FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Electrical Engineering Department

Telecommunication Systems

2012/2013

Laboratory Work 1:
Application using sockets TCP and UDP

Integrated Master in Electrical Engineering and Computers

http://tele1.dee.fct.unl.pt

Index

1. Objective

Familiarization with the use of sockets for communication between machines and the functioning of the programming interface of Java sockets.

The work consists in developing a system for transferring files with an initial registration requests.

> **Suggestions**: In certain parts of this document appears some text formatted differently that begins with the word "Suggestion". It is not mandatory to follow what is written there, but may be important for students or groups where there is not yet at ease with programming, data structures and algorithms.

2. Specifications

It is intended to develop a file transfer service with prior registration of the clients. The transfer system behaves as a portal for clients because they have to know only one address and access port. The keys service and file service access ports are not publicly known.

The general idea is as follows: clients access the registration server (trader) for the address and port of the keys server and then, when accessing the keys server, they get a key that will be submitted when accessing to any service registered in the portal. In this work, the portal only has one service – the file service. Before accessing the file service, the clients go again to the registration service to obtain the file service's port. In the possession of the key along with the file server's port, the clients access the file service to transfer the file. In response to the request, the client receives information concerning the length of the file (0 means empty file and -1 means inexistent) and the content of the file. The key expires after **one minute** if not used for a request to the file server. It is always necessary to ask for one key for each file transferred, even if the file transfer fails because the file does not exist.

The structure of the work consists of a server block (process) and a client block (process). The server block has the following components:

- Registration server (*trader*) – It is always waiting for requests from clients to one of the two other services. It answers to each valid request with the server port and each invalid request by stating "invalid".
- Keys server – Receives an enrolment request and returns a valid key for a transfer, which has to be performed in less than one minute.
- File server – Receives a request to transfer binary files, in which the parameters are the file name and key. If the key is not valid, or if the file does not exist returns a length equal to or below 0. Otherwise returns the file contents.

The client block runs the procedures to transfer a file.

2.1. Message Flow

The message sequence is shown in Figure 1, and consists of the following messages (assuming that no errors exist of the type of invalid service, invalid keys, etc.):

(1) – The client sends a request message to the registration server for the keys service port;
(2) – The registration server answers with the socket address of the keys server;
(3) – The client requests a key to keys service;
(4) – The keys service generates a key and sends it to the client. The time of one minute starts counting from the moment and ends when the client accesses to the file server;
(5) – The client sends a request message to the registration server for the file service port;

(6) – The server answers with the socket address of the file server;
(7) – The client proceeds to transfer the file:
- (7_a) – The client connects to the file server;
- (7_b) – Initially the client sends two lines of text across the TCP connection with the key and file name;
- (7_c-7_{n-1}) – The file server answers with the file size which can take values of zero or negative to indicate errors. In case there is the requested file, the server sends the file;
- (7_n) – The server closes the connection.

The communication with the registration server and keys server is done using *datagram sockets*. *Connection-oriented sockets* are used for communication with the file service.

The steps 7c - 7n correspond to the transmission of the length and data of the file. No one knows for sure how many packets will be received because the connection-oriented socket is a stream of bytes and not of packets (for students to better realize this difference data must be written on the socket using blocks of 1000 bytes at a time and the client must read blocks of 2000 bytes each).

