Practical Exercises in Computer Networks

Introduction to Java Network Programming with Datagram Sockets (WIP)

© 2013, 2014 José María Foces Morán

This practical exercise introduces how to make program that allow computer connected to networks to communicate. The programming language used is Java alongside DatagramSocket and DatgramPacket APIs. In the development of this practical work we will program a few simple example programs like a UDP version of ping, an RTT estimator and a program to wake computers remotely.

Java UDP Sockets

The Internet as a whole can be considered a packet-switching network where any information we want to send must be broken down into a series of small chunks before they are transmitted, these pieces are named packets and each is transmitted and transported independently of the others. Packets are similar to letters in many respects, for example, suppose you need to send fifty A4-sized written sheets to a friend in New York City, you will have to break down the 50-page block into smaller blocks such that each fits within the envelope size you own at the moment. Each of the 5 10-page blocks will have to be addressed to your friend. Potentially, some of them might be transported over different routes than the others and they might arrive in an order other than the original order. We must not forget an important fact now: any of the outgoing letters could be lost by the mail system or get damaged. The example just developed briefly represents the nature of Internet which is technically known as the Internet Model of Service.

The terms used in Internet parlance are not the same used to describe the physical world: the 50-page block would be named the message to send, each of the 5 10-page blocks would be called a packet and the network (The Internet) could deliver those packets out-of-order, duplicated, with errors or, even not deliver some of them at all. The Internet Model of Service is a *best effort model*, one which guarantees are very loose. There is an Internet model of service property that has no counterpart in the real world example: the network can, unexpectedly duplicate packets erroneously; this does not happen in the mail example - that figures. The key word here is *packet*, a sort of envelope that contains the Internet address of the host to which it is meant to be delivered. This is the basic nature of Internet: lack of deterministic guarantees, but this seems to leave us in a disadvantageous position as beginners, for, what is the rationale of a network that offers us no delivery guarantee? We will delve into the Internet Service Model in upcoming lectures, therefore, for the time being we will want to recover the lost trust, somehow: what the network can not guarantee us, we will compensate for those guarantees by increasing the responsibilities of the sending and the receiving hosts (An end-to-end responsibility, not the network's).

The key word is packet. The network switches packets, pretty much like the switchboard operator of the old circuit-switched telephone networks switched circuits, though you already know there are a lot of differences between both approaches, then, we build a packet and submit it to the network for transfer to its destination. The packet is the envelope in the mail analogy, then, where is the letter? The letter we name it *datagram* in the context of this practical

exercise, that suffices for the time being.

All in all: we have some message that we want to deliver to some application running on a destination host, the *message gets* broken up into separate datagrams, each of which subsequently gets encapsulated into an IP packet, each of these is eventually submitted to the network for transfer to its destination host. How can we illustrate this in a practical manner, in Java? The Java language offers us two essential APIs for carrying out the process that we just explained: DatagramPacket which corresponds to the Datagram concept explained above and DatagramSocket that corresponds to the encapsulation of a datagram into an IP packet for transmission. An example should help us gain further intuition.

A brief introduction to UDP: the simple demultiplexer

The example we are building now is a **ping program** in Java using the classes mentioned above: DatagramPacket and DatagramSocket. Recall that the real ping command is based on an Internet control-plane protocol named ICMP; here, we set out to emulate ping by programming against UDP (A layer-4 protocol) instead of against ICMP (A 3rd layer protocol). How come UDP, what protocol is that?

Let's assume that the host where we are programming these examples has a single NIC which IP address is 193.146.101.46 whose DNS name is paloalto.unileon.es. The correctly assigned IP identifies a single host in Internet and each IP packet directed toward it is ultimately delivered to it (If all is functioning correctly). With IP addresses we solve the problem of delivering packets to the correct destination host, but, we need to decide how to deliver the packet's payload to a specific process. We are assuming that the host is running a multitasking operating system like Linux where a multitude of processes will be running concurrently. UDP (User Datagram Protocol) is capable of identifying a single process on the destination host, it does so by providing a *name space for processes*.

