Network Programming – Sockets

1. Objectives

Learn Java programming techniques using sockets.

2. Preliminary notions

Computers connected by network communicate with each other using TCP (Transport Control Protocol) and UDP (User Datagram Protocol) protocols according to the diagram:

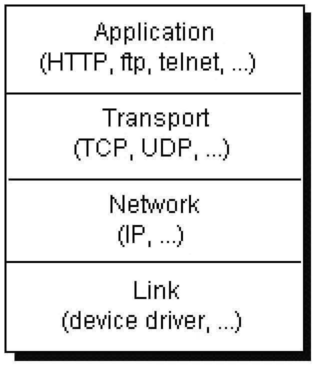


Fig1. Network communication levels

To implement programs that communicate via network using java, the classes in the **java.net** package are used. This package offers the classes required for implementing network programs that are independent from the operating systems.

The following table presents the main classes that are used to build network programs.

Class	Scop
URL	Represents a URL
URLConnection	Returns the content addressed by URL objects
Socket	Create a TCP socket
ServerSocket	Create a TCP server socket
DatagramSocket	Create a UDP socket
DatagramPacket	Represents a datagram sent through a DatagramSocket object
InetAddress	Represents the name of a PC on the network, or the corresponding IP

Java offers two different approaches to implement network programs. The two approaches are associated with classes:

• Socket, DatagramSocket and ServerSocket

• URL, URLEncoder, and URLConnection

Socket programming represents a bottom-up approach, whereby two computers can be connected to exchange data. As a basic principle, socket programming makes it possible to communicate in full-duplex mode between the client and the server. The communication is done by **byte streams**.

In order for the corresponding communication to proceed properly, the programmer will need to implement a communication protocol (dialogue rules) that the client and the server will follow.

Identifying a computer on the network

Any computer connected to the Internet is uniquely identified by its **IP** address (IP is the acronym of the Internet Protocol). This is a 32-bit number, usually in the form of 4 bytes, such as: 193.226.5.33 and is called a numerical IP address. Corresponding to a numeric address, there exist a symbolic IP address, such as utcluj.ro. Also, every computer on a local network has a unique name that can be used to identify it locally.

The Java class that represents the notion of IP address is **InetAddress**. To build an object, use the command:

InetAddress address =InetAddress.getByName("121.3.1.2");

To see all the ways in which **InetAddress** objects can be built, study the documentation of this class.

Generally, a computer has only one physical connection to the network. Any information that is destined to a machine is obligated to specify the IP address of that machine. But there may be several processors on a single computer that have established a network connections, waiting for diverse information. Therefore, data sent to a destination must specify the process to which that information is directed besides the IP address of that computer. The processes are identified through ports.

Any application communicating on the network is uniquely identified through a port so that the incoming packets on the host computer can be properly routed to the destination application.

The values that a port number can take are between 0 and 65535 (because they are 16-bit numbers), however, numbers between 0 and 1023 are reserved for some system services, they are not recommended for use.

Socket definition: A socket is a connection point on a TCP \ IP network. When two programs exist on two computers in a network wish to communicate, each one of them uses a socket. One of the programs (the server) will open a socket and then will wait for connections, the other program (the client) will connect to the server and so the exchange of information can begin. To establish a connection, the client will need to know the destination address (of the computer where the socket is opened) and the port number of the opened socket.

The main operations that are made by sockets are:

- Accept connections
- Connect to another socket
- Sending data
- Receiving data
- Closing connection.

3. Client-Server Application with a Single-threaded Server

To run a client-server program, the **ServerSocket** and Socket classes are used.

The server program will need to open a port and wait for connections. For this purpose, the **ServerSocket** class is used. When creating a **ServerSocket** object, the port (where waiting will be initiated) is specified. The listening port is started using the **accept()** method. When a client is connected, the **accept()** method returns a Socket object.

In turn, for the client to connect to a server, you will need to create a Socket object which will receive the parameters: the server's address and the port that waits for connections.

At both server and client levels, once Socket objects are created, read and write streams will be obtained. For this purpose, **getInputStream()** and **getOutputStream()** methods are used.

The listing below illustrates the server program.