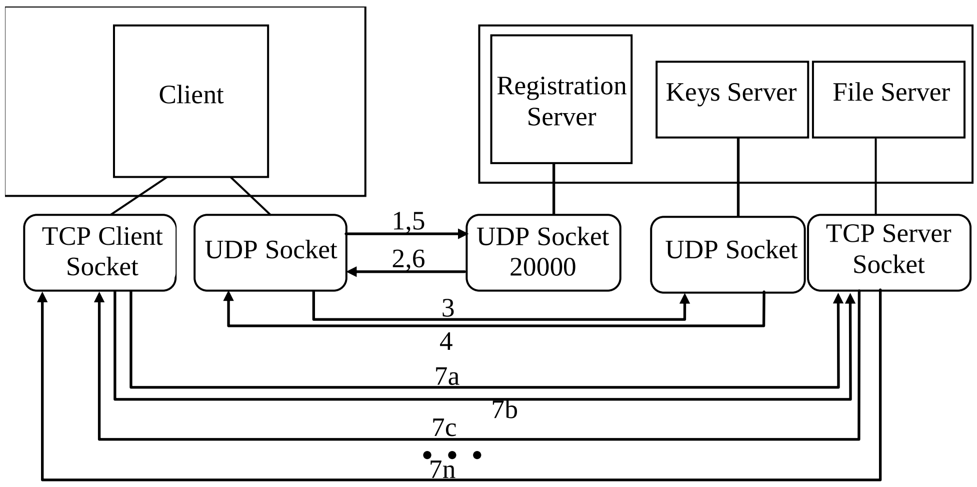
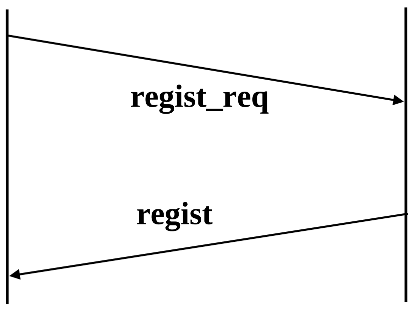


Fig. 1 – Sequence of messages exchanged

2.2. REGISTRATION SERVICE

The access to registration service uses datagram sockets. The access is represented by messages 1,2 and 5,6 of Fig. 1 and obey to the following request-response protocol:



```
Message regist_req:// sequence of
 int     port;  // UDP port of the client
 short codServ;//code of the requested service
   /* keys service code = 15 */
   /* file service code = 31 */


Message regist: // sequence of
 boolean validSer; // validity of service
   /*  false - invalid service */
   /*  true  - valid service   */
 int     portSer; // server's port
```

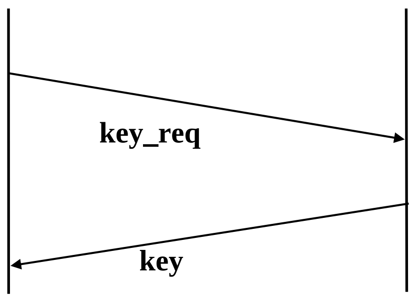
Messages *regist_req* and *regist* have the two parameters represented. If the port in *regist_req* does not match the port of the sender's UDP socket, a "non-existent service" should be returned.

As mentioned, the port of the registration server's socket is the only known. It is customary to call this a *well-known socket* (the IP address is associated to the machine; what is really known is the port). All other sockets must be free ports since their values are supplied by the various components. The known port of the first server running is **20020**. If more than one server runs on the same machine, they will have the port numbers 20021, 20022, etc.

Suggestion: The registration server must know what is the port of the datagram socket associated with the keys service and with the stream socket of the file server. The ports can be obtained using the method `getLocalPort` of the objects associated of type `DatagramSocket` and `ServerSocket`.

2.3. KEYS SERVICE

Access to keys service uses datagram sockets and is exemplified by messages 3 and 4, which obey to the protocol:



```
Message key_req: //sequence of
 int port;  // UDP Port of the client


Message key:  //sequence of
 short  key_len; // number of byte of key
 byte[] key;     // key
```

The client sends the request using the port that was given by the registration server. The response contains the key.

