CLIENT

```
package clientPackage;

import java.io.File;
import java.io.IOException;
import java.net.InetAddress;

import javax.swing.UIManager;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class ClientMain {

    static int REGISTRY_PORT = 0;
    static int TCP_PORT = 0;
    static int UDP_PORT = 0;
    static String REGISTRY_HOST = null;
    static InetAddress MULTICAST_ADDRESS = null;

    public static void main(String[] args) {

        String configFileName = "ClientConfiguration.xml";

        if (args.length == 1) configFileName = args[0];
        else if (args.length > 1) {
            System.out.println("Troppi argomenti; è possibile eseguire il programma con:");
            System.out.println("0 argomenti: viene preso il nome del file di configurazione \"ClientConfiguration.xml\"");
            System.out.println("1 argomento: viene preso il nome del file di configurazione indicato");
            System.exit(-1);
        }

        try {
            File inputFile = new File(configFileName);
            DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
            DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
            Document doc = dBuilder.parse(inputFile);
            doc.getDocumentElement().normalize();

            NodeList els = doc.getElementsByTagName("property");

            REGISTRY_HOST = els.item(0).getFirstChild().getTextContent();
            REGISTRY_PORT = Integer.parseInt(els.item(1).getFirstChild().getTextContent());
            TCP_PORT = Integer.parseInt(els.item(2).getFirstChild().getTextContent());
            UDP_PORT = Integer.parseInt(els.item(3).getFirstChild().getTextContent());
            String mult_addr_name = els.item(4).getFirstChild().getTextContent();

            MULTICAST_ADDRESS = InetAddress.getByName(mult_addr_name);
            if (!MULTICAST_ADDRESS.isMulticastAddress()) {
                System.out.println("Questo indirizzo non è Multicast. Impossibile continuare");
                System.exit(-1);
            }

        } catch (IOException ex) {
            System.out.println("Impossibile convertire la stringa data nell'indirizzo del server");
            ex.printStackTrace();
```

```
67              System.exit(-1);
68          } catch (NumberFormatException ex) {
69              System.out.println("Errore nella lettura di una porta del file di configurazione.
                Impossibile continuare");
70              System.exit(-1);
71          } catch (ParserConfigurationException e2) {
72              System.out.println("Errore nella conversione del file XML. Impossibile continuare");
73              e2.printStackTrace();
74              System.exit(-1);
75          } catch (SAXException e2) {
76              System.out.println("Errore nella lettura del file XML. Impossibile continuare");
77              e2.printStackTrace();
78              System.exit(-1);
79          }
80
81
82          java.awt.EventQueue.invokeLater(new Runnable() {
83              public void run() {
84                  try {
85                      UIManager.setLookAndFeel(
86                                  //  "javax.swing.plaf.metal.MetalLookAndFeel");
87                                  //  "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
88                                  UIManager.getCrossPlatformLookAndFeelClassName());
89                  } catch (Exception ex) {
90                      ex.printStackTrace();
91                  }
92                  new GUI_logic(REGISTRY_PORT, REGISTRY_HOST, TCP_PORT, UDP_PORT,
                    MULTICAST_ADDRESS).setVisible(true);
93              }
94          });
95      }
96
97
98  }
99
100
101 package clientPackage;
102
103 import java.util.ArrayList;
104
105 public class GamesContainer {
106
107     ArrayList<GamesData> games;
108
109     GamesContainer() {
110         this.games = new ArrayList<GamesData>();
111     }
112
113     public GamesData getGameByID(int gameID) {
114
115         for (GamesData game: games) {
116             if (game.getID() == gameID) return game;
117         }
118         return null;
119     }
120
121     public void addGame(GamesData game) {
122         if (game == null) return;
123         games.add(game);
124     }
125
126     public void removeGame(GamesData game) {
127         if (game == null) return;
128         games.remove(game);
129     }
130
131 }
132
133
```

```
134 package clientPackage;
135
136 public class GamesData {
137
138     private String creator;
139     private int gameID;
140     private int multicastGamePort;
141
142     GamesData(String creator, int gameID, int multicastGamePort) {
143         this.creator = creator;
144         this.gameID = gameID;
145         this.multicastGamePort = multicastGamePort;
146     }
147
148     public String getCreator() { return this.creator; }
149     public int getID() { return this.gameID; }
150     public int getPort() { return this.multicastGamePort; }
151
152     public String toString() {
153         return new String(creator + "  (GAME ID: " + gameID + ")" );
154     }
155
156 }
157
158
159
160 package clientPackage;
161
162 import java.rmi.RemoteException;
163 import java.rmi.server.RemoteObject;
164
165 import commonPackage.RMI_client_interface;
166
167 public class RMI_client extends RemoteObject implements RMI_client_interface{
168
169     private static final long serialVersionUID = 8520172379567158477L;
170
171     private GUI_logic clientGUI;
172
173     RMI_client(GUI_logic clientGUI) {
174         this.clientGUI = clientGUI;
175     }
176
177
178     /*
179      * (non-Javadoc)
180      * @see commonPackage.RMI_client_interface#gameCall(java.lang.String, int, int)
181      */
182     @Override
183     public void gameCall(String creator, int gameID, int multicastPort) throws RemoteException {
184
185         // not adding the game if it belongs to the creator
186         if (!creator.equals(clientGUI.username)) clientGUI.addGameRequest(creator, gameID,
            multicastPort);
187
188     }
189
190     /*
191      * (non-Javadoc)
192      * @see commonPackage.RMI_client_interface#gameCancelled(java.lang.String, int)
193      */
194     @Override
195     public void gameCancelled(String creator, int gameID) throws RemoteException {
196
197         clientGUI.removeGameRequest(gameID);
198
199     }
200
201     /*
```

```
202      * (non-Javadoc)
203      * @see commonPackage.RMI_client_interface#gameCancelled(java.lang.String, int)
204      * This methos have the only meaning to let the Server know if the
205      * Client is still online. If the Client fail to respond
206      * (a RemoteException is thrown), the Server sets the client offline.
207     */
208     @Override
209     public boolean isOnline() throws RemoteException {
210         return true;
211     }
212 
213 }
214 
215 
216 
217 package clientPackage;
218 
219 import java.awt.Color;
220 import java.awt.Dimension;
221 import java.awt.Font;
222 import java.awt.Toolkit;
223 import java.awt.event.ActionEvent;
224 import java.awt.event.ActionListener;
225 import java.awt.event.WindowEvent;
226 import java.awt.event.WindowListener;
227 import java.io.ByteArrayInputStream;
228 import java.io.ByteArrayOutputStream;
229 import java.io.IOException;
230 import java.io.ObjectInputStream;
231 import java.io.ObjectOutputStream;
232 
233 import java.net.DatagramPacket;
234 import java.net.DatagramSocket;
235 import java.net.InetAddress;
236 import java.net.MulticastSocket;
237 import java.net.Socket;
238 import java.net.SocketException;
239 import java.net.UnknownHostException;
240 
241 import java.rmi.NotBoundException;
242 import java.rmi.RemoteException;
243 import java.rmi.registry.LocateRegistry;
244 import java.rmi.registry.Registry;
245 import java.rmi.server.UnicastRemoteObject;
246 
247 import java.util.ArrayList;
248 import java.util.concurrent.ExecutionException;
249 
250 import javax.swing.*;
251 import javax.swing.event.ListSelectionEvent;
252 import javax.swing.event.ListSelectionListener;
253 
254 import commonPackage.Multicast_rankings;
255 import commonPackage.RMI_client_interface;
256 import commonPackage.RMI_server_interface;
257 import commonPackage.UDP_words;
258 
259 import static javax.swing.GroupLayout.Alignment.*;
260 
261 public class GUI_logic extends JFrame implements ActionListener, ListSelectionListener{
262 
263     public enum GUI_state {
264         LOGIN, LOBBY, NEWGAME, PLAYING
265     }
266 
267     // Action commands:
268     private final String
269             LOGIN = "Login", REGISTER = "Register", NEWGAME = "New Game", ACCEPTGAME = "Accept
            Game",
```

```
270             DECLINEGAME = "Decline Invite", RANKINGS = "Update Ranking", LOGOUT = "Logout",
271             SENDINVITES = "Send invites", CANCEL = "Cancel", ADDALL = "Add All", SWITCH = "< >",
272             UPDATE = "Refresh", REMOVEALL = "Remove All", SENDWORD = "Send Word", SURREND =
                "Surrend",
273             HELP = "Help", TIMER = "Timer", GAMETIMER = "GameTimer";
274
275     private static final long serialVersionUID = -4932694628587849289L;
276     private GUI_state state;
277     private RMI_server_interface serverRMI;
278     private RMI_client_interface userStub;
279     private int REGISTRY_PORT;
280     private String REGISTRY_HOST;
281     private Registry registry;
282     private InetAddress serverAddress;
283     private InetAddress multicastAddress;
284     private int TCP_PORT;
285     private int UDP_PORT;
286     private DatagramSocket dataSocket;
287
288     private Dimension screenDimension;
289     public String username;
290     private String password;
291     private GamesData currentPlayingGame;
292     private GamesContainer games = new GamesContainer();
293
294     /*
295      * The session ID is used to comunicate with Server (look at
296      * RMI_server_interface).