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

public class SimpleServer {
  public static void main(String[] args) throws IOException{

    ServerSocket ss=null;
    Socket socket=null;
    try{
      String line="";
      ss = new ServerSocket(1900); //creates a serversocket object
      socket = ss.accept(); //start waiting at port 1900
      //when a client is connected ss.accept() retruns
      //a socket that identifies the connection

      //creates input/output streams
      BufferedReader in = new BufferedReader(
           new InputStreamReader(socket.getInputStream()));

      PrintWriter out = new PrintWriter(
           new BufferedWriter(new OutputStreamWriter(
             socket.getOutputStream())),true);

      while(!line.equals("END")){
        line = in.readLine(); //reads the data from the client
        out.println("ECHO "+line); //sends data to the client
      }

    }catch(Exception e){e.printStackTrace();}
     finally{
      ss.close();
      if(socket!=null) socket.close();
     }
  }
}
```

The client program is shown in the following listing:

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

public class SimpleClient {

  public static void main(String[] args)throws Exception{
    Socket socket=null;
    try {
    //creating object address that identifies the server address
      InetAddress address =InetAddress.getByName("localhost");
    //the alternative could be used: InetAddress.getByName ("127.0.0.1")
   socket = new Socket(address,1900);
```

```
      BufferedReader in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Output is automatically flushed
      // by PrintWriter:
      PrintWriter out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())),true);

      for(int i = 0; i < 10; i ++) {
        out.println("message " + i);
        String str = in.readLine(); //send message
        System.out.println(str); //waits for response
      }
      out.println("END"); //sends a message that causes the server to close the connection

    }
    catch (Exception ex) {ex.printStackTrace();}
    finally{
      socket.close();
    }
  }
}
```

For verification, the server will start right after the client.

4. Port Forwarding

4.1. Network architecture and IP visibility

Most of the times, computer networks architectures are connections of several star-shaped sub-networks (LANs is an acronym for Local Area Networks), where the center of the star is usually a router (or server). This node (router) is the connection point of the sub-network (LAN) to the outside realm (internet) and handles all data transfer from outside to nodes (eg. computers) inside the sub-network and vice-versa. Each node in the network, and also the router itself, can have a firewall for protection.

