Lab #7: UDP S*ockets* Programming

Lab #7 introduces the programming of UDP sockets in Java. UDP protocol is easier than TCP protocol, however its programming can be initially more complex.

UDP protocol is a "connectionless” protocol, that is, not connection establishment is required before data is transmitted between client and server. For this reason, every time that a UDP datagram is sent, it is necessary to indicate the IP address and port of the destination.

In addition, the transmitted data may arrive out of order or even be lost. For this reason, the application itself must check the delivery of the data if reliability is needed. Making and analogy with daily life, we could say that UDP operates as the postal mail. The "letter" (like the UDP datagram) carries both, the sender address and the destination address. If the sender does not receive a response to the message sent, the sender can’t know whether the message was received or not (it was lost), or whether the response was sent and lost or simply it was not sent.

It is recommended to read previously Section 2.8 of Kurose's book, " Computer Networking A Top-Down Approach 5th.Edition", which explains the programming of UDP sockets in java.

2. Lab #7 aims

At the end of lab #7 you should be able to program a UDP client and a UDP server in Java. In particular you will be able to:

1) Use **`InetAddress`** class to represent an Internet IP address.
2) Use **`DatagramSocket`** class to create a UDP socket of both clients (bind a datagram socket to any available port on the local host machine) or servers (bind a datagram socket to the specified port on the local host machine).
3) Use the **`send()`**and **`receive()`**methods of the **`DatagramSocket`** class to send and receive UDP datagrams respectively.
4) Use different **`DatagramPacket`** class constructors to create the appropriate datagrams to be sent or be received.
5) Use the **`getPort()`**, **`getData()`**, **`getLengh()`** and **`getAddress()`**methods of **`DatagramPacket`** class to get the information contained in a **`DatagramPacket.`**

6) Use the **`setPort(), setData(), setLengh()`** and **`setAddress()`** methods of **`DatagramPacket`** class to set the information to send in a **`DatagramPacket`**.
7) Generate a text string from an array of bytes to display easily the received information.

3. InetAddress class

Because UDP is a connectionless protocol, one of the first problems encountered by the UDP client is how to specify the address of the destination. In java for this purpose, an object of the **`InetAddress`** class, belonging to the *java.net* package, is usually used. Among the methods offered by this class, the **`getByName`** () method accepts as a parameter a text string that represents the name of a host or its IP address, and returns an object of type **`InetAddress`** that contains the associated IP address.

Let's see some examples of it:

```
InetAddress ipServer = InetAddress.getByName(“www.upv.es”);
InetAddress ipServer = InetAddress.getByName(args[0]);
InetAddress ipServer = InetAddress.getByName(“127.0.0.1”);
```

When the parameter contains an IP address, for example, "127.0.0.1", only it is checked that the address has a valid format. On the other hand, if the argument contains a domain name such as "www.upv.es", an attempt will be made to resolve the name, generating if necessary a query to the local DNS server. If the resolution of the name fails, the exception UnknownHostException will be generated.

Note that **`getByName()`** is a static method, which means that it is always available directly by prefixing the name of the class (**`InetAddress`**). You do not need to have instantiated an object previously. The particularity of these methods is that the word **static** appears in the definition of the method (or attribute).

Another method of the **`InetAddress`** class that may be useful is the **`toString`**() method that translates the **`InetAddress`** object information into a string of text. The string that it returns has the format: `"<host name> / <IP address>".` If you do not have information about the host name, the host name part is shown as an empty string.

Exercise 1:

Write a Java program, called **`dnslookup`**, that accepts the name of a host as input argument and displays the host name and IP address or an error message indicating that the name could not be translated.

Use the **`getByName()`**method of the **`InetAddress`** class to translate the name and the **`toString()`** method to display it.

The program must be executed from a console. The use of the program should be: **`java dnslookup <hostname>`**

Run your program to find out the IP addresses of "www.eltiempo.es" and the web server of the UPV, "www.upv.es".