The User Datagram Protocol (UDP) data unit is named Datagram, every datagram has a source port field and a destination port field, these ports somehow map the originating process to the destination process in the destination host. Your Java program runs in a process space in the host operating system and it may acquire a UDP port by creating what is known as a UDP socket. Once your Java program has hold of the UDP socket it can use it to transmit and receive datagrams over it, observe the following sketch of the steps required:

1. Your program acquires a UDP socket on port 50001, for example
2. Now, your program is bound to that port: the UDP socket knows your program and your program knows the UDP socket
3. When your host receives an IP packet that contains a UDP datagram whose destination port is 50001, the datagram's payload (Typically an application-level block of information) will be delivered to your Java program.

All in all: An IP's identifies a host and a UDP port identifies a single process running in that host –today, instead of a single process, we should say *a single thread*.

A ping application based on UDP

Java has a series of APIs under java.net that represent the UDP sockets mentioned above: DatagramSocket for sending and receiving DatagramPackets, as you have already guessed DatagramPacket is the Java class that is used to create UDP datagrams suitable for transferring them over DatagramSockets, those UDP datagrams will carry the information you want transferred from the originating host-process to the receiving host-process.

We are going to write the ping server which will bounce back the information it receives and the client which will simply send a few datagrams to the server and will test each datagram is correctly received back in turn.

The server's Java class will be named UDPPingServer.java following the class naming conventions of Java. When run, this program will wait for a datagram to arrive and then it will re-send it to the host that sent the original datagram, but for further clarity, as an example data processing, we will capitalize the original message so that when the response datagram is received we can be convinced that it arrived at its destination and was processed and sent back. In the development of this practical example, in case of doubt regarding DatagramPacket and DatagramSocket, search the Javadoc pages at Oracle by googling "java 6 datagrampacket" for example. Also, you can skim the Oracle's Java Tutorial section devoted to Java UDP programming. Striving for simplicity in these initial *toy* examples, we will not follow a detailed object oriented design process, certainly you will have to follow such process when you write full applications so that your software gets an appropriate life cycle.

UDP programming in Java gravitates around DatagramSocket and DatagramPacket classes. DatagramSocket represents in Java a UDP port in your operating system's TCP/IP **network stack**, therefore, using its methods you can basically do two things with it:

1. Transfer outbound information through it, that info will be identified with your IP and the port you specified when you created the DatagramSocket. In the diagram of fig. 1, the DatagramSocket **ds** is associated with UDP port 50001, therefore, all the datagrams (Java DatagramPacket) that you transfer using **ds** will get its source port set to 50001 and will subsequently be encapsulated in an IP packet whose source IP will be 193.146.101.46. In the example, the blue DatagramPacket is being sent to host IP 70.80.90.100, destination UDP port 2000. You use the DatagramSocket's send() method to carry out the transfer.
2. Receive inbound information through it. All the datagrams that are received which destination UDP port is number 50001 will eventually be conveyed to your Java program. The method in this case is receive().

The following figure highlights the two UDP Java programming classes, DatagramSocket and DatagramPacket, without coming down into much detail, however, in order for us to get faithful information at this stage it is necessary to comment that every UDP datagram gets encapsulated in an IP packet which will be addressed with the destination host's IP and that it will also contain the source IP address.

