Интернет програмиране с Java – част 3

Автор: Светлин Наков,
Софийски Университет "Св. Климент Охридски"
Web-site: http://www.nakov.com

Последна промяна: 10.06.2002

Работа с UDP сокети

В предходната тема изяснихме как се разработват Java приложения, които си комуникират чрез TCP сокети. Както знаем от първата част на настоящия курс, TCP е протокол, който осигурява надеждна двупосочна поточно-ориентирана комуникационна линия. В тази тема ще се занимаен със средствата, които платформата Java ни дава за комуникация чрез единични UDP пакети.

Както знаем, протоколът UDP осигурява изпращане и получаване на единични пакети с данни, пристигането на които не е гарантирано. Поради факта, че не установява връзка между двата компютъра, които комуникират по между си, UDP генерира по-малък мрежов трафик отколкото TCP и затова осигурява по-голяма бързина. Той може да се използва, когато трябва да се изпращат малки по размер и независими едно от друго съобщения, но не е подходящ за изпращане на съобщения с голям размер или ако редът на тяхното пристигане е важен. Освен това, както вече знаем, успешното изпращане на един UDP пакет не гарантира че той ще пристигне или пък че ако изпратим два UDP пакета, те ще пристигнат в същия ред, в който са изпратени. Ето защо преди да се вземе решение да се използва комуникация по UDP, трябва да се прецени дали този протокол е подходящ.

В Java за поддръжката на UDP сокети разполагаме с класовете `java.net.DatagramSocket` и `java.net.DatagramPacket`. Класът `DatagramSocket` ни дава възможност да се свърваме към определен мрежов интерфейс и порт, да изпращаме пакети и да получаваме пакети. Класът `DatagramPacket` представлява структура от данни, която описва един UDP пакет. Да илюстрираме използването на тези класове чрез един пример. Нека създадем приложението UDPDataServer, което отговаря на клиентски UDP заявки за взимане на текущата дата:

```
import java.net.*;

public class UDPDateServer {
    public static void main(String[] args)
    throws Exception {
        DatagramSocket socketIn =
            new DatagramSocket(12345);
        DatagramSocket socketOut =
            new DatagramSocket();
        System.out.println(
            "UDP Date Server started.");
        while (true) {
            // Receive a request from a client
            byte[] requestBuf = new byte[256];
            DatagramPacket packetIn =
                new DatagramPacket(requestBuf,
                    requestBuf.length);
            socketIn.receive(packetIn);
            int len = packetIn.getLength();
            String request =
                new String(requestBuf, 0, len);

            // Send a response to the client
            if (request.startsWith("PORT=")) {
                int port = new Integer(request.
                    substring(5)).intValue();
                String resp = new java.util.
                    Date().toString();
                byte[] respBuf =
                    resp.getBytes();
                DatagramPacket packetOut =
                    new DatagramPacket(respBuf,
```

```
                        respBuf.length,
                        packetIn.getAddress(),
                        port);
                socketOut.send(packetOut);
            }
        }
    }
}
```

Както се вижда от кода, сървърът отваря два UDP сокета – един на порт 12345 за получаване на UDP пакети и още един на случаен порт, през който да изпраща отговорите на клиентите. След това в безкраен цикъл получава клиентска заявка, като счита че тя не надвишва 256 байта, извлича от нея IP адреса и порта, където да изпрати отговор, създава отговор (символен низ, съдържащ текущата дата и час) и го изпраща на клиента. IP адресът на клиента се взима от адреса, от който е дошъл пакета със заявката, а портът се взима от текста на клиентската заявка, която трябва да е във формат "`PORT=число`". Важна особеност на програмирането с UDP сокети с Java е, че един обект от класа `DatagramSocket` може да се използва или само за получаване, или само за изпращане на UDP пакети. Ако един сървър получава пакет, а след това изпраща отговор, той трябва да създаде два сокета, с които да извършва това, както е в нашия пример.

