Segurança e Confiabilidade
2015-16

Grupo XXIII

fc41941 – Rodrigo Reis

fc41964 – Tito Oliveira

fc45773 – Jonas Ferreira

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

- Crypto.java – Trata de todas as funcionalidades da criptografia do projeto

- 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

Alterações efectuadas ao projecto:

- O servidor autentica com sucesso os utilizadores caso as informações inseridas por estes sejam válidas.
- O servido guarda uma síntese de “password_salt” em formato de texto. No inicio da execução do servidor tem que ser dada uma password para utilizar nos MAC’S dos ficheiros de configuração.
- Os ficheiros que são protegidos por MAC são: ficheiro de policy do server, ficheiro de credenciais, ficheiro de configuraçao pessoal de utilizador (username.cfg) e ficheiro de configuraçao dos grupos (groupname.cfg)
- O cliente e o servidor utilizam sockets TLS/SSL e ambos encontram se com a sandbox activa.

- O cliente consegue cifra qualquer mensagem ou ficheiros com cifras hibridas seja qual for o destinatário.
- A funcionalidade de adicionar e remover um utilizador a um grupo.

Os algoritmos de síntese utilizados no trabalho para confidencialidade das passwords são HmacSHA256 Base64

O algoritmo utilizado nas cifras hibridas e nos MAC'S é HmacSHA256

Funcionalidades que não foram implementadas:

- Assinaturas digitais.
- Decifragem de Mensagens e Ficheiros por parte do Cliente

```java
package functionality;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Scanner;

/**
 * This class handles the authentication process
 */
public class Authentication {

    /************************************************************
     * CLIENT SIDE
     ************************************************************/

    /**
     * Authenticates with the server
     *
     * @param in
     *            - ObjectInputStream
     * @param out
     *            - ObjectOutputStream
     * @param username
     *            - User's username
     * @param password
     *            TODO
     * @return true if user was successfully logged in
     */
    public static boolean login(ObjectInputStream in, ObjectOutputStream out, String username, String password) {

        boolean userExists = false;
        boolean usernameAvailable = false;
        boolean successLogin = false;
        int salt = 0;

        try {
            System.out.println("Trying to authenticate...");

            out.writeObject(username);
            out.flush();

            // Receive if user exists
            userExists = in.readBoolean();

            // If user is doesn't exist
            if (!userExists) {

                // Check if available
                usernameAvailable = in.readBoolean();

                if (usernameAvailable) {

                    @SuppressWarnings("resource")
                    Scanner read = new Scanner(System.in);
                    String ans = "";

                    // Keep prompting for a valid answer to register or not
                    while (!ans.equals("n") && !ans.equals("y")) {

                        System.out.println("This username is not registered. Do you want to register it? (Y/N)");
                        ans = read.nextLine().toLowerCase();
```

```
63                          System.out.println(ans);
64                      }
65
66                      // read.close();
67
68                      if (ans.equals("y")) {
69                          out.writeBoolean(true);
70                          out.flush();
71
72                          // Receber o salt
73                          salt = in.readInt();
74
75                          // Saltear e enviar a pass
76                          String hashedPass = Crypto.hashPassword(password, salt);
77
78                          out.writeObject(hashedPass);
79                          out.flush();
80
81                      } else {
82                          out.writeBoolean(false);
83                          out.flush();
84                      }
85                  }
86
87                  successLogin = in.readBoolean();
88
89                  if (successLogin) {
90                      System.out.println("User registered successfully!");
91                  } else {
92                      System.out.println("User not registered!");
93                  }
94
95                  // Se o user existir verificar se foi logado com sucesso
96              } else {
97
98                  // Receber o salt
99                  salt = in.readInt();
100
101                 // Saltear e enviar a pass
102                 String hashedPass = Crypto.hashPassword(password, salt);
103
104                 out.writeObject(hashedPass);
105                 out.flush();
106
107                 successLogin = in.readBoolean();
108
109                 if (successLogin) {
110                     System.out.println("Successfull Authentication!");
111                     return true;
112                 } else {
113                     System.out.println("Failed Authentication! Incorrect
    Credentials!");
114                     return false;
115                 }
116             }
117
118         } catch (IOException e) {
119
120             System.err.println("Error during autentication!");
121             // e.printStackTrace();
122         }
123
124         return false;
125     }
```

```
126
127     /*************************************************************
128      * SERVER SIDE
129      *************************************************************/
130
131     /**
132      * Authenticates a user - If user exists checks for password, if not,
133      * registers
134      *
135      * @param in
136      *            - ObjectInputStream
137      * @param out
138      *            - ObjectOutputStream
139      * @param username
140      *            - Credentials supplied during login in format user:password
141      * @return True if password correct or new user was registered, False
142      *         otherwise
143      * @throws ClassNotFoundException
144      */
145     public static boolean authenticateUser(ObjectInputStream in, ObjectOutputStream
    out, String username,
146             String serverpass) throws ClassNotFoundException {
147
148         String creds;
149         String pass;
150
151         boolean register = false;
152
153         try {
154             // Verify if user exists (returns user:passhash:salt)
155             if ((creds = User.userExists(username, serverpass)) != "") {
156
157                 // Send that the user exists
158                 out.writeBoolean(true);
159                 out.flush();
160
161                 String[] registeredCredentials = creds.split(":");
162
163                 // Send salt
164                 out.writeInt(Integer.parseInt(registeredCredentials[2]));
165                 out.flush();
166
167                 // Receive hashed password
168                 pass = (String) in.readObject();
169
170                 // If password matches the one registered
171                 if (registeredCredentials[1].equals(pass)) {
172                     System.out.println(registeredCredentials[0] + " logged in with
    success");
173                     out.writeBoolean(true);
174                     out.flush();
175                     return true;
176
177                 } else {
178
179                     System.out.println(registeredCredentials[0] + "'s password
    doesn't match");
180                     out.writeBoolean(false);
181                     out.flush();
182                     return false;
183                 }
184
185                 // User doesn't exist, register it
186             } else {
```

```
187
188                 // Send that the user doesn't exist
189                 out.writeBoolean(false);
190                 out.flush();
191
192                 // Verify if a group exists with the same name
193                 if (!Group.groupExists(username, serverpass)) {
194
195                     // Tell username is available
196                     out.writeBoolean(true);
197                     out.flush();
198
199                     register = in.readBoolean();
200
201                     if (register) {
202
203                         int salt = Crypto.generateRandomSixDigit();
204
205                         // Send salt
206                         out.writeInt(salt);
207                         out.flush();
208
209                         // Receive hashed password
210                         pass = (String) in.readObject();
211
212                         boolean result = User.createUser(username + ":" + pass +
    ":" + salt, serverpass);
213                         out.writeBoolean(result);
214                         out.flush();
215                         return result;
216
217                     } else {
218                         System.out.println("User not registered!");
219                         out.writeBoolean(false);
220                         out.flush();
221                         return false;
222                     }
223
224                 } else {
225
226                     // Send username is not available
227                     out.writeBoolean(false);
228                     out.flush();
229                     return false;
230                 }
231             }
232
233         } catch (IOException e) {
234
235             System.err.println("Error during authentication!");
236             // e.printStackTrace();
237         }
238
239         return false;
240     }
241 }
242
```

```java
package functionality;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

import javax.net.ssl.SSLSession;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;

/**
 * This class handles communication between client/server
 */
public class Communication {

    /**
     * Connects to a server in the address and port provided
     *
     * @param addressAndPort
     *            - Server's address and port in X.X.X.X:YYYY FORMAT
     * @return A socket with the connection to the server, null in case of fail
     */
    public static Socket connect(String addressAndPort) {

        Socket sock = null;

        try {
            if (addressAndPort.matches("\\d+\\.\\d+\\.\\d+\\.\\d+\\:\\d+")) {
                String[] addPort = addressAndPort.split(":");
                SSLSocketFactory ssf = (SSLSocketFactory) SSLSocketFactory.getDefault();
                sock = ssf.createSocket(addPort[0], Integer.parseInt(addPort[1]));

                SSLSession session = ((SSLSocket) sock).getSession();

                System.out.println("The Certificates used by peer");

                System.out.println("Peer host is " + session.getPeerHost());
                System.out.println("Cipher is " + session.getCipherSuite());
                System.out.println("Protocol is " + session.getProtocol());
                System.out.println("Session created in " + session.getCreationTime());
                System.out.println("Session accessed in " + session.getLastAccessedTime());
            }
        } catch (IOException e) {

            System.err.println("Error connecting to server!\nCheck if server or connection are down!");
            // e.printStackTrace();
        }
        return sock;
    }

    /**
     * Sends a command flag and it's arguments to the server
     *
     * @param out
     *            - ObjectOutputStream
     * @param flag
     *            - The command flag for the operation
     * @param args
     *            - The arguments for the operation
```

```
61      */
62     public static void sendCommand(ObjectOutputStream out, String flag, String[]
   args) {
63
64         try {
65             out.writeObject(flag);
66             out.writeObject(args);
67             out.flush();
68
69         } catch (IOException e) {
70             System.err.println("Error sending flag and arguments to server!");
71             // e.printStackTrace();
72         }
73     }
74
75     public static void sendFileOrGroup(ObjectOutputStream out, String dest, String
   serverpass) {
76
77         String[] userList = null;
78
79         try {
80             // Enviar se é cliente ou grupo
81             // 0 - Group
82             // 1 - User
83             int val = Group.isUserOrGroup(dest, serverpass);
84
85             // Se der erro
86             if (val == -1) {
87                 out.writeInt(-1);
88                 out.flush();
89
90                 // É Grupo
91             } else if (val == 0) {
92
93                 out.writeInt(0);
94                 out.flush();
95                 userList = Group.membersList(Configurations.GROUPS_FOLDER + "/" +
   dest + ".cfg");
96                 out.writeObject(userList);
97                 out.flush();
98
99                 // É user
100            } else if (val == 1) {
101
102                userList = new String[] { dest };
103                out.writeInt(1);
104                out.flush();
105                out.writeObject(userList);
106                out.flush();
107
108            }
109
110        } catch (IOException e) {
111            // TODO Auto-generated catch block
112            e.printStackTrace();
113        }
114    }
115
116    public static String[] receiveUserOrGroup(ObjectInputStream in) {
117
118        String[] memberList = null;
119
120        try {
121            // receive group/user/error indication
```

```
122
123             int val = in.readInt();
124
125             // Error
126             if (val == -1) {
127                 System.err.println("Error receiving if user or group!");
128
129                 // GROUP
130             } else if (val == 0) {
131
132                 memberList = (String[]) in.readObject();
133
134                 // USER
135             } else if (val == 1) {
136                 memberList = (String[]) in.readObject();
137             }
138
139         } catch (ClassNotFoundException e) {
140             // TODO Auto-generated catch block
141             e.printStackTrace();
142         } catch (IOException e) {
143             // TODO Auto-generated catch block
144             e.printStackTrace();
145         }
146
147         return memberList;
148     }
149 }
150
```

```java
package functionality;

/**
 * The configurations class contains the definition of various paths and data
 * sizes
 */
public class Configurations {

    // Credential file and group list file
    public final static String CREDENTIALS_FILENAME = "!credentialsFile";
    public final static String GROUPS_FILENAME = "!groupsFile";

    // User and group data saving destination
    public final static String USERS_FOLDER = "users";
    public final static String GROUPS_FOLDER = "groups";

    // Messages saving destination
    public final static String MESSAGES_FOLDER = "messages";

    // Path to files received from -r DEST FILENAME
    public final static String DOWNLOAD_FOLDER = "downloads";

    // Client policy file
    public final static String CLIENT_POLICY = "client.policy";

    // Server policy file
    public final static String SERVER_POLICY = "server.policy";

    // Server Keystore File
    public final static String KEYSTORE_NAME = "keystore.jks";

    // Client Truststore File
    public final static String TRUSTSTORE_NAME = "truststore.jks";

    // Data block size to send and receive files
    public final static int DATA_BLOCK = 1024;
}