Suggestion 1 – keys creation

To simplify the work, the keys server can return keys that are initially the value of an integer variable incremented (perhaps starting a non-zero value). If you have time, you should replace this original form by random keys, for example, constructed from a random number (see code fragment below). In this case, you must ensure that keys that exist at a given moment are unique, and that you do not have two equal keys active at any given time.

```
private Random keygen= new Random();
int number= keygen.nextInt(1000000);         // Random number between 0 and 999999
```

Suggestion 2 – Several active clients

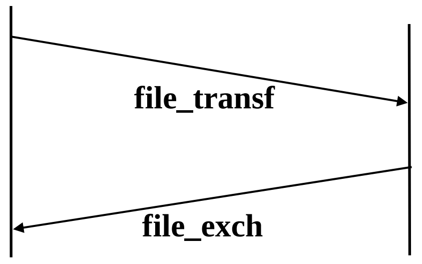
Initially you can admit that there is at most one client active in every moment (if you think that it simplifies programming). Should you have time after the work, you should consider supporting multiple clients in parallel. In this case, when the keys server generates a key for a customer, it must put it accessible to the file server. It is recommended to use a list to store the allocated keys and to manage the expiration time of the keys. When there is a file transfer, or when time passes, the keys must be removed from the list. To save the key, students must create a new class, with all the data that they consider relevant to manage time.

Suggestion 3 – Keys removal from list

After a minute, the key not used is invalid and must be removed from the list. This operation can be performed by a timer, but cannot be held with only one passage on the list, because the iterator used blocks any removal of list elements. It is necessary to allocate an auxiliary list or array of variable size for storing outdated keys, which could be removed only at the end.

2.4. File service

Access to the service file transfer must use connection-oriented sockets and obey the protocol:



```
file_transf : //sequence of
 String key + "\n";      // Key
 String filename + "\n"; // File name

file_exch : //sequence of
 String file_length +"\n";
   /* -2 - invalid key        */
   /* -1 - inexistent file    */
   /*  0 - empty file         */
   /*  file length            */
 contents /* sequence of bytes with
             file contents */
```

The communication of messages *file_transf* and *file_exch* will be accomplished through the exchange of strings delimited by line breaks. The exception is the content of the files to be transmitted in binary format without modifying the transmitted data. **Do not forget** that the server sends, in each cycle of transmission, only 1000 bytes of the file (if you send the whole file in a single writing will be penalized in the final evaluation of the work). In the end, the server closes the connection. If you decide to try to support multiple clients in parallel, remember you need to have a thread for each client, which should keep all relevant parameters for that connection.

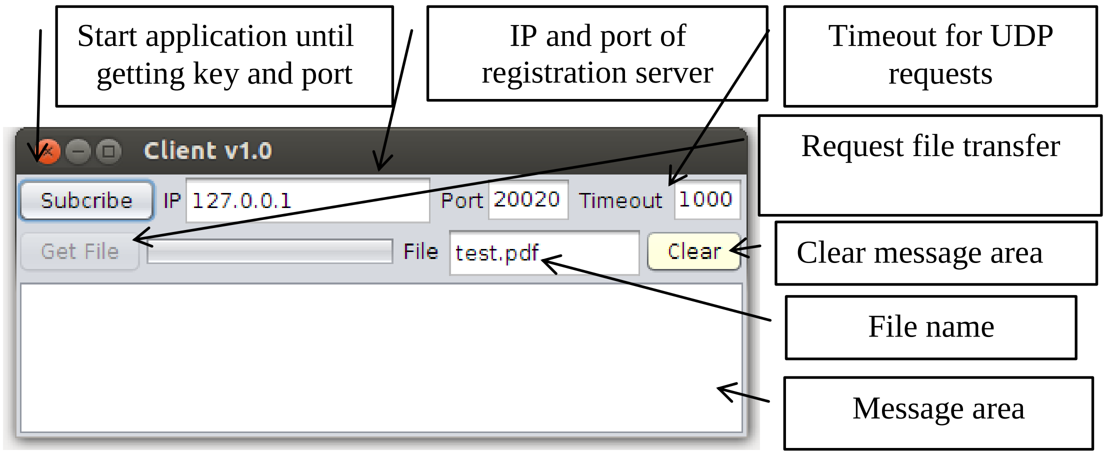
Suggestion: Do not forget that you should not under any circumstances block the main thread (linked to the graphical interface). Any blocking operation (e.g. read or write the socket) should be performed within a thread.