4. DatagramSocket class

`DatagramSocket` class represents a socket for sending and receiving UDP datagrams. The two constructors that we are going to see can generate exceptions of the **`SocketException`** type, which is a subclass of the input / output exceptions, **`IOException`**:

- **`DatagramSocket() throws SocketException`**.

  Constructs a datagram socket and binds it to any available port on the local host machine. Usually, we will use this constructor to create client sockets. A SocketException exception will be generated if the socket can't be created, for example, because there are no free UDP ports left in the system.

- **`DatagramSocket(int port) throws SocketException`**.

  Constructs a datagram socket and binds it to the specified port on the local host machine. Usually, we will use this constructor to create server sockets, since servers are interested in listening on a specific port number. If the argument **`port`** = 0 is used, the behaviour is equivalent to use the **`DatagramSocket()`** constructor (without parameters).

The **`DatagramSocket`** class has the **`getLocalPort()`** method. It returns the port number on the local host to which the socket is bound.

Example:

```
DatagramSocket ds = new DatagramSocket();
int p = ds.getLocalPort();
```

Exercise 2:

Create a UDP client using the **`DatagramSocket()`**constructor (without specifying the associated port number).

The UDP client will show on the screen (writing in the standard output (stdout)) the port number that has been assigned by the operating system to the UDP socket. Also, you will check how this port number changes after each execution.

Once a UDP socket is created, usually we are interested in using it to send and receive datagrams. To do this, we can use the **`send()`** and **`receive()`** methods of the **`DatagramSocket`** class:

- **`send(DatagramPacket p) throws IOException`**
- **`receive(DatagramPacket p) throws IOException`**

Both methods have a **`DatagramPacket`** class objetct as argument (**`DatagramPacket`** class will be seen in the following section). In the case of the **`send()`** method, the **`DatagramPacket`** will contain **`the`** message data, the message length and the IP address and port number of the destination. In the case of the **`receive()`**method, the **`DatagramPacket`** will be empty, it will be used to store the received message data, the received message length and the IP address and port number of the sender. Both the **`send()`** and the **`receive()`**method can generate an IOException.

The **`send()`** method returns as soon as the datagram passes the datagram to the lower level, responsible for transmitting it to its destination. However, the **`receive()`**method will block the program until a datagram is received, unless a listening timeout is established on the socket. Therefore, to avoid the execution of other instructions of the program, the **`receive()`**should be run in a separate thread. The idea is similar to the one we saw in the Lab #6 of concurrent TCP servers in the chat exercise.

On the other hand, the operating system will discard the received UDP datagrams destined to a closed destination port (there is no active UDP socket binds to the port).

As mentioned, sometimes it is interesting to limit the maximum waiting time in a **`receive()`**method to avoid the process waits indefinitely for a datagram that does not arrive. This limitation can be easily achieved through the **`setSoTimeout()`** method of the **`DatagramSocket`** class:

`setSoTimeout(int timeout) throws SocketTimeoutException`

This method sets the maximum waiting interval (expressed in milliseconds) that the **`receive()`**method will block the process waiting for a packet. If the established interval (the **`timeout`** argument of the previous declaration) timeouts and nothing has been received, an exception of type **`SocketTimeoutException`** will be generated. Obviously, it is necessary to set the time before invoking the **`receive()`**method.

Example:

```
DatagramSocket socket = new DatagramSocket(4444);
DatagramPacket p = new DatagramPacket(new byte[1024], 1024);
socket.setSoTimeout(3000); //waiting time of 3 s
// The execution will be blocked 3 s at most
socket.receive(p);
```

5. DatagramPacket class

This class represents a UDP datagram to be transmitted or received. The **`DatagramPacket`** objects contain a data field, an IP address and a port number. These last two values correspond to the sender when the datagram is received and to the destination when the datagram is transmitted.

Message (Array of bytes)	Message Length	IP Address	Port Number

6. Sending Packets

When UDP datagram is sent, it must be specified, in addition to the message data and its length, the IP address and the port number of the destination. These values can be included directly when the **`DatagramPacket`** is created using its constructor:

```
DatagramPacket (byte buf[ ], int longitud, InetAddress dirIP, int puerto) throws SocketException
```

Once the **`DatagramPacket`** is created, it can be sent just including it as argument in the **`send()`**method of the **`DatagramSocket`**.

