Л Е К Ц И Я №4

Мрежово програмиране с Java

I. ВЪВЕДЕНИЕ

Едно от основните предимства на глобалната мрежа е масовата достъпност до множество услуги като електронна поща, обмен на файлове, достъп до Web страници, електронна търговия, банкиране и много др. Всички услуги, независимо от тяхната сложност, се реализират благодарение на определени мрежови протоколи. Програмният достъп до различните протоколи, а следователно и до съответните услугите, се реализира чрез мрежовите библиотеки, които съответният програмен език предоставя. Колекцията от мрежови нива (дефинирани от TCP/IP мрежовия модел), до които имаме програмен достъп, е прието да се нарича *стек*. Когато един език за програмиране предоставя функции за работа в мрежова среда на всички нива (приложно, транспортно, мрежово и канално) се казва, че поддържа пълен TCP/IP стек. На практика най-голям практически интерес представлява достъпът до протоколите от приложно (HTTP, SMTP, POP3, FTP, Telnet) и транспортно (TCP и UDP) ниво. Огромната част от мрежовите услуги се базират на TCP или UDP комуникации. Най-често програмните езици, които позволяват работа в мрежова среда, поддържат TCP, UDP и протоколи от приложно ниво, които се използват с цел достъп до най-използваните мрежови услуги: HTTP, SMTP, POP3, IMAP4, DNS и SNMP.

По-голямата част от мрежовите услуги използват комуникационния модел "клиент-сървър". От гледна точка на комуникация в мрежова среда е без значение кой е клиента и кой е сървъра, но този модел помага за по-лесното разбиране на функционирането на мрежовите услуги. От програмна гледна точка най-често инициаторът на обмена е клиентът. Клиентите изпращат заявки, които идентифицират каква информация се желае. Сървърът отговаря на заявките, като статично или динамично формира желаната от клиента информация. Форматът на обмен е най-често *поток от байтове*. Когато клиентът е инициатор на обмена се говори за **pull** комуникации. В последните няколко години все по-голямо приложение, основно в областта на мобилните комуникации, намират тъй нар. **push** комуникации. При тях сървърът може да е и инициатор на обмена. Най-често това решение се използва с цел доставяне на информация за която клиентът се е абонирал и промяната на която е важна за него. Вместо клиентът да генерира през определен интервал от време заявки, сървърът уведомява клиента при всяка промяна на информацията, например: промяна на пътната обстановка, промяна в работното състояние на технологичен процес, временно блокиране на услуга и др.

II. TCP КОМУНИКАЦИИ

Основната част от мрежовите услуги използват транспортен протокол TCP. Причина за това е, че този протокол *гарантира* доставянето на информацията, предавана между два хоста. Хостът е компютър или мрежов хардуер, който еднозначно може да се идентифицира в мрежова среда. При протокол UDP *не се гарантира* доставяне на информацията. В този случай, програмно се проверява дали информацията е била доставена. Поради тази причина протокол UDP се използва при услуги, които работят в

интранет среда или при които загубата на данни не е фатално, например доставяне на потоково видео и аудио съдържание.

При комуникация в Интернет среда идентификаторът на хардуера е число, наречено **IP адрес**. Този адрес е 32-битов или 128-битов, в зависимост от версията на протокол IP с която се работи (IP v.4 или IP v.6). За да реализираме комуникация между два хоста, IP адресите им не са достатъчни. Причината за това е, че на един ход може да бъдат инсталирани както много на брой клиенти, така и много на брой сървъри. Разпознаването на клиентите или сървърите от един и същи хост се реализира от друго число (16 бита) наречено **порт**. Следователно, еднозначната идентификация на услуги в Интернет се реализира чрез комбинацията между IP адрес и номер на порт. Тази комбинация се нарича **сокет**. Програмният език Java, чрез своите мрежови библиотеки, позволява създаване на всички видове TCP и UDP сокети.

