DSMWare Laboratory 1: Simple Client-Server with Sockets

Alberto ARDUSSO
ardusso@eurecom.fr

Last change: Thursday 17th March, 2016 at 03:15

Contents

1 Development of a tiny web server

1.1 Your first server using sockets

I refer to Appendix A for my implementation of the tiny web server.

1.2 Analyzing client requests

Analyzing the output of my program I noticed that the browser by default send two HTTP messages of type GET. One for the root of the website and the second for favicon.ico

1. request of the root page of the website and header of the answer

```
GET / HTTP/1.1
Host: 127.0.0.1:7777
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like
    Gecko) Chrome/49.0.2623.87 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: it-IT,it;q=0.8,en-US;q=0.6,en;q=0.4

##############################

HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Request Headers
view source
```

Listing 1: HTTP request of the default file (index file)

2. request of `favicon.ico` and header of the answer

```
GET /favicon.ico HTTP/1.1
Host: 127.0.0.1:7777
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like
    Gecko) Chrome/49.0.2623.87 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: it-IT,it;q=0.8,en-US;q=0.6,en;q=0.4

##########################

HTTP/1.1 404 NOT FOUND
Connection: close
Content-Type: text/html
```

Listing 2: HTTP request of the favicon.ico

1.3 Enforcing access control

In the case of a single client the simplest way to solve the problem is to keep track of the state in which the client is with an attribute. In my implementation in my code there is the variable `state` of type `int`. `state` can assume values 0, 1, 2. After each request the state is updated; a request is satisfied only if `state` has the requested value.
In a more complex scenario in which more clients exist a solution could be to implement a class `Client` whose identifier is the ip address or, better, a nonce delivered to the client by the cookies mechanism.

1.4 Managing multiple clients

For this question I implemented a sketch of the cookie schema. The server sets a cookie on the client via the header `Set-Cookie: state=1`. Then the protocol sends it back during the next requests so that

the server can parse it and get again the information about the state of the client. This solution is not secure because is very simple for the client tamper with the value of the cookie. A better way to make i work could be the one explained in the section before, that is assign to each client a nonce very difficult to guess and keep the information about the state in the server. Then would also be better to encrypt the communication to avoid impersonification.

2 Implementation of an FTP client

2.1 About my implementation

I implemented my FTP client using the State Design Pattern, that well adapts at programs with the structure of a Finite State Machine. The main class `FTPClient.java` contains an object, instance of the class `ProtocolIterpreter.java`. In the main method a loop is performed on the functions of the `ProtocolIterpreter` object. This object contains the information relative to the state in which the protocol is, along with the socket objects and others. When one of the methods that needs to change behaviour depending on the state is called, the real implementation of the function is executed in its place. Here there is an example.

```
1  package Protocol;
2
3  import java.io.*;
4  import java.net.*;
5
6  enum StateEnum {
7    CONNECTED,
8    AUTHSENDUSER,
9    AUTHSENDPASSWD,
10   READY,
11   EXECUTING,
12   EXITING,
13   ERROR;
14 }
15
16 public class ProtocolInterpreter {
17
18   ...
19
20   protected State[] states = {
21       new State.Connected(),
22       new State.AuthSendUser(),
23       new State.AuthSendPasswd(),
24       new State.Ready(),
25       new State.Executing(),
26       new State.Exiting(),
27       new State.Error()
28   };
29   protected StateEnum currentState;
30   ...
31
32
33   //Should change the current state
34   public String interpretServerResponse() throws Exception{
35     String toShowToUser = states[currentState.ordinal()].interpretServerResponse(this);
36     lastResponse = null;
37     return toShowToUser;
38   }
39
40   ...
41 }
```

Listing 3: Call of a method in the class ProtocolInterpreter

The real implementation of the methods are in the classes derived from `State` In this case the states are represented by subclasses of a State class. In each of these states the methods that change behaviour are overridden.

