном и том же хосте?

Задание 2. Дополните программу из листинга 1.3 выводом MAC-адресов каждого из сетевых интерфейсов компьютера (в качестве разделителей для частей MAC-адреса можно использовать и «:» и «-»).

Задание 3. Дополните программу из листинга 1.5 возможностью принимать несколько одновременных подключений клиентов при помощи потоков исполнения.

Задание 4. Модифицируйте серверное приложение из листинга 1.5 таким образом, чтобы обработка клиентов выполнялась отдельными процессами, которые запускает для каждого из подключающихся клиентов программа-менеджер.

СПИСОК ЛИТЕРАТУРЫ

1. Calvert, Kenneth L., TCP/IP Sockets in Java: Practical Guide for Programmers, Second Edition / Kenneth L. Calvert, Michael J. Donahoo. – Morgan Kaufmann Publishers., 2008.

2. Elliotte Rusty Harold, Java Network Programming, Fourth Edition. – O'Reilly, 2014.

3. Jan Graba, An Introduction to Network Programming with Java, 3rd ed. – Springer., 2013.

4. java.net (Java SE 21 & JDK 21) [электронный ресурс] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/net/package-summary.html. Copyright © 1993, 2023, Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065 USA.

5. Richard M Reese, Learning Network Programming with Java. – Packt Publishing., 2015.