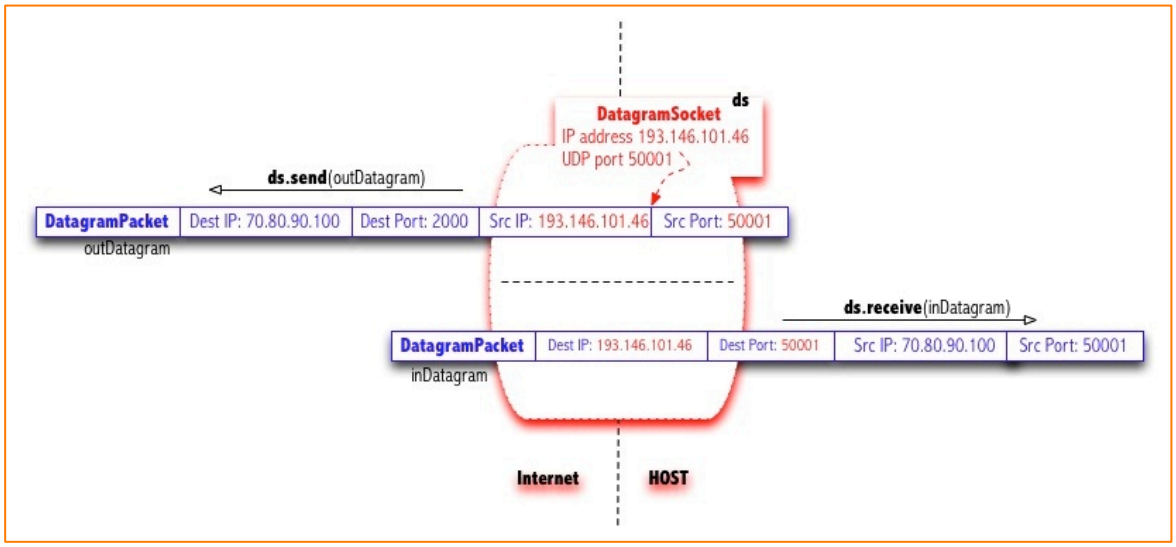

Fig. 1. DatagramPacket object being transferred to its destination host and destination port by using a DatagramSocket

In the following paragraphs we are going to explain the highlights of the UDPPingServer.java program in what is relevant to programming with DatagramSocket and DatagramPacket. You can download their source code from the following URLs:

http://paloalto.unileon.es/cn/UDPPingServer.java
http://paloalto.unileon.es/cn/UDPPingClient.java

The DatagramSocket() constructor used in the example server program receives a single integer that represents the UDP port number associated with the new DatagramSocket, in our case variable PORT has been assigned a value of 50001. You must include the constructor invocation in a try/catch that might capture any network-related problem and give you a chance to recover or to break the program's execution altogether as in our case, for simplicity reasons.

```
DatagramSocket ds = null;

   public UDPPingServer() {

      try {

         ds = new DatagramSocket(PORT);

      } catch (SocketException ex) {
         Logger.getLogger(UDPPingServer.class.getName()).log(Level.SEVERE, null, ex);
         System.exit(-1);
      }

   }
```

At this stage we have a new DatagramSocket like the one in the figure. Now we will illustrate how to create a fresh

DatagramPacket and expect our counterpart client to send us some test packet that we are going to bounce back to it, after all, we aim to emulate ping, right? In the following code snippet we create a DatagramPacket and give it a block of storage for it to be able to store the bytes it receives from the TCP/IP stack in a storage accessible by our Java program.

```
byte[] inData  = new byte[MAXPACKETSIZE];

        /*
         * Create a new datagram with a byte buffer of size MAXPACKETSIZE
         */
        DatagramPacket inDatagram = new DatagramPacket(inData, inData.length);
```

Now, we can call DatagramPacket's receive() method, this method will block until a UDP datagram arrives, that datagram's payload will be transferred to the byte array inData in the preceding code snippet.

```
        try {

            /* Block until a new datgram arrives at this IP address
             * on ds's port
             */
            ds.receive(inDatagram);

         } catch (IOException ex) {
            Logger.getLogger(UDPPingServer.class.getName()).log(Level.SEVERE, null, ex);
         }
```

The aforementioned receive() method blocks, i.e., it will not return until a datagram arrives, meanwhile your program will be suspended by the Java Virtual Machine.

IP addresses in Java are represented by a new java.net class whose name is InetAddress, we will need an instance of InetAddress where we can save the originating IP address (The ping client). You can obtain that IP address by calling DatagramPacket getAddress() method.

The rest of the server program processes the incoming data which are supposed to be the byte-array representation of a String object sent by the client. Once we have the data that we want to bounce back to the client, we build a new DatagramPacket provide storage for its payload by means of a new byte[] array named outData, ultimately we send the new DatagramPacket over the DatagramSocket and loop back again:

```
         /*
         * Create a datagram to hold the echo back data
         */
        byte[] outData = outString.getBytes();
        DatagramPacket outDatagram =
                new DatagramPacket(outData,
                outData.length,
                inDatagram.getAddress(),
                inDatagram.getPort());
```

```
        /*
         * Send the datagram over the ds DatagramSocket
         */
        try {
            ds.send(outDatagram);
        } catch (IOException ex) {
            Logger.getLogger(UDPPingServer.class.getName()).log(Level.SEVERE, null, ex);
        }
```