2.2 Connecting to the server

Here there is the example of how work the connection to an FTP server by an FTP client with command lie interface. The lines with the prompt `ftp>` are written by the user, the lines that start with `--->` are the messages sent toward the server and the other lines are the answers. I used a local server on my computer because ftp.eurecom.fr is not accessible from outside eurecom.

```
1  ftp> open 127.0.0.1 2121
2  Connesso a 127.0.0.1.
3  220 pyftpdlib 1.5.0 ready.
4  ---> OPTS UTF8 ON
5  530 Log in with USER and PASS first.
6  Utente (127.0.0.1:(none)): anonymous
7  ---> USER anonymous
8  331 Username ok, send password.
9  Password:
10 ---> PASS alberto
11 230 Login successful.
12 ftp> cd Documents
13 ---> CWD Documents
14 250 "/Documents" is the current directory.
15 ftp>
```

Listing 4: HTTP request of the favicon.ico

And here is the output of my program:

```
1  220 pyftpdlib 1.5.0 ready.
2
3  220 pyftpdlib 1.5.0 ready.
4
5  Connection success
6  Username: anonymous
7  331 Username ok, send password.
8
9  331 Username ok, send password.
10
11 Password: hgfd
12 230 Login successful.
13 230 Login successful.
14
15 User has been authenticated
```

Listing 5: HTTP request of the favicon.ico

2.3 Data exchange

The File Transfer Protocol involves two different channel: the Control Channel and the Data Channel. In the first commands from the client and response from the server are delivered; this is created by the client when first connect to the server (usually on port 21). The second is used to transmit the data (content of files or result of other commands such as LIST). The peculiarity is that the Data Channel is (by default) established by the server that try to connect the client.

2.3.1 PORT command

The PORT command is necessary for the client to tell the server in which port he's accepting the connection for the Data Channel. The format of the command is

```
PORT ip[0],ip[1],ip[2],ip[3],port[0],port[1]
```

where the square brackets select respectively the bytes of the ip address and the tcp port in network order.

2.3.2 LIST command

After sending the PORT command the server tries to connect on the port specified. So before sending the PORT command the client has to bind a socket (`ServerSocket` in Java) that will accept connection. Then there are two solution:

1. Run another thread that will wait for connection on the socket for the data channel and will manage the data received
2. Do not execute immediately the `accept()` method on the socket, but send the LIST (or one other command that uses the data channel) and only after a positive response from the server (code 125 for example) wait for a connection. When the transfer is completed the protocol can continue and the socket is closed.

```
1  ls - list the content of current directory
2  cd [directory] - changes the current directory to [directory]
3  get [file] - download [file]
4  put [file] - upload [file]
5  exit - close the connection and exit
6  ? -
7  ls
8  --->PORT 192,168,0,26,254,110
9
10
11 <---200 Active data connection established.
12
13 PORT command succeded
14 Preparing for download
15 --->LIST
16 <---125 Data connection already open. Transfer starting.
17
18 Starting to download...
19 drwxrwxrwx   1 owner    group        4096 Mar 11 10:37 .eclipse
20 drwxrwxrwx   1 owner    group           0 Mar 08 09:36 AppData
21 dr-xr-xr-x   1 owner    group           0 Mar 11 13:19 Contacts
22 drwxrwxrwx   1 owner    group           0 Mar 08 09:36 Cookies
23 drwxrwxrwx   1 owner    group           0 Mar 08 09:36 Dati applicazioni
24 dr-xr-xr-x   1 owner    group        4096 Mar 11 13:25 Desktop
25 drwxrwxrwx   1 owner    group           0 Mar 08 09:36 Documenti
26 ...
27 drwxrwxrwx   1 owner    group        4096 Mar 16 23:02 workspace
28 File successfully downloaded
29
30 <---226 Transfer complete.
```

Listing 6: Communication for the commands PORT and LIST

In the listing the lines that starts with "$---->$" are messages from the client, with "$<---$" are from the server and the others are console messages.

2.3.3 PASV command

In environment in which the client cannot be reached by TCP connections (NAT, Firewall,...) the only way to exchange data with this protocol is use the passive mode. With this command the client asks the server to provide another socket waiting for connection, so that revert the precedent behaviour.

2.4 RETR command

The difference from this command and LIST command is that the result will be written in a file. There is then some problems for the managing of binary data. The default encoding used in an FTP connection is ASCII (7 bits). We need then to change the encoding to download binary file with the command:

TYPE [A,E,I]

Where A is ASCII type, E is EBCDCI text and I stands for IMAGE (pure binary).