297      * if it's set to 0 it means it's not connected to the Server
298     */
299     private int sessionID = 0;
300
301     // Elements of Login GUI (must be global in the class to access them in other methods)
302     private GroupLayout layout = null;
303     private JTextField txtUsername = null;
304     private JPasswordField txtPassword = null;
305
306     // Elements of Lobby GUI
307     private JLabel lblUsername = null;
308     private JList<GamesData> jListInvites;
309     private JButton butAcceptInvite = null;
310     private JButton butDeclineInvite = null;
311     private DefaultListModel<GamesData> listInv = new DefaultListModel<GamesData>();
312     private DefaultListModel<String> listRank = new DefaultListModel<String>();
313
314     // Elements of New Game GUI
315     private JList<String> jListOnline;
316     private JList<String> jListInvited;
317     private JButton butSwitch = null;
318     private JButton butSend = null;
319     private DefaultListModel<String> listOn = null;
320     private DefaultListModel<String> listIn = null;
321
322     // Elements of Game GUI
323     private JLabel lblTimer = null;
324     private JLabel lblState = null;
325     private JLabel lblErrors = null;
326     private JLabel lblLetters = null;
327     private JLabel lblRanking = null;
328     private JButton butSurrend = null;
329     private JButton butSendWord = null;
330     private int timeLeft = 0;
331     private boolean gameStarted = false;
332     private Timer waitingForWords = null;
333     private Timer gameTimer = null;
334     private JTextField txtSendWord = null;
335     private DefaultListModel<String> listFWords = null;
336     private DefaultListModel<String> listLRanking = null;
337     // To stop the worker thread whos waiting for the word
```

```
338     private AcceptGameTask acceptTask = null;
339
340
341
342     GUI_logic(int REGISTRY_PORT, String REGISTRY_HOST, int TCP_PORT, int UDP_PORT, InetAddress
        MULT_ADDR) {
343
344         this.REGISTRY_PORT = REGISTRY_PORT;
345         this.REGISTRY_HOST = REGISTRY_HOST;
346         this.TCP_PORT = TCP_PORT;
347         this.UDP_PORT = UDP_PORT;
348         this.multicastAddress = MULT_ADDR;
349
350         try {
351
352             serverAddress = InetAddress.getByName(REGISTRY_HOST);
353             dataSocket = new DatagramSocket();
354             dataSocket.setSoTimeout(2000);
355
356             RMI_client_interface user = new RMI_client(this);
357
358             // Client Stub for callback comunications
359             userStub = (RMI_client_interface) UnicastRemoteObject.exportObject(user, 0);
360
361             // Server RMI element for RMI comunication
362             registry = LocateRegistry.getRegistry(this.REGISTRY_HOST, this.REGISTRY_PORT);
363             serverRMI = (RMI_server_interface) registry.lookup(RMI_server_interface.OBJECT_NAME);
364
365         } catch (NotBoundException e) {
366             System.out.println("Errore nella localizzazione del registro RMI. Impossibile
                continuare");
367             e.printStackTrace();
368             System.exit(-1);
369         } catch (RemoteException e) {
370             System.out.println("Errore nell'esportazione dello stub RMI. Impossibile
                continuare");
371             e.printStackTrace();
372             System.exit(-1);
373         } catch (SocketException e) {
374             System.out.println("Errore nella creazione del socket UDP. Impossibile continuare");
375             e.printStackTrace();
376             System.exit(-1);
377         } catch (UnknownHostException e) {
378             System.out.println("Impossibile leggere l'indirizzo del server. Impossibile
                continuare");
379             e.printStackTrace();
380             System.exit(-1);
381         }
382
383         setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
384
385         // SETS RIGHT OPERATION WHEN TRYING TO CLOSE
386         this.addWindowListener(new ClosingOp());
387
388         screenDimension = Toolkit.getDefaultToolkit().getScreenSize();
389
390         createAndShowLoginGUI();
391     }
392
393
394
395     /*
396      * The next functions are helpers made to easily create objects
397      *
398      */
399
400     private JTextField makeText(String command) {
401         JTextField txt = new JTextField("");
402         txt.setActionCommand(command);
```

```
403         txt.addActionListener(this);
404         return txt;
405     }
406 
407     private JButton makeButton(String caption) {
408         JButton b = new JButton(caption);
409         b.setActionCommand(caption);
410         b.addActionListener(this);
411         return b;
412     }
413 
414     private <U> JList<U> makeListScroller(DefaultListModel<U> list) {
415 
416         JList<U> jList = new JList<U>(list);
417         jList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
418         jList.setLayoutOrientation(JList.VERTICAL_WRAP);
419         jList.setVisibleRowCount(-1);
420         jList.setBorder(BorderFactory.createEmptyBorder(3, 3, 3, 3));
421         return jList;
422     }
423 
424     private GroupLayout setLayout(int width, int height) {
425 
426         GroupLayout layout = new GroupLayout(getContentPane());
427         getContentPane().setPreferredSize(new Dimension(width, height));
428         getContentPane().removeAll();
429         getContentPane().setLayout(layout);
430 
431         layout.setAutoCreateGaps(true);
432         layout.setAutoCreateContainerGaps(true);
433         return layout;
434     }
435 
436 
437     /*
438      * This are the main functions used to set the graphics.
439      *
440      * - LoginGUI is the login interface, where the user log in and register.
441      *
442      * - LobbyGUI is the lobby interface, where the user can accept, decline,
443      *      create new games and see the global ranking.
444      *
445      * - NewGameGUi is an interface made to create new game, and in it the
446      *      User can select the online player with he wants to play.
447      *
448      * - GameGUI is the interface to play the game.
449      *
450      */
451 
452     private void createAndShowLoginGUI() {
453 
454         state = GUI_state.LOGIN;
455 
456         //Create and set up the window.
457         JLabel lblUsername = new JLabel("Username: ");
458         JLabel lblPassword = new JLabel("Password: ");
459         JButton butLogin = makeButton(LOGIN);
460         JButton butRegister = makeButton(REGISTER);
461         txtPassword = new JPasswordField("");
462         txtPassword.setActionCommand(LOGIN);
463         txtPassword.addActionListener(this);
464         txtUsername = makeText(LOGIN);
465 
466         // Dimension for Login GUI
467         GroupLayout layout = setLayout(screenDimension.width / 4 - 10, screenDimension.height /
            8);
468 
469         layout.setHorizontalGroup(layout.createSequentialGroup()
470             .addGroup(layout.createParallelGroup()
```

```
                .addComponent(lblUsername)
                .addComponent(lblPassword))
            .addGroup(layout.createParallelGroup()
                .addComponent(txtUsername)
                .addComponent(txtPassword)
                .addGroup(layout.createSequentialGroup()
                    .addComponent(butLogin)
                    .addComponent(butRegister)))
        );

        layout.linkSize(SwingConstants.HORIZONTAL, butLogin, butRegister);

        layout.setVerticalGroup(layout.createSequentialGroup()
            .addGroup(layout.createParallelGroup()
                .addComponent(lblUsername)
                .addComponent(txtUsername))
            .addGroup(layout.createParallelGroup()
                .addComponent(lblPassword)
                .addComponent(txtPassword))
            .addGroup(layout.createParallelGroup(BASELINE)
                .addComponent(butLogin)
                .addComponent(butRegister))
        );

        setTitle("Login / Register");

        //Display the window.