```

```java
package functionality;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileReader;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.UnsupportedEncodingException;

import java.security.InvalidKeyException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Signature;
import java.security.SignatureException;
import java.security.UnrecoverableKeyException;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.util.Arrays;
import java.util.Base64;

import javax.crypto.Cipher;
import javax.crypto.CipherOutputStream;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.Mac;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;

public class Crypto {

    /**
     * Writes a file with a mac
     *
     * @param mac
     *            - Mac to be written
     * @param filename
     *            - File to associate the mac with
     */
    public static void writeMacFile(byte[] mac, String filename) {

        try {

            if (mac.length != 0) {
                FileOutputStream fos = new FileOutputStream(filename + ".mac",
false);
                ObjectOutputStream out = new ObjectOutputStream(fos);
                out.writeObject(mac);
                out.close();
                fos.close();
            } else {
                File f = new File(filename + ".mac");
                f.createNewFile();
            }

```

```
64          } catch (IOException e) {
65
66              // TODO Auto-generated catch block
67              e.printStackTrace();
68          }
69      }
70
71      /**
72       * Reads the mac from a file
73       *
74       * @param filename
75       *            - File tor ead the mac from
76       * @return Byte array with mac
77       */
78      public static byte[] readMacFile(String filename) {
79
80          byte[] mac = null;
81
82          try {
83
84              FileInputStream fis = new FileInputStream(filename + ".mac");
85              ObjectInputStream out = new ObjectInputStream(fis);
86              mac = (byte[]) out.readObject();
87              out.close();
88              fis.close();
89
90          } catch (IOException | ClassNotFoundException e) {
91
92              // TODO Auto-generated catch block
93              // e.printStackTrace();
94          }
95
96          return mac;
97      }
98
99      /**
100      * Reads a file a generates a mac from it
101      *
102      * @param password
103      *            - Password to associate to this mac
104      * @param filename
105      *            - File to have the mac calculated
106      * @return Byte with mac generated from file and password
107      */
108     public static byte[] calculateFileMAC(String password, String filename) {
109
110         byte[] digest = null;
111
112         try {
113
114             SecretKey key = new SecretKeySpec(password.getBytes(), "HmacSHA256");
115
116             Mac mac = Mac.getInstance("HmacSHA256");
117             mac.init(key);
118
119             BufferedReader bf = new BufferedReader(new FileReader(filename));
120
121             String tmp = "";
122             while ((tmp = bf.readLine()) != null) {
123                 mac.update(tmp.getBytes("UTF-8"));
124             }
125             bf.close();
126             digest = mac.doFinal();
127
```

```
128         } catch (NoSuchAlgorithmException e) {
129             // TODO Auto-generated catch block
130             e.printStackTrace();
131         } catch (InvalidKeyException e) {
132             // TODO Auto-generated catch block
133             e.printStackTrace();
134         } catch (UnsupportedEncodingException e) {
135             // TODO Auto-generated catch block
136             e.printStackTrace();
137         } catch (FileNotFoundException e) {
138             // TODO Auto-generated catch block
139             e.printStackTrace();
140         } catch (IllegalStateException e) {
141             // TODO Auto-generated catch block
142             e.printStackTrace();
143         } catch (IOException e) {
144             // TODO Auto-generated catch block
145             e.printStackTrace();
146         }
147 
148         return digest;
149     }
150 
151     /**
152      * Checks if a .mac file exists for filename
153      *
154      * @param filename
155      *            - The file's name to be checked
156      * @return True if a mac file exists for filename File
157      */
158     public static boolean isMacProtected(String filename) {
159 
160         File f = new File(filename + ".mac");
161 
162         return (f.exists() && !f.isDirectory());
163     }
164 
165     /**
166      * Verifies if file has valid mac
167      *
168      * @param filename
169      *            - The file's name
170      * @param serverpass
171      *            - The server running pass
172      * @return True if file is correct, System shutdown otherwise
173      */
174     public static boolean hasValidMac(String filename, String serverpass) {
175 
176         // CALCULATE MAC
177         byte[] credentialsMacDigest = Crypto.calculateFileMAC(serverpass,
    filename);
178 
179         // CHECK IF PASS FILE IS PROTECTED WITH MAC
180         if (Crypto.isMacProtected(filename)) {
181 
182             byte[] storedMac = Crypto.readMacFile(filename);
183 
184             // IF SO OK
185             if (Arrays.equals(credentialsMacDigest, storedMac)) {
186                 return true;
187 
188                 // IF NOT ERROR AND CLOSE
189             } else {
190                 System.err.println("Error! <" + filename + "> MAC mismatch! File
```

```
might have been tampered with!");
                System.exit(-1);
            }

        } else {
            System.err.println(filename + " is not MAC protected. Cannot check for valid MAC");
        }

        return false;
    }

    /**
     * Returns a random 6 digit integer
     *
     * @return a random 6 digit integer
     */
    public static int generateRandomSixDigit() {
        SecureRandom sr = new SecureRandom();

        return sr.nextInt(899999) + 100000;
    }

    /**
     * Salts and hashes a password
     *
     * @param password
     *            - The password
     * @param salt
     *            - The salt
     * @return - A salted and hashed password in base64
     */
    public static String hashPassword(String password, int salt) {

        String sal = new Integer(salt).toString();
        String ingredientes = password + sal;

        SecretKey key = new SecretKeySpec(ingredientes.getBytes(), "HmacSHA256");

        String salteadinho = Base64.getEncoder().encodeToString(key.getEncoded());

        return salteadinho;
    }

    /**
     * Calculates and updates a file's MAC, creates it if doesn't exist
     *
     * @param filename
     *            - The file to be Mac'd
     * @param serverpass
     *            - The server's pass
     */
    public static void updateMAC(String filename, String serverpass) {

        byte[] mac = Crypto.calculateFileMAC(serverpass, filename);
        Crypto.writeMacFile(mac, filename);
    }

    public static void removeMAC(String filename) {
        Files.removeFile(filename);
    }

    static byte[] computeSignature(String filename, String certUser, String certPass, byte[] byteArray) {
```

```
252
253         byte[] signature = null;
254
255         try {
256             // Start the digital signature algorithm with server's private key
257             Signature sign = Signature.getInstance("SHA256withRSA");
258
259             FileInputStream kfile = new
    FileInputStream(Configurations.KEYSTORE_NAME);
260             KeyStore kstore = KeyStore.getInstance("JKS");
261
262             kstore.load(kfile, "storepass".toCharArray());
263
264             PrivateKey key = (PrivateKey) kstore.getKey(certUser,
    certPass.toCharArray());
265             sign.initSign(key);
266
267             if (filename != null && byteArray == null) {
268
269                 String tmp;
270                 BufferedReader br = new BufferedReader(new FileReader(filename));
271
272                 while ((tmp = br.readLine()) != null) {
273                     sign.update(tmp.getBytes());
274                 }
275
276                 br.close();
277
278             } else if (filename == null && byteArray != null) {
279
280                 sign.update(byteArray);
281
282             } else {
283
284                 return null;
285             }
286
287             signature = sign.sign();
288
289         } catch (UnrecoverableKeyException e) {
290             // TODO Auto-generated catch block
291             e.printStackTrace();
292         } catch (InvalidKeyException e) {
293             // TODO Auto-generated catch block
294             e.printStackTrace();
295         } catch (NoSuchAlgorithmException e) {
296             // TODO Auto-generated catch block
297             e.printStackTrace();
298         } catch (FileNotFoundException e) {
299             // TODO Auto-generated catch block
300             e.printStackTrace();
301         } catch (KeyStoreException e) {
302             // TODO Auto-generated catch block
303             e.printStackTrace();
304         } catch (CertificateException e) {
305             // TODO Auto-generated catch block
306             e.printStackTrace();
307         } catch (SignatureException e) {
308             // TODO Auto-generated catch block
309             e.printStackTrace();
310         } catch (IOException e) {
311             // TODO Auto-generated catch block
312             e.printStackTrace();
313         }
```

```
314
315         return signature;
316     }
317
318     public static void writeSignature(byte[] signature, String filename) {
319
320         try {
321
322             FileOutputStream fos = new FileOutputStream(filename + ".sig", false);
323             ObjectOutputStream out = new ObjectOutputStream(fos);
324             out.writeObject(signature);
325             out.close();
326             fos.close();
327
328         } catch (IOException e) {
329
330             // TODO Auto-generated catch block
331             e.printStackTrace();
332         }
333     }
334
335     /**
336      *
337      * @param filename
338      * @return
339      */
340     public static boolean hasSignature(String filename) {
341
342         File f = new File(filename + ".sig");
343
344         return (f.exists() && !f.isDirectory());
345     }
346
347     /**
348      *
349      * @return
350      */
351     public static SecretKey randomAESKey() {
352
353         SecretKey key = null;
354
355         try {
356
357             KeyGenerator kg = KeyGenerator.getInstance("AES");
358             kg.init(128);
359             key = kg.generateKey();
360
361         } catch (NoSuchAlgorithmException e) {
362             // TODO Auto-generated catch block
363             e.printStackTrace();
364         }
365
366         return key;
367     }
368
369     public static void encryptFile(String filename, SecretKey key) {
370
371         try {
372             File file = new File(filename);
373             if (file.length() == 0)
374                 return;
375
376             // Initialize the encode cipher
377             Cipher aes = Cipher.getInstance("AES");
```

```
378             aes.init(Cipher.ENCRYPT_MODE, key);
379
380             // Temporary File to Encode
381             File tempFile = new File("encoding.temp");
382
383             // Input
384             FileInputStream input = new FileInputStream(file);
385
386             // Cypher file output
387             FileOutputStream output = new FileOutputStream(tempFile);
388             CipherOutputStream encryptedOut = new CipherOutputStream(output, aes);
389
390             // Encrypt
391             byte[] b = new byte[128];
392             int redBytes = input.read(b);
393
394             while (redBytes != -1) {
395                 encryptedOut.write(b, 0, redBytes);
396                 redBytes = input.read(b);
397             }
398
399             encryptedOut.close();
400             input.close();
401
402             // Delete Original and rename encoded file
403             file.delete();
404             tempFile.renameTo(file);
405
406         } catch (Exception e) {
407             e.printStackTrace();
408             System.err.println("Error encoding file");
409         }
410
411     }
412
413     public static byte[] encryptMessage(byte[] byteArray, SecretKey key) {
414
415         byte[] toReturn = null;
416
417         try {
418
419             // Initialize the encode cipher
420             Cipher aes = Cipher.getInstance("AES");
421             aes.init(Cipher.ENCRYPT_MODE, key);
422
423             toReturn = aes.doFinal(byteArray);
424
425         } catch (Exception e) {
426             System.err.println("Error encoding file");
427         }
428
429         return toReturn;
430     }
431
432     public static byte[][] encryptWithPrivateKey(byte[] toEncrypt, String[]
    members, SecretKey key,
433             String keystoreFile) {
434
435         byte[][] keyEncoded = new byte[members.length][];
436
437         try {
438
439             FileInputStream kfile = new FileInputStream(keystoreFile);
440             KeyStore kstore = KeyStore.getInstance("JKS");
```

```
441
442             kstore.load(kfile, "storepass".toCharArray());
443
444             int index = 0;
445             for (String s : members) {
446
447                 // PUBLICK KEY
448                 Certificate cert = kstore.getCertificate(s);
449                 Cipher rsa = Cipher.getInstance("RSA");
450                 PublicKey publicKey = cert.getPublicKey();
451                 rsa.init(Cipher.WRAP_MODE, publicKey);
452
453                 keyEncoded[index] = rsa.wrap(key);
454                 index++;
455
456             }
457
458             kfile.close();
459
460         } catch (NoSuchAlgorithmException e) {
461             // TODO Auto-generated catch block
462             e.printStackTrace();
463
464         } catch (KeyStoreException e) {
465             // TODO Auto-generated catch block
466             e.printStackTrace();
467         } catch (FileNotFoundException e) {
468             // TODO Auto-generated catch block
469             e.printStackTrace();
470         } catch (CertificateException e) {
471             // TODO Auto-generated catch block
472             e.printStackTrace();
473         } catch (IOException e) {
474             // TODO Auto-generated catch block
475             e.printStackTrace();
476         } catch (InvalidKeyException e) {
477             // TODO Auto-generated catch block
478             e.printStackTrace();
479         } catch (NoSuchPaddingException e) {
480             // TODO Auto-generated catch block
481             e.printStackTrace();
482         } catch (IllegalBlockSizeException e) {
483             // TODO Auto-generated catch block
484             e.printStackTrace();
485         }
486
487         return keyEncoded;
488     }
489
490 }
491
```

```java
package functionality;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

/**
 * This class handles files and folder operations
 */
public class Files {

    /**
     * Returns a String array with a list of folder names in path
     *
     * @param path
     *             - The path to map
     * @return - String array with the folders in path
     */
    public static String[] listFolders(String path) {

        File directory = new File(path);

        File[] folderList = directory.listFiles();
        List<String> list = new ArrayList<String>();

        for (File folder : folderList) {
            if (folder.isDirectory()) {
                list.add(folder.getName());
            }
        }

        return list.toArray(new String[list.size()]);
    }

    /**
     * Returns a String array with a list of file names in path
     *
     * @param path
     *             - The path to map
     * @return - String array with the file names in path
     */
    public static String[] listFiles(String path) {

        File directory = new File(path);

        File[] fileList = directory.listFiles();
        List<String> list = new ArrayList<String>();

        if (fileList != null) {
            for (File file : fileList) {
                if (file.isFile()) {
                    list.add(file.getName());
                }
            }
        }
        return list.toArray(new String[list.size()]);
    }

    /**
```