Appendices

```
1  //HttpServer.java
2
3  import java.net.*;
4  import java.util.*;
5  import java.io.*;
6
7
8  class HttpServer {
9    PrintStream ps;
10
11   int state = 0;
12   HashMap<InetAddress,Integer> clients;
13   int i, j;
14
15   public void initWebServer(int port) {
16     try {
17       // byte[] buf = new byte[1000];
18       int i,j;
19       // InputStream is;
20       BufferedReader bis;
21       String page_requested = "/"; // page requested ("/" by default)
22       ServerSocket master_sock = null;
23       InetAddress currentAddr;
24       try {
25         master_sock = new ServerSocket(port);
26       } catch (Exception e) {
27         System.err.println(e);
28         e.printStackTrace();
29       }
30       Socket socket;
31       System.err.println("Web Server started");
32       String requestLine;
33       Integer currentState;
34       //Instantiate clients
35       clients = new HashMap<InetAddress,Integer>(20);
36
37       while (true) {
38         page_requested = "/";
39         socket = master_sock.accept();
40
41         currentAddr = socket.getInetAddress();
42         currentState = clients.get(currentAddr);
43         if(currentState == null)
44           currentState = new Integer(0);
45         System.err
46             .println("-----------------------------------------------");
47
48         bis = new BufferedReader(new InputStreamReader(
49             socket.getInputStream()));
50         ps = new PrintStream(socket.getOutputStream());
51
52         // display request contents
53         System.err.println();
54         System.err.println("REQUEST:");
55         System.err.println("--------");
56         try {
57           // Request
58           requestLine = bis.readLine();
59           System.err.println("First line "
60               + requestLine);
61           // Parse request
62           if(requestLine==null){
63             continue;
64           }
65           i = requestLine.indexOf("GET");
66           j = requestLine.indexOf("HTTP");
67           if (i == -1) {
68             System.err.println("Invalid request type!"
69                 + requestLine);
70             System.exit(-1);
```

```
71            }
72            i += 3;
73            String page = requestLine.substring(i,j);
74            page = page.trim();
75            if (page.equals(page_requested)) {
76              page = "/page1";
77            }
78            page_requested = page;
79            System.err.println("Page requested: "
80                + page_requested);
81            while ((requestLine = bis.readLine()) != null) {
82              // Headers (Ignored);
83              System.err.print(requestLine);
84              if(requestLine.length() <= 2)
85                break;
86            }
87          } catch (IOException e) {
88            ;// Possible that the request is finished
89          }
90          System.err.println();
91          System.err.println("writing back page:" + page_requested);
92          sendHeader();
93          displayPage(page_requested,currentAddr,currentState);
94          socket.close();
95        }
96      } catch (Exception e) {
97        System.err.println(e);
98        e.printStackTrace();
99      }
100   }
101
102   public void sendHeader() {
103     System.err.println("Sending header");
104     ps.print("HTTP/1.1 200 OK\r\n");
105     ps.print("Connection:\t close\r\n");
106     ps.print("Content-Type: text/html\r\n");
107     ps.print("\r\n");
108     System.err.println("Header sent");
109   }
110
111   public void sendError() {
112     ps.print("HTTP/1.1 404 NOT FOUND\r\n");
113     ps.print("Connection:\t close\r\n");
114     ps.print("Content-Type: text/html\r\n");
115     ps.print("\r\n\r\n");
116     ps.println("<HTML>\n<title>404 - Not Found</title>");
117     ps.println("<H1>Ahhhhhh</H1>");
118     ps.println("<P>There's not such a page :P");
119     ps.println("</P>");
120     ps.println("");
121     ps.println("<HR>\n</HTML>");
122   }
123
124   public void displayPage(String name, InetAddress current, Integer state) {
125     if(name.equals("/page1")){
126       clients.put(current,1);
127       displayPage1();
128     }else if(name.equals("/page2")){
129       if(state == 1){
130         clients.put(current,2);
131         displayPage2();
132       }else{
133         displayPage2NotAuthorized();
134       }
135     }else if(name.equals("/page3")){
136       if(state == 2){
137         clients.put(current,0);
138         displayPage3();
139       }else{
140         displayPage3NotAuthorized();
141       }
142     }else{
143       sendError();
```

```
144     }
145
146   }
147   public void displayPage1() {
148     state = 1;
149     System.err.println("Sending page");
150     ps.println("<HTML>\n<title>Java Socket Web Server Page 1</title>");
151     ps.println("<H1>Java Socket Web Server - Welcome to page 1</H1>");
152     ps.println("<P>This server is powered by Java Sockets.");
153     ps.println("This is not so neat, but not so big either</P>");
154     ps.println("Want another cup of java? Click <A HREF=\"page2\">here</A>");
155     ps.println("<HR>\n</HTML>");
156     System.err.println("Page sent");
157   }
158   public void displayPage2() {
159     state = 2;
160     ps.println("<HTML>\n<title>Java Socket Web Server Page 2</title>");
161     ps.println("<H1>Java Socket Web Server - Welcome to page 2</H1>");
162     ps.println("<P>This server is powered by Java Sockets.");
163     ps.println("This page is nearly the same as page 1, why bother?</P>");
164     ps.println("Go to <A HREF=\"page3\">page 3</A>");
165     ps.println("<HR>\n</HTML>");
166   }
167   public void displayPage3() {
168     state = 0;
169     ps.println("<HTML>\n<title>Java Socket Web Server Page 3</title>");
170     ps.println("<H1>Java Socket Web Server - Welcome to page 3</H1>");
171     ps.println("Page 3 at last !!!</P>");
172     ps.println("<HR>\n</HTML>");
173   }
174   public void displayPage2NotAuthorized() {
175     ps.println("<HTML>\n<title>Error !!</title>");
176     ps.println("<H1>Page 2 cannot be accessed directly</H1>");
177     ps.println("<H1>You must read page 1 !</H1>");
178   }
179   public void displayPage3NotAuthorized() {
180     ps.println("<HTML>\n<title>Error !!</title>");
181     ps.println("<H1>Page 3 cannot be accessed directly</H1>");
182     ps.println("<H1>You must read page 1 and 2 first !</H1>");
183   }
184 }
```