В мрежовото програмиране се работи с два вида сокети (ще разгледаме само TCP сокети):

- **Работни** сокети. Това са сокетите, получени от клас Socket. Те имат за задача да идентифицират участниците във всеки край на комуникационния канал. От страна на клиента се получават чрез създаване на обект от клас Socket, а от сървърната страна се генерират от метод accept().
- **Сървърни** сокети. Както подсказва името им, те се използват само от сървърната страна. Те нямат за задача да формират комуникационен канал с клиента, а само *подслушват* указан порт за заявки на клиенти. За целта се използва метод accept() от клас ServerSocket. Методът чака до получаване на заявка от клиент. В този момент метод accept() връща обект-сокет чрез който клиента ще комуникира със сървъра. Този сокет е от клас Socket и той може да се използва за изграждане на комуникационен канал.

Всеки сървър обработва в даден момент определен брой клиенти. За всеки клиент се създава програмно отделен комуникационен канал. Тези канали са програмни обекти, които трябва да са изолирани един от друг с цел сигурност и да могат да работят паралелно. Това е възможно само ако: 1) сървърът има няколко процесора или ако процесорът е многоядрен; и 2) ОС „вижда" програмния код на комуникационния канал като отделен процес (програма, която вече е стартирана и е заредена в паметта). Условие 2 лесно се изпълнява, ако програмният код за комуникация между двата хоста е написан като *програмна нишка*.

III. ПРОГРАМНИ НИШКИ

Съвременните ОС предоставят възможността за псевдо-паралелно изпълнение не само на ниво процеси, но и на ниво програмни модули в рамките на един процес. В този случай се казва, че ОС поддържа работа с множество *програмни нишки*.

> ***Нишката*** (*thread*) е програмен код (функция), част от една програма, с ясно дефинирани начало и край. Този код се обработва от страна на ОС като отделен процес с тази разлика, че за него не се заделят системни ресурси, освен стек. Нишките използват ресурсите на програмата в която са дефинирани. За всяка нишка може да се зададе различен приоритет на изпълнение от страна на процесора.

Всяко приложение има поне един път на своето изпълнение – така наречената *главна нишка*. Нейната задача, от гледна точка на многонишковото програмиране, е да създаде, синхронизира и унищожи останалите нишки.

Основните предимства на многонишковото програмиране са:

- Възможност за псевдо-паралелно (едно-процесорна платформа) или паралелно (много-процесорна или многоядрена платформа) изпълнение на програмни модули от едно приложение.
- Реализация на изчисления и комуникации във фонов режим, без да се налага блокиране на интерфейса с потребителя. Нишката, отговаряща за интерфейса с потребителя, трябва да има по-висок приоритет от нишките реализиращи изчислителни процедури или мрежова комуникация!

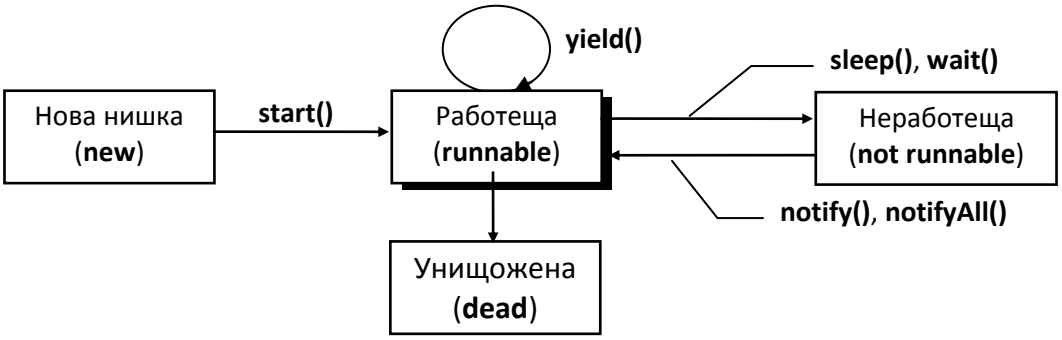
Програмният език Java предоставя добри възможности за поддръжка на програмни нишки. Планировчикът на задачите на Java използва проста стратегия за превключване на нишките - fixed-priority scheduling. Една нишка, която се изпълнява, може да бъде прекъсната, ако:

- Трябва да се активира нишка с по-висок приоритет.
- Нишката доброволно преотстъпва процесора на друга нишка.
- Нишката се унищожи.
- Изтекъл е интервалът от време за работа на нишката, който е заделен от планировчика на задачите. След този интервал, планировчика активира друга нишка.