The relevant counterpart code sections in the client that have been explained in the server follow, beginning with the client's DatagramSocket creation, you may wonder why in this case the constructor will get no port argument, the reason is that port in clients don't necessarily have to be known, only server's ports must be known (Well known ports): the empty constructor will build a DatagramSocket with an associated ephemeral port, that is, any number greater than 1024.

```
//Global object's state
    DatagramSocket ds = null;
    InetAddress remoteAddress = null;
    int remotePort;

    /*
     * Constructor receives command line parameters remote IP address and the
     * remote port on which the server expects to receive bounceback requests
     * (The server has a wired-in UDP port number of 50001)
     */
    public UDPPingClient(InetAddress remoteAddress, int remotePort) {

        try {

            this.remoteAddress = remoteAddress;
            this.remotePort = remotePort;
            ds = new DatagramSocket();

        } catch (SocketException ex) {
            Logger.getLogger(UDPPingClient.class.getName()).log(Level.SEVERE, null, ex);
            System.exit(-1);
        }

    }
```

The following client code corresponds to the test packet send operation and the subsequent section contains the reception of the bounced-back response.

```
 //outData will hold EXACTLY the size of the text to send
byte[] outData = textToSend.getBytes();

/*
 * Constructs a datagram packet for sending packets of length
 * outData.length to the specified port number on the specified host.
 * The length argument must be less than or equal to buf.length
 *
 * Only outData.length bytes will be transmitted, i.e., the actual
 * size of the outData can be greater than the second argument, only
 * the number of bytes indicated in it will eventually get transmitted
 */
DatagramPacket outDatagram =
        new DatagramPacket(outData, outData.length, remoteAddress, remotePort);

try {
    ds.send(outDatagram);
} catch (IOException ex) {
    Logger.getLogger(UDPPingClient.class.getName()).log(Level.SEVERE, null, ex);
}


/* This array will hold the received bytes which maximum size
 * is MAXPACKETSIZE. The ACTUAL size of the data received can be
 * found out by inDatagram.getLength()
 */
byte[] inData = new byte[MAXPACKETSIZE];
DatagramPacket inDatagram =
        new DatagramPacket(inData, inData.length);

//Receive a new DatagramPacket over the DatagramSocket ds:
try {
    ds.receive(inDatagram);
} catch (IOException ex) {
    Logger.getLogger(UDPPingClient.class.getName()).log(Level.SEVERE, null, ex);
}
```

By now, after this brief introduction, you should try to compile both programs and run the server first and then the client; also, it is advisable that you study thoroughly both programs by scanning its source code and the Oracle javadoc entries corresponding to the APIs explained.

Exercises

1. Donwload, compile and run the server and client ping programs in different hosts in the laboratory, watch the results and take note of your possible doubts. You will need to run the Linux network commands explained in Lab 1 to find out the IP addresses of the hosts with which you are carrying out this exercise. Also, it might be useful for you to skim the Oracle's **Java Tutorial** section on UDP sockets and also take a look at the DatagramSocket and DatagramPacket Javadocs.

2. Can you explain how one can create a new instance of Java InetAddress? By using the new operator? If that is not the case, study how you can request a new instance of InetAddress by looking it up in Oracle javadocs, particularly the -static- factory methods present in them.

3. While your UDP ping server program is running, check with netstat the corresponding UDP socket, its port, etc. Do the same with the client program, check with netstat that it has a UDP socket open, check both the source and the destination port numbers.

4. The ping command offers you information regarding the statistical distribution of the RTT's measured (Round trip times), now, we want the same capabilities incorporated in our UDP ping client/server, to that end we provide you a skeleton client program in which you will have to program the RTT measurements, you can use the same bounce-back ping server UDPPingServer.java that we wrote in the preceding sections. Download the source code file from the following URL and read the comments so you understand your task better:

http://paloalto.unileon.es/cn/RttClient.java

5. Test the RttClient program against protocol.unileon.es, carry out a few RTT measurements with it and contrast the results obtained to those obtained by using the ping command. Summarize your results in tabular form and analyze them, explain the most important conclusions you have reached.

6. Study the relationships existing between the design of the RttClient.java program and the performance concepts explained in chapter 1, of particular interest is the size of the data to be sent to the remote host in the measurement of the RTT.

7. Now, after measuring the RTT we are asked to measure the throughput to the same hosts with which we carried out the RTT measurements. Using the same RttClient.java source code as a base code, write a new client and a new server programs for estimating the throughput between two internet hosts.