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

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

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

```
254                 // Answers if file already exists
255                 File checkIfExists = new File(fullPath);
256 
257                 if (checkIfExists.exists()) {
258                     System.out.println("File already exists locally!");
259                     out.writeBoolean(true);
260                     out.flush();
261                     return false;
262 
263                 } else {
264                     System.out.println("Receive file!");
265                     out.writeBoolean(false);
266                     out.flush();
267 
268                     // Receive file size
269                     long fileSize = in.readLong();
270 
271                     // Receive file
272                     long recvd = 0;
273                     if (fileSize > 0) {
274 
275                         file = new FileOutputStream(fullPath);
276 
277                         int count;
278                         recvd = 0;
279                         while (recvd < fileSize) {
280                             count = in.read(buffer);
281                             file.write(buffer, 0, count);
282                             recvd += count;
283                         }
284 
285                         file.close();
286 
287                     } else {
288 
289                         checkIfExists.createNewFile();
290                     }
291 
292                     // Send received bytes amnount
293                     out.writeLong(recvd);
294                     out.flush();
295                     return true;
296                 }
297 
298             } else {
299                 System.out.println("User or group does not exist!");
300                 out.writeBoolean(false);
301                 out.flush();
302                 return false;
303             }
304 
305         } catch (IOException e) {
306             System.err.println("Error receiving file!");
307             // e.printStackTrace();
308         }
309 
310         return false;
311     }
312 
313     /**
314      * Removes a file with filename
315      *
316      * @param filename
317      *            - The path to the file to be removed
```

```
318      */
319     public static void removeFile(String filename) {
320         File f = new File(filename);
321         f.delete();
322     }
323 
324     public static void sendCodedKeys(ObjectOutputStream out, String[] memberList,
    byte[][] keys) {
325 
326         try {
327 
328             out.writeObject(memberList);
329             out.flush();
330             out.writeObject(keys);
331             out.flush();
332 
333         } catch (IOException e) {
334             // TODO Auto-generated catch block
335             e.printStackTrace();
336         }
337 
338     }
339 
340     public static void receiveCodedKeysAndWrite(ObjectInputStream in, String
    filename, String destination) {
341         try {
342 
343             String[] memberList = (String[]) in.readObject();
344             byte[][] keys = (byte[][]) in.readObject();
345             int index = 0;
346 
347             for (String s : memberList) {
348 
349                 File keyFile = new File(
350                         Configurations.MESSAGES_FOLDER + "/" + destination + "/" +
    filename + ".key." + s);
351                 FileOutputStream fos = new FileOutputStream(keyFile);
352                 fos.write(keys[index]);
353                 index++;
354                 fos.flush();
355                 fos.close();
356             }
357 
358         } catch (ClassNotFoundException e) {
359             // TODO Auto-generated catch block
360             e.printStackTrace();
361         } catch (IOException e) {
362             // TODO Auto-generated catch block
363             e.printStackTrace();
364         }
365 
366     }
367 }
368 
```

```java
package functionality;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

/**
 * This class handles group operations
 */
public class Group {

    /**
     * Verifies if a groups exists in the groups file
     *
     * @param groupName
     *            - Name of the group to be searched for
     * @param serverpass
     *            TODO
     * @return True if group exists in groups file
     */
    static boolean groupExists(String groupName, String serverpass) {

        // Check if groups file MAC is valid
        if (Crypto.hasValidMac(Configurations.GROUPS_FILENAME, serverpass)) {
            try {

                BufferedReader br = new BufferedReader(new
 FileReader(Configurations.GROUPS_FILENAME));

                String line;
                while ((line = br.readLine()) != null) {

                    if (line.equals(groupName)) {
                        br.close();
                        System.out.println("Group exists in groups file!");
                        return true;
                    }
                }

                br.close();

            } catch (IOException e) {
                System.err.println("Error verifying if group exists!");
                // e.printStackTrace();
            }
        }
        return false;
    }

    /**
     * Checks if contact is a client or a group
     *
     * @param input
     *            - the contact's id
     * @param serverpass
     *            TODO
```

```
64      * @return 1 if it's a user, 0 if it's a group and -1 in case of error
65      */
66     public static int isUserOrGroup(String input, String serverpass) {
67 
68         try {
69             // Check if is User
70             if (User.userExists(input, serverpass) != "") {
71                 return 1;
72 
73                 // Check if is group
74             } else if (groupExists(input, serverpass)) {
75                 return 0;
76             }
77 
78         } catch (IOException e) {
79             System.err.println("Erro verifying if it's a user or a group!");
80             // e.printStackTrace();
81         }
82         return -1;
83     }
84 
85     /**
86      * Creates a new group if it doesn't exist, or adds a member if it does
87      *
88      * @param username
89      *             - Username of the user to be added
90      * @param groupName
91      *             - Name of the group to be created
92      * @param callingUser
93      *             - The user who invokes the method
94      * @param out
95      *             - ObjectOutputStream
96      * @param in
97      *             - ObjectInputStream
98      * @param serverpass
99      *             TODO
100     * @return True if group was created/user added to group successfully, False
101     *         otherwise
102     * @throws IOException
103     */
104    public static boolean addGroup(String username, String groupName, String
    callingUser, ObjectOutputStream out,
105            ObjectInputStream in, String serverpass) throws IOException {
106 
107        try {
108            // Check if there's a user with the supplied group name
109            if (User.userExists(groupName, serverpass).equals("")) {
110 
111                // Check if group already exists
112                if (groupExists(groupName, serverpass)) {
113 
114                    // If group exists, check if calling user is admin
115                    if (isAdmin(groupName, callingUser, serverpass)) {
116 
117                        // Check if user to be added already exists in group
118                        // If it doesn't, add it!
119                        if (!isInGroup(groupName, username, serverpass)
120                                && !(User.userExists(username,
    serverpass).equals(""))) {
121 
122                            if (Crypto.hasValidMac(Configurations.GROUPS_FOLDER +
    "/" + groupName + ".cfg",
123                                    serverpass)) {
124 
```

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

```
179                         bw.newLine();
180
181                         // If user not trying to add himself
182                     } else {
183                         // Adicionar linha do admin
184                         bw.append(callingUser);
185                         bw.newLine();
186
187                         if (Crypto.hasValidMac(Configurations.USERS_FOLDER + "/" +
    callingUser + ".cfg", serverpass)) {
188                             // Add group to calling user user file
189                             BufferedWriter callingUserPersonalFile = new
    BufferedWriter(
190                                     new FileWriter(Configurations.USERS_FOLDER +
    "/" + callingUser + ".cfg", true));
191                             callingUserPersonalFile.write(groupName);
192                             callingUserPersonalFile.newLine();
193                             callingUserPersonalFile.close();
194
195                             Crypto.updateMAC(Configurations.USERS_FOLDER + "/" +
    callingUser + ".cfg", serverpass);
196                         }
197
198                         // Check if user to be added exists
199                         if (!User.userExists(username, serverpass).equals("")) {
200                             // If so add user to group list member
201                             bw.append(username);
202                             bw.newLine();
203
204                             if (Crypto.hasValidMac(Configurations.USERS_FOLDER +
    "/" + username + ".cfg", serverpass)) {
205                                 // Add group to user file
206                                 BufferedWriter usrPersonalFile = new
    BufferedWriter(
207                                         new FileWriter(Configurations.USERS_FOLDER
    + "/" + username + ".cfg", true));
208                                 usrPersonalFile.write(groupName);
209                                 usrPersonalFile.newLine();
210                                 usrPersonalFile.close();
211
212                                 Crypto.updateMAC(Configurations.USERS_FOLDER + "/"
    + username + ".cfg", serverpass);
213                             }
214
215                             System.out.println("Group created succesfully");
216                             out.writeObject("Group created succesfully");
217                             out.flush();
218
219                             // User to eb added is not registered
220                         } else {
221
222                             out.writeObject("Group created but user " + username
223                                     + " could not be added because it doesn't
    exist");
224                             out.flush();
225                             System.out.println("Group created but user " + username
226                                     + " could not be added because it doesn't
    exist");
227                         }
228                     }
229
230                     bw.close();
231                     Crypto.updateMAC(Configurations.GROUPS_FOLDER + "/" + groupName
    + ".cfg", serverpass);
```

```
232
233                     if (Crypto.hasValidMac(Configurations.GROUPS_FILENAME,
    serverpass)) {
234                         // Register group in groups file
235                         BufferedWriter gf = new BufferedWriter(new
    FileWriter(Configurations.GROUPS_FILENAME, true));
236                         gf.write(groupName);
237                         gf.newLine();
238                         gf.close();
239
240                         Crypto.updateMAC(Configurations.GROUPS_FILENAME,
    serverpass);
241                     }
242                 }
243
244                 // If the group to be created conflicts with an already
245                 // registered group/user
246             } else {
247                 out.writeObject("Failed to create group! A user with this name
    already exists!");
248                 out.flush();
249                 System.out.println("Failed to create group! A user with this name
    already exists");
250                 return false;
251             }
252
253         } catch (IOException e) {
254             System.out.println("Failed to create group/add new element!");
255             out.writeObject("Failed to create group! A user with this name already
    exists!");
256             out.flush();
257             // e.printStackTrace();
258         }
259
260         return false;
261     }
262
263     /**
264      * Verifies if a user is admin of a group
265      *
266      * @param groupName
267      *             - The name of the group for this query
268      * @param username
269      *             - The username of the client for this query
270      * @param serverpass
271      *             TODO
272      * @return True if user is admin of group, otherwise, False
273      */
274     private static boolean isAdmin(String groupName, String username, String
    serverpass) {
275
276         if (Crypto.hasValidMac(Configurations.GROUPS_FOLDER + "/" + groupName +
    ".cfg", serverpass)) {
277             try {
278
279                 BufferedReader br = new BufferedReader(
280                         new FileReader(Configurations.GROUPS_FOLDER + "/" +
    groupName + ".cfg"));
281
282                 String line = br.readLine();
283
284                 if (line.equals(username)) {
285                     br.close();
286                     System.out.println("User " + username + " is Admin of group " +
```

```
groupName);
287                     return true;
288                 }
289                 br.close();
290 
291             } catch (IOException e) {
292                 // e.printStackTrace();
293                 System.err.println("Error checking if admin of group!");
294             }
295         }
296         return false;
297     }
298 
299     /**
300      * Verifies if user is in group
301      *
302      * @param groupName
303      *            - The name of the group
304      * @param username
305      *            - The user's username
306      * @param serverpass
307      *            TODO
308      * @return True if
309      */
310     private static boolean isInGroup(String groupName, String username, String
serverpass) {
311 
312         if (Crypto.hasValidMac(Configurations.GROUPS_FOLDER + "/" + groupName +
".cfg", serverpass)) {
313             try {
314 
315                 BufferedReader br = new BufferedReader(
316                         new FileReader(Configurations.GROUPS_FOLDER + "/" +
groupName + ".cfg"));
317 
318                 String line;
319 
320                 while ((line = br.readLine()) != null) {
321                     if (line.equals(username)) {
322                         br.close();
323                         System.out.println("User " + username + " is in group " +
groupName);
324                         return true;
325                     }
326                 }
327                 br.close();
328 
329             } catch (IOException e) {
330                 System.err.println("Error verifying if user belongs to group!");
331                 // e.printStackTrace();
332             }
333         }
334         return false;
335     }
336 
337     /**
338      * Removes an entry from a text file
339      *
340      * @param filename
341      *            - File to be redacted
342      * @param entry
343      *            - Entry to redact
344      * @param serverpass
345      *            TODO
```

```
346      * @return True if entry was successfully redacted
347      */
348     public static boolean removeEntry(String filename, String entry, String
    serverpass) {
349 
350         if (Crypto.hasValidMac(filename, serverpass)) {
351             try {
352                 BufferedReader in = new BufferedReader(new FileReader(filename));
353                 StringBuilder sb = new StringBuilder();
354 
355                 // Stripe the entry
356                 String line;
357                 while ((line = in.readLine()) != null) {
358                     // If it's not the entry we're looking for, append
359                     if (!line.equals(entry)) {
360                         sb.append(line);
361                         sb.append("\n");
362                     }
363                 }
364 
365                 in.close();
366 
367                 // Write the new file
368                 BufferedWriter out = new BufferedWriter(new FileWriter(filename));
369                 out.write(sb.toString());
370                 out.close();
371 
372                 Crypto.updateMAC(filename, serverpass);
373 
374                 return true;
375 
376             } catch (IOException e) {
377                 System.err.println("Error in file operations for entry removal!");
378                 // e.printStackTrace();
379             }
380         }
381         return false;
382 
383     }
384 
385     /**
386      * Removes a user from a group and cleans the respective configuration files
387      *
388      * @param username
389      *            - The username of the user to be removed
390      * @param groupName
391      *            - The groups name
392      * @param callingUser
393      *            - The user who invokes this method
394      * @param serverpass
395      *            TODO
396      * @return True if user was successfully removed, False otherwise
397      */
398     public static boolean removeFromGroup(String username, String groupName, String
    callingUser, String serverpass) {
399 
400         try {
401             // Verificar se quem chama o metodo e o admin do grupo
402             if (isAdmin(groupName, callingUser, serverpass)) {
403                 // Se o utilizador existe e pertence ao grupo
404                 if (User.userExists(username, serverpass) != null &&
    isInGroup(groupName, username, serverpass)) {
405 
406                     // Se o utilizador a remover Â´e admin apaga o grupo todo
```

```
407                     if (username.equals(callingUser)) {
408
409                         // Pega na lista de elementos do grupo e percorre-a
410                         BufferedReader in = new BufferedReader(
411                                 new FileReader(Configurations.GROUPS_FOLDER + "/" +
    groupName + ".cfg"));
412
413                         // Remove as entradas do ficheiro pessoal
414                         String line;
415                         while ((line = in.readLine()) != null) {
416
417                             removeEntry(Configurations.USERS_FOLDER + "/" + line +
    ".cfg", groupName, serverpass);
418
419                         }
420
421                         // apaga o ficheiro de membros
422                         Files.removeFile(Configurations.GROUPS_FOLDER + "/" +
    groupName + ".cfg");
423
424                         Crypto.removeMAC(Configurations.GROUPS_FOLDER + "/" +
    groupName + ".cfg");
425
426                         in.close();
427                         return true;
428                         // Just delete the element
429                     } else {
430
431                         // Remover do ficheiro do grupo
432                         removeEntry(Configurations.GROUPS_FOLDER + "/" + groupName
    + ".cfg", username, serverpass);
433
434                         // remover do ficheiro pessoal
435                         removeEntry(Configurations.USERS_FOLDER + "/" + username +
    ".cfg", groupName, serverpass);
436
437                         return true;
438
439                     }
440
441                 } else {
442                     System.out.println("User " + username + "doesn't exist or is
    not a member of " + groupName);
443                     return false;
444                 }
445
446             } else {
447                 System.out.println(callingUser + " is not an admin of " + groupName
    + " and cannot delete " + username);
448                 return false;
449             }
450         } catch (IOException e) {
451             // TODO Auto-generated catch block
452             System.err.println("Error removing user from group!");
453             e.printStackTrace();
454         }
455
456         return false;
457     }
458
459     /**
460      * Returns a String array with all the members of the group
461      *
462      * @param groupFile
```

```
463      *            - The path to the groups config file
464      * @return String array with members name, null in case of error
465      */
466     public static String[] membersList(String groupFile) {
467 
468         BufferedReader br;
469         List<String> list = new ArrayList<String>();
470 
471         try {
472             br = new BufferedReader(new FileReader(groupFile));
473 
474             String tmp;
475             while ((tmp = br.readLine()) != null) {
476 
477                 list.add(tmp);
478             }
479 
480             br.close();
481         } catch (FileNotFoundException e) {
482             // TODO Auto-generated catch block
483             e.printStackTrace();
484         } catch (IOException e) {
485             // TODO Auto-generated catch block
486             e.printStackTrace();
487         }
488 
489         return list.toArray(new String[0]);
490     }
491 }
492 
```

```java
package functionality;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Date;

/**
 * This class handles Message operations
 */
public class Message {

    /**
     * Builds a conversation from the individual message files
     *
     * @param dest
     *            - The destinatary
     * @param callingUser
     *            - The sender
     * @param lastestOnly
     *            - If it shows only the last message exchanged with
     * @param serverpass
     *            TODO
     * @return A string with the conversation
     */
    public static String buildConversation(String dest, String callingUser, boolean lastestOnly, String serverpass) {

        // RECONSTRUCT CONVERSATION
        StringBuilder conversationBack = new StringBuilder();
        ;
        try {
            System.out.println("Send all communications with " + dest);

            String destination = Files.getDestination(dest, callingUser, serverpass);

            String[] filesList = Files.listFiles(Configurations.MESSAGES_FOLDER + "/" + destination);

            if (filesList.length != 0) {
                // SHOW WHO IS THIS CONVERSATION WITH
                conversationBack.append("Contact: " + dest + "\n");

                if (lastestOnly) {
                    String s = filesList[filesList.length - 1];

                    if (s.matches("([^\\s]+(\\.(?i)(msg))$)")) {

                        conversationBack.append(Message.parseMessage(
                                Configurations.MESSAGES_FOLDER + "/" + destination + "/" + s, callingUser));

                    }

                } else {

                    for (String s : filesList) {

