

Segurança e Confiabilidade

2015-16



Grupo XXIII

fc41941 – Rodrigo Reis
fc41964 – Tito Oliveira
fc45773 – Jonas Ferreira

No trabalho, todos os objectivos requisitados foram concretizados sendo que a única limitação do projecto é ser permitido apenas enviar ficheiros para o servidor, que estejam na raiz do projecto e não devem ser enviados ficheiros com a extensão .msg.

O projecto está dividido em 3 pacotes client, functionality e server.

No pacote client está a classe myWhats.java que é a concretização do cliente.

No pacote server está a classe myWhatsServer.java que é a concretização do servidor.

No pacote functionality encontramos várias classes que contêm a concretização dos métodos necessários ao funcionamento do cliente e do servidor, sendo estas:

- Authentication.java – Trata a autenticação de um utilizador server sided e client sided
- Communication.java – Trata da abertura de sockets e envio de comandos para o servidor
- Configurations.java – Contém a definição de vários paths e valores necessários a troca de blocos de dados entre cliente e servidor
- Files.java – Trata das funcionalidades relacionada com o envio, recepção de ficheiros e operações com ficheiros e pastas
- Group.java – Trata das funcionalidade relacionadas com os grupos
- Message.java – Trata das funcionalidades relacionadas com as mensagens, escrita, leitura, recepção, envio e parsing das mesmas para um formato adequado
- Usage.java – Apresenta a utilização correcta do cliente
- User.java – Trata de todas as operações relacionadas com os utilizadores

A sandbox do Servidor permite que este abra uma socket que está a escuta, aceita pedidos e resolve urls para IP.

Em termos de permissões de ficheiros, é permitido a leitura, escrita e deleção dos ficheiros de credenciais e listagem de grupos. É também permitida a leitura e escrita das pastas users, groups e messages e ficheiros no seu interior.

A sandbox do Cliente permite que uma socket faça conexões e resolva urls para endereços de ip.

Em termos de permissões de ficheiros, é permitida a leitura e escrita de todos os ficheiros na raiz do projecto.

As mensagens trocadas consiste maioritariamente no uso de booleanos, para haver coordenação entre os métodos do servidor e do cliente, e em determinados casos, como na recepção das conversas, mensagens de erro e envio de flags e parâmetros, strings.

Os requisitos que esta aplicação deve conter são a Autenticação, sendo usado um sistema de username/password para garantir esta característica; Privacidade, sendo que cada utilizador só consegue ler as suas conversas, as dos grupos a que pertence e é apenas possível que este transfira ficheiros pertencentes às mesmas; Disponibilidade, que é garantida através do uso de um servidor multithread que atende vários clientes simultaneamente; Integridade, sendo que um utilizador só pode enviar mensagens em seu nome e não pode substituir ficheiros já existentes no servidor.

myWhats.java

```
1 package client;
2
3 import java.io.File;
4 import java.io.IOException;
5 import java.io.ObjectInputStream;
6 import java.io.ObjectOutputStream;
7 import java.net.Socket;
8 import java.net.UnknownHostException;
9 import java.util.Scanner;
10
11 import functionality.*;
12
13 /**
14  * This class represents the client myWhats
15  */
16 public class myWhats {
17
18     public static void main(String[] args) throws UnknownHostException,
19         IOException, ClassNotFoundException {
20
21         // Verify the giver argument number
22         if (args.length < 3) {
23
24             System.out.println("Incorrect parameters!\n");
25
26             // Print correct usage
27             Usage.printUsage();
28
29         } else {
30
31             Socket sock = null;
32
33             String user = "";
34             String pass = "";
35
36             // Automatic password handling
37             if (args.length >= 4) {
38                 user = args[0];
39                 pass = args[3];
40             }
41
42             // NO PASSWORD PROVIDED
43             if (args.length == 3) {
44
45                 // Check if -p
46                 if (args[2].equals("-p")) {
47
48                     Scanner sc = new Scanner(System.in);
49
50                     user = args[0];
51
52                     while (pass.length() < 2) {
53
54                         System.out.print("Password: ");
55                         pass = sc.nextLine();
56
57                     }
58
59                     // Close scanner
60                     //sc.close();
61                     //Warn user only registering or login will happen
62                     System.out
63                         .println("\nNOTICE: Your only arguments are your
64 username, "
```

myWhats.java

```

64         + "server and password!\n"
65         + "This will only register you or try to log
you in.\n");
66
67         sock = Communication.connect(args[1]);
68         if(sock != null){
69             ObjectOutputStream out = new ObjectOutputStream(
70                 sock.getOutputStream());
71             ObjectInputStream in = new ObjectInputStream(
72                 sock.getInputStream());
73             Authentication.login(in, out, user, pass);
74             out.close();
75             in.close();
76             sock.close();
77         }
78
79
80     } else {
81
82         System.out.println("Incorrect parameters!\n");
83         Usage.printUsage();
84     }
85
86     // JUST THE PASSWORD
87     } else if (args.length == 4 && args[2].equals("-p")) {
88         //Warn user only registering or login will happen
89         System.out
90             .println("\nNOTICE: Your only arguments are your username,
"
91                 + "server and password!\n"
92                 + "This will only register you or try to log you
in.\n");
93
94         sock = Communication.connect(args[1]);
95         if(sock != null){
96             ObjectOutputStream out = new ObjectOutputStream(
97                 sock.getOutputStream());
98             ObjectInputStream in = new ObjectInputStream(
99                 sock.getInputStream());
100             Authentication.login(in, out, user, pass);
101             out.close();
102             in.close();
103             sock.close();
104         }
105
106
107     // PASSWORD AND -r (no args)
108     } else if (args.length == 5) {
109
110         if (args[4].equals("-r")) {
111
112             System.out.println("-r | Send all the latest comms!");
113
114             sock = Communication.connect(args[1]);
115             if(sock != null){
116                 ObjectOutputStream out = new ObjectOutputStream(
117                     sock.getOutputStream());
118                 ObjectInputStream in = new ObjectInputStream(
119                     sock.getInputStream());
120                 Authentication.login(in, out, user, pass);
121                 String [] snd = {};
122                 Communication.sendCommand(out, "-r", snd);
123
124                 //Receive result