        pack();
        setVisible(true);

        this.setLocation((screenDimension.width - this.getWidth()) / 2 - 30
                , (screenDimension.height - this.getHeight()) / 2 - 30);
    }


    private void createAndShowLobbyGUI() {

        state = GUI_state.LOBBY;

        JLabel lblGameRequest = new JLabel("Games Request:");
        JLabel lblRanking = new JLabel("Rankings:");
        JButton butNewGame = makeButton(NEWGAME);
        JButton butRankings = makeButton(RANKINGS);
        JButton butHelp = makeButton(HELP);
        JButton butLogout = makeButton(LOGOUT);
        JScrollPane listInvites = new JScrollPane(jListInvites = makeListScroller(listInv));
        JScrollPane listRanking = new JScrollPane(makeListScroller(listRank));
        butAcceptInvite = makeButton(ACCEPTGAME);
        butAcceptInvite.setEnabled(false);
        butDeclineInvite = makeButton(DECLINEGAME);
        butDeclineInvite.setEnabled(false);
        lblUsername = new JLabel("Welcome " + username + "!");

        jListInvites.addListSelectionListener(this);
        listRank.removeAllElements();

        butNewGame.setToolTipText("Start a new game!");

        layout = setLayout(screenDimension.width / 2, screenDimension.height / 2);

        layout.setHorizontalGroup(layout.createSequentialGroup()
            .addGroup(layout.createParallelGroup()
                .addComponent(lblUsername)
                .addComponent(butNewGame)
                .addComponent(butAcceptInvite)
                .addComponent(butDeclineInvite)
                .addComponent(butRankings)
                .addComponent(butHelp)
                .addComponent(butLogout))
```

```
            .addGroup(layout.createParallelGroup(GroupLayout.Alignment.CENTER)
                .addComponent(lblGameRequest)
                .addComponent(listInvites))
            .addGroup(layout.createParallelGroup(GroupLayout.Alignment.CENTER)
                .addComponent(lblRanking)
                .addComponent(listRanking))
        );

        layout.linkSize(SwingConstants.HORIZONTAL, butNewGame, butAcceptInvite,
                        butDeclineInvite, butRankings, butLogout, butHelp);
        layout.linkSize(SwingConstants.VERTICAL, lblGameRequest, lblRanking);

        layout.setVerticalGroup(layout.createSequentialGroup()
            .addGroup(layout.createParallelGroup()
                .addComponent(lblUsername)
                .addComponent(lblGameRequest)
                .addComponent(lblRanking))
            .addGroup(layout.createParallelGroup()
                .addGroup(layout.createSequentialGroup()
                    .addComponent(butNewGame)
                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED,
                            GroupLayout.DEFAULT_SIZE, 10)
                    .addComponent(butAcceptInvite)
                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED,
                            GroupLayout.DEFAULT_SIZE, 10)
                    .addComponent(butDeclineInvite)
                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED,
                            GroupLayout.DEFAULT_SIZE, 10)
                    .addComponent(butRankings)
                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED,
                            GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                    .addComponent(butHelp)
                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED,
                            GroupLayout.DEFAULT_SIZE, 10)
                    .addComponent(butLogout))
                .addComponent(listInvites)
                .addComponent(listRanking))
            .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED,
                    10, 10)
        );

        setTitle("Lobby");
        pack();
        this.setLocation((Toolkit.getDefaultToolkit().getScreenSize().width - this.getWidth()) /
        2,
                (Toolkit.getDefaultToolkit().getScreenSize().height - this.getHeight()) / 2);

    }


    private void createAndShowNewGameGUI() {

        state = GUI_state.NEWGAME;

        JLabel lblOnlinePlayers = new JLabel("Online Players: ");
        JLabel lblSelectedPlayers = new JLabel("Invited Players: ");
        listOn = new DefaultListModel<String>();
        listIn = new DefaultListModel<String>();
        JScrollPane listOnline = new JScrollPane(jListOnline = makeListScroller(listOn));
        JScrollPane listInvited = new JScrollPane(jListInvited = makeListScroller(listIn));
        JButton butCancel = makeButton(CANCEL);
        JButton butAddAll = makeButton(ADDALL);
        JButton butRemoveAll = makeButton(REMOVEALL);
        JButton butUpdate = makeButton(UPDATE);
        butSend = makeButton(SENDINVITES);
        butSend.setEnabled(false);
        butSwitch = makeButton(SWITCH);
        butSwitch.setEnabled(false);

```

```
608         jListInvited.addListSelectionListener(this);
609         jListOnline.addListSelectionListener(this);
610 
611         GroupLayout layout = setLayout(screenDimension.width / 3 - 10, screenDimension.height /
            3);
612 
613         layout.setHorizontalGroup(layout.createSequentialGroup()
614             .addGroup(layout.createParallelGroup(GroupLayout.Alignment.CENTER)
615                 .addComponent(lblOnlinePlayers)
616                 .addComponent(listOnline)
617                 .addComponent(butSend))
618             .addGroup(layout.createParallelGroup()
619                 .addComponent(butSwitch)
620                 .addComponent(butAddAll)
621                 .addComponent(butRemoveAll)
622                 .addComponent(butUpdate))
623             .addGroup(layout.createParallelGroup(GroupLayout.Alignment.CENTER)
624                 .addComponent(lblSelectedPlayers)
625                 .addComponent(listInvited)
626                 .addComponent(butCancel))
627         );
628 
629         layout.setVerticalGroup(layout.createSequentialGroup()
630             .addGroup(layout.createParallelGroup()
631                 .addComponent(lblOnlinePlayers)
632                 .addComponent(lblSelectedPlayers))
633             .addGroup(layout.createParallelGroup(GroupLayout.Alignment.CENTER)
634                 .addComponent(listOnline)
635                 .addGroup(layout.createSequentialGroup()
636                     .addComponent(butSwitch)
637                     .addComponent(butAddAll)
638                     .addComponent(butRemoveAll)
639                     .addComponent(butUpdate))
640                 .addComponent(listInvited))
641             .addGroup(layout.createParallelGroup()
642                 .addComponent(butSend)
643                 .addComponent(butCancel))
644         );
645 
646         layout.linkSize(SwingConstants.HORIZONTAL, butSend, butCancel);
647         layout.linkSize(SwingConstants.HORIZONTAL, butSwitch, butAddAll, butRemoveAll,
            butUpdate);
648 
649         setTitle("New Game");
650         pack();
651 
652         this.setLocation((Toolkit.getDefaultToolkit().getScreenSize().width - this.getWidth()) /
            2,
653                 (Toolkit.getDefaultToolkit().getScreenSize().height - this.getHeight()) / 2);
654         // Request a list of online users
655         (new FindOnlineUsersTask()).execute();
656     }
657 
658 
659     private void createAndShowGameGUI() {
660 
661         state = GUI_state.PLAYING;
662 
663         JLabel lblLet = new JLabel("Letters: ");
664         JLabel lblFWords = new JLabel("Found words: ");
665         JPanel pnlErrors = new JPanel();
666         JSeparator separator = new JSeparator(SwingConstants.VERTICAL);
667         JSeparator separator2 = new JSeparator(SwingConstants.VERTICAL);
668         JButton butHelp = makeButton(HELP);
669         butSendWord = makeButton(SENDWORD);
670         listFWords = new DefaultListModel<String>();
671         listLRanking = new DefaultListModel<String>();
672         JScrollPane listFoundWords = new JScrollPane(makeListScroller(listFWords));
673         JScrollPane listLocalRanking = new JScrollPane(makeListScroller(listLRanking));
```

```
674         butSurrend = makeButton(SURREND);
675 
676         // Setting paramters for game start
677         gameStarted = false;
678         lblState = new JLabel("Waiting for other players");
679         lblTimer = new JLabel("Time Left: 420");
680         lblErrors = new JLabel("");
681         lblLetters = new JLabel("");
682         lblRanking = new JLabel("Rankings:");
683         waitingForWords = new Timer(1000, this);
684         waitingForWords.setActionCommand(TIMER);
685         timeLeft = 420; // 7 minutes to receive the words from the server
686         txtSendWord = makeText(SENDWORD);
687         txtSendWord.setEnabled(false);
688         lblLetters.setFont(new Font("Courier New", Font.CENTER_BASELINE, 16));
689         lblLetters.setForeground(Color.BLUE);
690         lblErrors.setForeground(Color.RED);
691         lblRanking.setForeground(Color.RED);
692         pnlErrors.add(lblState);
693         pnlErrors.add(lblTimer);
694         pnlErrors.add(lblErrors);
695         pnlErrors.setBorder(BorderFactory.createEmptyBorder(4, 4, 4, 4));
696         pnlErrors.setBackground(Color.GRAY);
697 
698         waitingForWords.start();
699 
700         GroupLayout layout = setLayout(screenDimension.width / 3 + 90, 100 +
            screenDimension.height / 3);
701 
702         // used to allign butSurrend and butHelp
703         int maxSize;
704         if (butSurrend.getPreferredSize().width > butHelp.getPreferredSize().width)
705             maxSize = butSurrend.getPreferredSize().width;
706         else maxSize = butHelp.getPreferredSize().width;
707 
708         layout.setHorizontalGroup(layout.createSequentialGroup()
709             .addGroup(layout.createParallelGroup(GroupLayout.Alignment.CENTER)
710                 .addComponent(lblLet)
711                 .addComponent(lblFWords)
712                 .addComponent(listFoundWords))
713             .addGroup(layout.createParallelGroup(GroupLayout.Alignment.CENTER)
714                 .addComponent(lblLetters)
715                 .addGroup(layout.createSequentialGroup()
716                     .addComponent(separator, 3, 3, 3)
717                     .addGroup(layout.createParallelGroup()
718                         .addComponent(txtSendWord, 0, 2 * maxSize + 5, Short.MAX_VALUE)
719                         .addComponent(butSendWord, 0, 2 * maxSize + 5, Short.MAX_VALUE)
720                         .addGroup(layout.createSequentialGroup()
721                             .addComponent(butHelp)
722                             .addComponent(butSurrend))
723                         .addComponent(pnlErrors, 0, 2 * maxSize + 5, 2 * maxSize + 5))
724                     .addComponent(separator2, 3, 3, 3)))
725             .addGroup(layout.createParallelGroup(GroupLayout.Alignment.CENTER)
726                 .addComponent(lblRanking)
727                 .addComponent(listLocalRanking))
728         );
729 
730         layout.setVerticalGroup(layout.createSequentialGroup()
731             .addGroup(layout.createParallelGroup()
732                 .addComponent(lblLet)
733                 .addComponent(lblLetters))
734             .addGroup(layout.createParallelGroup()
735                 .addComponent(lblRanking)
736                 .addComponent(lblFWords))
737             .addGroup(layout.createParallelGroup()
738                 .addComponent(listFoundWords)
739                 .addComponent(separator)
740                 .addGroup(layout.createSequentialGroup()
741                     .addComponent(txtSendWord,
```

```
                            txtSendWord.getPreferredSize().height,
                            txtSendWord.getPreferredSize().height,
                            txtSendWord.getPreferredSize().height)
                    .addComponent(butSendWord)
                    .addGroup(layout.createParallelGroup()
                        .addComponent(butHelp)
                        .addComponent(butSurrend))
                    .addComponent(pnlErrors))
                .addComponent(separator2)
                .addComponent(listLocalRanking))
        );

        layout.linkSize(SwingConstants.HORIZONTAL, butSurrend, butHelp);
        layout.linkSize(SwingConstants.HORIZONTAL, butSendWord, txtSendWord);

        setTitle("Playing");
        pack();
        this.setLocation((Toolkit.getDefaultToolkit().getScreenSize().width - this.getWidth()) /
        2,
                (Toolkit.getDefaultToolkit().getScreenSize().height - this.getHeight()) / 2);

    }


    /*
     * valueChanged and actionPerformed are action listener made to
     * perform the right action after an User made an action.