```

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

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

```
181         // Create a conversation folder
182         File messagePath = new File(Configurations.MESSAGES_FOLDER + "/" +
    destination);
183         if (!messagePath.exists()) {
184             messagePath.mkdir();
185         }
186 
187         Date date = new Date();
188 
189         // Write the message to disk
190         BufferedWriter msg = new BufferedWriter(
191                 new FileWriter(Configurations.MESSAGES_FOLDER + "/" + destination +
    "/" + date.getTime() + ".msg"));
192         msg.write(message);
193         msg.close();
194     }
195 
196     /*************************************************************
197      * CLIENT SIDE
198      *************************************************************/
199 
200     /**
201      * Sends a message to the server
202      *
203      * @param out
204      *            - Output Stream
205      * @param in
206      *            - Input Stream
207      * @param msg
208      *            TODO
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,
    Object msg) {
215 
216         try {
217             out.writeObject(msg);
218             out.flush();
219 
220             // Receive status from message delivery
221             String result = (String) in.readObject();
222             System.out.println(result);
223 
224         } catch (IOException | ClassNotFoundException e) {
225             // TODO Auto-generated catch block
226             System.err.println("Error sending message to server!");
227             // e.printStackTrace();
228         }
229     }
230 }
231 
```

```java
package client;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Scanner;

import javax.crypto.SecretKey;

import functionality.*;

/**
 * This class represents the client myWhats
 */
public class myWhats {

    public static void main(String[] args) {

        System.setProperty("javax.net.ssl.trustStore", Configurations.TRUSTSTORE_NAME);
        System.setProperty("javax.net.ssl.trustStorePassword", "storepass");


        try {
            // Verify the giver argument number
            if (args.length < 3) {

                System.err.println("Incorrect parameters!\n");

                // Print correct usage
                Usage.printUsage();

            } else {

                if (args[0].contains("#")) {
                    System.err.println("Invalid username. Usernames cannot contain \'#\'");
                    System.err.println("Exiting now...");
                    System.exit(-1);
                }

                Socket sock = null;

                String user = "";
                String pass = "";

                // Automatic password handling
                if (args.length >= 4) {
                    user = args[0];
                    pass = args[3];

                }

                // NO PASSWORD PROVIDED
                if (args.length == 3) {

                    // Check if -p
                    if (args[2].equals("-p")) {