Нека сега напишем клиент за нашия сървър. Той трябва да изпраща на сървъра заявка, в която да задава на кой порт да се получи отговора и след като получи този отговор, да го отпечата. Разбира се, получаването на отговор не е гарантирано и затова чакането му трябва да е ограничено откъм време. Често срещана грешка е да се опитваме да получим отговора на същия порт, от който сме изпратили заявката. На пръв поглед това изглежда разумно, защото няма да има нужда да изпращаме порта, на който да получаваме отговора. Сървърът би могъл да го вземе от пакета, с който е получил заявката. На практика този подход е грешен, защото както вече споменахме, не можем да използваме един и същ обект от класа DatagramSocket хем за изпращане, хем за получаване на пакети. Ако използваме два DatagramSocket обекта – един за изпращане и един за получаване, не можем да ги накараме да използват един и същ порт, защото създаването на DatagramSocket обект изисква свободен порт. Ако се опитаме след изпращането на заявката да освободим порта, от който тя е изпратена и веднага да започнем да слушаме на него за отговор, има вероятност през времето, в което още не сме започнали да слушаме, отговорът вече да е пристигнал и да го изпуснем. Както се вижда и този подход не работи. Ето защо силно се препоръчва въпросът и отговорът при UDP комуникация да са на различни портове. Ето и примерна реализация на клиент за нашия сървър:

```
import java.net.*;

public class UDPDateClient {
    public static void main(String[] args)
    throws Exception {
        // Open an UDP socket for the response
        DatagramSocket socketIn =
            new DatagramSocket();

        // Send request to the UDP Date Server
        DatagramSocket socketOut =
            new DatagramSocket();
        String request = "PORT=" +
            socketIn.getLocalPort();
        byte[] requestBuf = request.getBytes();
        DatagramPacket packetOut =
            new DatagramPacket(requestBuf,
            requestBuf.length,
            InetAddress.getByName("localhost"),
            12345);
        socketOut.send(packetOut);
        socketOut.close();

        // Receive the server response
        byte[] responseBuf = new byte[256];
        DatagramPacket packetIn =
            new DatagramPacket(responseBuf,
                responseBuf.length);
        socketIn.setSoTimeout(5000);
        socketIn.receive(packetIn);
```

```
            int len = packetIn.getLength();
            String response =
                new String(responseBuf, 0, len);
            System.out.println(response);
            socketIn.close();
        }
    }
```

Както се вижда от кода, клиентът отваря UDP сокет за получаване на отговор от сървъра, подготвя един пакет, като записва в него този порт, след което отваря втори сокет и изпраща през него подготвения пакет на адрес `localhost:12345`, където се очаква да е стартиран сървъра. След това чака за отговор 5 секунди и ако за това време получи пакет с отговор, го отпечатва на конзолата, а в противен случай настъпва изключение.