8. Wake-on-Lan (WOL) is a capability of the modern NICs that allows them to wake up the computer system in which they are installed. To wake a computer remotely we have to find out the NIC's MAC address and then, we build and send an Ethernet frame that contains that MAC duplicated 16 times, when the NIC senses that frame on the network medium it will wake the computer system, which will start the bootstrap process and eventually boot an operating system. That special Ethernet frame is known as *the magic packet*, in Java we use DatagramPacket and DatagramSocket to build and send the magic packet. In this exercise we will test the WOL by modifying the following Java program according to the preceding explanation and executing it.

Download the source code for WakeOnLan program from http://paloalto.unileon.es/cn/WakeOnLan.java

Appendices

Source code

```
/**
 * *********************************************************************
 * Universidad de León, EIII Grado en Ingeniería Informática
 * Course on Computer Networks (C) 2013 José María Foces Morán, lecturer
 *
 * CN LAB 2
 * UDPPingServer.java
 *
 * Sends UDP datagrams to a UDP bounceback server, it tries to imitate the
 * utility provided by the UNIX ping program. Peer client is implemented in
 * UDPPingClient.java belonging to Computer Networks LAB 2.
 *
 * - Datagram socket echo server, responds to eight requests
 * from its counterpart client (EchoClient.java). Illustrates the use of the
 * DatagramSocket and DatagramPacket java classes,
 * these sockets use the underlying non reliable UDP transport (OSI level 4).
 *
 * - This application program roughly follows the Client/Server (C/S) model in
 * its design: the server program establishes a socket which, upon recpetion of
 * the first request honors it and restarts listening for more requests. Any
 * client program instance, contacts the server at UDP port 50001, sends its
 * request, gets the response and finishes. The server finishes after 64
 * requests have been serviced altogeher.
 * *********************************************************************
 */

package udpinitialexamples;

import java.io.IOException;
import java.net.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class UDPPingServer {

    //Global constants
    static final int PORT = 50001; //The UDP port is wired here
    static final int MAXPACKETSIZE = 64;

    //Global object's state
    DatagramSocket ds = null;

    public UDPPingServer() {

        try {

            ds = new DatagramSocket(PORT);

        } catch (SocketException ex) {
            Logger.getLogger(UDPPingServer.class.getName()).log(Level.SEVERE, null, ex);
            System.exit(-1);
```

```
    }

}



/*
 * The bounceback receive/send loop:
 *
 * Honor 64 echo requests of at most 64 bytes in length each, these
 * requests come from counterpart clients. The method responds to each
 * requests by:
 *
 * 1. Converting to  upper case what is suppossed to be a string of
 * 8-bit UTF-8 characters
 * 2. Returning the new sequence of uppercase characters alongside
 * with the request number (1..64)
 *
 */

private void serverLoop(){

    while(true){

       byte[] inData  = new byte[MAXPACKETSIZE];

       /*
        * Create a new datagram with a byte buffer of size MAXPACKETSIZE
        */
       DatagramPacket inDatagram = new DatagramPacket(inData, inData.length);

        try {

            /* Block until a new datgram arrives at this IP address
             * on ds's port
             */
            ds.receive(inDatagram);

        } catch (IOException ex) {
            Logger.getLogger(UDPPingServer.class.getName()).log(Level.SEVERE, null, ex);
        }


        InetAddress fromIP = inDatagram.getAddress();

        /*
         * Convert the inData byte buffer into a String, since we are
         * reusing it we will fetch the exact
         * number of bytes received in this datagram reception: the method
         * DatagramPacket.getLength() returns that exact number of bytes
         */
        String inString = new String(inData, 0, inDatagram.getLength());

        System.out.println("Received data length: " + inDatagram.getLength());
        System.out.println("Data: " + "'" + inString + "'");
        System.out.println("From IP: " + fromIP);

        String outString = processIncomingData(inString);
```

```
            System.out.println("Sending back:" + "'" + outString + "'");
            System.out.println();

            /*
             * Create a datagram to hold the echo back data
             */
            byte[] outData = outString.getBytes();
            DatagramPacket outDatagram =
                    new DatagramPacket(outData,
                    outData.length,
                    inDatagram.getAddress(),
                    inDatagram.getPort());

            /*
             * Send the datagram over the ds DatagramSocket
             */
            try {
                ds.send(outDatagram);
            } catch (IOException ex) {
                Logger.getLogger(UDPPingServer.class.getName()).log(Level.SEVERE, null, ex);
            }

        }