```

```
63                          @SuppressWarnings("resource")
64                          Scanner sc = new Scanner(System.in);
65
66                          user = args[0];
67
68                          while (pass.length() < 2) {
69
70                              System.out.print("Password: ");
71                              pass = sc.nextLine();
72
73                          }
74
75                          // Close scanner
76                          // sc.close();
77                          // Warn user only registering or login will happen
78                          System.out.println("\nNOTICE: Your only arguments are your
  username, "
79                                  + "server and password!\n" + "This will only
  register you or try to log you in.\n");
80
81                          sock = Communication.connect(args[1]);
82                          if (sock != null) {
83                              ObjectOutputStream out = new
  ObjectOutputStream(sock.getOutputStream());
84                              ObjectInputStream in = new
  ObjectInputStream(sock.getInputStream());
85                              Authentication.login(in, out, user, pass);
86                              out.close();
87                              in.close();
88                              sock.close();
89                          }
90
91                      } else {
92
93                          System.err.println("Incorrect parameters!\n");
94                          Usage.printUsage();
95                      }
96
97                      // JUST THE PASSWORD
98                  } else if (args.length == 4 && args[2].equals("-p")) {
99                      // Warn user only registering or login will happen
100                     System.out.println("\nNOTICE: Your only arguments are your
  username, " + "server and password!\n"
101                             + "This will only register you or try to log you
  in.\n");
102
103                     sock = Communication.connect(args[1]);
104                     if (sock != null) {
105                         ObjectOutputStream out = new
  ObjectOutputStream(sock.getOutputStream());
106                         ObjectInputStream in = new
  ObjectInputStream(sock.getInputStream());
107                         Authentication.login(in, out, user, pass);
108                         out.close();
109                         in.close();
110                         sock.close();
111                     }
112
113                     // PASWORD AND -r (no args)
114                 } else if (args.length == 5) {
115
116                     if (args[4].equals("-r")) {
117
118                         sock = Communication.connect(args[1]);
```

```
119                         if (sock != null) {
120                             ObjectOutputStream out = new
    ObjectOutputStream(sock.getOutputStream());
121                             ObjectInputStream in = new
    ObjectInputStream(sock.getInputStream());
122                             Authentication.login(in, out, user, pass);
123                             String[] snd = {};
124                             Communication.sendCommand(out, "-r", snd);
125 
126                             System.out.println("\nTrying to receive the lastest
    communications...");
127 
128                             // Receive result
129                             String convo = (String) in.readObject();
130                             System.out.println(convo);
131 
132                             out.close();
133                             in.close();
134                             sock.close();
135                         }
136 
137                     } else {
138 
139                         System.err.println("Incorrect parameters!\n");
140                         Usage.printUsage();
141 
142                     }
143 
144                     // PASWORD AND -r ARG1
145                 } else if (args.length == 6) {
146 
147                     if (args[4].equals("-r")) {
148 
149                         sock = Communication.connect(args[1]);
150                         if (sock != null) {
151                             ObjectOutputStream out = new
    ObjectOutputStream(sock.getOutputStream());
152                             ObjectInputStream in = new
    ObjectInputStream(sock.getInputStream());
153                             Authentication.login(in, out, user, pass);
154 
155                             System.out.println("\nTrying to receive all
    communications with " + args[5] + "...");
156 
157                             String[] snd = { args[5] };
158                             Communication.sendCommand(out, "-r", snd);
159 
160                             // Receive the answer
161                             String convo = (String) in.readObject();
162                             System.out.println(convo);
163 
164                             out.close();
165                             in.close();
166                             sock.close();
167                         }
168 
169                     } else {
170 
171                         System.err.println("Incorrect parameters!\n");
172                         Usage.printUsage();
173 
174                     }
175 
176                     // ALL OTHER FLAGS
```

```
177                 } else if (args.length == 7) {
178
179                     switch (args[4]) {
180
181                     // MESSAGE OPERATION
182                     case "-m":
183
184                         sock = Communication.connect(args[1]);
185                         if (sock != null) {
186                             ObjectOutputStream out = new
    ObjectOutputStream(sock.getOutputStream());
187                             ObjectInputStream in = new
    ObjectInputStream(sock.getInputStream());
188                             Authentication.login(in, out, user, pass);
189
190                             System.out.println("\nTrying to send a message to " +
    args[5] + "...");
191
192                             if (args[6].length() > 0) {
193                                 String[] snd = { args[5] };
194
195                                 Communication.sendCommand(out, "-m", snd);
196
197                                 //Construir a mensagem
198                                 // Build the message
199                                 StringBuilder sb = new StringBuilder();
200
201                                 sb.append(user);
202                                 sb.append("\n");
203                                 sb.append(args[6]);
204                                 sb.append("\n");
205
206                                 Date date = new Date();
207                                 SimpleDateFormat ft = new
    SimpleDateFormat("yyyy-MM-dd hh:mm");
208                                 sb.append(ft.format(date));
209
210                                 String message = sb.toString();
211
212                                 //1 - Receber a lista de membros
213                                 String [] memberList =
    Communication.receiveUserOrGroup(in);
214
215                                 //Mensagem enviada para um utilizador
216                                 if(memberList.length == 1){
217
218                                     memberList = new String[] {memberList[0],
    user};
219                                 }
220
221                                 //2 - Assinatura digital do ficheiro em claro
222                                 //????
223
224                                 //3 - Chave simetrica AES aleatória K
225                                 SecretKey key = Crypto.randomAESKey();
226
227                                 //4 - Cifra mensagem com K
228                                 byte [] cipheredMsg =
    Crypto.encryptMessage(message.getBytes(), key);
229
230                                 //5 -
231                                 File tempMSG = new File("tmpMSG");
232                                 FileOutputStream msgOut = new
    FileOutputStream(tempMSG);
```

```
233                                 msgOut.write(cipheredMsg);
234                                 msgOut.close();
235 
236                                 //5 - Envia para o servidor
237                                 Files.sendFile(in, out, "", "tmpMSG");
238                                 out.writeBoolean(false);
239                                 out.flush();
240                                 tempMSG.delete();
241 
242 
243                                 //6 - Cifra-se K com as chaves publicas dos destinatarios
244                                 //Obtidas da truststore e envia-se para o servidor
245 
246                                 byte[][] keys = Crypto.encryptWithPrivateKey(key.getEncoded(), memberList, key, Configurations.TRUSTSTORE_NAME);
247 
248                                 Files.sendCodedKeys(out, memberList, keys);
249 
250 
251                             } else {
252                                 System.err.println("ERROR: Unable to send empty message.");
253                             }
254 
255                             out.close();
256                             in.close();
257                             sock.close();
258                         }
259                         break;
260 
261                     // FILE OPERATION
262                     case "-f":
263 
264                         sock = Communication.connect(args[1]);
265                         if (sock != null) {
266                             ObjectOutputStream out = new ObjectOutputStream(sock.getOutputStream());
267                             ObjectInputStream in = new ObjectInputStream(sock.getInputStream());
268 
269                             Authentication.login(in, out, user, pass);
270 
271                             System.out.println("\nTrying to send the file " + args[6] + " to " + args[5] + "...");
272 
273                             // Check if file doesn't have a disallowed extension
274                             if (args[6].toLowerCase().endsWith(".mac") || args[6].toLowerCase().endsWith(".msg")) {
275                                 System.err
276                                         .println("WARNING: .mac and .msg extension files are not allowed to be sent.");
277                                 System.err.println("File " + args[6] + " not sent.");
278 
279                             } else {
280 
281                                 // Check if file exists
282                                 File file = new File(args[6]);
283 
284                                 // Don't bother server if file doesn't exist
285                                 // localy
286                                 if (file.exists() && !file.isDirectory()) {
```

```
287
288                                     String[] snd = { args[5], args[6] };
289
290                                     Communication.sendCommand(out, "-f", snd);
291
292                                     //1 - Receber a lista de membros
293                                     String [] memberList =
    Communication.receiveUserOrGroup(in);
294
295                                     //Mensagem enviada para um utilizador
296                                     if(memberList.length == 1){
297
298                                         memberList = new String[] {memberList[0],
    user};
299                                     }
300
301                                     //2 - Assinatura digital do ficheiro em claro
302                                     //???
303
304                                     //3 - Chave simetrica AES aleatória K
305                                     SecretKey key = Crypto.randomAESKey();
306
307                                     //4 - Cifra mensagem com K
308                                     Crypto.encryptFile(args[6], key);
309
310                                     //5 - Envia para o servidor
311                                     Files.sendFile(in, out, "", args[6]);
312                                     out.writeBoolean(true);
313                                     out.flush();
314
315                                     //6 - Cifra-se K com as chaves publicas dos
    destinatarios
316                                     //Obtidas da truststore e envia-se para o
    servidor
317                                     byte[][] keys =