     *
     */

    public void valueChanged(ListSelectionEvent e) {

        if (e.getSource() == jListInvites) {
            if (jListInvites.getSelectedIndex() == -1) {

                //No selection, disable buttons.
                butAcceptInvite.setEnabled(false);
                butDeclineInvite.setEnabled(false);
            } else {

                //Selection, enable the buttons.
                butAcceptInvite.setEnabled(true);
                butDeclineInvite.setEnabled(true);
            }
        } else {
            if (jListOnline == null || jListInvited == null) return;
            if (jListOnline.getSelectedIndex() == -1 && jListInvited.getSelectedIndex() == -1) {

                //No selection, disable button.
                butSwitch.setEnabled(false);
            } else {

                //Selection, enable the button.
                if (e.getSource() == jListInvited) {
                    jListOnline.clearSelection();
                    jListInvited.setSelectedIndex(jListInvited.getSelectedIndex());
                    butSwitch.setEnabled(true);
                }
                else {
                    jListInvited.clearSelection();
                    jListOnline.setSelectedIndex(jListOnline.getSelectedIndex());
                    butSwitch.setEnabled(true);
                }
            }
        }
    }


    @Override
```

```
810     public void actionPerformed(ActionEvent action) {
811
812         switch(action.getActionCommand()) {
813             case LOGIN : verifyAndStartLogin(); break;
814             case REGISTER : verifyAndStartRegister(); break;
815             case NEWGAME : createAndShowNewGameGUI(); break;
816             case ACCEPTGAME : tryToAccept(true); break;
817             case DECLINEGAME : tryToAccept(false); break;
818             case RANKINGS : (new RankingTask()).execute(); break;
819             case LOGOUT : (new LogoutTask()).execute(); break;
820             case SENDINVITES : (new SendGameRequestTask()).execute(); break;
821             case CANCEL : createAndShowLobbyGUI(); break;
822             case ADDALL : addAllList(); break;
823             case REMOVEALL : removeAllList(); break;
824             case SWITCH : switchList(); break;
825             case UPDATE : update(); break;
826             case SURREND : if (gameTimer != null) gameTimer.stop();
827                            if (waitingForWords != null) waitingForWords.stop();
828                            if (acceptTask != null) acceptTask.interrupt();
829                            createAndShowLobbyGUI(); break;
830             case SENDWORD : addAGameWord(); break;
831             case TIMER : updateTimerWait(); break;
832             case GAMETIMER : gameEnd(); break;
833             case HELP : help(); break;
834             default : System.out.println("Errore nella lettura del comando: comando non
                esistente.");
835         }
836     }
837
838
839     /*
840      * This are self-explanatory function made to interact with
841      * worker threads.
842      * Some of them are made to create and start new worker threads,
843      * and some to let the GUI thread do the remaining work that
844      * a normal worker thread can't do.
845      */
846
847     private void addAGameWord() {
848
849         if (txtSendWord == null || lblLetters.getText() == null) return;
850         if (lblLetters.getText().equals("")) return;
851         String fWord = txtSendWord.getText();
852         if (fWord.equals("")) {
853             System.out.println("You should set a word!");
854             lblErrors.setText("Choose a word!");
855             lblErrors.setForeground(Color.RED);
856             return;
857         }
858         if (isValidWord(fWord, lblLetters.getText())) {
859             listFWords.addElement(txtSendWord.getText());
860             txtSendWord.setText("");
861             lblErrors.setText("Added word!");
862             lblErrors.setForeground(Color.GREEN);
863
864         } else {
865             lblErrors.setText("Not valid word!");
866             lblErrors.setForeground(Color.RED);
867         }
868     }
869     private boolean isValidWord(String word, String base) {
870
871         // controls if it is an already choosen word
872         for (int i = 0; i < listFWords.size(); i++) {
873             if (word.equals(listFWords.getElementAt(i))) return false;
874         }
875
876         ArrayList<String> w = new ArrayList<String>(), b = new ArrayList<String>();
877         for (int j = 0; j < word.length(); j++) w.add(word.substring(j, j+1));
```

```
878         for (int j = 0; j < base.length(); j++) b.add(base.substring(j, j+1));
879 
880         // Strings implements comparable
881         b.sort(null);
882         w.sort(null);
883 
884         if (b.size() < w.size()) return false;
885 
886         // the words single letters group must be a subset of the base single letter group so,
887         // when searching for a letter in a position i in the first array, it can already start
888         // at position i in the second array.