Listing 7: Implementation of the HTTP Server (HttpServer.java)

```
1  package FTPClient;
2  import java.io.*;
3  import java.net.*;
4
5  import Protocol.*;
6
7  public class Client {
8
9    public static void main(String[] args) {
10     try {
11       if (args.length < 2) {
12         System.out.println("Usage: java ip port");
13         return;
14       }
15
16       InetAddress remote = InetAddress.getByName(args[0]);
17       ProtocolInterpreter PI = new ProtocolInterpreter(remote,Integer.parseInt(args[1]))
     ;
18       BufferedReader userInputStream = new BufferedReader(new InputStreamReader(System.
     in));
19       String recv;
20       String string, fromUser;
21
22       PI.connect();
23       recv = PI.receiveMessage();
24       System.err.println(recv);
25       string = PI.interpretServerResponse();
26       if(string != null)
27         System.out.print(string);
```

```
28
29        while(PI.alive){
30          string = PI.messageForUser();
31          if(string != null)
32            System.out.print(string);
33          if(PI.needUserInput()){
34            fromUser = userInputStream.readLine();
35            while(userInputStream.ready()){
36              userInputStream.read();
37            }
38            PI.userInputEval(fromUser);
39          }
40          PI.sendMessageToServer();
41          recv = PI.receiveMessage();
42          System.err.println(recv);
43          //Here change the state
44          string = PI.interpretServerResponse();
45          if(string != null)
46            System.out.print(string);
47        }
48
49      } catch (Exception e) {
50        System.err.println(e);
51        e.printStackTrace();
52      }
53    }
54
55    static int getCommand(){
56      System.out.println();
57      return 0;
58    }
59  }
```