Основният клас, който се използва при работа с нишки в Java е **Thread** от библиотека java.lang. Той предоставя методи за стартиране, спиране, промяна на приоритета и унищожаване на нишки.

3.1. Жизнен цикъл на нишките

Жизненият цикъл дефинира основните *състояния* на една нишка от момента на създаването, до момента на унищожаването ѝ. Жизненият цикъл на Java нишките е показан на фиг. 4.1. Показани са и методите, които привеждат нишките от едно състояние в друго. Текущото състояние на всяка нишка може да се получи чрез метод getState() от клас Thread.



Фиг. 4.1. Жизнен цикъл на нишките в Java

В клас Thread са дефинирани и 6 константи, чрез които числово се кодират възможните състояния на една нишка:

- Thread.NEW (нова нишка);
- Thread.RUNNABLE (работеща);
- Thread.TERMINATED (унищожена);

- Thread.TIME_WAITING (неработеща, изчаква интервал от време);
- Thread.WAITING (неработеща, изчаква събитие);
- Thread.BLOCKED (неработеща, блокирана).

3.2. Приоритет на нишките

Клас Thread дефинира други две константи чрез които може да се разбере най-ниското и най-високо ниво на приоритет, които дадена JVM поддържа: **MIN_PRIORITY** и **MAX_PRIORITY**. По всяко време може да получите нивото на приоритет на една нишка чрез метод getPriority() или да го промените чрез метод setPriority(). Може да зададете за ниво на приоритет всяко цяло число в интервала [MIN_PRIORITY, MAX_PRIORITY].

3.4. Създаване на нови нишки

В Java се предоставят множество начини за създаване на нова нишка, два от които се използват по-често:

- Създаване на обект от клас, наследил клас **Thread**. В тялото на класа трябва да се дефинира метод run(). Кодът от тялото на този метод е кодът на нишката.

```
class MyThread extends Thread {
    . . . . . .
    public void run() {
        // тяло на нишката
    }
}
```

- Реализация на интерфейс ***Runnable***. Аналогично и тук в тялото на класа се дефинира метод run().

```
class MyThread impliments Runnable {
    . . . . . .
    public void run() {
        // тяло на нишката
    }
}
```

Когато е необходима само една нишка, може да се използва интерфейс *Runnable*, в противен случай трябва да се използва наследяване на клас Thread. След създаване на обект от клас Thread състоянието на нишката е **new**. В този момент все още не са заделени ресурси за нишката. Следва примерно деклариране и създаване на обект-нишка:

```
1   Thread myThread1 = new Thread(this);
2   Thread myThread2 = new Thread(”thread2”);
```

При вариант 1 се използва конструкторът на клас Thread, който изисква един атрибут - референция към обекта с който се асоциира нишката (this = текущия обект). При вторият вариант като аргумент се предава името на нишката. То може да бъде получено по всяко време чрез метод getName(). Името може да се използва като идентификатор на обекти-нишки, получени от един и същи клас, наследник на Thread.

3.5. Стартиране на нишки

След като е създаден обект от клас Thread нишката може да се стартира чрез метод **start()**. Това е единственият метод, който може да изпълните при състояние „new” на нишката. При изпълнение на кой да е друг метод за работа с нишки се генерира изключение от клас IllegalThreadStateException. След успешно изпълнение на метод run нишката преминава в състояние „runnable” и започва да се обслужва от процесора.

```
myThread1.start();
```

3.6. Унищожаване на нишки

Съществуват няколко начина за унищожаване на една нишка. Ще използваме само един от тях, който винаги е безопасен и не води до блокиране на програмния код поради слаби познания на програмиста от гледна точка на thread-safe програмиране. Ако оставим метод run() да върне от управление, нишката *автоматично* се унищожава. За целта кодът, който реализира логиката на работа на нишката се поставя в тялото на while конструкция. При изход от while цикъла, метод run() завършва своята работа и нишката се унищожава от JVM. Следва пример:

```
public void run() {
    while(!exitFlag) {
        // код, реализиращ логиката на нишката
    }
}
```

Преди да се стартира нишката, на флаг exitFlag се задава стойност false. Когато желаем нишката да се самоунищожи, флагът трябва да приеме стойност true.