    Crypto.encryptWithPrivateKey(key.getEncoded(), memberList, key,
    Configurations.TRUSTSTORE_NAME);
318
319                                     Files.sendCodedKeys(out, memberList, keys);
320
321
322                                 } else {
323                                     System.out.println("File " + args[6] + " not
    found!");
324                                 }
325                             }
326                             out.close();
327                             in.close();
328                             sock.close();
329                         }
330                         break;
331
332                     // REVIEW OPERATION
333                     case "-r":
334
335                         sock = Communication.connect(args[1]);
336                         if (sock != null) {
337                             ObjectOutputStream out = new
    ObjectOutputStream(sock.getOutputStream());
338                             ObjectInputStream in = new
    ObjectInputStream(sock.getInputStream());
339                             Authentication.login(in, out, user, pass);
340
341                             System.out.println("\nTrying to receive file " +
```

```
   args[6] + " from " + args[5] + "...");
342
343                             String[] snd = { args[5], args[6] };
344                             System.out.println("-r " + args[5] + " " + args[6]);
345                             Communication.sendCommand(out, "-r", snd);
346
347                             // Check if DOWNLOADS FOLDER exist
348                             File downloadsFolder = new
   File(Configurations.DOWNLOAD_FOLDER);
349                             if (!downloadsFolder.exists()) {
350
351                                 downloadsFolder.mkdir();
352                             }
353
354                             boolean fileExists = in.readBoolean();
355
356                             if (fileExists) {
357                                 Files.receiveFile(in, out,
   Configurations.DOWNLOAD_FOLDER, snd[1]);
358                             } else {
359                                 System.out.println("The file you requested does not
   exist!");
360                             }
361
362                             out.close();
363                             in.close();
364                             sock.close();
365                         }
366                         break;
367
368                     // GROUP ADD OPERATION
369                     case "-a":
370
371                         sock = Communication.connect(args[1]);
372                         if (sock != null) {
373                             ObjectOutputStream out = new
   ObjectOutputStream(sock.getOutputStream());
374                             ObjectInputStream in = new
   ObjectInputStream(sock.getInputStream());
375                             Authentication.login(in, out, user, pass);
376
377                             System.out.println("\nTrying to add user " + args[5] +
   " to  group " + args[6] + "...");
378
379                             System.out.println("-a " + args[5] + " " + args[6]);
380                             String[] snd = { args[5], args[6] };
381                             Communication.sendCommand(out, "-a", snd);
382
383                             // receive response
384                             String rsp = (String) in.readObject();
385                             System.out.println(rsp);
386
387                             out.close();
388                             in.close();
389                             sock.close();
390                         }
391                         break;
392
393                     // GROUP REMOVE OPERATION
394                     case "-d":
395
396                         sock = Communication.connect(args[1]);
397                         if (sock != null) {
398                             ObjectOutputStream out = new
```

```
   ObjectOutputStream(sock.getOutputStream());
399                             ObjectInputStream in = new
   ObjectInputStream(sock.getInputStream());
400                             Authentication.login(in, out, user, pass);
401
402                             System.out.println("\nTrying to remove user " + args[5]
   + " from group " + args[6] + "...");
403
404                             String[] snd = { args[5], args[6] };
405                             System.out.println("-d " + args[5] + " " + args[6]);
406                             Communication.sendCommand(out, "-d", snd);
407
408                             out.close();
409                             in.close();
410                             sock.close();
411                         }
412                         break;
413
414                     // INVALID PARAMETERS
415                     default:
416                         System.err.println("Incorrect parameters!\n");
417                         Usage.printUsage();
418                     }
419                 } else {
420                     System.err.println("Incorrect parameters!\n");
421                     Usage.printUsage();
422                 }
423             }
424
425         } catch (ClassNotFoundException | IOException e) {
426             // TODO Auto-generated catch block
427             // e.printStackTrace();
428             System.err.println("Error during execution. Server might be down.");
429         }
430     }
431 }
432
```

```java
package server;

/*****************************************************************************
 *                    Seguranca e Confiabilidade 2015/16
 *****************************************************************************/

import java.io.File;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Scanner;

import javax.crypto.SecretKey;
import javax.net.ssl.SSLServerSocketFactory;
import javax.net.ssl.SSLSession;
import javax.net.ssl.SSLSocket;

import functionality.*;

//Servidor do servico myWhatsServer
public class myWhatsServer {

    private static String serverpass;

    // MAIN
    public static void main(String[] args) {

        System.setProperty("javax.net.ssl.keyStore", "keystore.jks");
        System.setProperty("javax.net.ssl.keyStorePassword", "storepass");
        myWhatsServer sv = new myWhatsServer();
        sv.startServer();
    }

    // START SERVER
    @SuppressWarnings("resource")
    public void startServer() {

        ServerSocket sSoc = null;

        serverpass = "";
        boolean validPassLength = false;
        Scanner sc = new Scanner(System.in);

        // ASK FOR SERVER PASSWORD
        System.out.println("Server starting...");
        System.out.print("Password: ");
        while (!validPassLength) {

            serverpass = sc.nextLine();

            if (serverpass.length() < 4) {
                System.err.println("Password must have at least 4 chars.");
            } else {
                validPassLength = true;
            }
        }

        try {

            SSLServerSocketFactory ssf = (SSLServerSocketFactory)
```

```
SSLServerSocketFactory.getDefault();
65              sSoc = ssf.createServerSocket(23456);
66
67          } catch (IOException e) {
68
69              System.err.println(e.getMessage());
70              System.exit(-1);
71
72          }
73
74          //
************************************************************************
75          // CHECK FOR FILE STRUCTURE INTEGRITY
76          //
************************************************************************
77
78          // Check if USERS file exist
79          File credentials = new File(Configurations.CREDENTIALS_FILENAME);
80          if (!credentials.exists() && !credentials.isDirectory()) {
81
82              try {
83
84                  credentials.createNewFile();
85
86              } catch (IOException e) {
87                  System.err.println("Error creating user file");
88                  e.printStackTrace();
89
90              }
91          }
92
93          // Check if GROUPS file exist
94          File groups = new File(Configurations.GROUPS_FILENAME);
95          if (!groups.exists() && !groups.isDirectory()) {
96
97              try {
98
99                  groups.createNewFile();
100
101             } catch (IOException e) {
102                 System.err.println("Error creating groups file");
103                 e.printStackTrace();
104
105             }
106         }
107
108         // Check if users FOLDER exist
109         File usersFolder = new File(Configurations.USERS_FOLDER);
110         if (!usersFolder.exists()) {
111
112             usersFolder.mkdir();
113         }
114
115         // Check if groups FOLDER exist
116         File groupsFolder = new File(Configurations.GROUPS_FOLDER);
117         if (!groupsFolder.exists()) {
118
119             groupsFolder.mkdir();
120         }
121
122         // Check if messages FOLDER exist
123         File messagesFolder = new File(Configurations.MESSAGES_FOLDER);
124         if (!messagesFolder.exists()) {
125
```

```
126             messagesFolder.mkdir();
127         }
128 
129         //
    ********************************************************************************
130         // CHECK FOR MAC FILE
131         //
    ********************************************************************************
132 
133         // CALCULATE MAC
134         byte[] credentialsMacDigest = Crypto.calculateFileMAC(serverpass,
    Configurations.CREDENTIALS_FILENAME);
135 
136         // CHECK IF PASS FILE IS PROTECTED WITH MAC
137         if (Crypto.isMacProtected(Configurations.CREDENTIALS_FILENAME)) {
138 
139             // IF SO OK
140             System.out.println("Checking credentials file for integrity...");
141             if (Crypto.hasValidMac(Configurations.CREDENTIALS_FILENAME,
    serverpass)) {
142                 System.out.println("OK");
143 
144                 // IF NOT ERROR AND CLOSE
145             } else {
146                 System.err.println("The credentials' file calculated mac doesn't
    match the stored mac!");
147                 System.exit(-1);
148             }
149 
150             // IF NOT PROTECT
151         } else {
152             System.out.println("The credentials file is not protected by MAC, do
    you wish to protect it (Y/N)?");
153 
154             Scanner read = new Scanner(System.in);
155             String ans = "";
156 
157             while (!ans.equals("n") && !ans.equals("y")) {
158                 ans = read.nextLine().toLowerCase();
159             }
160 
161             if (ans.equals("y")) {
162 
163                 Crypto.writeMacFile(credentialsMacDigest,
    Configurations.CREDENTIALS_FILENAME);
164                 // System.out.println("Server started!");
165 
166             } else {
167                 System.err.println("Terminating execution.");
168                 System.exit(-1);
169             }
170         }
171 
172         //
    ********************************************************************************
173         // CHECK OTHER CONFIG FILES FOR MAC FILE
174         //
    ********************************************************************************
175 
176         // GROUPS FILE //
177         // Se estiver, verifica se é vaído
178         if (Crypto.isMacProtected(Configurations.GROUPS_FILENAME)) {
179             System.out.println("Checking groups config file for integrity...");
180             Crypto.hasValidMac(Configurations.GROUPS_FILENAME, serverpass);
```

```
181             System.out.println("OK");
182             // Caso contrario protege
183         } else {
184             byte[] groupsFile = Crypto.calculateFileMAC(serverpass,
    Configurations.GROUPS_FILENAME);
185             Crypto.writeMacFile(groupsFile, Configurations.GROUPS_FILENAME);
186         }
187 
188         // SERVER FILE //
189         // Se estiver, verifica se é vaído
190         if (Crypto.isMacProtected(Configurations.SERVER_POLICY)) {
191             System.out.println("Checking server policy file for integrity...");
192             Crypto.hasValidMac(Configurations.SERVER_POLICY, serverpass);
193             System.out.println("OK");
194 
195             // Caso contrario protege
196         } else {
197             byte[] groupsFile = Crypto.calculateFileMAC(serverpass,
    Configurations.SERVER_POLICY);
198             Crypto.writeMacFile(groupsFile, Configurations.SERVER_POLICY);
199         }
200 