Listing 8: Main class of FTP client (Client.java)

```
1  package Protocol;
2
3  import java.io.*;
4  import java.net.*;
5
6  enum StateEnum {
7    CONNECTED,
8    AUTHSENDUSER,
9    AUTHSENDPASSWD,
10   READY,
11   EXECUTING,
12   DOWNLOADING,
13   EXITING,
14   ERROR;
15 }
16
17 public class ProtocolInterpreter {
18
19   protected InetAddress server;
20   protected int port;
21   protected Socket socket;
22   protected InetAddress local;
23   protected int data_port;
24   protected ServerSocket server_data_socket = null;
25   protected Socket data_socket = null;
26   protected BufferedReader bis;
27   protected PrintStream ps;
28   protected String lastResponse;
29   protected String toBeSent;
30   protected State[] states = {
31       new State.Connected(),
32       new State.AuthSendUser(),
33       new State.AuthSendPasswd(),
34       new State.Ready(),
35       new State.Executing(),
36       new State.Downloading(),
37       new State.Exiting(),
38       new State.Error()
```

```
39    };
40    protected StateEnum currentState;
41    protected String currentUserCommand;
42    public boolean alive;
43
44    public String filename = null;
45    public String path = "C:\\Users\\ardus\\workspace\\tmp\\";
46
47    public ProtocolInterpreter(InetAddress server,int port){
48      this.server = server;
49      this.port = port;
50    }
51
52    public void connect() throws IOException{
53      this.socket = new Socket(server,port);
54      local = socket.getLocalAddress();
55      bis = new BufferedReader(new InputStreamReader(socket.getInputStream()));
56      ps = new PrintStream(socket.getOutputStream());
57      currentState = StateEnum.CONNECTED;
58      alive = true;
59    }
60
61    public String receiveMessage() throws IOException{
62      return states[currentState.ordinal()].receiveMessage(this);
63    }
64
65    //Most likely changes the current state
66    public String interpretServerResponse() throws Exception{
67      String toShowToUser = states[currentState.ordinal()].interpretServerResponse(this);
68      lastResponse = null;
69      return toShowToUser;
70    }
71
72    public String messageForUser(){
73      return states[currentState.ordinal()].messageForUser(this);
74    }
75
76    public boolean needUserInput(){
77      return states[currentState.ordinal()].needUserInput(this);
78    }
79
80    public void userInputEval(String string)throws Exception{
81      states[currentState.ordinal()].userInputEval(string,this);
82    }
83
84    public void sendMessageToServer()throws Exception{
85      states[currentState.ordinal()].sendMessageToServer(this);
86      toBeSent = null;
87    }
88
89    public Integer lastResponeCode(){
90      return states[currentState.ordinal()].lastResponseCode(this);
91    }
92
93    //If there is no socket listening, search for free port
94    public void createDataSocket() throws IOException{
95      if(server_data_socket != null){
96        try{
97          server_data_socket.close();
98        }catch(Exception e){
99          System.err.println("Strange but we continue");
100       }
101       server_data_socket = null;
102     }
103     server_data_socket = new ServerSocket();
104     InetSocketAddress addr = new InetSocketAddress(local, 0);
105     server_data_socket.bind(addr);
106
107     data_port = server_data_socket.getLocalPort();
108
109   }
110
111   public void openDataConnection() throws IOException{
```

```
112       data_socket = server_data_socket.accept();
113     }
114
115     public void prepareFile(String filename){
116       this.filename = filename;
117     }
118
119     public void writeDataOnStdout() throws IOException{
120       InputStreamReader input;
121       char[] buffer = new char[8*1024];
122       int n;
123
124       input = new InputStreamReader(data_socket.getInputStream());
125       do {
126         n = input.read(buffer);
127         if (n == 0 || n == -1)
128           break ;
129         System.out.print(String.valueOf(buffer, 0, n));
130       }while (true);
131       System.err.println("File successfully downloaded");
132     }
133
134     public void writeDataOnFile(String filename) throws IOException{
135       InputStreamReader input;
136       char[] buffer = new char[8*1024];
137       int n;
138       BufferedWriter downloadedFile;
139
140       downloadedFile = new BufferedWriter(new FileWriter(path + filename));
141
142       input = new InputStreamReader(data_socket.getInputStream());
143       if(downloadedFile != null){
144         try{
145           do {
146             n = input.read(buffer);
147             if (n == 0 || n == -1)
148               break ;
149             downloadedFile.write(buffer, 0, n);
150           }while (true);
151         }finally{
152           downloadedFile.close();
153           downloadedFile = null;
154         }
155       }
156       System.err.println("File successfully downloaded");
157     }
158
159     //Ftp commands
160     public void commandPORT() throws IOException{
161       int port_h,port_l;
162       String[] ip;
163
164       this.createDataSocket();
165       ip = local.getHostAddress().split("\\.");
166       port_h = data_port/256;
167       port_l = data_port%256;
168
169       toBeSent = new StringBuilder("PORT ")
170               .append(ip[0])
171               .append(",")
172               .append(ip[1])
173               .append(",")
174               .append(ip[2])
175               .append(",")
176               .append(ip[3])
177               .append(",")
178               .append(port_h)
179               .append(",")
180               .append(port_l)
181               .append("\r\n")
182               .toString();
183       System.err.println(toBeSent);
184     }
```

```
185
186   public void commandQUIT(){
187     toBeSent = "QUIT\r\n";
188   }
189 }
```