889         int j = 0, i = 0;
890 
891         while (i < w.size() && j < b.size()) {
892             if (w.get(i).equals(b.get(j))) {
893                 i++;
894                 j++;
895             } else j++;
896         }
897         return (i == w.size());
898 
899     }
900     private void gameEnd() {
901 
902         if (timeLeft > 0) {
903             timeLeft--;
904             lblState.setText("Games started!");
905             lblTimer.setText("Time left: " + timeLeft);
906         } else {
907 
908             gameTimer.stop();
909 
910             String[] words = new String[listFWords.size()];
911             for (int i = 0; i < listFWords.size(); i++) words[i] = listFWords.getElementAt(i);
912 
913             (new SendWordsTask(words)).execute();
914             lblErrors.setText("Waiting for results!");
915             lblState.setText("Games ended!");
916             lblTimer.setText("---");
917         }
918     }
919     private void updateTimerWait() {
920         if (!gameStarted) {
921 
922             if (timeLeft > 0) {
923 
924                 timeLeft--;
925                 lblTimer.setText("Time left: " + timeLeft);
926             }
927             else {
928                 // Game cancelled; it still needs to wait until the server cancell it
929                 waitingForWords.stop();
930             }
931         }
932     }
933     private void update() {
934         switch(state) {
935             case LOGIN: break;
936             case LOBBY: break;
937             case NEWGAME: (new FindOnlineUsersTask()).execute(); break;
938             case PLAYING: break;
939             default: System.out.println("Errore: stato inconsistente.");;
940         }
941     }
942     private void addAllList() {
943         int dim = listOn.size();
944         for (int i = 0; i < dim; i++) {
945             listIn.addElement(listOn.getElementAt(0));
946             listOn.remove(0);
```

```
947         }
948         butSend.setEnabled(!listIn.isEmpty());
949     }
950     private void removeAllList() {
951         int dim = listIn.size();
952         for (int i = 0; i < dim; i++) {
953             listOn.addElement(listIn.getElementAt(0));
954             listIn.remove(0);
955         }
956         butSend.setEnabled(!listIn.isEmpty());
957     }
958     private void switchList() {
959         if (jListInvited.isSelectionEmpty()) {
960             if (!jListOnline.isSelectionEmpty()) {
961                 String val = jListOnline.getSelectedValue();
962                 listOn.remove(jListOnline.getSelectedIndex());
963                 listIn.addElement(val);
964             }
965         } else {
966             if (jListOnline.isSelectionEmpty()) {
967                 String val = jListInvited.getSelectedValue();
968                 listIn.remove(jListInvited.getSelectedIndex());
969                 listOn.addElement(val);
970             }
971         }
972         butSend.setEnabled(!listIn.isEmpty());
973     }
974     private void verifyAndStartLogin() {
975 
976         if (!state.equals(GUI_state.LOGIN)) state = GUI_state.LOGIN;
977         if (txtUsername == null || txtUsername.getText().equals("")) {
978             System.out.println("Devi inserire un username!");
979             JOptionPane.showMessageDialog(this,
980                     "You must choose an username!",
981                     "Missing Username",
982                     JOptionPane.ERROR_MESSAGE);
983         }
984         else if (txtPassword == null || txtPassword.getPassword().length == 0) {
985             System.out.println("Devi inserire una password!");
986             JOptionPane.showMessageDialog(this,
987                     "You must choose a password!",
988                     "Missing Password",
989                     JOptionPane.ERROR_MESSAGE);
990 
991         }
992         else {
993             password = String.valueOf(txtPassword.getPassword());
994             username = txtUsername.getText();
995             txtUsername.setText("");
996             txtPassword.setText("");
997             (new LoginTask()).execute();
998         }
999     }
1000     private void verifyAndStartRegister() {
1001 
1002         if (!state.equals(GUI_state.LOGIN)) state = GUI_state.LOGIN;
1003         if (txtUsername == null || txtUsername.getText().equals("")) {
1004             System.out.println("Devi inserire un username!");
1005             JOptionPane.showMessageDialog(this,
1006                     "You must choose an username!",
1007                     "Missing Username",
1008                     JOptionPane.ERROR_MESSAGE);
1009         }
1010         else if (txtPassword == null || txtPassword.getPassword().length == 0) {
1011             System.out.println("Devi inserire una password!");
1012             JOptionPane.showMessageDialog(this,
1013                     "You must choose a password!",
1014                     "Missing Password",
1015                     JOptionPane.ERROR_MESSAGE);
```

```
1016         }
1017         else {
1018             password = String.valueOf(txtPassword.getPassword());
1019             username = txtUsername.getText();
1020             txtUsername.setText("");
1021             txtPassword.setText("");
1022             (new RegisterTask()).execute();
1023         }
1024     }
1025     private void tryToAccept(boolean accept) {
1026 
1027         if (jListInvites.isSelectionEmpty()) {
1028             System.out.println("Nessun oggetto selezionato!");
1029             JOptionPane.showMessageDialog(this,
1030                     "You must select an object!",
1031                     "No object selected",
1032                     JOptionPane.ERROR_MESSAGE);
1033         } else {
1034 
1035             if (accept) {
1036 
1037                 /*
1038                  * Accept the game:
1039                  * erase every element on game invite list,
1040                  * and prepare an array to tell wich gameID
1041                  * he's refusing by accepting this game
1042                  *
1043                  */
1044 
1045                 currentPlayingGame = jListInvites.getSelectedValue();
1046 
1047                 listInv.remove(jListInvites.getSelectedIndex());
1048 
1049                 int[] refusedGames = new int[listInv.size()];
1050 
1051                 for (int i = 0; i < listInv.size(); i++) {
1052                     refusedGames[i] = listInv.getElementAt(i).getID();
1053                 }
1054                 listInv.removeAllElements();
1055 
1056                 acceptTask = new AcceptGameTask(accept, refusedGames);
1057                 acceptTask.execute();
1058 
1059                 createAndShowGameGUI();
1060 
1061             } else {
1062 
1063                 /*
1064                  * Refusing the game:
1065                  * erase the refused game from game invite list,
1066                  * sets the refuse array to this only element.
1067                  *
1068                  */
1069 
1070                 int[] refusedGame = new int[1];
1071 
1072                 refusedGame[0] = listInv.getElementAt(jListInvites.getSelectedIndex()).getID();
1073                 listInv.remove(jListInvites.getSelectedIndex());
1074 
1075                 acceptTask = new AcceptGameTask(false, refusedGame);
1076                 acceptTask.execute();
1077 
1078             }
1079         }
1080     }
1081     private void help() {
1082 
1083         if (state.equals(GUI_state.LOBBY)) {
1084             JOptionPane.showMessageDialog(this,
```

```
1085                    "This is the Lobby interface.                " + System.lineSeparator() +
1086                    "From here you can:                          " + System.lineSeparator() +
1087                    "- Create a new game, inviting online players," + System.lineSeparator() +
1088                    "- Accept  a  game  invite from other players," + System.lineSeparator() +
1089                    "- Refuse  a  game  invite from other players," + System.lineSeparator() +
1090                    "- Update  the  Rankings and see you position," + System.lineSeparator() +
1091                    "- Logout."       +      System.lineSeparator() + System.lineSeparator() +
1092                    "- Creating   and   accepting   a   game  will" + System.lineSeparator() +
1093                    "   automatically   refuse   all  the  others." + System.lineSeparator(),
1094                    "Help",
1095                    JOptionPane.INFORMATION_MESSAGE);
1096        } else if (state.equals(GUI_state.PLAYING)){
1097            JOptionPane.showMessageDialog(this,
1098                    "This  is  the interface of a game session.   " + System.lineSeparator() +
1099                    "From here you can play your game:           " + System.lineSeparator() +
1100                    "- Write  a word and press Send to send it,   " + System.lineSeparator() +
1101                    "- You  can't send the same word two times,   " + System.lineSeparator() +
1102                    "- You  should  use only the given letters,   " + System.lineSeparator() +
1103                    "- After  the  end of the timer, the result   " + System.lineSeparator() +
1104                    "    will  be  sent  and shown in rankings,   " + System.lineSeparator() +
1105                    "- You  can  surrend  by  pressing Surrend;   " + System.lineSeparator() +
1106                    "    By  surrending  the game will continue   " + System.lineSeparator() +
1107                    "    for  other  players  and you can still   " + System.lineSeparator() +
1108                    "    and you can find your result in global   " + System.lineSeparator() +
1109                    "    rankings.",
1110                    "Help",
1111                    JOptionPane.INFORMATION_MESSAGE);
1112        }
1113
1114    }
1115    private Socket openConnection() throws IOException {
1116        Socket socket = null;
1117        socket = new Socket(REGISTRY_HOST, TCP_PORT);
1118        return socket;
1119    }
1120    private void closeConnection(Socket socket, ObjectOutputStream out, ObjectInputStream in)
1121            throws IOException {
1122        socket.close();
1123        out.close();
1124        in.close();
1125    }
1126    public void addGameRequest(String creator, int gameID, int multicastPort) {
1127
1128        java.awt.EventQueue.invokeLater(new Runnable() {
1129            public void run() {
1130
1131                // can't accept games request while not online
1132                if (!state.equals(GUI_state.LOBBY)) {
1133                    int[] refusedGame = new int[1];
1134                    refusedGame[0] = gameID;
1135                    (new AcceptGameTask(false, refusedGame)).execute();
1136                    return;
1137                }
1138
1139                GamesData game = new GamesData(creator, gameID, multicastPort);
1140                games.addGame(game);
1141                listInv.addElement(game);
1142            }
1143        });
1144    }
1145    public void removeGameRequest(int gameID) {
1146
1147        java.awt.EventQueue.invokeLater(new Runnable() {
1148            public void run() {
1149                GamesData game = games.getGameByID(gameID);
1150                games.removeGame(game);
1151                listInv.removeElement(game);