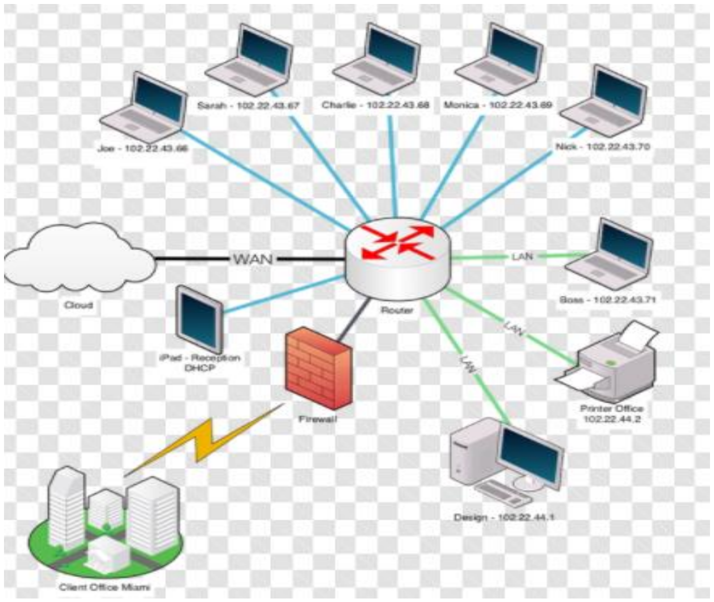


Fig 2. Network Architecture

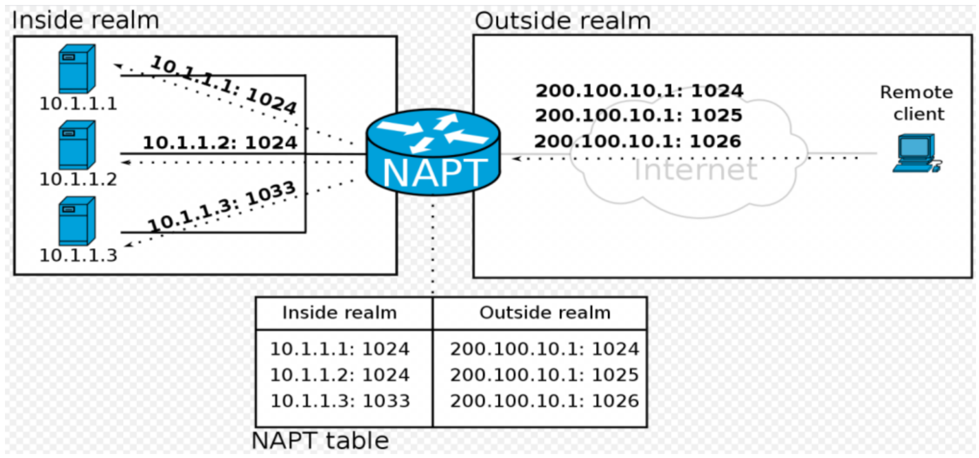
Every node inside the LAN is uniquely identified using the IP address, usually of the form 192.168.0.5, etc, which are dynamically allocated (can change when a node enters or leaves the system). In different LANs, the same IP can therefore correspond to different computers. From the LAN's perspective, the router usually has a fixed IP adress, usually 192.168.0.1. Nodes inside the LAN are not visible to the outside network.

Outside the LAN, there are static and dynamic IP addresses. The router itself has a dynamic IP address of the form 187.25.148.5 (Ipv4), which can be used to identify the router from outside the LAN.

Because nodes from the LAN are not visible outside (only the router is visible), TCP/IP connection fails because the server IP address cannot be identified.

4.2. Port forwarding

The router can be configured to redirect requests (or any data transfers) through one port to a certain node (computer) in the LAN. To the outside world, it is as if the router processes all the requests. This is called **port forwarding**, an application of NAT (Network address translation). In complex applications, several ports can be forwarded to different nodes in the LAN, as în the picture below, resulting a routing table of the form below. NAPT is an acronym for Network Address Port Translation.



Inside realm	Outside realm
10.1.1.1: 1024	200.100.10.1: 1024
10.1.1.2: 1024	200.100.10.1: 1025
10.1.1.3: 1033	200.100.10.1: 1026

Fig.3 NAPT

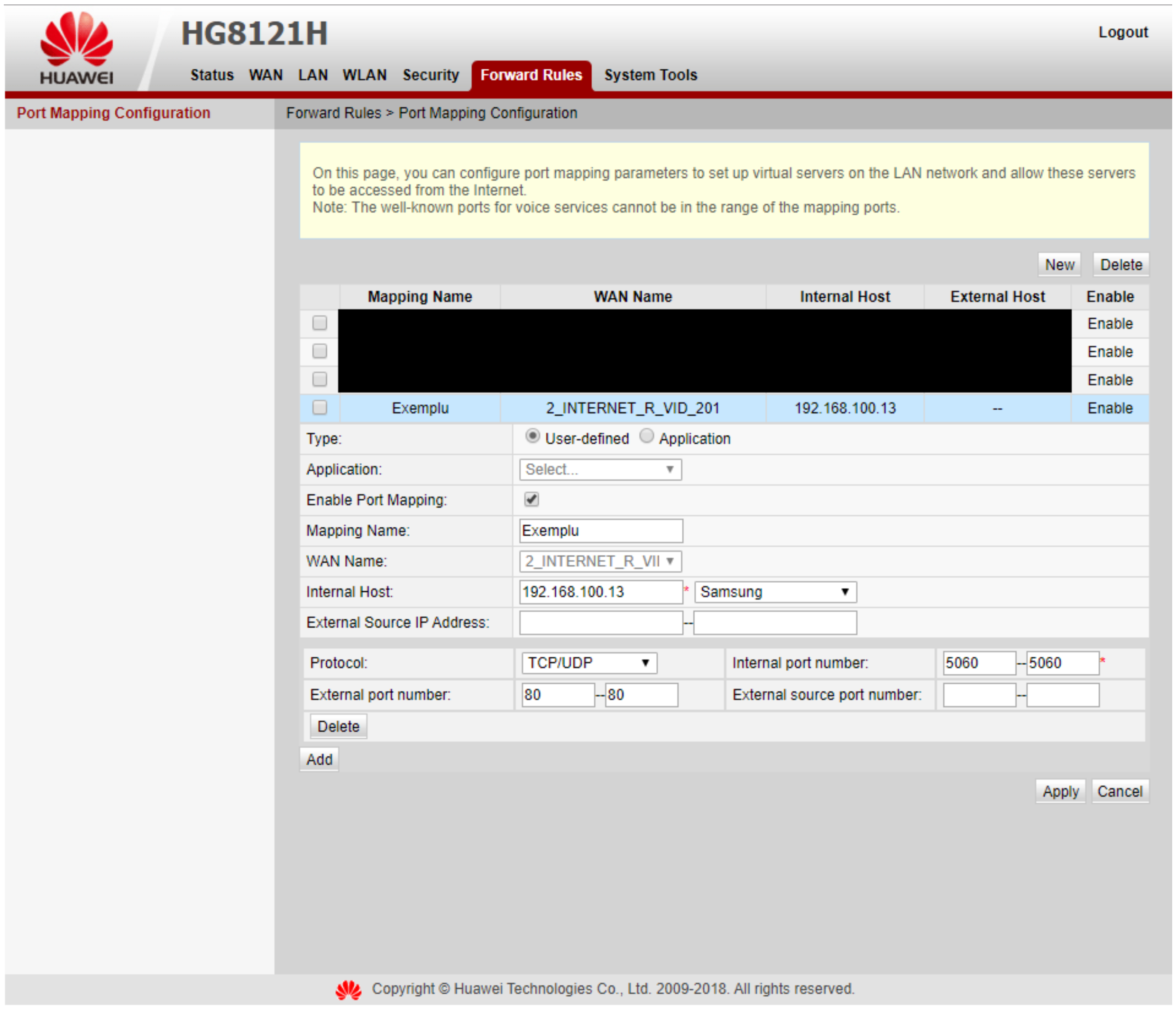
4.3. Configure port forwarding on router