Listing 9: Class that implements the protocol (ProtocolInterpreter.java)

```
1  package Protocol;
2  import java.io.IOException;
3
4  //Classes that define the behaviour of the Protocol Interpreter based on the current
       state
5  public class State {
6    //Return a message to be printed for the user
7    public String messageForUser(ProtocolInterpreter PI){
8      return null;
9    }
10
11   //Must be overridden if there's need of user input
12   public boolean needUserInput(ProtocolInterpreter PI){
13     return false;
14   }
15
16   //Used to get commands or data from the user
17   public void userInputEval(String string,ProtocolInterpreter PI) throws Exception{
18     throw new Exception("Getting the input in an undesired moment O.o\n");
19   }
20
21   //Write on the socket a message (non need to override)
22   public void sendMessageToServer(ProtocolInterpreter PI) throws Exception{
23     PI.ps.print(PI.toBeSent);
24     PI.toBeSent = null;
25   }
26
27   //Method to receive a response from the server
28   //    probably no need to override
29   public String receiveMessage(ProtocolInterpreter PI) throws IOException{
30     StringBuilder builder = new StringBuilder();
31     String line;
32     do {
33       line = PI.bis.readLine();
34       if (line == null) {
35         break ;
36       }
37       if (line.length() > 0) {
38         builder.append(line).append ("\n");
39       }
40     }while (line.length() > 0 && line.matches("[0-9]{3}-.*"));
41     PI.lastResponse = builder.toString();
42     return PI.lastResponse;
43   }
44
45   public Integer lastResponseCode(ProtocolInterpreter PI){
46     String[] lines = PI.lastResponse.split("\n");
47     //Returns the number represented by the first three character of
48     //the last line of the last response; null if error occurred
49     return Integer.parseInt(lines[lines.length-1].substring(0, 3));
50   }
51
52   //Method to parse the response of the server
53   //    MUST be overridden
54   public String interpretServerResponse(ProtocolInterpreter PI) throws Exception{
55     throw new Exception("Cannot parse... I know nothing\n");
56     //return null;
57   }
58
59   public static class Connected extends State{
60     //If 220 response then go in AUTHSENDUSER state
61     public String interpretServerResponse(ProtocolInterpreter PI) throws Exception{
62       Integer responseCode;
63       try{
64         responseCode = lastResponseCode(PI);
```

```
65        }catch(NumberFormatException e){
66          PI.currentState = StateEnum.ERROR;  //Error
67          return null;
68        }
69        StringBuilder forUser = new StringBuilder();
70
71        forUser.append(PI.lastResponse).append('\n');
72
73        if(responseCode.intValue() == 220){
74          forUser.append("Connection success\n");
75          PI.currentState = StateEnum.AUTHSENDUSER; //AuthSendUser
76        }else{
77          forUser.append("An error occured in the connection.\nResponse code: ").append(
      responseCode.toString());
78          PI.currentState = StateEnum.ERROR;
79        }
80        return forUser.toString();
81      }
82    }
83
84    public static class AuthSendUser extends State{
85      public String messageForUser(ProtocolInterpreter PI){
86        return "Username: ";
87      }
88
89      public boolean needUserInput(ProtocolInterpreter PI){
90        return true;
91      }
92      //Prepare the message to be sent with the user data
93      public void userInputEval(String string,ProtocolInterpreter PI) throws Exception{
94        PI.toBeSent = "USER "+string+"\r\n";
95      }
96      //Check if need of the password then go to AUTHSENDPASSWD state
97      //Else to the READY state
98      public String interpretServerResponse(ProtocolInterpreter PI) throws Exception{
99        Integer responseCode;
100       try{
101         responseCode = lastResponseCode(PI);
102       }catch(NumberFormatException e){
103         PI.currentState = StateEnum.ERROR;  //Error
104         return null;
105       }
106       StringBuilder forUser = new StringBuilder();
107
108       forUser.append(PI.lastResponse).append('\n');
109       if(responseCode.intValue() == 331){
110         PI.currentState = StateEnum.AUTHSENDPASSWD;
111       }else if(responseCode.intValue() == 230){
112         forUser.append("User has been authenticated\n");
113         PI.currentState = StateEnum.READY;
114       }else{
115         forUser.append("An error occured in the connection.\nResponse code: ").append(
      responseCode.toString());