```

```

myWhats.java

125         String convo = (String) in.readObject();
126         System.out.println(convo);
127
128         out.close();
129         in.close();
130         sock.close();
131     }
132
133     } else {
134
135         System.out.println("Incorrect parameters!\n");
136         Usage.printUsage();
137
138     }
139
140     // PASSWORD AND -r ARG1
141     } else if (args.length == 6) {
142
143         if (args[4].equals("-r")) {
144
145             System.out.println("-r | Send all the comms with " + args[5]);
146
147             sock = Communication.connect(args[1]);
148             if(sock != null){
149                 ObjectOutputStream out = new ObjectOutputStream(
150                     sock.getOutputStream());
151                 ObjectInputStream in = new ObjectInputStream(
152                     sock.getInputStream());
153                 Authentication.login(in, out, user, pass);
154
155                 String [] snd = {args[5]};
156                 Communication.sendCommand(out, "-r", snd);
157
158                 //Receive the answer
159                 String convo = (String) in.readObject();
160                 System.out.println(convo);
161
162                 out.close();
163                 in.close();
164                 sock.close();
165             }
166
167         } else {
168
169             System.out.println("Incorrect parameters!\n");
170             Usage.printUsage();
171
172         }
173
174     // ALL OTHER FLAGS
175     } else if (args.length == 7) {
176
177         switch (args[4]) {
178
179             //MESSAGE OPERATION
180             case "-m":
181                 System.out.println("-m | Send message \"" + args[6] + "\" to "
+ args[5]);
182
183                 sock = Communication.connect(args[1]);
184                 if(sock != null){
185                     ObjectOutputStream out = new ObjectOutputStream(
186                         sock.getOutputStream());
187                     ObjectInputStream in = new ObjectInputStream(

```

```

188         sock.getInputStream());
189     Authentication.login(in, out, user, pass);
190     String [] snd = {args[5], args[6]};
191     Communication.sendCommand(out, "-m", snd);
192     Message.sendMessage(out, in);
193     out.close();
194     in.close();
195     sock.close();
196 }
197 break;
198
199 //FILE OPERATION
200 case "-f":
201
202     sock = Communication.connect(args[1]);
203     if(sock != null){
204         ObjectOutputStream out = new ObjectOutputStream(
205             sock.getOutputStream());
206         ObjectInputStream in = new ObjectInputStream(
207             sock.getInputStream());
208
209         Authentication.login(in, out, user, pass);
210
211         //Check if file exists
212         File file = new File(args[6]);
213
214         //Don't bother server if file doesn't exist locally
215         if(file.exists() && !file.isDirectory()){
216             String [] snd = {args[5], args[6]};
217             System.out.println("-f " + args[5] + " " + args[6]);
218             Communication.sendCommand(out, "-f", snd);
219             Files.sendFile(in, out, "", args[6]);
220
221         }else{
222             System.out.println("File " + args[6] + " not found!");
223         }
224
225         out.close();
226         in.close();
227         sock.close();
228     }
229     break;
230
231 //REVIEW OPERATION
232 case "-r":
233     System.out.println("-r " + args[5] + " " + args[6]);
234
235     sock = Communication.connect(args[1]);
236     if(sock != null){
237         ObjectOutputStream out = new ObjectOutputStream(
238             sock.getOutputStream());
239         ObjectInputStream in = new ObjectInputStream(
240             sock.getInputStream());
241         Authentication.login(in, out, user, pass);
242         String [] snd = {args[5], args[6]};
243         System.out.println("-r " + args[5] + " " + args[6]);
244         Communication.sendCommand(out, "-r", snd);
245
246         // Check if DOWNLOADS FOLDER exist
247         File downloadsFolder = new
File(Configurations.DOWNLOAD_FOLDER);
248         if (!downloadsFolder.exists()) {
249
250             downloadsFolder.mkdir();

```

```

myWhats.java

251     }
252
253     boolean fileExists = in.readBoolean();
254
255     if (fileExists) {
256         Files.receiveFile(in, out,
Configurations.DOWNLOAD_FOLDER, snd[1]);
257     } else {
258         System.out.println("The file you requested does not
exist!");
259     }
260
261     out.close();
262     in.close();
263     sock.close();
264 }
265 break;
266
267 //GROUP ADD OPERATION
268 case "-a":
269
270     sock = Communication.connect(args[1]);
271     if(sock != null){
272         ObjectOutputStream out = new ObjectOutputStream(
273             sock.getOutputStream());
274         ObjectInputStream in = new ObjectInputStream(
275             sock.getInputStream());
276         Authentication.login(in, out, user, pass);
277         System.out.println("-a " + args[5] + " " + args[6]);
278         String [] snd = {args[5], args[6]};
279         Communication.sendCommand(out, "-a", snd);
280
281         //receive response
282         String rsp = (String) in.readObject();
283         System.out.println(rsp);
284
285         out.close();
286         in.close();
287         sock.close();
288     }
289     break;
290
291 //GROUP REMOVE OPERATION
292 case "-d":
293
294     sock = Communication.connect(args[1]);
295     if(sock != null){
296         ObjectOutputStream out = new ObjectOutputStream(
297             sock.getOutputStream());
298         ObjectInputStream in = new ObjectInputStream(
299             sock.getInputStream());
300         Authentication.login(in, out, user, pass);
301
302         String [] snd = {args[5], args[6]};
303         System.out.println("-d " + args[5] + " " + args[6]);
304         Communication.sendCommand(out, "-d", snd);
305
306         out.close();
307         in.close();
308         sock.close();
309     }
310     break;
311
312 //INVALID PARAMETERS

```


myWhats.java

```
313         default:
314             System.out.println("Incorrect parameters!\n");
315             Usage.printUsage();
316
317
318         }
319     }
320 }
321
322 }
323
```

Authentication.java

```

1 package functionality;
2
3 import java.io.IOException;
4 import java.io.ObjectInputStream;
5 import java.io.ObjectOutputStream;
6 import java.util.Scanner;
7
8 /**
9  * This class handles the authentication process
10 */
11 public class Authentication {
12
13     /*****
14      * CLIENT SIDE
15      *****/
16
17     /**
18      * Authenticates with the server
19      *
20      * @param in
21      *         - ObjectInputStream
22      * @param out
23      *         - ObjectOutputStream
24      * @param username
25      *         - User's username
26      * @param password
27      *         - User's password
28      * @return true if user was successfully logged in
29      */
30     public static boolean login(ObjectInputStream in, ObjectOutputStream out,
31                                String username, String password) {
32
33         boolean userExists = false;
34         boolean usernameAvailable = false;
35         boolean successLogin = false;
36
37         try {
38             System.out.println("Trying to authenticate...");
39
40             out.writeObject(username);
41             out.writeObject(password);
42             out.flush();
43
44             // Receive if user exists
45             userExists = in.readBoolean();
46
47             // If user is doesn't exist
48             if (!userExists) {
49
50                 // Check if available
51                 usernameAvailable = in.readBoolean();
52
53                 if (usernameAvailable) {
54
55                     Scanner read = new Scanner(System.in);
56                     String ans = "";
57
58                     // Keep prompting for a valid answer to register or not
59                     while (!ans.equals("n") && !ans.equals("y")) {
60
61                         System.out.println("This username is not registered. Do you
62 want to register it? (Y/N)");
63                         ans = read.nextLine().toLowerCase();
64                         System.out.println(ans);