Find the **router IP inside the LAN**: open **cmd**, type *ipconfig*, select the *Default Gateway*.
Find the **computer IP**: open **cmd**, type *ipconfig*, select *IPv4 Address*.
Find the **router IP as seen from outside the LAN**: use **https://whatismyipaddress.com/**

Access the configuration page of the router by typing the router IP inside the LAN in a browser.
The configuration page differs a bit for each router model, but the settings are similar!! Open the tab **Forward Rules -> Port Mapping Configuration**. Here, create a new rule with *Enable Port Mapping* checked and provide the following details:

Internal Host = the computer IP (you may want to select the host from the dropdown)
External Source IP Address = the outside computer IP which will have access (leave empty => any outside computer IP will have access). Leave empty for now.
Protocol = UTP/TCP
Internal port number = the port on your computer to which traffic will be forwarded (5060 for example)
External port number = the port on the router which will be used from outside (80 for example)

The TCP application will be configured such that:
1. the server application runs on the computer inside the LAN with IP = **computer** IP. The server will start listening on port *Internal port number* (5060 for example)
2. the client application will connect to the server using the **router IP as seen from outside the LAN**, using the port *External port number* (80 for example)


HG8121H
Logout
Status WAN LAN WLAN Security Forward Rules System Tools
Port Mapping Configuration
Forward Rules > Port Mapping Configuration
New Delete
Add
Apply Cancel

On this page, you can configure port mapping parameters to set up virtual servers on the LAN network and allow these servers to be accessed from the Internet.
Note: The well-known ports for voice services cannot be in the range of the mapping ports.

	Mapping Name	WAN Name	Internal Host	External Host	Enable
☐	[redacted]	[redacted]	[redacted]	[redacted]	Enable
☐	[redacted]	[redacted]	[redacted]	[redacted]	Enable
☐	[redacted]	[redacted]	[redacted]	[redacted]	Enable
☐	Exemplu	2_INTERNET_R_VID_201	192.168.100.13	--	Enable

Type:	[✓ User-defined] Application
Application:	Select...
Enable Port Mapping:	✓
Mapping Name:	Exemplu
WAN Name:	2_INTERNET_R_VII
Internal Host:	192.168.100.13 * Samsung
External Source IP Address:	--

Protocol:	TCP/UDP	Internal port number:	5060 -- 5060 *
External port number:	80 -- 80	External source port number:	--
Delete			

Copyright © Huawei Technologies Co., Ltd. 2009-2018. All rights reserved.