1152            }
1153        });
```

```
1154      }
1155      private void sendAllertMessage(String mex, String label, int err) {
1156
1157          JOptionPane.showMessageDialog(this, mex, label, err);
1158      }
1159      private void sendCloseMessage() {
1160
1161          if (JOptionPane.showConfirmDialog(this, "Are you sure ?") == JOptionPane.OK_OPTION){
1162              this.setVisible(false);
1163              try {
1164                  if (username != null && password != null) serverRMI.logout(username, password);
1165              } catch (RemoteException e) {
1166                  e.printStackTrace();
1167              }
1168              System.exit(0);
1169
1170          }
1171      }
1172
1173
1174      private class ClosingOp implements WindowListener {
1175
1176          // USED TO ASK FOR CLOSING
1177
1178          @Override
1179          public void windowClosed(WindowEvent e) {}
1180
1181          @Override
1182          public void windowActivated(WindowEvent e) {}
1183
1184          @Override
1185          public void windowClosing(WindowEvent e) {
1186              sendCloseMessage();
1187          }
1188
1189          @Override
1190          public void windowDeactivated(WindowEvent e) {}
1191
1192          @Override
1193          public void windowDeiconified(WindowEvent e) {}
1194
1195          @Override
1196          public void windowIconified(WindowEvent e) {}
1197
1198          @Override
1199          public void windowOpened(WindowEvent e) {}
1200
1201      }
1202
1203
1204      private class LoginTask extends SwingWorker<Void, Void> {
1205
1206          private int result = -3;
1207
1208          @Override
1209          protected Void doInBackground() {
1210
1211              // comunicate with server: tries to login with RMI
1212              // and gives him a callback for game invites
1213
1214              try {
1215                  result = serverRMI.login(username, password, userStub);
1216              } catch (RemoteException e) {
1217                  result = -4;
1218              }
1219              return null;
1220          }
1221
1222          @Override
```

```
1223          protected void done() {
1224
1225              if (result >= 0) {
1226                  System.out.println("Login avvenuto con sucesso");
1227                  sessionID = result;
1228
1229                  createAndShowLobbyGUI();
1230              } else if (result == -1) {
1231
1232                  sendAllertMessage("You chose the wrong password!",
1233                              "Wrong Password",
1234                              JOptionPane.ERROR_MESSAGE);
1235
1236              } else if (result == -2) {
1237
1238                  sendAllertMessage("This Account doens't Exist!",
1239                              "Account non existent",
1240                              JOptionPane.ERROR_MESSAGE);
1241
1242              } else if (result == -3) {
1243
1244                  sendAllertMessage("This user is already logged in!",
1245                              "Already Logged User",
1246                              JOptionPane.ERROR_MESSAGE);
1247              } else if (result == -4) {
1248
1249                  sendAllertMessage("Comunication Error; impossible to login!",
1250                              "Error",
1251                              JOptionPane.ERROR_MESSAGE);
1252              }
1253          }
1254      }
1255
1256
1257      private class RegisterTask extends SwingWorker<Void, Void> {
1258
1259          private boolean success = false;
1260
1261          @Override
1262          protected Void doInBackground() {
1263
1264              // comunicates with server: tries to register with RMI
1265              try {
1266                  success = serverRMI.register(username, password);
1267              } catch (RemoteException e) {
1268
1269              }
1270
1271              return null;
1272          }
1273
1274          @Override
1275          protected void done() {
1276
1277              if (success) {
1278
1279                  System.out.println("Registrazione avvenuta con sucesso");
1280                  // starts login automatically
1281
1282                  (new LoginTask()).execute();
1283              } else {
1284
1285                  sendAllertMessage("This account already exist!",
1286                              "Account already existent",
1287                              JOptionPane.ERROR_MESSAGE);
1288              }
1289          }
1290      }
1291
```

```
1292
1293     private class RankingTask extends SwingWorker<Multicast_rankings, Void> {
1294
1295         private boolean success = false;
1296
1297         @Override
1298         protected Multicast_rankings doInBackground() {
1299
1300             Multicast_rankings rankings = null;
1301             try {
1302
1303                 // TCP comunication to get rankings
1304                 Socket socket = openConnection();
1305                 ObjectOutputStream writer = new ObjectOutputStream (socket.getOutputStream());
1306                 ObjectInputStream reader = new ObjectInputStream (socket.getInputStream());
1307
1308                 writer.writeInt(2);
1309                 writer.flush();
1310
1311                 rankings = (Multicast_rankings) reader.readObject();
1312
1313                 success = true;
1314
1315                 closeConnection(socket, writer, reader);
1316
1317             } catch (IOException e) { // to ignore errors in socket closing
1318             } catch (ClassNotFoundException e) {
1319             }
1320             return rankings;
1321         }
1322
1323         @Override
1324         protected void done() {
1325
1326             if (success && state.equals(GUI_state.LOBBY)) {
1327
1328                 try {
1329
1330                     Multicast_rankings rankings = get();
1331
1332                     System.out.println("Lettura classifica avvenuta con successo");
1333
1334                     String[] rankEl = rankings.getStringElements();
1335                     listRank.clear();
1336                     for (int i = 0; i < rankEl.length; i++)
1337                         listRank.addElement((i+1) + "°: " + rankEl[i]);
1338
1339
1340                 } catch (InterruptedException | ExecutionException e) {
1341                     System.out.println("Impossibile leggere la classifica");
1342                     e.printStackTrace();
1343                 }
1344             } else {
1345
1346                 sendAllertMessage("Comunication Error; impossible to read rankings!",
1347                         "Error",
1348                         JOptionPane.ERROR_MESSAGE);
1349             }
1350         }
1351     }
1352
1353
1354     private class LogoutTask extends SwingWorker<Void, Void> {
1355
1356         private boolean success = false;
1357
1358         @Override
1359         protected Void doInBackground() {
1360
```

```
1361            // tries to logout via RMI
1362            try {
1363                success = 0 == serverRMI.logout(username, password);
1364            } catch (RemoteException e) {
1365                System.out.println("Impossibile effettuare il logout."
1366                        + " Persa la connessione col Server.");
1367
1368            }
1369
1370            return null;
1371        }
1372
1373        @Override
1374        protected void done() {
1375
1376            if (success) {
1377                System.out.println("Logout successful");
1378                username = "";
1379                password = "";
1380                sessionID = 0;
1381                createAndShowLoginGUI();
1382            } else {
1383
1384                System.out.println("Internal error: cannot logout");
1385                // It still logout
1386                username = "";
1387                password = "";
1388                sessionID = 0;
1389                sendAllertMessage("Errors doing Logout",
1390                        "Error with Server",
1391                        JOptionPane.ERROR_MESSAGE);
1392                createAndShowLoginGUI();
1393            }
1394        }
1395    }
1396
1397
1398    private class AcceptGameTask extends SwingWorker<Void, Void> {
1399
1400        private boolean accepted;
1401        private int[] refusedGame;
1402        private boolean success = false;
1403        private String word = null;
1404        Socket socket;
1405
1406        AcceptGameTask(boolean accepted, int[] refusedGame) {
1407            this.accepted = accepted;
1408            this.refusedGame = refusedGame;
1409        }
1410
1411        @Override
1412        protected Void doInBackground() {
1413
1414            try {
1415
1416                /*
1417                 * Use TCP comunication to answer a game.
1418                 *
1419                 * This method could be invoked for 3 different reasons:
1420                 * 1 - an user accepted a game and want to warn the server
1421                 *      (and refuse all the other games he's been invited)
1422                 *
1423                 * 2 - an user refused a game and want to warn the server
1424                 *
1425                 * 3 - an user created a game, and want to refuse all the
1426                 *      game invited.
1427                 *
1428                 * In any case, some other games may be refused.
1429                 *
```

```
1430                 */
1431
1432                 socket = openConnection();
1433                 ObjectOutputStream writer = new ObjectOutputStream (socket.getOutputStream());
1434                 ObjectInputStream reader = new ObjectInputStream (socket.getInputStream());
1435
1436                 writer.writeInt(1);
1437                 writer.writeUTF(username);
1438                 writer.writeInt(sessionID);
1439
1440                 // It writed the refused game (if it refused one)
1441                 // or the accepted game.
1442                 if (!accepted && refusedGame != null){
1443                     writer.writeInt(refusedGame[0]);
1444                 } else {
1445                     writer.writeInt(currentPlayingGame.getID());
1446                 }
1447                 writer.writeBoolean(accepted);
1448
1449                 // After accepting a game, it writes the other refused game
1450                 // (to accept this one). It doesn't do anything if it's just
1451                 // refusing a game.
1452                 int len = 0;
1453                 if (accepted && refusedGame != null) len = refusedGame.length;
1454
1455                 writer.writeInt(len);
1456
1457                 for (int i = 0; i < len; i++) {
1458                     writer.writeInt(refusedGame[i]);
1459                 }
1460
1461                 // if he refused the game, the server automatically closes the connection.
1462                 writer.flush();
1463                 switch (reader.readInt()) {
1464                     case -1:
1465                     case -2:
1466                     case -3: return null;
1467                     case 0:
1468
1469                         try {
1470                             word = reader.readUTF();
1471                             success = true;
1472                         } catch (IOException e) {
1473
1474                             // If it arrives here, the server annulled the game.