```

Authentication.java

```

63         }
64
65         //read.close();
66
67         if (ans.equals("y")) {
68             out.writeBoolean(true);
69             out.flush();
70         } else {
71             out.writeBoolean(false);
72             out.flush();
73         }
74     }
75
76     successLogin = in.readBoolean();
77
78     if (successLogin) {
79         System.out.println("User registered successfully!");
80     } else {
81         System.out.println("User not registered!");
82     }
83
84     // Se o user existir verificar se foi logado com sucesso
85 } else {
86
87     successLogin = in.readBoolean();
88
89     if (successLogin) {
90         System.out.println("Successfull Authentication!");
91         return true;
92     } else {
93         System.out.println("Failed Authentication! Incorrect
94 Credentials!");
95         return false;
96     }
97
98 } catch (IOException e) {
99
100     System.err.println("Error during autentication!");
101     // e.printStackTrace();
102 }
103
104 return false;
105 }
106
107 /*****
108  * SERVER SIDE
109  *****/
110
111 /**
112  * Authenticates a user - If user exists checks for password, if not,
113  * registers
114  *
115  * @param in
116  *      - ObjectInputStream
117  * @param out
118  *      - ObjectOutputStream
119  * @param login
120  *      - Credentials supplied during login in format user:password
121  * @return True if password correct or new user was registered, False
122  *      otherwise
123  */
124 public static boolean authenticateUser(ObjectInputStream in, ObjectOutputStream
out, String login) {

```

```

125
126 String creds;
127 String[] parsedCreds = login.split(":");
128 boolean register = false;
129
130 try {
131     // Verify if user exists
132     if ((creds = User.userExists(parsedCreds[0])) != "") {
133
134         // Send that the user exists
135         out.writeBoolean(true);
136         out.flush();
137
138         String[] registeredCredentials = creds.split(":");
139
140         // If password matches the one registered
141         if (registeredCredentials[1].equals(parsedCreds[1])) {
142             System.out.println(registeredCredentials[0] + " logged in with
143 success");
144             out.writeBoolean(true);
145             out.flush();
146             return true;
147         } else {
148
149             System.out.println(registeredCredentials[0] + "'s password
150 doesn't match");
151             out.writeBoolean(false);
152             out.flush();
153             return false;
154         }
155
156         // User doesn't exist, register it
157     } else {
158
159         // Send that the user doesn't exist
160         out.writeBoolean(false);
161         out.flush();
162
163         // Verify if a group exists with the same name
164         if (!Group.groupExists(parsedCreds[0])) {
165
166             // Tell username is available
167             out.writeBoolean(true);
168             out.flush();
169
170             register = in.readBoolean();
171
172             if (register) {
173                 boolean result = User.createUser(login);
174                 out.writeBoolean(result);
175                 out.flush();
176                 return result;
177             } else {
178                 System.out.println("User not registered!");
179                 out.writeBoolean(false);
180                 out.flush();
181                 return false;
182             }
183
184         } else {
185
186             // Send username is not available

```

Authentication.java

```
187         out.writeBoolean(false);
188         out.flush();
189         return false;
190     }
191 }
192
193 } catch (IOException e) {
194
195     System.err.println("Error during authentication!");
196     // e.printStackTrace();
197 }
198
199 return false;
200 }
201 }
202
```

```

1 package functionality;
2
3 import java.io.IOException;
4 import java.io.ObjectOutputStream;
5 import java.net.Socket;
6
7 /**
8  * This class handles communication between client/server
9  */
10 public class Communication {
11
12     /**
13      * Connects to a server in the address and port provided
14      *
15      * @param addressAndPort
16      *      - Server's address and port in X.X.X.X:YYYY FORMAT
17      * @return A socket with the connection to the server, null in case of fail
18      */
19     public static Socket connect(String addressAndPort) {
20
21         Socket sock = null;
22
23         try {
24             if (addressAndPort.matches("\\d+\\.\\d+\\.\\d+\\.\\d+\\:\\d+")) {
25                 String[] addPort = addressAndPort.split(":");
26                 sock = new Socket(addPort[0], Integer.parseInt(addPort[1]));
27             }
28         } catch (IOException e) {
29
30             System.err.println("Error connecting to server!\nCheck if server or
31 connection are down!");
32             // e.printStackTrace();
33         }
34         return sock;
35     }
36
37     /**
38      * Sends a command flag and it's arguments to the server
39      *
40      * @param out
41      *      - ObjectOutputStream
42      * @param flag
43      *      - The command flag for the operation
44      * @param args
45      *      - The arguments for the operation
46      */
47     public static void sendCommand(ObjectOutputStream out, String flag, String[]
48 args) {
49
50         try {
51             out.writeObject(flag);
52             out.writeObject(args);
53             out.flush();
54         } catch (IOException e) {
55             System.err.println("Error sending flag and arguments to server!");
56             // e.printStackTrace();
57         }
58     }
59 }

```

Configurations.java

```
1 package functionality;
2
3 /**
4  * The configurations class contains the definition of various paths and data sizes
5  */
6 public class Configurations {
7
8     //Credential file and group list file
9     public final static String CREDENTIALS_FILENAME = "!credentialsFile";
10    public final static String GROUPS_FILENAME = "!groupsFile";
11
12    //User and group data saving destination
13    public final static String USERS_FOLDER = "users";
14    public final static String GROUPS_FOLDER = "groups";
15
16    //Messages saving destination
17    public final static String MESSAGES_FOLDER = "messages";
18
19    //Path to files received from -r DEST FILENAME
20    public final static String DOWNLOAD_FOLDER = "downloads";
21
22    //Data block size to send and receive files
23    public final static int DATA_BLOCK = 1024;
24 }
25
```

Files.java

```
1 package functionality;
2
3 import java.io.File;
4 import java.io.FileInputStream;
5 import java.io.FileOutputStream;
6 import java.io.IOException;
7 import java.io.ObjectInputStream;
8 import java.io.ObjectOutputStream;
9 import java.util.ArrayList;
10 import java.util.List;
11
12 /**
13  * This class handles files and folder operations
14  */
15 public class Files {
16
17     /**
18      * Returns a String array with a list of folder names in path
19      *
20      * @param path
21      *      - The path to map
22      * @return - String array with the folders in path
23      */
24     public static String[] listFolders(String path) {
25
26         File directory = new File(path);
27
28         File[] folderList = directory.listFiles();
29         List<String> list = new ArrayList<String>();
30
31         for (File folder : folderList) {
32             if (folder.isDirectory()) {
33                 list.add(folder.getName());
34             }
35         }
36
37         return list.toArray(new String[list.size()]);
38     }
39
40     /**
41      * Returns a String array with a list of file names in path
42      *
43      * @param path
44      *      - The path to map
45      * @return - String array with the file names in path
46      */
47     public static String[] listFiles(String path) {
48
49         File directory = new File(path);
50
51         File[] fileList = directory.listFiles();
52         List<String> list = new ArrayList<String>();
53
54         if (fileList != null) {
55             for (File file : fileList) {
56                 if (file.isFile()) {
57                     list.add(file.getName());
58                 }
59             }
60         }
61         return list.toArray(new String[list.size()]);
62     }
63
64     /**
```