Fig 4. Router configuration example

NOTE1: It may be that some ports are blocked by your InternetServiceProvider. If this is the case, select other port numbers.
NOTE2: It may be that your firewall (or the firewall of the router) prevents communication. If this is the case, disable firewalls for now (or if you are an advanced user, set the TCP application as exception for the firewall)

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

public class NATClient {

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

```
		Socket socket = null;
		try {
			// create address object that identifies the server address
			InetAddress address = InetAddress.getByName("192.168.100.29");
			// the alternative could be used: InetAddress.getByName("127.0.0.1")
			socket = new Socket(address, 8081);
			BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
			PrintWriter out = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream())),
					true);
			out.println("sending trallala...");
			String line = "";
			while (!line.equals("END")) {
				line = in.readLine();
				System.out.println("received: " + line);
			}

			socket.close();
		} catch (Exception ex) {
			ex.printStackTrace();
		} finally {
			if (socket != null)
				socket.close();
		}
	}

}
```

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

public class NATServer {

	public static void main(String[] args) throws IOException {
		ServerSocket ss = null;
		Socket socket = null;

		try {
			ss = new ServerSocket(8081); // create the serversocket object
			socket = ss.accept(); // start waiting for the 5000 port
```

```
                        BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
                        PrintWriter out = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream())),true);

                        String receivedMessage = in.readLine();
                        System.out.println("received message: " + receivedMessage);
                        for (int i=0;i<10;i++) {
                                out.println("iaca: " + i);
                                Thread.sleep(100);
                        }

                        out.println("END");
                        socket.close();
                        ss.close();
                } catch (Exception e) {
                        e.printStackTrace();
                } finally {
                        ss.close();
                        if (socket != null)
                                socket.close();
                }
        }
}
```

5. Client-Server Application with a Multi-threaded Server

When analyzing the server program that is presented in the previous section, it can be seen that it can serve only one client at a time. In order for the server to be able to serve multiple clients simultaneously, multi-threaded programming will be used.

The basic idea is simple, that is, the server will wait for connections by calling the **accept()** method. When a client is connected on and the **accept()** method has returned a Socket, it will create a thread of execution that will serve that client, and the sever will return to waiting.

The next program presents a multi-thread server - capable of serving many customers simultaneously.

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

```
public class MultiThreadedServer
{
            public static final int PORT = 1900;
            void startServer()
            {
                        ServerSocket ss=null;
                        try
                        {
                        ss = new ServerSocket(PORT);
                        while (true)
                        {
                                    Socket socket = ss.accept();
                                    new TreatClient(socket).start();
                        }

                        }catch(IOException ex)
                        {
                                    System.err.println("Error :"+ex.getMessage());
                        }
                        finally
                        {
                                    try{ss.close();}catch(IOException ex2){}
                        }
            }

            public static void main(String args[])
            {
                        MultiThreadedServer smf = new MultiThreadedServer();
                        smf.startServer();
            }
 }

class TreatClient extends Thread
{
             private Socket socket;
             private BufferedReader in;
            private PrintWriter out;
             TreatClient(Socket socket)throws IOException
             {
                        this.socket = socket;
                   in = new BufferedReader(new InputStreamReader(socket.getInputStream()), true);
                              out  =  new  PrintWriter(new  BufferedWriter(new  OutputStreamWriter(
socket.getOutputStream())));
             }
```