Multicast сокети

Понякога се налага един пакет да бъде изпратен до много получатели. Обикновено тези получатели са приложения, които са заявили, че искат да получават тези пакети, т.е. те са се абонирали предварително за тях. Да вземем за пример една организация. Служителите в нея са разделени на работни групи и всеки служител иска да комуникира постоянно с колегите от своята група като им изпраща съобщения. От началството пък искат да могат да разпращат важни съобщения до всички служители. Една проста комуникационна система може да се изгради по следния начин: Софтуерът на всеки служител отваря TCP сокет до централен сървър, през който минават всички съобщения. Сървърът знае за всеки служител към кои групи принадлежи и когато получи съобщение предназначено за някоя група, го разпраща до всички нейни членове. По подобен начин, отново с централен сървър, цялата система може да се изгради и чрез UDP сокети. При големи натоварвания, обаче, централният сървър би могъл да не издържи или да работи, но неприемливо бавно. Представете си ако системата служи за комуникация не между хора, а между различни софтуерни компоненти на една сложна информационна система, където има десетки сървъри, като всеки от тях участва в хиляди групи и изпраща стотици съобщения в секунда. Очевидно натоварването е голямо, а рискът от срив на сървъра също не е малък. За решаването на такива проблеми са разработени multicast сокетите. Те много приличат на UDP сокети, но при тях има три основни операции – абониране за група, изпращане на съобщение до група, отказване от група. Всяка група се идентифицира с уникален IP адрес в локалната мрежа. Един компютър може да е едновременно в много групи. Изпращането на пакет до всички членове на група става като се изпрати този пакет до IP адреса на групата. Работата с multicast групи много прилича на работата с UDP сокети. Да разгледаме един пример – клиентско приложение, което се абонира за групата 224.0.0.1 и отпечатва на екрана всичи съобщения, получени в тази група на порт 2002.

```
import java.net.*;

public class MulticastListener {
    public static void main(String[] args)
    throws Exception {
        InetAddress multicastAddr =
            InetAddress.getByName("224.0.0.1");
        byte[] buf = new byte[1024];
        DatagramPacket packet = new
            DatagramPacket(buf,buf.length);
        MulticastSocket multicastSocket =
            new MulticastSocket(2002);
        multicastSocket.joinGroup(
            multicastAddr);
        while (true) {
            multicastSocket.receive(packet);
            String msg = new String(
                packet.getData(),
                0, packet.getLength());
            System.out.println(msg);
        }
    }
}
```

Както виждаме от кода, работата с multicast сокети съвсем не е сложна. Създаваме се multicast сокет, извикваме метода `joinGroup()`, с който се абонираме за някой multicast адрес и след това получаваме в цикъл UDP пакетите, предназначени за тази група и избрания порт. Класът, който използвахме `java.net.MulticastSocket`, има няколко основни метода – `joinGroup()` за присъединяване към multicast група, `leaveGroup()` за напускане на multicast група, `getTTL()` и `setTTL()` за извличане и промяна на параметъра TTL (time to live). Преди да изясним този параметър, трябва да си изясним механизма, по който работи multicasting-а. Груповото разпращане на съобщения (multicasting) се базира на протокола IGMP (Internet Group Management Protocol). Това е мрежов протокол, част от комплекта протоколи TCP/IP, за които говорихме в първата част на настоящия учебен материал. Multicast съобщенията се разпространяват по локална мрежа във вид на IGMP пакети. При подходящи настройки на мрежата тези пакети могат да преминават през маршрутизаторите в локалната мрежа. Преминаването през един маршрутизатор намалява стойността TTL с единица. Достигането на стойност 0 прекратява разпространението на пакета. Стойността TTL означава максималния брой маршрутизатори, през които съобщението може да премине. По стандарт в организацията на IP адресното пространство е предвидена специална зона от адреси от 224.0.0.1 до 239.255.255.255, които са предназначени за multicasting. Всеки от тези адреси би могъл да бъде използван за адрес на multicast група. Абонаментът за multicast услуги става по IP адрес на групата, но в рамките на тази група може да има 65535 различни услуги, съответстващи на различните възможни номера на портове. Поради факта, че за разпространението IGMP пакетите се грижи мрежовия хардуер и софтуер на IP ниво, този метод за разпращане на съобщение до група потребители е изключително ефективен и многократно по-бърз от подходите, които разгледахме по-горе (с централен сървър и TCP или UDP базирана комуникация). Изпращането на един пакет до хиляди компютри абонирани за някой multicast адрес отнема точно толкова време, колкото изпращането на един пакет до един компютър. Това означава, че изпращането на едно UDP съобщение до група получатели по multicast сокет може да е хиляди пъти по-бързо отколкото изпращането на същото съобщение до същата група получатели чрез TCP или UDP. Заради това multicasting-ът е предпочитан метод за работа в локална мрежа на приложения, при които високата производителност е от жизнена важност. Важно е да знаем, че multicasting-ът е свързан с някои особености на локалните мрежи, поради което не е приложим за Интернет. Едно от най-типичните приложения на този вид комуникация е при поддръжката на клъстери от сървъри. При тях multicast-ингът много често се използва за синхронизация на данните между сървърите, които работят в клъстер, заради високата скорост на разпространение и краткото време необходимо за изпращане на един пакет. Както вероятно повечето от вас знаят, клъстерът представлява група от компютри, които работят заедно като едно цяло. Клъстерът ни дава две основни предимства – скалируемост (scalability) и устойчивост на повреди (fault tolerance). Скалируемостта се постига от това, че имаме възможност да използваме няколко компютъра, които заедно работят много по-бързо отколкото един самостоятелен компютър и като си разпределят по равно работата, повишават многократно производителността на системата. Устойчивостта на повреди осигурява стабилност на системата, така че ако един от компютрите в клъстера се повреди или връзката се него се прекъсне, останалите да поемат неговата работа и клиентите на системата да не разберат за повредата. За да се постигне това, е необходимо в рамките на един клъстер компютрите да са взаимно заменяеми, т.е. всеки от тях да разполага с всичките данни, с които разполагат и останалите, за да може при евентуална повреда на някой от тях да няма загуба на данни. За синхронизацията на тези данни, когато се работи в локална мрежа, най-често се използват multicast сокети, заради огромната им скорост. Освен това при тях отпада необходимостта от централен сървър. Те дават и възможност включването и изключването на компютри в клъстер да може да става по всяко време.