Files.java

```

65  * Creates the correct path according if destinatory is a group or a user
66  *
67  * @param destinatory
68  *      - The destinatory
69  * @param callingUser
70  *      - The user who calls the method (remetent)
71  * @return A#B destinatory is a user, B if destinatory is group, or "" if
72  *         error
73  */
74  public static String getDestination(String destinatory, String callingUser) {
75
76      int val = Group.isUserOrGroup(destinatory);
77      String destination = "";
78
79      // Destinatory is user
80      if (val == 1) {
81
82          // WRITE A MESSAGE SHOWING A FILE WAS SENT
83          StringBuilder dst = new StringBuilder();
84
85          if (destinatory.compareTo(callingUser) <= 0) {
86
87              dst.append(destinatory);
88              dst.append("#");
89              dst.append(callingUser);
90
91          } else {
92
93              dst.append(callingUser);
94              dst.append("#");
95              dst.append(destinatory);
96
97          }
98
99          destination = dst.toString();
100
101          // Destinatory is group
102      } else if (val == 0) {
103
104          destination = destinatory;
105
106      }
107
108      return destination;
109  }
110
111  /**
112   * Deletes a folder and it's contents
113   *
114   * @param folder
115   *      - A File object with the folder's path
116   */
117  static void deleteFolder(File folder) {
118
119      // List all files
120      File[] files = folder.listFiles();
121
122      if (files != null) {
123          // Delete one by one
124          for (File f : files) {
125
126              if (f.isDirectory()) {
127                  deleteFolder(f);
128              } else {

```

```

129         f.delete();
130     }
131 }
132 }
133 // Delete folder
134 folder.delete();
135 }
136
137 /**
138  * Sends a file
139  *
140  * @param in
141  *      - ObjectInputStream
142  * @param out
143  *      - ObjectOutputStream
144  * @param filePath
145  *      - The path of the file
146  * @param fileName
147  *      - The file name
148  * @return True if sending was successful, False otherwise
149  */
150 public static boolean sendFile(ObjectInputStream in, ObjectOutputStream out,
String filePath, String fileName) {
151
152     byte[] buffer = new byte[Configurations.DATA_BLOCK];
153     FileInputStream file = null;
154
155     try {
156
157         file = new FileInputStream(filePath + fileName);
158
159         // Asks if destination is valid
160         boolean validDestination = in.readBoolean();
161
162         if (validDestination) {
163
164             // Asks if file exists remotely
165             boolean fileExistsAtDestination = in.readBoolean();
166
167             // If file doesn't exist send
168             if (!fileExistsAtDestination) {
169                 // Send file size
170                 long fileSize = file.getChannel().size();
171                 out.writeLong(fileSize);
172                 out.flush();
173
174                 // Send file
175                 int count;
176                 while ((count = file.read(buffer)) > 0) {
177                     out.write(buffer, 0, count);
178                     out.flush();
179                 }
180                 file.close();
181
182                 // Receive received byte amount
183                 long rcvdBytes = in.readLong();
184
185                 if (rcvdBytes == fileSize) {
186                     System.out.println("File sent with success!");
187                     return true;
188                 } else {
189                     System.out.println("Failed to send file!");
190                     return false;
191                 }

```

```

192
193         } else {
194             file.close();
195             System.out.println("File already exists!");
196             return false;
197         }
198
199     } else {
200         file.close();
201         System.out.println("User or group doesn't exist!");
202     }
203
204     } catch (IOException e) {
205         System.err.println("Error sending file!");
206         e.printStackTrace();
207     }
208
209     try {
210         file.close();
211     } catch (IOException e) {
212         // e.printStackTrace();
213         System.err.println("File not found!");
214     }
215     return false;
216 }
217
218 /**
219  * Receives a file
220  *
221  * @param in
222  *      - ObjectInputStream
223  * @param out
224  *      - ObjectOutputStream
225  * @param path
226  *      - The path to save the file
227  * @param fileName
228  *      - The file name
229  * @return True if file received successfully, False otherwise
230  */
231 public static boolean receiveFile(ObjectInputStream in, ObjectOutputStream out,
232 String path, String fileName) {
233
234     byte[] buffer = new byte[Configurations.DATA_BLOCK];
235     FileOutputStream file = null;
236     String fullPath;
237
238     try {
239         // Answers if destination is valid
240         if (path.equals("") || path == null) {
241             fullPath = fileName;
242         } else {
243             fullPath = path + "/" + fileName;
244         }
245
246         File dest = new File(path);
247         if (dest.exists() && dest.isDirectory()) {
248
249             out.writeBoolean(true);
250             out.flush();
251
252             // Answers if file already exists
253             File checkIfExists = new File(fullPath);
254             if (checkIfExists.exists()) {

```

```

255         System.out.println("File already exists locally!");
256         out.writeBoolean(true);
257         out.flush();
258         return false;
259
260     } else {
261         System.out.println("Receive file!");
262         out.writeBoolean(false);
263         out.flush();
264
265         // Receive file size
266         long fileSize = in.readLong();
267
268         // Receive file
269         long recvd = 0;
270         if (fileSize > 0) {
271
272             file = new FileOutputStream(fullPath);
273
274             int count;
275             recvd = 0;
276             while (recvd < fileSize) {
277                 count = in.read(buffer);
278                 file.write(buffer, 0, count);
279                 recvd += count;
280             }
281
282             file.close();
283         }
284
285         // Send received bytes amount
286         out.writeLong(recvd);
287         out.flush();
288         return true;
289     }
290
291     } else {
292         System.out.println("User or group does not exist!");
293         out.writeBoolean(false);
294         out.flush();
295         return false;
296     }
297
298     } catch (IOException e) {
299         System.err.println("Error receiving file!");
300         // e.printStackTrace();
301     }
302
303     return false;
304 }
305 }
306

```

Group.java

```

1 package functionality;
2
3 import java.io.File;
4 import java.io.BufferedReader;
5 import java.io.BufferedWriter;
6 import java.io.FileReader;
7 import java.io.FileWriter;
8 import java.io.IOException;
9 import java.io.ObjectInputStream;
10 import java.io.ObjectOutputStream;
11
12 /**
13  * This class handles group operations
14  */
15 public class Group {
16
17     /**
18      * Verifies if a groups exists in the groups file
19      *
20      * @param groupName
21      *      - Name of the group to be searched for
22      * @return True if group exists in groups file
23      */
24     static boolean groupExists(String groupName) {
25
26         try {
27
28             BufferedReader br = new BufferedReader(new
29             FileReader(Configurations.GROUPS_FILENAME));
30
31             String line;
32             while ((line = br.readLine()) != null) {
33
34                 if (line.equals(groupName)) {
35                     br.close();
36                     System.out.println("Group exists in groups file!");
37                     return true;
38                 }
39
40             }
41
42             br.close();
43
44         } catch (IOException e) {
45             System.err.println("Error verifying if group exists!");
46             // e.printStackTrace();
47         }
48         return false;
49     }
50
51     /**
52      * Checks if contact is a client or a group
53      *
54      * @param input
55      *      - the contact's id
56      * @return 1 if it's a user, 0 if it's a group and -1 in case of error
57      */
58     public static int isUserOrGroup(String input) {
59
60         try {
61             // Check if is User
62             if (User.userExists(input) != "") {
63                 return 1;
64
65             // Check if is group
66
67         }
68     }
69 }