3.7. Привеждане на нишките в неработно състояние

Не трябва да се оставя една нишка постоянно да изисква обработка от страна на процесора, дори ако тя реализира изчисления, които трябва да се изпълнят максимално бързо. Това е логично - нишките с по-нисък приоритет ще бъдат блокирани, докато нишката, извършваща изчисленията, не върне от управление. За да се предотврати подобна ситуация е необходимо нишките да преминават в състояние „not runnable" за определен интервал от време. В Java това може да се реализира по няколко начина:

1. Вика се статичен метод **sleep()**

Методът изисква един атрибут от тип long – интервал от време в ms за който нишката ще бъде в състояние „not runnable", по-точно TIME_WAITING. При изпълнение на метода може да се генерира изключение от клас InterruptedException (нишката се прекъсва от друга нишка). Следва пример за използване на метод sleep(). В този случай, програмният код, който реализира логиката на работа на нишката, ще се активира през 100 ms.

```
public void run() {
    while(!exitFlag) {
        // код, реализиращ логиката на нишката
        try {
            sleep(100);
        } catch(InterruptedException ie) {}
    }
}
```

2. Вика се метод **wait()**

Ако нишката трябва да се активира повторно след възникване на някакво събитие, то метод sleep() не може да се използва. В този случай трябва да се използва метод wait() от клас Object. Методът има две възможни декларации: без аргумент или с един аргумент – времето в ms след което нишката се връща в работно състояние. На пръв поглед функционалността на втория вариант съвпада с тази на метод sleep(), но това не е така.

Връщането в работно състояние може да възникне в два случая: 1) генерирано е следеното събитие или 2) изтекъл е указания интервал от време. Изход от метод wait(), когато се следи събитие, се реализира чрез метод notify() или notifyAll() от клас Object. Методи wait() и notify() трябва да бъдат в тялото на try клауза.

3. Вика се метод **yield()**

В този случай нишката доброволно преотстъпва своето процесорно време на други нишки с еднакъв или по-нисък приоритет.

4. Вика се метод **suspend()**

При активиране на този метод нишката преминава в неработно състояние, докато се изпълни метод resume().

5. Чакане на входно-изходна операция или освобождаване на защитен ресурс

Трябва да се има предвид, че повечето входно-изходни операции блокират нишката в която са активирани. Нишката остава блокирана докато операцията не завърши. Друга честа причина за блокиране на нишка е когато тя прави опит за достъп до ресурси, защитени от друга нишка. Блокировката е в сила, докато ресурсите не се освободят.

IV. СИНХРОНИЗАЦИЯ НА ДОСТЪПА ДО СПОДЕЛЕНИ РЕСУРСИ

Често при използване на множество нишки се налага две или повече от тях да желаят достъп до едни и същи ресурси (методи и променливи). Може да ви се струва, че това не е проблем, но не е така. Планировчикът на задачите, превключва нишките по време, без да се интересува от логиката, която те изпълняват. Представете си, че две нишки използват даден метод. Ако едната го извика и по време на изпълнението му се прекъсне от другата нишка, тя също ще го извика. В този случай, ако методът манипулира променливи, първата нишка може да продължи работа с грешни данни, след като се активира повторно. За да се елиминира този потенциален проблем се реализира *заключване* на споделените ресурси. За целта в Java се използва ключовата дума **synchronized**. Ако една променлива е защитена, до нея в даден момент има достъп само една нишка, например:

```
synchronized(this) {
    // заключване на променлива counter
    counter++;
}
// отключване на променливата
```

Частта от кода, който е в тялото на synchronized, е защитен и се нарича *критична секция*. След ключовата дума synchronized в скоби се задава референцията към обекта за който е в сила заключването. В конкретният случай чрез ключова дума this е указан текущият обект. Ако една нишка направи опит за достъп до ресурси от синхронизиран блок, тя се привежда в блокирано състояние, докато ресурсите не се освободят. Не използвайте synchronized за променливи, тъй като това изисква много ресурси, включително и време. В този случай е по-добре да зададете ключова дума **volatile** за съответната променлива.