1475                             success = false;
1476                         }
1477
1478                         break;
1479                     default:;
1480                 }
1481
1482                 closeConnection(socket, writer, reader);
1483
1484             } catch (IOException e) {
1485             }
1486
1487             return null;
1488         }
1489
1490
1491         @Override
1492         protected void done() {
1493
1494             if (waitingForWords != null) waitingForWords.stop();
1495             if (!state.equals(GUI_state.PLAYING)) return;
1496             if (success) startGame(word);
1497             else if (accepted) {
1498
```

```
1499                System.out.println("Partita annullata");
1500                sendAllertMessage("The game has been canceled!",
1501                        "canceled game",
1502                        JOptionPane.ERROR_MESSAGE);
1503
1504                createAndShowLobbyGUI();
1505            }
1506        }
1507
1508        public void interrupt() {
1509            try {
1510                if (socket != null) socket.close();
1511            } catch (IOException e) {
1512                // il socket non si può chiudere o è già chiuso
1513            }
1514        }
1515    }
1516    private void startGame(String word) {
1517
1518        lblLetters.setText(word);
1519        txtSendWord.setEnabled(true);
1520        gameTimer = new Timer(1000, this);
1521        gameTimer.setActionCommand(GAMETIMER);
1522        gameTimer.start();
1523        timeLeft = 120;
1524        gameStarted = true;
1525    }
1526
1527
1528    private class FindOnlineUsersTask extends SwingWorker<ArrayList<String>, Void> {
1529
1530        @Override
1531        protected ArrayList<String> doInBackground() {
1532
1533            ArrayList<String> onlineUsers = null;
1534
1535            try {
1536                onlineUsers = serverRMI.requestOnlineUsers();
1537            } catch (RemoteException e) {
1538                System.out.println("Impossibile accedere alla lista degli utenti online");
1539                e.printStackTrace();
1540            }
1541
1542            return onlineUsers;
1543        }
1544
1545        @Override
1546        protected void done() {
1547
1548            System.out.println("Refresh avvenuto con successo");
1549            try {
1550                ArrayList<String> onlineUsers = get();
1551                if (onlineUsers != null) {
1552                    listOn.removeAllElements();
1553                    listIn.removeAllElements();
1554                    for (String user: onlineUsers) if
                        (!user.equals(username))listOn.addElement(user);
1555                }
1556            } catch (InterruptedException e) {
1557                e.printStackTrace();
1558            } catch (ExecutionException e) {
1559                e.printStackTrace();
1560            }
1561        }
1562    }
1563
1564
1565    private class SendGameRequestTask extends SwingWorker<Void, Void> {
1566
```

```
1567          private boolean success = false;
1568
1569          @Override
1570          protected Void doInBackground() {
1571
1572              try {
1573
1574                  // Communicates with server with TCP to create a new game.
1575                  // After the communication, if it went right, it accepts
1576                  // automatically the game request from the server.
1577
1578                  Socket socket = openConnection();
1579                  ObjectOutputStream writer = new ObjectOutputStream (socket.getOutputStream());
1580                  ObjectInputStream reader = new ObjectInputStream (socket.getInputStream());
1581
1582                  writer.writeInt(0);
1583                  writer.writeInt(listIn.size());
1584                  writer.writeUTF(username);
1585                  writer.writeInt(sessionID);
1586                  for (int i = 0; i < listIn.size(); i++) {
1587                      writer.writeUTF(listIn.getElementAt(i));
1588                  }
1589                  writer.flush();
1590
1591                  if (success = reader.readBoolean()) {
1592                      int gameID = reader.readInt();
1593                      int gamePort = reader.readInt();
1594                      currentPlayingGame = new GamesData(username, gameID, gamePort);
1595
1596                      closeConnection(socket, writer, reader);
1597                  }
1598
1599                  System.out.println("Invio dati server");
1600
1601              } catch (IOException e) {
1602                  System.out.println("errore nell'invio della lista di inviti al server");
1603              }
1604
1605              return null;
1606          }
1607
1608          @Override
1609          protected void done() {
1610
1611              if (success) {
1612
1613                  System.out.println("invio richiesta partita avvenuto con successo");
1614
1615                  callAcceptGame();
1616                  createAndShowGameGUI();
1617              }
1618          }
1619
1620      }
1621      private void callAcceptGame() {
1622
1623          int[] refusedGames = new int[listInv.size()];
1624
1625          for (int i = 0; i < listInv.size(); i++) {
1626              refusedGames[i] = listInv.getElementAt(i).getID();
1627          }
1628
1629          listInv.removeAllElements();
1630          acceptTask = new AcceptGameTask(true, refusedGames);
1631          acceptTask.execute();
1632      }
1633
1634
1635      private class SendWordsTask extends SwingWorker<Void, Void> {
```

```
1636
1637        String[] words;
1638
1639        SendWordsTask(String[] words) {
1640            this.words = words;
1641        }
1642
1643        @Override
1644        protected Void doInBackground() {
1645
1646            try {
1647
1648                ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
1649                ObjectOutputStream os = new ObjectOutputStream(outputStream);
1650                // data dimension is limited by 1'000'000 bytes.
1651                byte[] data = null;
1652                UDP_words sendWords = new UDP_words(words, currentPlayingGame.getID(), username,
                    sessionID);
1653                os.writeObject(sendWords);
1654                data = outputStream.toByteArray();
1655
1656                DatagramPacket sendPacket = new DatagramPacket(data, data.length, serverAddress,
                    UDP_PORT);
1657                dataSocket.send(sendPacket);
1658
1659                System.out.println("Pacchetto UDP inviato con le parole scelte");
1660
1661                System.out.println("Avvio il task per la lettura multicast dei risultati");
1662
1663                (new ReadMulticastRankings()).execute();
1664
1665
1666            } catch (IOException e) {
1667                System.out.println("Errore nell'invio delle parole.");
1668                e.printStackTrace();
1669            }
1670
1671            return null;
1672        }
1673
1674        @Override
1675        protected void done() {
1676
1677            txtSendWord.setEnabled(false);
1678            butSendWord.setEnabled(false);
1679        }
1680    }
1681
1682
1683    private class ReadMulticastRankings extends SwingWorker<Multicast_rankings, Void> {
1684
1685        @Override
1686        protected Multicast_rankings doInBackground() {
1687
1688            // rankings data is limited by 1'000'000 bytes.
1689            byte [ ] date = new byte[10000000];
1690            Multicast_rankings rankings = null;
1691
1692            try {
1693                DatagramPacket dp = new DatagramPacket (date, date.length);
1694                MulticastSocket ms = new MulticastSocket (currentPlayingGame.getPort());
1695                ms.joinGroup(multicastAddress);
1696
1697                ms.receive(dp);
1698
1699                ByteArrayInputStream in = new ByteArrayInputStream(dp.getData());
1700                ObjectInputStream is = new ObjectInputStream(in);
1701
1702
```

```
1703                 rankings = (Multicast_rankings) is.readObject();
1704 
1705                 ms.close();
1706                 in.close();
1707 
1708             } catch(IOException e) {
1709 
1710                 System.out.println("Errore di comunicazione, impossibile leggere la classifica");
1711                 e.printStackTrace();
1712             } catch (ClassNotFoundException e) {
1713 
1714                 System.out.println("Errore nella conversione dell'oggetto della classe
                    specificata!");
1715                 e.printStackTrace();
1716             }
1717 
1718             return rankings;
1719         }
1720 
1721         @Override
1722         protected void done() {
1723 
1724             Multicast_rankings rankings = null;
1725             try {
1726                 rankings = get();
1727             } catch (InterruptedException | ExecutionException e) {
1728                 // Errors in reading rankings
1729                 lblRanking.setForeground(Color.gray);
1730                 lblErrors.setText("Errore nella ricezione della classifica!");
1731                 butSurrend.setText("    Exit    ");
1732                 pack();
1733             }
1734             if (rankings != null) {
1735 
1736                 // Adds the elements to the list after the comunication with server
1737                 System.out.println("Lettura classifica avvenuta con successo");
1738                 lblRanking.setForeground(Color.blue);
1739                 lblErrors.setText("Classifica arrivata!");
1740                 lblErrors.setForeground(Color.BLUE);
1741                 butSurrend.setText("    Exit    ");
1742 
1743                 pack();
1744 
1745                 String[] elements = rankings.getStringElements();
1746                 String userPosition = "-- :";
1747                 listLRanking.clear();
1748                 int pos = 1;
1749                 for (String el: elements) {
1750                     listLRanking.addElement(pos + "°: " + el);
1751                     if (el.split(" ")[0].equals(username)) userPosition = pos + "°!";
1752                     pos++;
1753                 }
1754                 sendAllertMessage(
1755                         "You arrived " + userPosition,
1756                         "Congratulations!",
1757                         JOptionPane.INFORMATION_MESSAGE);
1758 
1759             }
1760         }
1761     }
1762 }
1763 
1764 
1765 
1766 
1767 
1768 
1769 
1770 
```

COMMON PACKAGE

```
package commonPackage;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class Multicast_rankings implements Serializable{

    private static final long serialVersionUID = -3754637793233493207L;

    private ArrayList<RankingItem> ranks;

    public Multicast_rankings(ArrayList<RankingItem> ranks) {
        this.ranks = ranks;
    }

    public void orderRanks() {
        // Sorting
        Collections.sort(ranks, new Comparator<RankingItem>() {
                @Override
                public int compare(RankingItem user1, RankingItem user2)
                {
                    if (user1.points < user2.points) return 1;
                    else if (user1.points > user2.points) return -1;
                    else return 0;
                }
            });
    }

    public String[] getStringElements() {
        String[] els = new String[ranks.size()];

        for (int i = 0; i < ranks.size(); i++) els[i] = ranks.get(i).toString();

        return els;
    }

}




package commonPackage;

import java.io.Serializable;

public class RankingItem implements Serializable{

    private static final long serialVersionUID = 3873024188698404430L;

    public String user;
    public int points;

    public RankingItem(String user, int points) {
        this.user = user;
        this.points = points;
    }

```

```
1840     public String toString() {
1841         return user + " - " + points + " Points";
1842     }
1843 
1844 }
1845 
1846 
1847 
1848 
1849 package commonPackage;
1850 
1851 import java.rmi.Remote;
1852 import java.rmi.RemoteException;
1853 
1854 public interface RMI_client_interface extends Remote {
1855 
1856     public final static String OBJECT_NAME="RMI_client";
1857 
1858     /*
1859      * If some other Client creates a game, the server invokes this
1860      * method to warn all the invited Clients.