    }

    /*
     * Simulate some data processing over the incoming data, in this case,
     * convert the string to upper case
     */
    private String processIncomingData(String s){

        return s.toUpperCase();

    }

    public static void main(String[] args) {

        UDPPingServer up = new UDPPingServer();

        System.out.println("UDPPingServer @ port running " + PORT);
        System.out.flush();

        up.serverLoop();

    }
}
```

```
/**
 * ********************************************************************
 * Universidad de León, EIII Grado en Ingeniería Informática
 * Course on Computer Networks (C) 2013 José María Foces Morán, lecturer
 *
 * CN LAB 2
 * UDPPingClient.java
 *
 * Sends UDP datagrams to a UDP bounceback server, it tries to emulate the
 * utility provided by the UNIX ping program. Peer server is implemented in
 * UDPPingServer.java belonging to Computer Networks LAB 2.
 *
 * - Sends a UDP echo request to a server at port 50001 at localhost, gets the
 * response and lays it out on the console. Illustrates the use of the
 * DatagramSocket and DatagramPacket java classes,
 * these sockets use the underlying non-reliable UDP transport (OSI level 4).
 *
 * - This application program roughly follows the Client/Server (C/S) model in
 * its design: the server program establishes a socket which, upon reception of
 * the first request datagram honors it and begins to wait for more requests.
 * Any client program instance, contacts the server at UDP port 50001, sends its
 * request, gets the response and finishes. The server finishes after 64
 * requests have been serviced altogeher.

 * ********************************************************************
 */

package udpinitialexamples;

import java.io.IOException;
import java.net.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class UDPPingClient {

    //Global constants
    static final int MAXPACKETSIZE = 64;
    static final int NITERATIONS = 64;

    /* counter serves to count the number of datagrams sent from the client to
     * the server, counter is appended to a base, test text message
     */
    int counter = 0;

    //Global object's state
    DatagramSocket ds = null;
    InetAddress remoteAddress = null;
    int remotePort;

    /*
     * Constructor receives command line parameters remote IP address and the
     * remote port on which the server expects to receive bounceback requests
     * (The server has a wired-in UDP port number of 50001)
     */
    public UDPPingClient(InetAddress remoteAddress, int remotePort) {
```

```
        try {

            this.remoteAddress = remoteAddress;
            this.remotePort = remotePort;
            ds = new DatagramSocket();

        } catch (SocketException ex) {
            Logger.getLogger(UDPPingClient.class.getName()).log(Level.SEVERE, null, ex);
            System.exit(-1);
        }

    }

    /*
     * Provide a inter-request delay of t seconds as a way of limiting the network
     * resources employed by the program
     */
    public static void pause(long t) {

        try {
            Thread.sleep(1000 * t);
        } catch (InterruptedException ex) {
            Logger.getLogger(UDPPingClient.class.getName()).log(Level.INFO, null, ex);
            System.exit(-1);
        }

    }

    /*
     * The bounceback send/receive loop
     */
    private void clientLoop() {

        //The message to send
        String testMessage = new String("Computer Networks");


        for (int i = 0; i < NITERATIONS; i++) {

            counter++;

            String textToSend = testMessage + " #" + counter;
            System.out.println("Client sending: " + textToSend);
            System.out.flush();

            System.out.println("To " + remoteAddress + " port " + remotePort);
            System.out.flush();

            //outData will hold EXACTLY the size of the text to send
            byte[] outData = textToSend.getBytes();

            /*
             * Constructs a datagram packet for sending packets of length
             * outData.length to the specified port number on the specified host.
             * The length argument must be less than or equal to buf.length
             *
             * Only outData.length bytes will be transmitted, i.e., the actual
```

```
             * size of the outData can be greater than the second argument, only
             * the number of bytes indicated in it will eventually get transmitted
             */
            DatagramPacket outDatagram =
                    new DatagramPacket(outData, outData.length, remoteAddress,
remotePort);

            try {
                ds.send(outDatagram);
            } catch (IOException ex) {
                Logger.getLogger(UDPPingClient.class.getName()).log(Level.SEVERE, null,
ex);
            }


            /* This array will hold the received bytes which maximum size
             * is MAXPACKETSIZE. The ACTUAL size of the data received can be
             * found out by inDatagram.getLength()
             */
            byte[] inData = new byte[MAXPACKETSIZE];
            DatagramPacket inDatagram =