The following code generates a datagram with the content "hello" destined to port 7777 of the computer where the program is running (since the chosen destination is **`localhost`**):

```
String ms = new String("hola\n");
DatagramPacket dp = new DatagramPacket(ms.getBytes(),
  ms.getBytes().length, InetAddress.getByName("localhost"),7777);
```

Exercise 3:

Write a Java program that sends a UDP datagram with your name and surname to port 7777 on your computer and ends. Remember that you can identify your computer as "**`localhost`**". To check the right behaviour of the program, use the command **`nc -u -l 7777`**, which establishes a UDP server that listens on port 7777 of your computer.

Run the program several times. Interestingly, you will notice that only the name is received the first time (we justify this behaviour below).

Once you verify that your program works correctly, change the destination address so that the teacher's computer receives it.

A curiosity

Why has not your name appeared several times on the screen when executing the exercise 3 program? Actually, the UDP datagram that contained the name has been received but the **`nc`** program has discarded it. The interface of the sockets allows to restrict the communication of a UDP socket to a single destination. For this purpose, the **`DatagramSocket`** class provides the **`connect()`**method. After executing this method, the socket is connected to a remote address and port, and packets may only be sent to or received from that address and port, and will discard datagrams from another source. An attempt to send a datagram to a destination other than the specified one will generate an exception. Note that the **`connect()`** method is strictly a local operation. That is, unlike TCP, the execution of **`connect()`** does not produce any exchange of packages with the other end.

Taking into account the previous explanation, what would have happened if your UDP client instead of sending a single datagram, sent that same datagram three times inside of a loop? Think about it.

7. Receiving Packets

To receive UDP datagrams it is sufficient to create a DatagramPacket object with buffer associated. We will use the constructor

```
DatagramPacket(byte buf[ ], int length)
```

This constructor constructs a **`DatagramPacket`** for receiving packets of length length.

The length argument must be less than or equal to buf.length.

When a message is received, it is stored in the **`DatagramPacket`** together with its length, the IP address and the port number of the socket from which the datagram was sent.

Example of **`DatagramPacket`** use for reception:

```
byte[] buffer = new byte[1000];
DatagramPacket p = new DatagramPacket(buffer, 1000);
DatagramaSocket ds = new DatagramSocket(7777);
ds.receive(p);
```

The received message can be obtained through the **`getData()`** method that returns the data buffer associated with the data field of the **`DatagramPacket`**. All bytes of the buffer! Since **`getData()`** returns the complete array of bytes, to know the length of the received datagram, and therefore, obtain the array data that really interests us, we must use the **`getLength()`** method. The use of this method is especially important when a **`DatagramPacket`** object is reused for several receptions, since it could remain garbage from some previous reception in the buffer.

How to print the content of the data

Many times we need to convert the received array of bytes into a string to be able to print it or compare it with others. There are many ways to make that conversion. A simple possibility goes through the use of one of the **`String`** constructors:

```
String s = new String(p.getData(), 0, p.getLength());
```

The rest of the relevant information of the datagram can be extracted using the **`getPort()`** and **`getAddress()`** methods of the DatagramPacket class, to obtain the port and the IP address of the datagram sender.

- **`InetAddress getAddress()`**
- **`int getPort()`**

Sending responses

The response to a received datagram is greatly simplified if the same packet that was received is reused, since it contains the IP address and port of the new destination (the sender of the received datagram). It will be sufficient to rewrite the response data using the **`setData()`** method of the **`DatagramPacket`** class. The length of the data can be established with the **`setLength()`** method of the **`DatagramPacket`** class.

- **`void setData(byte[] buf)`**
- **`void setLengt(int length)`**

Example:

```
ds.receive(dp);    //receive the DatagramPacket
String name = “hola\n”;
dp.setData(name.getBytes());
dp.setLength(name.length());
ds.send(dp); //send the DatagramPacket
```

What size of buffer to choose?