Могат да се заключават и цели методи. За целта преди името на метода се задава ключовата дума synchronized. Ако една нишка извика заключен метод от даден обект, то другите нишки нямат достъп до кой да е защитен метод от този обект. Отключването се реализира след като методът върне от управление. Трябва да се има предвид, че заключването на методи забавя тяхното викане и връщане от управление. Ако обаче трябва да се гарантира синхронизация на достъпа до споделени ресурси, то заключването

няма алтернатива. Друга особеност на Java е, че когато наследявате даден клас със синхронизирани методи, те не са синхронизирани в дъщерния клас. В някои случаи защитата на ресурси може да доведе до блокиране, тъй нар. “мъртва хватка” (deadlock). Този тип блокировка възниква когато няколко нишки в затворен цикъл се опитват да реализират достъп до един споделен ресурс. Някои Java компилатори предупреждават за възможни deadlock ситуации, но други - не.

Колкото с повече нишки се налага да се работи, толкова е по-голяма вероятността за проблеми със стабилността на програмния код. Синхронизирането на множество нишки е сложен процес, който се реализира от програмисти с опит. Поради тази причина, част от програмните езици имат специални класове и методи, които гарантират до голяма степен писането на thread-safe код.

В Java има специални класове и интерфейси за работа с множество нишки (thread pool). Те са от библиотека java.util.concurrent. Класове Executor и ThreadPoolExecutor позволяват работа с определен брой програмни нишки, които се синхронизират вътрешно. Имате възможност за създаване на thread pool с фиксиран брой нишки или да оставите кода от библиотеката да прецени колко нишки да отдели. Изборът на оптималния брой програмни нишки в thread pool е сложен процес, тъй като той зависи от множество фактори, например:

- Броя на ядрата на процесора.
- Време на работа на нишките.
- Време на изчакване.

Една проста формула за определене на броя на необходимите нишки е следната:

Number of threads = Number of Available Cores * (1 + Wait time / Service time)

V. СЪЗДАВАНЕ НА TCP КЛИНТ И СЪРВЪР

Както вече споменахме, методите чрез които се изграждат входно-изходни канали, блокират нишката от чието тяло се генерират. Блокировката е в сила докато методът не върне от управление. Това означава, че ако се използва еднонишково приложение и се реализира комуникация в мрежова среда, то интерфейсът с потребителя ще бъде блокиран при всяка входно-изходна операция. Изходът от тази ситуация е прост - методите, реализиращи мрежов обмен, трябва да бъдат викани от тялото на работна нишка. В зависимост от необходимият брой комуникационни канали трябва да изберем броя на нишките, които ще ги обслужват.

За да демонстрираме използването на множество нишки ще реализираме клиент и сървър, които си комуникират на ниво низове (заявки и отговори). Ще реализираме две конзолни приложения:

- TCP сървър, който подслушва указан порт и връща като отговор заявката на клиента.
- TCP клиент, който генерира заявки към сървъра под формата на низ. Край на комуникацията - ред със съдържание “--end--”.

Необходими разяснения

1. Библиотеките, които са необходими за реализацията на поставената задача, са java.net и java.io.