```

Group.java

```

64         } else if (groupExists(input)) {
65             return 0;
66         }
67
68     } catch (IOException e) {
69         System.err.println("Error verifying if it's a user or a group!");
70         // e.printStackTrace();
71     }
72     return -1;
73 }
74
75 /**
76  * Creates a new group if it doesn't exist, or adds a member if it does
77  *
78  * @param username
79  *     - Username of the user to be added
80  * @param groupName
81  *     - Name of the group to be created
82  * @param callingUser
83  *     - The user who invokes the method
84  * @param out
85  *     - ObjectOutputStream
86  * @param in
87  *     - ObjectInputStream
88  * @return True if group was created/user added to group successfully, False
89  *         otherwise
90  * @throws IOException
91  */
92 public static boolean addGroup(String username, String groupName, String
callingUser, ObjectOutputStream out,
93                               ObjectInputStream in) throws IOException {
94
95     try {
96         // Check if there's a user with the supplied group name
97         if (User.userExists(groupName).equals("")) {
98
99             // Check if group already exists
100             if (groupExists(groupName)) {
101
102                 // If group exists, check if calling user is admin
103                 if (isAdmin(groupName, callingUser)) {
104
105                     // Check if user to be added already exists in group
106                     // If it doesn't, add it!
107                     if (!isInGroup(groupName, username) && !
(User.userExists(username).equals(""))) {
108
109                         BufferedWriter bw = new BufferedWriter(
110                             new FileWriter(Configurations.GROUPS_FOLDER +
111                             "/" + groupName + ".cfg", true));
112
113                         bw.append(username);
114                         bw.newLine();
115                         bw.close();
116
117                         out.writeObject("User added with sucess!");
118                         out.flush();
119                         return true;
120
121                     // If it's already added to the group
122                 } else {
123                     out.writeObject("Failed to add " + username + " to
group " + groupName
+ "\nUser doesn't exist or it's already on

```

Group.java

```

    this group");
124         out.flush();
125         System.out.println("Failed to add " + username + " to
group " + groupName
126             + "\nUser doesn't exist or it's already on
this group");
127         return false;
128     }
129
130     // The user is not an admin
131 } else {
132     out.writeObject("Failed to add " + username + " to the
group " + groupName
133         + ". You are not the admin of this group.");
134     out.flush();
135     System.out.println("Failed to add " + username + " to the
group " + groupName
136         + ". You are not the admin of this group.");
137     return false;
138 }
139
140 // The group doesn't exist
141 } else {
142
143     // Create group Folder
144     File createFolder = new File(Configurations.MESSAGES_FOLDER +
"/" + groupName);
145
146     createFolder.mkdir();
147
148     // Writes the group properties file
149     BufferedWriter bw = new BufferedWriter(
150         new FileWriter(Configurations.GROUPS_FOLDER + "/" +
groupName + ".cfg", true));
151
152     // Check if user is trying to add himself
153     // If that's the case, write only one line
154     if (username.equals(callingUser)) {
155
156         bw.append(username);
157         bw.newLine();
158
159         // If user not trying to add himself
160     } else {
161         // Adicionar linha do admin
162         bw.append(callingUser);
163         bw.newLine();
164
165         // Add group to calling user user file
166         BufferedWriter callingUserPersonalFile = new
BufferedWriter(
167             new FileWriter(Configurations.USERS_FOLDER + "/" +
callingUser + ".cfg", true));
168         callingUserPersonalFile.write(groupName);
169         callingUserPersonalFile.newLine();
170         callingUserPersonalFile.close();
171
172         // Check if user to be added exists
173         if (!User.userExists(username).equals("")) {
174             // If so add user to group list member
175             bw.append(username);
176             bw.newLine();
177
178

```

Group.java

```

179         // Add group to user file
180         BufferedWriter usrPersonalFile = new BufferedWriter(
181             new FileWriter(Configurations.USERS_FOLDER +
182                 "/" + username + ".cfg", true));
183         usrPersonalFile.write(groupName);
184         usrPersonalFile.newLine();
185         usrPersonalFile.close();
186
187         System.out.println("Group created succesfully");
188         out.writeObject("Group created succesfully");
189         out.flush();
190
191         // User to eb added is not registered
192     } else {
193         out.writeObject("Group created but user " + username
194             + " could not be added because it doesn't
195             exist");
196         out.flush();
197         System.out.println("Group created but user " + username
198             + " could not be added because it doesn't
199             exist");
200     }
201
202     bw.close();
203
204     // Register group in groups file
205     BufferedWriter gf = new BufferedWriter(new
206         FileWriter(Configurations.GROUPS_FILENAME, true));
207     gf.write(groupName);
208     gf.newLine();
209     gf.close();
210
211     // If the group to be created conflicts with an already
212     // registered group/user
213     } else {
214         out.writeObject("Failed to create group! A user with this name
215             already exists!");
216         out.flush();
217         System.out.println("Failed to create group! A user with this name
218             already exists");
219         return false;
220     }
221
222     } catch (IOException e) {
223         System.out.println("Failed to create group/add new element!");
224         out.writeObject("Failed to create group! A user with this name already
225             exists!");
226         out.flush();
227         // e.printStackTrace();
228     }
229
230     return false;
231 }
232
233 /**
234  * Verifies if a user is admin of a group
235  *
236  * @param groupName
237  *     - The name of the group for this query
238  * @param username
239  *     - The username of the client for this query

```


Group.java

```

236     * @return True if user is admin of group, otherwise, False
237     */
238     private static boolean isAdmin(String groupName, String username) {
239
240         try {
241
242             BufferedReader br = new BufferedReader(
243                 new FileReader(Configurations.GROUPS_FOLDER + "/" + groupName +
244                     ".cfg"));
245
246             String line = br.readLine();
247
248             if (line.equals(username)) {
249                 br.close();
250                 System.out.println("User " + username + " is Admin of group " +
251                     groupName);
252                 return true;
253             }
254             br.close();
255
256         } catch (IOException e) {
257             // e.printStackTrace();
258             System.err.println("Error removing user from group!");
259         }
260         return false;
261     }
262
263     /**
264     * Verifies if user is in group
265     *
266     * @param groupName
267     *     - The name of the group
268     * @param username
269     *     - The user's username
270     * @return True if
271     */
272     private static boolean isInGroup(String groupName, String username) {
273
274         try {
275
276             BufferedReader br = new BufferedReader(
277                 new FileReader(Configurations.GROUPS_FOLDER + "/" + groupName +
278                     ".cfg"));
279
280             String line;
281
282             while ((line = br.readLine()) != null) {
283                 if (line.equals(username)) {
284                     br.close();
285                     System.out.println("User " + username + " is in group " +
286                         groupName);
287                     return true;
288                 }
289             }
290             br.close();
291
292         } catch (IOException e) {
293             System.err.println("Error verifying if user belongs to group!");
294             // e.printStackTrace();
295         }
296         return false;
297     }
298
299     /**