```
  public void run()
   {
   try {
       while (true)
       {
         String str = in.readLine();
         if (str.equals("END")) break;
         System.out.println("Echoing: " + str);
         out.println(str);
        }//.while
    System.out.println("closing...");
    }
    catch(IOException e) {System.err.println("IO Exception");}
      finally {
           try {
              socket.close();
           }catch(IOException e) {System.err.println("Socket not closed");}
       }
  }//.run
}
```

6. Non-blocking sockets

The Java 2 Standard Edition 1.4 introduces a new network communication mechanism through the **non-blocking sockets** - they allow communication between applications without blocking the call processes for methods to open a connection, read, or write data.

The classic solution to build a server application that serves more clients is to use threads of execution technology and allocate a single thread of execution for each client served. By using the non-block sockets technology, the programmer will be able to deploy a server application for customer service **without having to explicitly call out threads of execution** to handle customer requests. Below are the basic principles of this technology and the way in which client applications and server applications can be built on non-blocking sockets technology.

The architecture of a system that uses non-blocking sockets for communication is illustrated in the following figure.

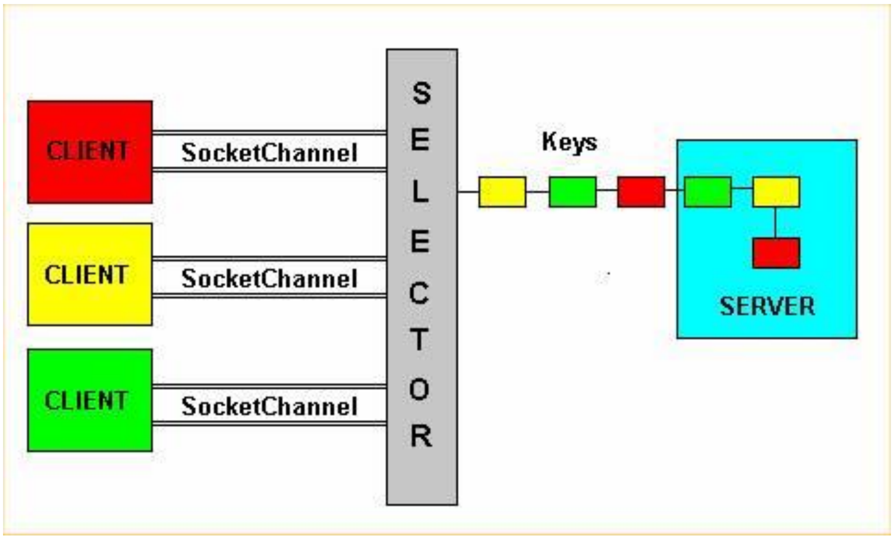


Fig. 5 Non-blocking sockets architecture

The implementation of the server consists of an infinite loop in which the selector waits for event generation. When an event occurred and a key was generated, the type of this key is checked. The possible types of **keys** are:

- **Acceptable** - associated with a request for a connection from a client event
- **Connectable** - associated with the connection acceptance by client event
- **Readable** - read data
- **Writeable** - write data

The **Selector** class is responsible for maintaining a set of keys that can be active while running the server program. When an event is generated by a client, a key is constructed.

Selector selector = Selector.open();

In order to demultiplex the data and have access to events, a communication channel must be built which will have to be registered in the selector object. Each registered channel will have to specify the type of event that it is interested in.

```
ServerSocketChannel channel = ServerSocketChannel.open();
channel.configureBlocking(false);
InetAddress lh = InetAddress.getLocalHost();
InetSocketAddress isa = new InetSocketAddress(lh, port );
channel.socket().bind(isa);

SelectionKey acceptKey = channel.register( selector, SelectionKey.OP_ACCEPT );
```

A channel that reads and writes data will be recorded as follows:

```
SelectionKey readWriteKey = channel.register( selector,
SelectionKey. OP_READ| SelectionKey. OP_WRITE );
```

The server application code that implements communication via non-blocking sockets is listed below:

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.channels.spi.*;
import java.nio.charset.*;
import java.net.*;
import java.util.*;

public class NonBlockingServer2 {

    public static void main(String[] args) throws Exception{

//          Create the server socket channel
        ServerSocketChannel server = ServerSocketChannel.open();
//          nonblocking I/O
        server.configureBlocking(false);
//          host-port 8000
        server.socket().bind(new java.net.InetSocketAddress("localhost",8000));
        System.out.println("Server waiting on port 8000");
//          Create the selector
        Selector selector = Selector.open();
//          Recording server to selector (type OP_ACCEPT)
        server.register(selector,SelectionKey.OP_ACCEPT);

//          Infinite server loop

        for(;;) {
        Thread.sleep(1000);
          // Waiting for events
          System.err.println("wait for event...");
          selector.select();

          // Get keys
          Set keys = selector.selectedKeys();
          Iterator i = keys.iterator();
          System.err.println("keys size="+keys.size());
          // For each keys...
          while(i.hasNext()) {
```

```
 // Obtain the interest of the key
SelectionKey key = (SelectionKey) i.next();

// Remove the current key
i.remove();


// if isAccetable = true
// then a client required a connection
if (key.isAcceptable()) {
  System.err.println("Key is of type acceptable");
  // get client socket channel
  SocketChannel client = server.accept();
  // Non Blocking I/O
  client.configureBlocking(false);
  // recording to the selector (reading)
  client.register(selector, SelectionKey.OP_READ);
  continue;
}

// if isReadable = true
// then the server is ready to read
if (key.isReadable()) {
  System.err.println("Key is of type readable");
  SocketChannel client = (SocketChannel) key.channel();

  // Read byte coming from the client
  int BUFFER_SIZE = 32;
  ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
  try {
    client.read(buffer);

  }
  catch (Exception e) {
    // client is no longer active
        client.close();

    e.printStackTrace();
    continue;
  }

  // Show bytes on the console
  buffer.flip();
  Charset charset=Charset.forName("ISO-8859-1");
```

```
                CharsetDecoder decoder = charset.newDecoder();
                CharBuffer charBuffer = decoder.decode(buffer);
                System.out.println(charBuffer.toString());
                continue;
               }


            }
            System.err.println("after while keys size="+keys.size());
           }
      }
}
```