3. Program development

3.1. Client

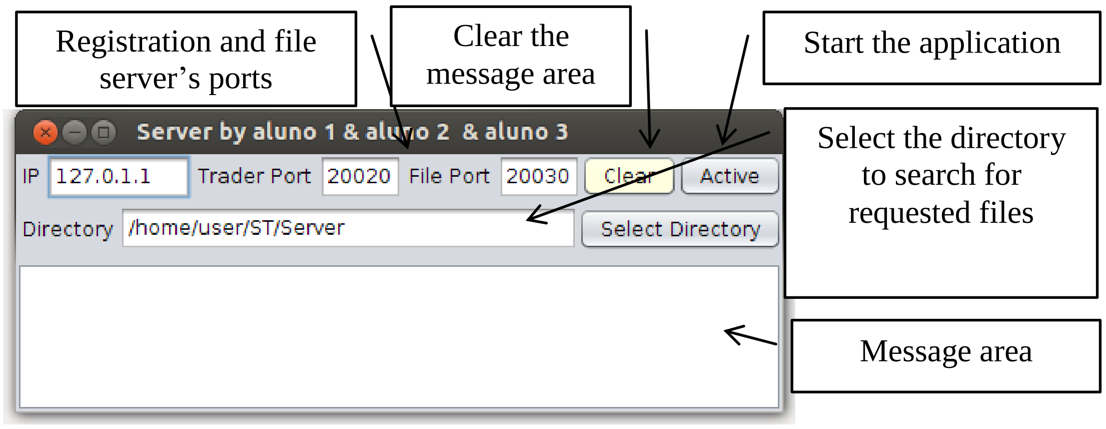
The client application is provided entirely implemented together with the statement of work. A Java application was developed with the graphical interface shown below. After pressing button "*Subscribe*" the client performs exchanges of messages between 1 and 6 depicted in Figure 1. Thus, the program is ready to communicate with the file server, getting unlocked the button "*Get File*". When pressed, exchanges of messages shown as 7 in Figure 1 take place. During the file transfer, the bar illustrates the evolution of the reception of data from the file. At any time you can abort a transfer by pressing the "*Subscribe*" button.

You can run the client from the terminal using the command: `java –jar Client.jar`



3.2. Server

The work consists of making the server block. To facilitate the development of the program and make possible the development of the program in the four classes provided, it is supplied an incomplete Server program, with the graphical interface shown below, which already performs part of the functionality requested. Each group can make all the changes you want to the base program, or even draw a GUI from scratch. However, we recommend that you invest the time in the correct implementation of the proposed protocols.



The program provided is composed by four classes:

- *Daemon_udp.java* (complete) – Thread that receives datagram packets;
- *Daemon_tcp.java* (complete) – Thread that receives connections for the file service;
- *SConn_tcp.java* (**to be completed**) – Thread that handles a connection of the file service;
- *Server.java* (**to be completed**) – Main class with the GUI, which manages and synchronizes the objects used in the project.

The source code supplied reuses with minor modifications the *Daemon_tcp* and *Daemon_udp* classes that were used in the laboratory work 0 and suggests two incomplete classes which support user interface (*Server*) and the communication in a TCP connection.

Class *Daemon_tcp* was already used in the third class of laboratory work 0.

The class *Daemon_udp* of second class of laboratory work 0 was modified in this project to allow being used with several UDP sockets in parallel. When you create an instance of this class, you define a unique type (a number). When it receives a packet, the method called (`receive_packet`) takes as its first parameter the value of the type. Therefore, several objects of this class can coexist as long as their types are different.

```
public class Daemon_udp extends Thread {
    Server root;                        // Main window object
    DatagramSocket ds;                  // datagram socket
    int type;                           // socket type

    public Daemon_udp(Server root, DatagramSocket ds, int type);

    public void run();  // UDP socket thread code
     // Calls root.receive_packet(type, dp, dis);
     //   everytime a new packet is received

    public void stopRunning();  // Interrupt a running thread
}
```

Class *SConn_tcp* defines a thread but is almost empty. The task of programming it was left the students. In the end, before finishing, the thread must call the `root.connection_thread_ended` informing the end of the connection.