116         PI.currentState = StateEnum.ERROR;
117       }
118       return forUser.toString();
119     }
120   }
121
122   public static class AuthSendPasswd extends State{
123     public String messageForUser(ProtocolInterpreter PI){
124       return "Password: ";
125     }
126
127     public boolean needUserInput(ProtocolInterpreter PI){
128       return true;
129     }
130
131     public void userInputEval(String string,ProtocolInterpreter PI) throws Exception{
132       PI.toBeSent = "PASS "+string+"\r\n";
133     }
134     //If good answer from the server then go in READY state
135     public String interpretServerResponse(ProtocolInterpreter PI) throws Exception{
```

```
136       Integer responseCode;
137       try{
138         responseCode = lastResponseCode(PI);
139       }catch(NumberFormatException e){
140         PI.currentState = StateEnum.ERROR;  //Error
141         return null;
142       }
143       StringBuilder forUser = new StringBuilder();
144
145       forUser.append(PI.lastResponse).append('\n');
146       if(responseCode.intValue() == 230){
147         forUser.append("User has been authenticated\n");
148         PI.currentState = StateEnum.READY;
149       }else{
150         forUser.append("An error occured in the connection.\nResponse code: ").append(
      responseCode.toString());
151         PI.currentState = StateEnum.ERROR;
152       }
153       return forUser.toString();
154     }
155   }
156
157   public static class Ready extends State{
158     boolean after_sending_open_data_socket = false;
159     //Lists the possible commands for the user
160     public String messageForUser(ProtocolInterpreter PI){
161       String commandList = new StringBuilder()
162       .append("ls - list the content of current directory\n")
163       .append("cd [directory] - changes the current directory to [directory]\n")
164       .append("get [file] - download [file]\n")
165       .append("put [file] - upload [file] \n")
166       .append("exit - close the connection and exit\n")
167       .toString();
168       String prompt = "? - ";
169       return commandList+prompt;
170     }
171
172     public boolean needUserInput(ProtocolInterpreter PI){
173       return true;
174     }
175     //Reads the command of the user and execute the corresponding functions
176     public void userInputEval(String fromUser, ProtocolInterpreter PI){
177       PI.currentUserCommand = fromUser.trim();
178       try{
179         if(PI.currentUserCommand.startsWith("ls")){
180           PI.commandPORT();
181         }else if(PI.currentUserCommand.startsWith("cd")){
182           String arg = null;
183           try{
184             arg = fromUser.split("[ \t]+")[1];
185           }catch(ArrayIndexOutOfBoundsException e){
186             arg = ".";
187           }
188           PI.toBeSent = "CWD "+arg+"\r\n";
189         }else if(PI.currentUserCommand.startsWith("get")){
190           PI.commandPORT();
191         }else if(PI.currentUserCommand.startsWith("put")){
192           PI.commandPORT();
193         }else if(PI.currentUserCommand.startsWith("exit")){
194           PI.commandQUIT();
195           PI.currentState = StateEnum.EXITING;
196         }
197       }catch(IOException e){
198         PI.currentState = StateEnum.ERROR;
199       }
200     }
201
202     public String interpretServerResponse(ProtocolInterpreter PI) throws Exception{
203       Integer responseCode;
204       String forUser = null;
205       try{
206         responseCode = lastResponseCode(PI);
207         if(PI.currentUserCommand.startsWith("ls") || PI.currentUserCommand.startsWith("
```

```
    get")){
208           if(responseCode == 200){
209             forUser = "PORT command succeded\n";
210             PI.openDataConnection();
211             PI.currentState = StateEnum.EXECUTING;
212           }
213         }
214       }catch(NumberFormatException e){
215         PI.currentState = StateEnum.ERROR;  //Error
216         return forUser;
217       }
218       return forUser;
219     }
220   }
221
222   public static class Executing extends State{
223     private String fileName = null;
224
225     public String messageForUser(ProtocolInterpreter PI){
226       String info = new StringBuilder()
227       .append("Preparing for download\n")
228       .toString();
229       return info;
230     }
231
232     public void sendMessageToServer(ProtocolInterpreter PI) throws Exception {
233       try{
234         if(PI.currentUserCommand.startsWith("ls")){
235           PI.toBeSent = "LIST\r\n";