The client application code that uses non-blocking sockets for communicating with a server is listed below:

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.channels.spi.*;
import java.nio.charset.*;
import java.net.*;
import java.util.*;

public class NonBlockingClient {

        public static void main(String[] args) throws IOException {

// Create client SocketChannel
          SocketChannel client = SocketChannel.open();

// nonblocking I/O
          client.configureBlocking(false);

// Connection to host port 8000
          client.connect(new java.net.InetSocketAddress("localhost",8000));

// Create selector
          Selector selector = Selector.open();

// Record to selector (OP_CONNECT type)
          SelectionKey
```

```
            clientKey = client.register(selector, SelectionKey.OP_CONNECT);

// Waiting for the connection
            while (selector.select(500)> 0) {
                System.err.println("Start communication...");

                // Get keys
                Set keys = selector.selectedKeys();
                Iterator i = keys.iterator();

                // For each key...
                while (i.hasNext()) {
                   SelectionKey key = (SelectionKey)i.next();

                   // Remove the current key
                   i.remove();

                   // Get the socket channel held by the key
                   SocketChannel channel = (SocketChannel)key.channel();

                    // Attempt a connection
                    if (key.isConnectable()) {

                    // Connection OK
                    System.out.println("Server Found");

                // Close pendent connections
                if (channel.isConnectionPending())
                  channel.finishConnect();

                // Write continuously on the buffer
                ByteBuffer buffer = null;
                int x=0;
                for (;x<7;) {
                      x++;
                  buffer =
                    ByteBuffer.wrap(
                      new String(" Client " + x + " "+x).getBytes());
                  channel.write(buffer);
                  buffer.clear();
                  try {Thread.sleep(2000);}
                  catch (InterruptedException e) {e.printStackTrace();}
            }
            channel.finishConnect();
            client.close();
          }
        }
      }
      System.err.println("Client terminated.");
    }
}
```

Exercises

Import the Eclipse environment into the project that exemplifies the concepts presented in this laboratory (proiect link).

The **lab.scd.socket** package exemplifies the construction of server and client applications that communicate using the TCP protocol.

1)

- Run the ServerSimplu application
- Without shutting down the ServerSimplu application, run the ClientSimplu application

2)

- Stop the ServerSimple and ClientSimple applications
- Launch the ServerMultifir application
- Without stopping the ServerMultifir application, launch two or more instances of ClientSimplu applications

The **lab.scd.datagram** package exemplifies the construction of server and client applications that communicate using the UDP protocol.

- Run the MulticastServer application
- Without stopping the MulticastServer application, run the MulticastClient application

The **lab.scd.broadcast** package exemplifies the construction of server and client applications that communicate using the UDP protocol.

- Launch the QuoteServerThread application
- Without quitting the QuoteServerThread application, run the QuoteClient application

The **lab.scd.seriering** package exemplifies the mechanism for serializing and deserializing objects.

The **lab.scd.net.httpexample** package shows how an HTTP server can be constructed in java. The example application partially implements the HTTP protocol, responding to GET requests. Analyze the way the application was built, run and test the application.

The **lab.scd.net.url_http** package shows the classes in java.net that can be used to work with URLs.

The **lab.scd.net.browser** package describes how an html page browser can be constructed in java.