                    new DatagramPacket(inData, inData.length);

            //Receive a new DatagramPacket over the DatagramSocket ds:
            try {
                ds.receive(inDatagram);
            } catch (IOException ex) {
                Logger.getLogger(UDPPingClient.class.getName()).log(Level.SEVERE, null,
ex);
            }

            System.out.println("Client received: " + new String(inDatagram.getData()));
            System.out.print(". Length: "+ inDatagram.getLength() +" bytes");
            System.out.println();

            pause(1L);

        }

    }

    public static void main(String[] args) {

        InetAddress remoteAddress = null;
        int remotePort = 0;

        if (args.length != 2) {
            System.out.println("UDPPingClient: <UDPPingSever IP address> <UDPPingServer
port>");
            System.exit(-1);
        } else {
            remotePort = Integer.parseInt(args[1]);

            /*
             * Call the factory method InetAddress.getByName(...) to convert the
             * textual representation of the server's IP address to a Java javanet
             * ip address InetAddress:
             *
```

```
             * Why InetAddress instances are not obtained by new inetAddress?
             * (There is no constructor for InetAddress)
             */
            try {
                remoteAddress = InetAddress.getByName(args[0]);
            } catch (UnknownHostException ex) {
                Logger.getLogger(UDPPingClient.class.getName()).log(Level.SEVERE, null,
ex);
                System.exit(-1);
            }
        }

        System.out.println("UDPPingClient: " + remoteAddress + " " + remotePort);
        System.out.flush();

        UDPPingClient up = new UDPPingClient(remoteAddress, remotePort);

        up.clientLoop();


    }
}
```

```
/*****************************************************************************
 * Universidad de León, EIII
 * Grado en Ingeniería Informática
 * Asignatura de Arquitectura, Diseño y Gestión de Redes de Computadores
 * Computer Networks, 2nd term 11/12 and 12/13
 * (C) José María Foces Morán, lecturer.
 *
 * RttClient.java
 *
 * - Datagram socket bounceback client for measuring the RTT of a UDP channel
 *
 * - This application program roughly follows the Client/Server (C/S) model in
 * its design: the remote server (UDPPingServer.java) program creates a UDP
 * socket at UDP port 50001, receives one byte and bounces it back and restarts
 * waiting for the next byte to arrive.
 *
 * This client program (RttClient.java) instance, contacts the server at UDP port
 * 50001, sends one byte, gets the response and measures the time interval between
 * the transmission of the single byte and the time at which the response
 * was received.
 *
 * ***************************************************************************
 */


package udpinitialexamples;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;
import java.net.InetSocketAddress;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author Chema
 */
public class RttClient {

    private static final int REMOTEPORT = 50001;
    private static final int NITERATIONS = 64;

    /*
     * Tries to create a Datagram client socket
     */
    private DatagramSocket setupClientSock() {

        DatagramSocket s = null;

        try {

            s = new DatagramSocket();

        } catch (SocketException ex) {
            Logger.getLogger(RttClient.class.getName()).log(Level.SEVERE, null, ex);
```

```
            System.exit(-1);
        }

        return s;

    }

    /*
     * Estimate the UDP channel RTT between UDP client and UDP server
     */
    void sampleRTT(DatagramSocket s) {

        //Build the least amount of information to send over the socket: 1 Byte
        byte outBuf[] = new byte[1];
        outBuf[0] = (byte) 'A';

        /*
         * If you want to use the Rtt server at protocol, substitute
         * "localhost" for "protocol.unileon.es" next:
         */
        InetSocketAddress server = new InetSocketAddress("localhost", REMOTEPORT);

        //Send one byte NITERATIONS times
        for (int i = 0; i < NITERATIONS; i++) {


            /*
             * CODE THAT YOU MUST COMPLETE:
             *
             * 0. Create DatagramSocket and DatagramPackets
             * 1. Call systemTimeMillis() to get current time
             * 2. Send a one-byte DatgramPacket over the DatagramSocket
             * 3. Receive the response (1 byte) from the server over
             *    the DatagramSocket
             * 4. Call systemTimeMillis() to get current time again
             * 5. Calculate time interval and store it for later reference
             *
             */

            //1-second pause between each send-receive cycle
            UDPPingClient.pause(1L);

        }



    }

    /*
     * Start the client program
     */
    void start() {

        DatagramSocket s = setupClientSock();

        sampleRTT(s);

    }
```

```
    public static void main(String[] args) {

        (new RttClient()).start();

        System.out.println("Client exiting");
    }
}
```