2. Идентификацията на участниците в комуникационния канал (клиентите и сървъра) се реализира на базата на сокети. Както вече знаем, те са програмни абстракции чрез които *еднозначно* се идентифицират участниците в един комуникационен канал. В конкретният случай номера на порта се задава като константа. За да се реализира комуникационен канал е необходимо комбинацията IP адрес и номер на порт (сокет) да е уникална.
3. За потребителите е по удобно да работят с мрежовото име на машините, а не с техните IP адреси. Java предоставя метод getByName() от клас InetAddress чрез който се преобразува мрежово име до IP адрес. Методът получава IP адреса на хоста чрез достъп до DNS сървър.
4. След като обектите-сокети са създадени, комуникацията е безпроблемна. Входящият канал се получава чрез метод **getInputStream()**, който създава обект от клас InputStream. Изходящият канал се получава чрез метод **getOutputStream()**, който създава обект от клас OutputStream. Комуникацията на ниво *поток от байтове* се конвертира до поток от символи чрез конвертиращите класове **InputStreamReader()** и **OutputStreamReader()**. При комуникация в мрежова среда входящите и изходящи канали трябва да се буферират. За целта ще използваме обекти от клас **BufferedReader** и **BufferedWriter**.

Една примерна реализация на сървъра е показана на Листинг 4.1.

```
1   import java.io.*;
2   import java.net.*;
3   public class Server {
4      final static int PORT = 8000;
5      public static void main(String[] args) {
6         ServerSocket s = new ServerSocket(PORT);
7         System.out.println(”Server has been started...”);
8         try {
9            while(true) {
10              Socket socket = s.accept();
11              System.out.println(”Incomming request:”);
12              try {
13                 new ServeClient(socket);
14              } catch(IOException ioe) {
15
16                 if(socket != null) socket.close();
17              }
18           }
19        } finally {
20           if (s != null) s.close();
21        }
22     }
23  }
```

```
24  class ServeClient extends Thread {
25     private PrintWriter out;
26     private BufferedReader in;
27     private Socket socket;

28     public ServeClient(Socket s) throws IOException {
29        socket = s;
30        in = new BufferedReader(new OnputStreamReader(
               socket.getInputStream() ));
31        out = new PrintWriter(new BufferedWriter(
               new OutputStreamWriter(
```

```
            socket.getOutputStream())), true);
32          start();
33      }

34      public void run() {
35          try {
36              while(true) {
37                  String inStr = in.readLine();
38                  if (inStr.equals("--end--")) break;
39                  System.out.println("Client: "+ inStr);
40                  out.println("Server: "+ inStr);
41              }
42              System.out.println("Clossing connection");
43          } catch(IOException ioe) {
44              System.out.println("Communication error");
45          } finally {
46              try {
47                  socket.close();
48              } catch(IOException ioe) {
49                  System.out.println("Can't close socket");
50              }
51          }
52      }
53  }
```

Листинг 4.1. Изходен код на сървъра

Следва кратко описание на изходния код по редове:

Редове 3-23 - Деклариране на клас Server;

Ред 6 - Създаване на сървърен сокет - обект s;

Ред 10 - Активиране на подслушването на указания порт (PORT);

Ред 13 - Създаване на нишка за обслужване на заявките на клиента. За целта се създава обект от клас ServeClient. На конструктура, като атрибут, се предава обекта-сокет, генериран от метод accept;

Редове 24-53 - Деклариране на клас ServeClient, който наследява клас Thread;

Ред 25 - Деклариране на изходящ канал (out) за комуникация с клиента;

Ред 26 - Деклариране на входящ канал (in) за комуникация с клиента;

Ред 27 - Буфериране на обекта-сокет, върнат от метод accept;

Редове 28-33 - Конструктор на клас ServeClient;

Ред 30 - Изграждане на входящия канал;

Ред 31 - Изграждане на изходящия канал. За да можем да използваме познатия ни метод println, за да предаваме данни към клиента, използваме обект от клас PrintWriter. В конкретният случай конструкторът му изисква два параметъра: обект от клас BufferedWriter и флаг с булева стойност, който указва при какво условие данните да се предадат към клиента. Ако стойността е true данните се предават веднага, дори буферът да не е запълнен;

Ред 32 - Стартиране на нишката, която ще обслужва клиента. За всеки клиент се стартира отделна нишка;

Редове 34-52 - Метод run - код на нишката;

Редове 36-41 - Цикъл за четене на данните от клиента и връщане на отговор. Комуникацията е на ниво ASCII ред. Комуникационният канал се унищожава, когато клиентът предаде ред със съдържание "--end--" Сървърът връща на клиента, това което той е предал (exo). Нишката се унищожава при изход от цикъла или изключение, тъй като това води и до изход от метод run.