След като изяснихме в общи линии какво е multicasting и за какво се използва, нека да напишем един сървър за клиента, с който започнахме.

```
import java.net.*;

public class MulticastSender {
    public static void main(String[] args)
    throws Exception {
        InetAddress multicastAddr =
            InetAddress.getByName("224.0.0.1");
```

```
            MulticastSocket multiSock =
                new MulticastSocket();
            multiSock.joinGroup(multicastAddr);
            while (true) {
                String message = "Hello " +
                    new java.util.Date();
                DatagramPacket packet =
                    new DatagramPacket(
                        message.getBytes(),
                        message.length(),
                        multicastAddr, 2002);
                multiSock.send(packet);
                Thread.sleep(1000);
            }
        }
    }
```

Както виждаме, кодът силно прилича на код, който разпраща UDP пакети. Единствената разлика е, че вместо класа DatagramSocket ползваме класа MulticastSocket и преди да започнем разпращането указваме групата, към която ще пращаме. Дадената сървърска програма в безкраен цикъл праща съобщение, съдържащо текущата дата, с цел да демонстрира получаването на multicast пакетите. За да тестваме клиента и сървъра, ни е необходим компютър включен в локална мрежа. На компютър без мрежова карта и двете програми не работят. Най-добре е ако имаме няколко компютъра и пуснем на всеки от тях няколко клиента и един сървър. Така всеки клиент ще получава всички изпратени пакети, а всеки сървър ще праща до всички клиенти. При желание могат да се добавят и още multicast групи.

Работа с URL ресурси

Още от времената, когато е била предназначена само за писане на аплети, Java се слави с това че, е силно Интернет ориентирана. Извличането на ресурси от Интернет с Java може да бъде изключително лесно, ако използваме класа `java.net.URL`. Нека първо изясним какво URL. Както при пощенските услуги, за доставяне на някаква поща е необходим адрес (държава, град, улица, номер и т.н.), така и в Интернет, в рамките на глобалната информационна система World Wide Web (WWW), за достъп до някакъв ресурс е необходим адрес. Адресите във WWW се наричат URL. URL е съкращение от Uniform Resource Locator – единен адрес на ресурс и има следния формат:

```
protocol://host[:port]/[resource]
```

Protocol е протоколът, по който е достъпен ресурсът, примерно `http`, `ftp` и др. Host е IP адресът или името на машината, от която е достъпен ресурсът, примерно `www.nakov.com` или `208.185.127.162`. Port е назадължително поле, което указва номера на порта на машината, зададена в полето host, примерно `80` или `8080`. Resource е пълното име на искания ресурс, като се включва и пътя до него. Ако не е зададен, се използва подразбиращият се ресурс. Например ако ресурсът, който искаме да извлечем е http://www.nakov.com/english/CV.html, протоколът е `http`, host-ът е `www.nakov.com`, портът не е зададен и се позразбира че е стандартния за http – `80`, а ресурсът е `english/CV.html`, като `english/` е пътят до ресурса, а `CV.html` е името му.