```

Group.java

```

296  * Removes an entry from a text file
297  *
298  * @param filename
299  *      - File to be redacted
300  * @param entry
301  *      - Entry to redact
302  * @return True if entry was successfully redacted
303  */
304  public static boolean removeEntry(String filename, String entry) {
305
306      try {
307          BufferedReader in = new BufferedReader(new FileReader(filename));
308          StringBuilder sb = new StringBuilder();
309
310          // Stripe the entry
311          String line;
312          while ((line = in.readLine()) != null) {
313              // If it's not the entry we're looking for, append
314              if (!line.equals(entry)) {
315                  sb.append(line);
316                  sb.append("\n");
317              }
318          }
319
320          in.close();
321
322          // Write the new file
323          BufferedWriter out = new BufferedWriter(new FileWriter(filename));
324          out.write(sb.toString());
325          out.close();
326
327          return true;
328
329      } catch (IOException e) {
330          System.err.println("Error in file operations for entry removal!");
331          // e.printStackTrace();
332      }
333
334      return false;
335
336  }
337
338  /**
339   * Removes a user from a group and cleans the respective configuration files
340   *
341   * @param username
342   *      - The username of the user to be removed
343   * @param groupName
344   *      - The groups name
345   * @param callingUser
346   *      - The user who invokes this method
347   * @return True if user was successfully removed, False otherwise
348   */
349  public static boolean removeFromGroup(String username, String groupName, String
callingUser) {
350
351      try {
352          // Verificar se quem chama o metodo e o admin do grupo
353          if (isAdmin(groupName, callingUser)) {
354              // Se o utilizador existe e pertence ao grupo
355              if (User.userExists(username) != null && isInGroup(groupName,
username)) {
356
357                  // Se o utilizador a remover É admin apaga o grupo todo

```

Group.java

```

358         if (username.equals(callingUser)) {
359
360             // Pega na lista de elementos do grupo e percorre-a
361             BufferedReader in = new BufferedReader(
362                 new FileReader(Configurations.GROUPS_FOLDER + "/" +
groupName + ".cfg"));
363
364             // Remove as entradas do ficheiro pessoal
365             String line;
366             while ((line = in.readLine()) != null) {
367
368                 removeEntry(Configurations.USERS_FOLDER + "/" + line +
".cfg", groupName);
369
370             }
371
372             // apaga o ficheiro de membros
373
374             File memberFile = new File(Configurations.GROUPS_FOLDER +
"/" + groupName + ".cfg");
375             memberFile.delete();
376
377             in.close();
378             return true;
379             // Just delete the element
380         } else {
381
382             // Remover do ficheiro do grupo
383             removeEntry(Configurations.GROUPS_FOLDER + "/" + groupName
+ ".cfg", username);
384
385             // remover do ficheiro pessoal
386             removeEntry(Configurations.USERS_FOLDER + "/" + username +
".cfg", groupName);
387
388             return true;
389
390         }
391
392     } else {
393         System.out.println("User " + username + "doesn't exist or is
not a member of " + groupName);
394         return false;
395     }
396
397 } else {
398     System.out.println(callingUser + " is not an admin of " + groupName
+ " and cannot delete " + username);
399     return false;
400 }
401 } catch (IOException e) {
402     // TODO Auto-generated catch block
403     System.err.println("Error removing user from group!");
404     e.printStackTrace();
405 }
406
407 return false;
408 }
409 }
410

```

Message.java

```

1 package functionality;
2
3 import java.io.BufferedReader;
4 import java.io.BufferedWriter;
5 import java.io.File;
6 import java.io.FileReader;
7 import java.io.FileWriter;
8 import java.io.IOException;
9 import java.io.ObjectInputStream;
10 import java.io.ObjectOutputStream;
11 import java.text.SimpleDateFormat;
12 import java.util.Date;
13
14 /**
15  * This class handles Message operations
16  */
17 public class Message {
18
19     /**
20      * Builds a conversation from the individual message files
21      *
22      * @param dest
23      *      - The destinatory
24      * @param callingUser
25      *      - The sender
26      * @param lastestOnly
27      *      - If it shows only the last message exchanged with
28      * @return A string with the conversation
29      */
30     public static String buildConversation(String dest, String callingUser, boolean
31     lastestOnly) {
32         // RECONSTRUCT CONVERSATION
33         StringBuilder conversationBack = new StringBuilder();
34         ;
35         try {
36             System.out.println("Send all communications with " + dest);
37
38             String destination = Files.getDestination(dest, callingUser);
39
40             String[] filesList = Files.listFiles(Configurations.MESSAGES_FOLDER +
41             "/" + destination);
42
43             if (filesList.length != 0) {
44                 // SHOW WHO IS THIS CONVERSATION WITH
45                 conversationBack.append("Contact: " + dest + "\n");
46
47                 if (lastestOnly) {
48                     String s = filesList[filesList.length - 1];
49
50                     if (s.matches("(^[\\s]+(\\. (?.i) (msg)))$")) {
51
52                         conversationBack.append(Message.parseMessage(
53                             Configurations.MESSAGES_FOLDER + "/" + destination
54                             + "/" + s, callingUser));
55
56                     }
57
58                 } else {
59
60                     for (String s : filesList) {
61
62                         if (s.matches("(^[\\s]+(\\. (?.i) (msg)))$")) {

```

Message.java

```

62         conversationBack.append(Message.parseMessage(
63             Configurations.MESSAGES_FOLDER + "/" +
destination + "/" + s, callingUser));
64     }
65     }
66     }
67     } else {
68         conversationBack.append("No conversations found for user/group " +
dest);
69     }
70
71     } catch (IOException e) {
72         System.err.println("Error building conversation!");
73         // e.printStackTrace();
74     }
75     return conversationBack.toString();
76 }
77
78 /**
79  * Presents a message properly
80  *
81  * @param path
82  *     - Path to the message file
83  * @param callingUser
84  *     - The sender
85  * @return A string with the parsed message
86  */
87 public static String parseMessage(String path, String callingUser) throws
IOException {
88
89     BufferedReader br;
90     StringBuilder sb = new StringBuilder();
91
92     br = new BufferedReader(new FileReader(path));
93
94     String tmp;
95     int pos = 0;
96     while ((tmp = br.readLine()) != null) {
97         if (pos == 0) {
98             if (tmp.equals(callingUser))
99                 sb.append("me: ");
100             else
101                 sb.append(tmp + ": ");
102         } else {
103             sb.append(tmp);
104             sb.append("\n");
105         }
106         pos++;
107     }
108     br.close();
109     return sb.toString();
110 }
111
112 /**
113  * Receives and stores a message properly
114  *
115  * @param out
116  *     - Output Stream
117  * @param in
118  *     - Input Stream
119  * @param callingUser
120  *     - The user currently logged it to save who sent the message
121  */
122 public static void receiveMessage(ObjectOutputStream out, ObjectInputStream in,