На Листинг 4.2 е показана реализацията на клиента.

```
1   import java.io.*;
2   import java.net.*;
3   public class Client {
4       static final int PORT = 8000;
5       static final int NUM_OF_PINGS = 10;

6       public static void main(String[] args) {
7          Socket socket = null;
8          InetAddress ip = InetAddress.getByName(”localhost”);
9          System.out.println(”Client IP = ”+ ip);
10         try {
11             socket = new Socket(ip, PORT);
12             BufferedReader in = new BufferedReader(
                     new OnputStreamReader(socket.getInputStream()));
13             PrintWriter out = new PrintWriter(
                     new BufferedWriter(new OutputStreamWriter(
                     socket.getOutputStream())), true);
14             for(int i=0; i<NUM_OF_PINGS; i++) {
15                 out.println(”Ping” + i);
16                 String inStr = in.readLine();
17                 System.out.println(”Server: ”+inStr);
18             }
19             out.println(”--end--”);
20         } catch(IOExceprion ioe) {
21             if (socket == null)
22                 System.out.println(”Server not responding);
23             else
24                 System.out.println(”Communication error);
25         } finally {
26             if (socket != null) socket.close();
27         }
28      }

29  }
```

Листинг 4.2. Изходен код на клиента

Следва кратко описание на изходния код по редове:

Ред 5 - Константата NUM_OF_PINGS задава колко реда данни да се предадат към сървъра;
Ред 7 - Деклариране на обект-сокет (socket);
Ред 8 - Преобразуване на името на машината до Интернет адрес (ip);
Ред 11 - Създаване на обекта-сокет;
Ред 12 - Изграждане на входящия канал за четене от сървъра (in);
Ред 13 - Изграждане на изходящия канал за предаване към сървъра (out);
Ред 14-18 - Цикъл за комуникация със сървъра. Информацията се предава ред по ред. След всеки ред данни към сървъра се изчаква за неговия отговор (**ред 16**);
Ред 19 - Предаване на ред за край на комуникацията (“--end--”).

На Фиг. 4.2 е показан резултатът от работа на клиента и сървъра.

```
Net address = localhost/127.0.0.1
Socket = Socket[addr=localhost/127.0.0.1,port=8000,localport=1349]
Server: Ping 0
Server: Ping 1
Server: Ping 2
Server: Ping 3
Server: Ping 4
Server: Ping 5
Server: Ping 6
Server: Ping 7
Server: Ping 8
Server: Ping 9
Press any key to continue...
```

```
Server starting ...
Client: Ping 0
Client: Ping 1
Client: Ping 2
Client: Ping 3
Client: Ping 4
Client: Ping 5
Client: Ping 6
Client: Ping 7
Client: Ping 8
Client: Ping 9
Client: --end--
Connection closed
```

Фиг. 4.2. Резултат от работа на клиента и сървъра

V. ВЪПРОСИ И ЗАДАЧИ ЗА ИЗПЪЛНЕНИЕ

1. Какви са предимствата и недостатъците на многонишковото програмиране?
2. Стартирайте сървъра от Листинг 4.1. След това стартирайте кой да е Web браузър (Chrome, Explorer, Opera, Firefox), като в URL реда задайте да се адресира порт 8000. Ако клиентът и сървърът са на една машина, заявката трябва да бъде следната:

   `http://localhost:8000 <Enter>.`

   Анализирайте данните, които Web браузъра изпраща.
3. Напишете Java конзолно приложение, което разпечатва на екрана всички IP адреси, които са назначени за хост с указано мрежово име. Защо някои Интернет услуги имат назначен повече от един логически адрес?

Информация за връзката между мрежово име и логическите адреси, които му съответстват, може да се получи от DNS сървър. За целта трябва да се използва права DNS заявка. Тази информация може да се получи чрез използване на метод getAllByName() от клас InetAddress.

Резултатът, получен при име на хоста www.yahoo.com, е следния:

```
www.yahoo.com/87.248.100.216
www.yahoo.com/87.248.100.215
```