The *Server* class is the main class. It includes the definition of the graphical interface, the start of the socket of the registration service (*trader*) and the TCP socket. It also includes the code to choose the directory to read files, and a set of empty callbacks, which should be done by the students throughout the work.

```
public class Server extends javax.swing.JFrame {
  public final static int MAX_MLENGTH;   // Maximum UDP message size
  public final static long KEY_VALIDITY; // Key validity (ms)

  /* Types of Service in Registration service */
  public final static byte KEY_SERVICE;  // Key service
  public final static byte FILE_SERVICE; // File Service

  public final static int TRADER_SOCKET = 101;  // Type of trader socket

  public void Log(String text); // Log function

  // Handle "Active" toggle button
  private void jToggleActiveActionPerformed(java.awt.event.ActionEvent evt);

  private void jButtonClearActionPerformed(java.awt.event.ActionEvent evt):

  public void close_all(); // Close all sockets and threads

  /* Logs an error message associated to the reception of a packet */
  public void Log_err(DatagramPacket dp, String type, String err);

  // Handle trader request - called by 'receive_packet'
  public void handle_trader_packet(DatagramPacket dp, DataInputStream dis);

  // Handle an UDP packet of type 'type'
  public void receive_packet(int type, DatagramPacket dp, DataInputStream dis);

  public File get_fileref(String filename); // Return the file descriptor associated
                                            //      to a filename

  SConn_tcp start_connection_thread(Socket s); // Handle TCP connections received;

  void connection_thread_ended(SConn_tcp th);  // Handle end of TCP Thread

// Variables declared
  private DatagramSocket tradersock;  // UDP socket of the trader service
  private Daemon_udp daemon_trader;  // Thread of the trader service
  private ServerSocket ss;  // TCP socket of the file service
  private Daemon_tcp daemon_tcp;  // Thread of the file service
  public SimpleDateFormat formatter;  // Formatter for dates
}
```

3.3. Goals

A sequence to implement the program Server can be:

1. Program the start of the socket and the thread of the keys service in the method `jToggleActiveActionPerformed` of class `Server`. You should also program the method `close_all` to close them;
2. Program the method that prepares the reply to requests to the registration service, reading and validating all fields in the request and preparing the reply;
3. Program the method that handles the requests to the keys service, reading and validating the fields of the request, and preparing a reply. In this first implementation you can return the value of an integer value incremented linearly;
4. Program the method `start_connection_thread` of class `Server` such that an object of class `SConn_tcp` is created and the thread associated is started. If you don't feel comfortable using threads, in a first approach you can prepare the program to have only one active thread, postponing the support of multiple threads;
5. Program the class `SConn_tcp` to implement the file service protocol (do not forget to send the file in blocks, to avoid penalizations). Program the method `connection_thread_ended` of class `Server`;
6. Program the support for multiples concurrent clients, including the possibility to stop all threads, in method `close_all`;
7. Implement the mechanism to validate the keys received in the file service, using random values. It is suggested that an auxiliary class is created, to store the information about the key and its validity. This class can then be used to create the list, with all allocated keys.
8. Implement an automatic cleaning mechanism to clear the list of key, supported by a timer that runs periodically.

ALL students must try to **complete phase 5**. In the first week of the work a general introduction to the work is made and phase 2 should be finished. At the end of the second week the phase 5 must have started. At the end of the third week you should have started phase 7. At the end of the fourth and final week you should try to achieve the maximum number of phases, taking into account that it is preferable to do less well (running and no errors), than everything and nothing works.

Phases 2-3 and 4-5 can be implemented concurrently for groups with two or three elements, shortening the work's implementation time.

Student Posture

Each group should consider the following:

- Do not waste time with the aesthetics of input and output data;
- Program in accordance with the general principles of good coding (using indentation for comments, using variables with names conform to its functions ...) and;
- Proceed so that the work is equally distributed to the two members of the group.