Ние всекидневно използваме URL адреси за достъп до различни сайтове в Интернет. Когато използваме стандартен Web-браузър, ние пишем в полето за адрес URL-то на сайта, който искаме да посетим и натискаме бутона за извличане на зададения адрес. Когато работим с Java отваряме URL като създаваме обект от класа `java.net.URL`, задаваме адреса на ресурса, от който се интересуваме и го извличаме използвайки стандартен входен поток. Нека илюстрираме това с един малък пример – програма, която извлича документа http://www.nakov.com/english/CV.html и го отпечатва на стандартния изход:

```
import java.net.*;
import java.io.*;

public class URLReaderExample {
    public static void main(String[] args)
    throws Exception {
        URL cv = new URL(
```

```
            "http://www.nakov.com/english/CV.html");
            BufferedReader in = new BufferedReader(
                new InputStreamReader(
                    cv.openStream()));
            String line;
            while ((line = in.readLine()) != null)
                System.out.println(line);
            in.close();
        }
    }
```

Кодът е съвсем кратък и ясен – създава се обект от класа URL, като му се подава адреса на ресурса, до който искаме да установим достъп, след това се отваря входен поток за четене на този ресурс с метода `openStream()` и от този поток се прочита целия ресурс ред по ред. Разбира се, разчитаме, че ресурсът е текстов документ и затова го четем с текстов поток. Ако трябваше да извлечем ресурс, който не е текстов, примерно картинка, трябваше да го четем с бинарен поток.

Класът `URL`, заедно с класа `URLConnection` могат да се използват не само за четене на ресурси, но и за писане в ресурси. При писане в ресурс на сървъра, който предоставя този ресурс се изпраща записаната от нас информация, оформена съгласно протокола, по който е достъпен този ресурс. Ето един пример как се прави това:

```
import java.io.*;
import java.net.*;

public class ReverseExample {
    public static void main(String[] args)
    throws Exception {
        String s = URLEncoder.encode(
            "Svetlin Nakov");
        URL url = new URL(
    "http://java.sun.com/cgi-bin/backwards");

        // Send request
        URLConnection connection =
            url.openConnection();
        connection.setDoOutput(true);
        PrintWriter out = new PrintWriter(
            connection.getOutputStream());
        out.println("string=" + s);
        out.close();

        // Retrieve response
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                connection.getInputStream()));
        String line;
        while ((line = in.readLine()) != null)
            System.out.println(line);
        in.close();
    }
}
```

В примера се осъществява достъп до услугата `http://java.sun.com/cgi-bin/backwards`, на която се подава един символен низ, а тя го връща в обратен ред. Тази услуга е реализирана като стандартен CGI script, който се извиква с параметри. Понеже услугата работи по протокола http, е необходимо всичко, което й се изпраща да се кодира съгласно http протокола, така че да бъде разпознато от нея. За целта се използва класа URLEncoder, с който параметрите към скрипта се кодират по стандарта за URL. След това се отваря изходен поток към услугата, подават й се кодираните параметри, после се отваря входен поток и се прочита резултатът от него. За писане в URL се използва класа URLConnection, обект от който се получава с метода `openConnection()`.

Повече информация относно класовете за работа със UDP сокети, multicast сокети и URL можете да намерите в документацията за Java 2 Standard Edition 1.4 на сайта на Sun – http://java.sun.com/j2se/1.4/docs/api/index.html, а също и на сайта на курса “Интернет програмиране с Java” – http://inetjava.sourceforge.net.