```

```

String contact, String message,
123     String callingUser) {
124
125     try {
126
127         String destination = Files.getDestination(contact, callingUser);
128         // -1 - not registered
129         // 0 - group
130         // 1 - user
131         if (destination == "") {
132
133             // Tells destination doesn't exist
134             // out.writeBoolean(false);
135             System.out.println("Message destinary doesn't exist!");
136             out.writeObject("Message destinary doesn't exist!");
137             out.flush();
138         } else {
139
140             writeMessage(message, callingUser, destination);
141             System.out.println("Message sent from " + callingUser + " to " +
contact);
142             out.writeObject("Message sent from " + callingUser + " to " +
contact);
143             out.flush();
144         }
145
146     } catch (IOException e) {
147
148         System.err.println("Error receiving message!");
149         try {
150             out.writeObject("Error sending message");
151             out.flush();
152         } catch (IOException e1) {
153             // e1.printStackTrace();
154         }
155         // e.printStackTrace();
156     }
157
158 }
159
160 /**
161  * Writes a message to disk
162  *
163  * @param message
164  *     - The message to be written
165  * @param callingUser
166  *     - The one who sends the message
167  * @param destination
168  *     - The one who receives the message
169  */
170 public static void writeMessage(String message, String callingUser, String
destination) throws IOException {
171     System.out.println("Message: " + message);
172
173     // Create a conversation folder
174     File messagePath = new File(Configurations.MESSAGES_FOLDER + "/" +
destination);
175     if (!messagePath.exists()) {
176         messagePath.mkdir();
177     }
178
179     // Build the message
180     StringBuilder sb = new StringBuilder();
181

```

Message.java

```

182     sb.append(callingUser);
183     sb.append("\n");
184     sb.append(message);
185     sb.append("\n");
186
187     Date date = new Date();
188     SimpleDateFormat ft = new SimpleDateFormat("yyyy-MM-dd hh:mm");
189     sb.append(ft.format(date));
190
191     // Write the message to disk
192     BufferedWriter msg = new BufferedWriter(
193         new FileWriter(Configurations.MESSAGES_FOLDER + "/" + destination +
194         "/" + date.getTime() + ".msg"));
195     msg.write(sb.toString());
196     msg.close();
197
198     /*****
199     * CLIENT SIDE
200     *****/
201
202     /**
203     * Sends a message to the server
204     *
205     * @param out
206     *      - Output Stream
207     * @param in
208     *      - Input Stream
209     * @param destination
210     *      - The user or group for the message to be sent
211     * @param message
212     *      - The message to be sent
213     */
214     public static void sendMessage(ObjectOutputStream out, ObjectInputStream in) {
215
216         try {
217             // Receive status from message delivery
218             String result = (String) in.readObject();
219             System.out.println(result);
220
221         } catch (IOException | ClassNotFoundException e) {
222             // TODO Auto-generated catch block
223             System.err.println("Error sending message to server!");
224             // e.printStackTrace();
225         }
226     }
227 }
228

```

Usage.java

```
1 package functionality;
2
3 /**
4  * This class handles usage display of application
5  */
6 public class Usage {
7
8     /**
9      * Prints the usage menu
10     */
11     public static void printUsage() {
12
13         System.out.println("myWhats <localUser> <serverAddress> [ -p <password> ]\n"
14                             + "                                [ -m <contact> "
15                             + "                                [ -f <contact> "
16                             + "                                [ -r [contact] "
17                             + "                                [ -a <user> <group> "
18                             + "                                [ -d <user> <group> "
19                             + "                                ]\n");
20     }
21 }
```


User.java

```
1 package functionality;
2
3 import java.io.BufferedReader;
4 import java.io.BufferedWriter;
5 import java.io.FileNotFoundException;
6 import java.io.FileReader;
7 import java.io.FileWriter;
8 import java.io.IOException;
9
10 /**
11  * This class handles User operations
12  */
13 public class User {
14
15     /**
16      * Adds a user to the credentials file
17      *
18      * @param login
19      *      - Credentials to be added
20      */
21     static boolean createUser(String login) {
22
23         try {
24
25             // Insert credentials in the creds file
26             BufferedWriter bw = new BufferedWriter(new
27 FileWriter(Configurations.CREDENTIALS_FILENAME, true));
28
29             bw.append(login);
30             bw.newLine();
31             bw.close();
32
33             // Create user's personal folder
34             String[] parseCreds = login.split(":");
35
36             BufferedWriter uf = new BufferedWriter(
37 new FileWriter(Configurations.USERS_FOLDER + "/" + parseCreds[0]
38 + ".cfg", true));
39             uf.close();
40
41             System.out.println("New user registered");
42
43             return true;
44
45         } catch (IOException e) {
46
47             System.err.println("Error opening credentials file for user insertion");
48             // e.printStackTrace();
49
50             return false;
51         }
52     }
53
54     /**
55      * Verifies if a username is present in the credentials file
56      *
57      * @param username
58      *      - User's username
59      *
60      * @return If username is present it's credentials will be returned
61      *      otherwise, empty will be returned
62      * @throws IOException
63      */
64     static String userExists(String username) throws IOException {
```

```
63     try {
64
65         BufferedReader br = new BufferedReader(new
66         FileReader(Configurations.CREDENTIALS_FILENAME));
67
68         String line;
69         String[] credLine;
70         while ((line = br.readLine()) != null) {
71
72             credLine = line.split(":");
73
74             if (credLine[0].equals(username)) {
75                 br.close();
76                 System.out.println("Username exists in credentials file");
77                 return line;
78             }
79
80             br.close();
81
82         } catch (FileNotFoundException e) {
83
84             System.err.println("Error opening credentials file for user
85             verification");
86             // e.printStackTrace();
87
88             System.out.println("Username not found");
89             return "";
90         }
91     }
92
```