201         System.out.println("File integrity verifications: PASSED");
202         System.out.println("\n\n==============");
203         System.out.println("SERVER STARTED");
204         System.out.println("==============");
205 
206         //
    ********************************************************************************
207         // SERVER MAIN LOOP
208         //
    ********************************************************************************
209 
210         // Server Main - Client reception and thread creation
211         while (true) {
212             try {
213                 Socket inSoc = sSoc.accept();
214 
215                 ServerThread newServerThread = new ServerThread(inSoc);
216                 newServerThread.start();
217             } catch (IOException e) {
218                 e.printStackTrace();
219             }
220 
221         }
222     }
223 
224     //
    ********************************************************************************
225 
226     // Threads utilizadas para comunicacao com os clientes
227     class ServerThread extends Thread {
228 
229         private Socket socket = null;
230 
231         ServerThread(Socket inSoc) {
232             socket = inSoc;
233             System.out.println("Client connected from " + socket.getInetAddress());
234 
235             SSLSession session = ((SSLSocket) inSoc).getSession();
236 
237             System.out.println("Peer host is " + session.getPeerHost());
238             System.out.println("Cipher is " + session.getCipherSuite());
239             System.out.println("Protocol is " + session.getProtocol());
```

```
240             System.out.println("Session created in " + session.getCreationTime());
241             System.out.println("Session accessed in " +
    session.getLastAccessedTime());
242         }
243 
244         // Run the thread
245         public void run() {
246 
247             try {
248                 // OPEN STREAMS
249                 ObjectOutputStream out = new
    ObjectOutputStream(socket.getOutputStream());
250                 ObjectInputStream in = new
    ObjectInputStream(socket.getInputStream());
251 
252                 String user = null;
253                 // END OPEN STEAMS
254 
255                 // RECEIVE CREDENTIALS
256                 try {
257                     user = (String) in.readObject();
258 
259                     System.out.println("Connection from " +
    socket.getInetAddress());
260 
261                 } catch (ClassNotFoundException e1) {
262                     System.err.println("Error receiving credentials");
263                     // e1.printStackTrace();
264                 }
265                 // END RECEIVE CREDENTIALS
266 
267                 // AUTHENTICATION
268                 boolean wasSuccessful = false;
269 
270                 if (user.length() > 0) {
271                     wasSuccessful = Authentication.authenticateUser(in, out, user,
    serverpass);
272                 }
273 
274                 // END AUTHENTICATION
275 
276                 if (wasSuccessful) {
277                     try {
278                         String flag = (String) in.readObject();
279                         String[] argArray = (String[]) in.readObject();
280                         String destination;
281 
282                         // Procede according to flag sent
283                         switch (flag) {
284                         case "-m":
285                             System.out.println("Message To: " + argArray[0]);
286 
287                             // 1 - Enviar a lista de membros
288                             Communication.sendFileOrGroup(out, argArray[0],
    serverpass);
289 
290                             destination = Files.getDestination(argArray[0], user,
    serverpass);
291 
292                             if (destination != "") {
293 
294                                 Date date = new Date();
295                                 SimpleDateFormat ft = new
    SimpleDateFormat("yyyy-MM-dd hh:mm");
```

```
296
297                                 String fileName = date.getTime() + ".msg";
298
299                                 Files.receiveFile(in, out,
    Configurations.MESSAGES_FOLDER + "/" + destination,
300                                         fileName);
301
302                                 boolean writeMsgFile = in.readBoolean();
303
304                                 if (writeMsgFile) {
305
306                                     Message.writeMessage(argArray[1], user,
    destination);
307                                 }
308
309                                 Files.receiveCodedKeysAndWrite(in, fileName,
    destination);
310
311                             } else {
312
313                                 // Send that is not a valid destination
314                                 out.writeBoolean(false);
315                                 out.flush();
316
317                                 System.out.println("ERROR! Not a user or group!");
318                             }
319
320                             break;
321
322                         case "-f":
323
324                             // 1 - Enviar a lista de membros
325                             Communication.sendFileOrGroup(out, argArray[0],
    serverpass);
326
327                             System.out.println("File to: " + argArray[0] + " - " +
    argArray[1]);
328
329                             destination = Files.getDestination(argArray[0], user,
    serverpass);
330
331                             if (destination != "") {
332
333                                 Files.receiveFile(in, out,
    Configurations.MESSAGES_FOLDER + "/" + destination,
334                                         argArray[1]);
335
336                                 boolean writeMsgFile = in.readBoolean();
337
338                                 if (writeMsgFile) {
339                                     Message.writeMessage(argArray[1], user,
    destination);
340                                 }
341
342                                 Files.receiveCodedKeysAndWrite(in, argArray[1],
    destination);
343
344                             } else {
345
346                                 // Send that is not a valid destination
347                                 out.writeBoolean(false);
348                                 out.flush();
349
350                                 System.out.println("ERROR! Not a user or group!");
```

```
351                             }
352                             break;
353
354                         case "-r":
355
356                             // TODO: Enviar os ficheiros de mensagem todos?
357                             // CIFRAR A MENSAGEM COM K
358                             // CIFRAR K COM A CHAVE PRIVADA DO SERVER
359                             // DECIFRAR COM A PUBLICA NO CLIENTE
360                             // DECIFRAR A MENSAGEM COM A CHAVE PUBLICA DO SERVER
361
362                             if (argArray.length == 0) {
363                                 System.out.println("Send all communications");
364
365                                 // LIST ALL FOLDERS THAT CONTAIN THE USER's
366                                 // USERNAME
367
368                                 String[] folderList =
    Files.listFolders(Configurations.MESSAGES_FOLDER);
369                                 StringBuilder convo = new StringBuilder();
370                                 for (String s : folderList) {
371
372                                     if (s.matches(".+#.+")) {
373
374                                         String[] tmp = s.split("#");
375
376                                         // RECONSTRUCT CONVERSATION
377                                         if (tmp[0].equals(user)) {
378
379                                             convo.append(Message.buildConversation(t
    mp[1], user, true, serverpass));
380                                         } else if (tmp[1].equals(user)) {
381                                             convo.append(Message.buildConversation(t
    mp[0], user, true, serverpass));
382                                         }
383                                     }
384                                 }
385
386                                 // SEND
387                                 out.writeObject(convo.toString());
388                                 out.flush();
389
390                             } else if (argArray.length == 1) {
391
392                                 String convo =
    Message.buildConversation(argArray[0], user, false, serverpass);
393
394                                 // SEND
395                                 out.writeObject(convo);
396                                 out.flush();
397
398                             } else if (argArray.length == 2) {
399                                 System.out.println("Get file " + argArray[1] + " in
    " + argArray[0]);
400
401                                 // OPEN CONVERSATION FOLDER
402                                 destination = Files.getDestination(argArray[0],
    user, serverpass);
403                                 File file = new File(
404                                         Configurations.MESSAGES_FOLDER + "/" +
    destination + "/" + argArray[1]);
405
406                                 if (file.exists()) {
407                                     out.writeBoolean(true);
```

```
408                                     out.flush();
409                                     Files.sendFile(in, out,
    Configurations.MESSAGES_FOLDER + "/" + destination + "/",
410                                             argArray[1]);
411                                 } else {
412                                     out.writeBoolean(false);
413                                     out.flush();
414                                 }
415
416                             }
417                             break;
418
419                         case "-a":
420                             System.out.println("Adds user " + argArray[0] + " to
    group " + argArray[1]);
421                             Group.addGroup(argArray[0], argArray[1], user, out, in,
    serverpass);
422
423                             break;
424
425                         case "-d":
426                             System.out.println("Delete user " + argArray[0] + "
    from group " + argArray[1]);
427                             Group.removeFromGroup(argArray[0], argArray[1], user,
    serverpass);
428                             break;
429
430                         default:
431                             System.out.println("Incorrect parameters!\n");
432                         }
433
434                     } catch (IOException | ClassNotFoundException e) {
435                         System.out.println("No command was sent from client!");
436                         // e.printStackTrace();
437                     }
438                 }
439
440                 // END SERVER MAIN
441
442                 System.out.println("-=*=-=*=-=*=- END OF SESSION -=*=-=*=-=*=-");
443                 // SOCKT AND STREAM CLOSING
444                 out.close();
445                 in.close();
446
447                 socket.close();
448                 // END SOCKT AND STREAM CLOSING
449
450             } catch (IOException e) {
451                 System.err.println("Error running server thread!");
452                 // e.printStackTrace();
453             } catch (ClassNotFoundException e) {
454                 // TODO Auto-generated catch block
455                 e.printStackTrace();
456             }
457         }
458     }
459 }
```

```java
package functionality;

/**
 * This class handles usage display of application
 */
public class Usage {

    /**
     * Prints the usage menu
     */
    public static void printUsage() {

        System.out.println("myWhats <localUser> <serverAddress> [ -p <password> ]\n"
                           + "                                  [ -m <contact> <message> ]\n"
                           + "                                  [ -f <contact> <file> ]\n"
                           + "                                  [ -r [contact] [file] ]\n"
                           + "                                  [ -a <user> <group> ]\n"
                           + "                                  [ -d <user> <group> ]\n");
    }
}