Although theoretically the maximum size of a UDP datagram almost reaches 64KB (65,531 bytes), many implementations do not support buffers larger than 8 KB (8,192 bytes). Larger packages are simply truncated, discarding the data that does not fit in the buffer (we must be careful with this, since java does not notify it to the application). Many protocols such as DNS or TFTP use packets of 512 bytes or less. The largest size in commonly used is 8,192 bytes for NFS. For our exercises, 512 bytes of buffer are usually enough.

Exercise 4:

Create a UDP client program that sends a datagram to port 7777 of your computer and then wait for a response from that same port, printing the contents of the received datagram on the screen (stdout). To test it, launch a UDP server on your computer that listens on port 7777, using the command "**`nc -u -l 7777`**". To generate the answer, write a text and press Enter.

Exercise 5:

Create a daytime UDP server that listens on port 7777, and return a datagram with the date and time in response to the reception of any datagram, regardless of its content. Try it using the **`nc`** program as a client, using the command "**`nc -u localhost 7777`**". The datagram sent by the client is not relevant, but at least one carriage return (Enter) must be entered so that it is transmitted and the server can reply.

In java you can get the date and time with the following instructions:

```
Date now = new Date();
String name = now.toString() + “\n”;
```

8. Optional extensions

Exercise 6:

Modify the program in Exercise 5 to wait for the client's datagram for a maximum time of 5 seconds. Check that the program finishes after 5 seconds, even if the client has not sent anything.

As a second optional extension, we will work with the DNS. Sometimes a domain name has several IP addresses associated. The **`getAllByName`** method allows us to obtain all of them in an **`InetAddress`** array:

```
static InetAddress[] getAllByName (String host) throws UnknownHostException
```

As was the case with **`getByName()`** method, if the resolution of the name fails, an **`UnknownHostException`** will be thrown.

Example:

```
InetAddress[] listaIp =
        InetAddress.getAllByName(“www.hotmail.es”);
```

The information obtained can be conveniently printed using the **`Arrays.toString(Object[] a)`** method, available in the **`java.util.Arrays`** class.

Example:

```
System.out.println(Arrays.toString
            (InetAddress.getAllByName("www.hotmail.es")));
```

Exercise 7:

Write a program in java that prints all the addresses of "www.google.es" on the screen (stdout).

Annexed

Some methods related to UDP sockets

InetAddress class

Returned Type	Method	Description
`static InetAddress`	**`getByName(String s)`**	Determines the IP address of a host, given the host's name. As it is a static method, it is always available directly by prefixing the name of the class (**`InetAddress`**).
`String`	**`toString()`**	Returns a string with the contents of the **`InetAddress`** object.

DatagramSocket class

Returned Type	Method	Description
`Int`	**`getLocalPort()`**	Returns the port number on the local host to which this socket is bound.
`InetAddress`	**`getLocalAddress()`**	Gets the local address to which the socket is bound. Not used in this Lab.
`Void`	**`send(DatagramPacket p)`**	Sends a datagram packet from this socket. El DatagramPacket includes the destination IP address and port, and the data to send.
`Void`	**`receive(DatagramPacket p)`**	Receives a datagram packet from this socket. It is a blocking method. The program waits in this instruction until the datagram is received. The received datagram is stored in p.

DatagramPacket class

Returned Type	**Method**	**Description**
`InetAddress`	**`getAddress()`**	Returns the IP address of the machine to which this datagram is being sent or from which the datagram was received.
`byte[]`	**`getData()`**	Returns the data buffer.
`Int`	**`getLength()`**	Returns the length of the data to be sent or the length of the data received.
`Int`	**`getPort()`**	Returns the port number on the remote host to which this datagram is being sent or from which the datagram was received.
`Void`	**`setData(byte[] buf)`**	Set the data buffer for this **`DatagramPacket`**.
`Void`	**`setLength(int length)`**	Set the length for this **`DatagramPacket`**.