236         }else if(PI.currentUserCommand.startsWith("get")){
237           String[] arg = PI.currentUserCommand.split("[ \t]+");
238           this.fileName = arg[1];
239           PI.toBeSent = "RETR "+this.fileName+"\r\n";
240         }
241         super.sendMessageToServer(PI);
242       }catch(IOException e){
243         PI.currentState = StateEnum.ERROR;
244       }
245     }
246
247     public String interpretServerResponse(ProtocolInterpreter PI){
248       Integer responseCode;
249       StringBuilder forUser = new StringBuilder();
250       try{
251         responseCode = lastResponseCode(PI);
252
253         if(responseCode == 150 || responseCode == 125){
254           forUser.append("Starting to download...\n");
255           if(PI.currentUserCommand.equals("ls")){
256             PI.prepareFile("@stdout");
257           }else if(PI.currentUserCommand.equals("get")){
258             System.err.print("Writing "+ this.fileName +" in filename\n");
259             PI.prepareFile(this.fileName);
260           }
261           PI.currentState = StateEnum.DOWNLOADING;
262         }else{
263            forUser.append("Maybe the file does not exist, or you don't have the rights
    to download it\n");
264         }
265       }catch(NumberFormatException e){
266         PI.currentState = StateEnum.ERROR;  //Error
267       }
268       return forUser.toString();
269     }
270   }
271
272   public static class Downloading extends State{
273
274     public String messageForUser(ProtocolInterpreter PI){
275       try{
276         if(PI.filename.equals("@stdout")){
277           PI.writeDataOnStdout();
278         }else{
```

```
279           PI.writeDataOnFile(PI.filename);
280         }
281       }catch(IOException e){
282         e.printStackTrace();
283         PI.currentState = StateEnum.ERROR;
284       }
285       String info = "\n";
286       return info;
287     }
288
289     public void sendMessageToServer(ProtocolInterpreter PI) throws Exception {
290       ;
291     }
292
293     public String interpretServerResponse(ProtocolInterpreter PI){
294       Integer responseCode;
295       StringBuilder forUser = new StringBuilder();
296       try{
297         responseCode = lastResponseCode(PI);
298         if(responseCode == 226 || responseCode == 250){
299           forUser.append(PI.lastResponse)
300             .append("\nCommand successfully completed\n");
301           PI.currentState = StateEnum.READY;
302         }else{
303           PI.currentState = StateEnum.ERROR;
304         }
305       }catch(NumberFormatException e){
306         PI.currentState = StateEnum.ERROR;  //Error
307       }
308       PI.currentUserCommand = null;
309       return forUser.toString();
310     }
311   }
312
313   public static class Exiting extends State{
314     public String interpretServerResponse(ProtocolInterpreter PI) throws Exception{
315       String forUser = null;
316       try{
317         forUser = "Exiting...";
318         PI.bis.close();
319         PI.data_socket.close();
320         PI.ps.close();
321         PI.server_data_socket.close();
322         PI.socket.close();
323       }catch(NumberFormatException e){
324         PI.currentState = StateEnum.ERROR;  //Error
325         return forUser;
326       }
327       PI.alive = false;
328       return forUser;
329     }
330   }
331
332   public static class Error extends State{
333     //Return a message to be printed for the user
334     public String messageForUser(ProtocolInterpreter PI){
335       return "A fatal error occured\n";
336     }
337
338     //Must be overridden if there's need of user input
339     public boolean needUserInput(ProtocolInterpreter PI){
340       return false;
341     }
342
343     //Write on the socket a message (non need to override)
344     public void sendMessageToServer(ProtocolInterpreter PI) throws Exception{
345       PI.toBeSent = null;
346     }
347
348     //Method to receive a response from the server
349     //    probably no need to override
350     public String receiveMessage(ProtocolInterpreter PI) throws IOException{
351       return null;
```

```
352     }
353
354     public Integer lastResponseCode(ProtocolInterpreter PI){
355       return -1;
356     }
357
358     //Method to parse the response of the server
359     //    MUST be overridden
360     public String interpretServerResponse(ProtocolInterpreter PI){
361       PI.alive = false;
362       return null;
363     }
364   }
365 }
```

Listing 10: Classes containing the different behaviour of the protocol(State.java)