```

```java
package functionality;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

/**
 * This class handles User operations
 */
public class User {

    /**
     * Adds a user to the credentials file
     *
     * @param login
     *            - Credentials to be added
     */
    static boolean createUser(String login, String serverpass) {

        if (Crypto.hasValidMac(Configurations.CREDENTIALS_FILENAME, serverpass)) {

            try {

                // Insert credentials in the creds file
                BufferedWriter bw = new BufferedWriter(new
FileWriter(Configurations.CREDENTIALS_FILENAME, true));

                bw.append(login);
                bw.newLine();
                bw.close();

                // Create user's personal folder
                String[] parseCreds = login.split(":");

                BufferedWriter uf = new BufferedWriter(
                        new FileWriter(Configurations.USERS_FOLDER + "/" +
parseCreds[0] + ".cfg", true));
                uf.close();

                Crypto.updateMAC(Configurations.USERS_FOLDER + "/" + parseCreds[0]
+ ".cfg", serverpass);

                System.out.println("New user registered");

                Crypto.updateMAC(Configurations.CREDENTIALS_FILENAME, serverpass);

                return true;

            } catch (IOException e) {

                System.err.println("Error opening credentials file for user
insertion");
                // e.printStackTrace();
            }
        }

        return false;
    }

    /**
     * Verifies if a username is present in the credentials file
```

```
61      *
62      * @param username
63      *            - User's username
64      * @param serverpass
65      *            TODO
66      *
67      * @return If username if present it's credentials will be returned
68      *         otherwise, empty will be returned
69      * @throws IOException
70      */
71     static String userExists(String username, String serverpass) throws IOException
   {
72
73         if (Crypto.hasValidMac(Configurations.CREDENTIALS_FILENAME, serverpass)) {
74
75             try {
76
77                 BufferedReader br = new BufferedReader(new
   FileReader(Configurations.CREDENTIALS_FILENAME));
78
79                 String line;
80                 String[] credLine;
81                 while ((line = br.readLine()) != null) {
82
83                     credLine = line.split(":");
84
85                     if (credLine[0].equals(username)) {
86                         br.close();
87                         // System.out.println("Username exists in credentials
88                         // file");
89                         return line;
90                     }
91                 }
92
93                 br.close();
94
95             } catch (FileNotFoundException e) {
96
97                 System.err.println("Error opening credentials file for user
   verification");
98                 // e.printStackTrace();
99             }
100         }
101
102         System.out.println("Username <" + username + "> not found");
103         return "";
104     }
105 }
106
```