myWhatsServer.java

```
1 package server;
2
3 /*****
4  *           Seguranca e Confiabilidade 2015/16
5  *****/
6
7 import java.io.File;
8 import java.io.IOException;
9 import java.io.ObjectInputStream;
10 import java.io.ObjectOutputStream;
11 import java.net.ServerSocket;
12 import java.net.Socket;
13
14 import functionality.*;
15
16 //Servidor do servico myWhatsServer
17 public class myWhatsServer {
18
19     //MAIN
20     public static void main(String[] args) {
21         System.out.println("Server running");
22         myWhatsServer sv = new myWhatsServer();
23         sv.startServer();
24     }
25
26     //START SERVER
27     @SuppressWarnings("resource")
28     public void startServer() {
29         ServerSocket sSoc = null;
30
31         try {
32
33             sSoc = new ServerSocket(23456);
34
35         } catch (IOException e) {
36
37             System.err.println(e.getMessage());
38             System.exit(-1);
39
40         }
41
42         //
43         *****/
44
45         // Check if USERS file exist
46         File credentials = new File(Configurations.CREDENTIALS_FILENAME);
47         if (!credentials.exists() && !credentials.isDirectory()) {
48
49             try {
50
51                 credentials.createNewFile();
52
53             } catch (IOException e) {
54                 System.err.println("Error creating user file");
55                 e.printStackTrace();
56
57             }
58
59         }
60
61         // Check if GROUPS file exist
62         File groups = new File(Configurations.GROUPS_FILENAME);
63         if (!groups.exists() && !groups.isDirectory()) {
64
65             try {
```

```

64
65         groups.createNewFile();
66
67     } catch (IOException e) {
68         System.err.println("Error creating groups file");
69         e.printStackTrace();
70
71     }
72 }
73
74 // Check if users FOLDER exist
75 File usersFolder = new File(Configurations.USERS_FOLDER);
76 if (!usersFolder.exists()) {
77
78     usersFolder.mkdir();
79 }
80
81 // Check if groups FOLDER exist
82 File groupsFolder = new File(Configurations.GROUPS_FOLDER);
83 if (!groupsFolder.exists()) {
84
85     groupsFolder.mkdir();
86 }
87
88 // Check if messages FOLDER exist
89 File messagesFolder = new File(Configurations.MESSAGES_FOLDER);
90 if (!messagesFolder.exists()) {
91
92     messagesFolder.mkdir();
93 }
94
95 //
96 *****
97 // Server Main - Client reception and thread creation
98 while (true) {
99     try {
100         Socket inSoc = sSoc.accept();
101         ServerThread newServerThread = new ServerThread(inSoc);
102         newServerThread.start();
103     } catch (IOException e) {
104         e.printStackTrace();
105     }
106 }
107 }
108 }
109
110 //
111 *****
112 // Threads utilizadas para comunicacao com os clientes
113 class ServerThread extends Thread {
114
115     private Socket socket = null;
116
117     ServerThread(Socket inSoc) {
118         socket = inSoc;
119         System.out.println("Client connected from " + socket.getInetAddress());
120     }
121
122     // Run the thread
123     public void run() {
124
125         try {

```

```

126         // OPEN STREAMS
127         ObjectOutputStream out = new
ObjectOutputStream(socket.getOutputStream());
128         ObjectInputStream in = new
ObjectInputStream(socket.getInputStream());
129
130         String user = null;
131         String passwd = null;
132         // END OPEN STREAMS
133
134         // RECEIVE CREDENTIALS
135         try {
136             user = (String) in.readObject();
137             passwd = (String) in.readObject();
138
139             System.out.println(
140                 "Authentication attempt\n>" + socket.getInetAddress() +
" | " + user + " | " + passwd);
141
142             } catch (ClassNotFoundException e1) {
143                 e1.printStackTrace();
144             }
145         // END RECEIVE CREDENTIALS
146
147         // AUTHENTICATION
148         boolean wasSuccessful = false;
149
150         if (user.length() > 0 && passwd.length() > 0) {
151             wasSuccessful = Authentication.authenticateUser(in, out, (user
+ ":" + passwd));
152         }
153
154         // END AUTHENTICATION
155
156         if (wasSuccessful) {
157             try {
158                 String flag = (String) in.readObject();
159                 String[] argArray = (String[]) in.readObject();
160                 String destination;
161
162                 // Proceede according to flag sent
163                 switch (flag) {
164                     case "-m":
165                         System.out.println("Message To: " + argArray[0] + " - "
+ argArray[1]);
166                         Message.receiveMessage(out, in, argArray[0],
argArray[1], user);
167
168                         break;
169
170                     case "-f":
171                         System.out.println("File to: " + argArray[0] + " - " +
argArray[1]);
172
173                         destination = Files.getDestination(argArray[0], user);
174
175                         if (destination != "") {
176
177                             boolean success = Files.receiveFile(in, out,
Configurations.MESSAGES_FOLDER + "/" +
178 destination, argArray[1]);
179
180                             if (success) {
181                                 Message.writeMessage(argArray[1], user,

```

```

destination);
182         }
183
184     } else {
185
186         // Send that is not a valid destination
187         out.writeBoolean(false);
188         out.flush();
189
190         System.out.println("ERROR! Not a user or group!");
191     }
192     break;
193
194     case "-r":
195         if (argArray.length == 0) {
196             System.out.println("Send all communications");
197
198             // LIST ALL FOLDERS THAT CONTAIN THE USER'S
199             // USERNAME
200
201             String[] folderList =
Files.listFolders(Configurations.MESSAGES_FOLDER);
202             StringBuilder convo = new StringBuilder();
203             for (String s : folderList) {
204
205                 if (s.matches(".*#.*")) {
206
207                     String[] tmp = s.split("#");
208
209                     // RECONSTRUCT CONVERSATION
210                     if (tmp[0].equals(user)) {
211
212                         convo.append(Message
213                             .buildConversation(tmp[1],
214 user, true));
215
216                         }else if( tmp[1].equals(user)){
217                             convo.append(Message
218                                 .buildConversation(tmp[0],
219 user, true));
220
221                         }
222                     }
223                 }
224
225                 // SEND
226                 out.writeObject(convo.toString());
227                 out.flush();
228
229             } else if (argArray.length == 1) {
230
231                 String convo =
Message.buildConversation(argArray[0], user, false);
232
233                 // SEND
234                 out.writeObject(convo);
235                 out.flush();
236
237             } else if (argArray.length == 2) {
238                 System.out.println("Get file " + argArray[1] + " in
" + argArray[0]);
239
240                 // OPEN CONVERSATION FOLDER
241                 destination = Files.getDestination(argArray[0],
242 user);
243
244                 File file = new File(Configurations.MESSAGES_FOLDER

```

```

+ "/" + destination + "/" + argArray[1]);
239
240         if(file.exists()){
241             out.writeBoolean(true);
242             out.flush();
243             Files.sendFile(in, out,
Configurations.MESSAGES_FOLDER + "/" + destination + "/", argArray[1]);
244         }else{
245             out.writeBoolean(false);
246             out.flush();
247         }
248     }
249     break;
250
251     case "-a":
252         System.out.println("Adds user " + argArray[0] + " to
group " + argArray[1]);
253         Group.addGroup(argArray[0], argArray[1], user, out,
in);
254
255         break;
256
257     case "-d":
258         System.out.println("Delete user " + argArray[0] + "
from group " + argArray[1]);
259         Group.removeFromGroup(argArray[0], argArray[1], user);
260         break;
261
262     default:
263         System.out.println("Incorrect parameters!\n");
264     }
265
266     } catch (IOException | ClassNotFoundException e) {
267         System.out.println("No command was sent from client or
unexpected data type!");
268         // e.printStackTrace();
269     }
270 }
271
272 // END SERVER MAIN
273
274 System.out.println("--*==*==*==*-- END OF SESSION --*==*==*==");
275 // SOCKET AND STREAM CLOSING
276 out.close();
277 in.close();
278
279 socket.close();
280 // END SOCKET AND STREAM CLOSING
281
282 } catch (IOException e) {
283     System.err.println("Error running server thread!");
284     //e.printStackTrace();
285 }
286
287 }
288
289 }

```