1861      */
1862     public void gameCall(String creator, int gameID, int multicastPort) throws RemoteException;
1863 
1864     /*
1865      * If a Client accept a game, and for some reasong the Server
1866      * cancell it, it invokes this methot to warn the Client
1867      */
1868     public void gameCancelled(String creator, int gameID) throws RemoteException;
1869 
1870     /*
1871      * This methos have the only meaning to let the Server know if the
1872      * Client is still online. If the Client fail to respond
1873      * (a RemoteException is thrown), the Server sets the client offline.
1874      */
1875     public boolean isOnline() throws RemoteException;
1876 
1877 }
1878 
1879 
1880 
1881 
1882 package commonPackage;
1883 
1884 import java.rmi.Remote;
1885 import java.rmi.RemoteException;
1886 import java.util.ArrayList;
1887 
1888 public interface RMI_server_interface extends Remote {
1889 
1890 
1891     public final static String OBJECT_NAME="RMI_server";
1892 
1893 
1894 
1895     /*
1896      * Return Encoding:
1897      * true  : registration successful
1898      * false : registration failed
1899      */
1900     public boolean register(String name, String password) throws RemoteException;
1901 
1902 
1903 
1904     /*
1905      * Return Encoding:
1906      * >0  : login successfull
1907      * -1 : account non existent
1908      * -2 : wrong password
```

```
     * -3 : already logged int
     *
     * The number returned from the login call is the session ID necessary
     * for future comunication with Server. If the Client needs to send
     * any type of message to the server, it must use this number,
     * so the Server can assure it's the right Client doing the operation.
     */
    public int login(String name, String password, RMI_client_interface clientCallback) throws
    RemoteException;



    /*
     * Return Encoding:
     * 0  : logout successfull
     * -1 : account non existent
     * -2 : wrong password
     *
     * the logout needs the password to prevent anyone to logout a generic account.
     */
    public int logout(String name, String password) throws RemoteException;



    /*
     * Returns a String[] containing all the users online in that moment
     */
    public ArrayList<String> requestOnlineUsers() throws RemoteException;

}




package commonPackage;

import java.io.Serializable;

public class UDP_words implements Serializable{

    private static final long serialVersionUID = 6317533870617383938L;

    public String[] words;
    public int gameID;
    public String player;
    public int sessionID;

    public UDP_words(String[] words, int gameID, String player, int sessionID) {

        this.words = words;
        this.gameID = gameID;
        this.player = player;
        this.sessionID = sessionID;
    }

}
```

SERVER

```
1977
1978
1979
1980
1981
1982
1983
1984
1985 package serverPackage;
1986 import org.w3c.dom.*;
1987 import org.xml.sax.SAXException;
1988
1989 import commonPackage.RMI_server_interface;
1990
1991 import javax.xml.parsers.*;
1992 import java.io.*;
1993 import java.net.DatagramSocket;
1994 import java.net.InetAddress;
1995 import java.net.ServerSocket;
1996 import java.net.Socket;
1997 import java.net.SocketException;
1998 import java.net.UnknownHostException;
1999 import java.rmi.RemoteException;
2000 import java.rmi.registry.LocateRegistry;
2001 import java.rmi.registry.Registry;
2002 import java.rmi.server.UnicastRemoteObject;
2003 import java.util.concurrent.Executors;
2004 import java.util.concurrent.ThreadPoolExecutor;
2005
2006 public class ServerMain {
2007
2008     public static void main(String[] args) {
2009
2010         String configFileName = "ServerConfiguration.xml";
2011         if (args.length == 1) configFileName = args[0];
2012         else if (args.length > 1) {
2013             System.out.println("Devi inserire il nome del file di configurazione");
2014         }
2015
2016         INIT_SERVER(configFileName);
2017
2018     }
2019
2020
2021     @SuppressWarnings("resource")
2022     private static void INIT_SERVER(String configFileName) {
2023
2024         Document doc = null;
2025
2026         try {
2027
2028             File inputFile = new File(configFileName);
2029             DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
2030             DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
2031             doc = dBuilder.parse(inputFile);
2032             doc.getDocumentElement().normalize();
2033
2034         } catch (IOException e2) {
2035             System.out.println("Errore nell'apertura del file di configurazione. Impossibile
                continuare");
2036             e2.printStackTrace();
2037             System.exit(-1);
2038         } catch (ParserConfigurationException e2) {
2039             System.out.println("Errore nella conversione del file XML. Impossibile continuare");
2040             e2.printStackTrace();
2041             System.exit(-1);
2042         } catch (SAXException e2) {
2043             System.out.println("Errore nella lettura del file XML. Impossibile continuare");
2044             e2.printStackTrace();
```

```
2045            System.exit(-1);
2046        }
2047
2048        int serverPort = 0;
2049        String serverName = null;
2050        int registryPort = 0;
2051        String wordsFileName = null;
2052        int startingMulticastPort = 0;
2053        int maxMulticastPort = 0;
2054        int multicastPort = 0;
2055        String multicastAddress = null;
2056        int udpPort = 0;
2057        DatagramSocket dataSocketUDP = null;
2058
2059        String stateFileName = null;
2060
2061        // Reading necessary items from configuration file
2062        try {
2063
2064            NodeList els = doc.getElementsByTagName("property");
2065
2066            serverPort = Integer.parseInt(els.item(0).getFirstChild().getTextContent());
2067            serverName = els.item(1).getFirstChild().getTextContent();
2068            registryPort = Integer.parseInt(els.item(2).getFirstChild().getTextContent());
2069            wordsFileName = els.item(3).getFirstChild().getTextContent();
2070            startingMulticastPort =
                Integer.parseInt(els.item(4).getFirstChild().getTextContent());
2071            maxMulticastPort = Integer.parseInt(els.item(5).getFirstChild().getTextContent());
2072            multicastAddress = els.item(6).getFirstChild().getTextContent();
2073            udpPort = Integer.parseInt(els.item(7).getFirstChild().getTextContent());
2074            stateFileName = els.item(8).getFirstChild().getTextContent();
2075
2076        } catch (NumberFormatException e) {
2077            System.out.println("Errore nel file di configurazione. Impossibile continuare.");
2078            System.exit(-1);
2079        }
2080
2081        multicastPort = startingMulticastPort;
2082
2083        UserContainer users = new UserContainer();
2084
2085        // trying to open and read the old state of the server
2086        ServerState serverState = new ServerState(stateFileName, users);
2087        serverState.recover();
2088
2089        // Initializing data
2090        RMI_server server = new RMI_server(serverName, serverPort, users, serverState);
2091
2092        File words = new File(wordsFileName);
2093        Game.createWordList(words);
2094
2095
2096        // Searching for multicast address
2097        InetAddress multAddress = null;
2098        try { multAddress = InetAddress.getByName(multicastAddress); }
2099        catch (UnknownHostException e2) {
2100            System.out.println("Impossibile determinare l'indirizzo dell'host " +
                multicastAddress);
2101        }
2102
2103        if (!multAddress.isMulticastAddress()) {
2104            System.out.println("Questo indirizzo non è Multicast.. chiusura forzata.");
2105            System.exit(-1);
2106        }
2107
2108
2109        // Creating datagram socket for multicast
2110        DatagramSocket dataSocketMUL = null;
2111        try {
```

```
2112            dataSocketMUL = new DatagramSocket();
2113        } catch (SocketException e2) {
2114            System.out.println("Errors in the opening of a datagram socket.");
2115            e2.printStackTrace();
2116        }
2117
2118        // Exporting RMI registry
2119        try{
2120
2121            RMI_server_interface serverRMI =
2122                    (RMI_server_interface) UnicastRemoteObject.exportObject(server, 0);
2123            Registry registry = LocateRegistry.createRegistry(registryPort);
2124            registry.rebind(RMI_server_interface.OBJECT_NAME, serverRMI);
2125
2126            System.out.println("Server RMI pronto.");
2127
2128        } catch(RemoteException e){
2129            System.out.println("Server error:" + e.getMessage());
2130            System.exit(-1);
2131        }
2132
2133        ThreadPoolExecutor thPool = (ThreadPoolExecutor) Executors.newFixedThreadPool(100);
2134
2135
2136
2137
2138        // TASK ESECUZIONE RICHIESTE UDP
2139        try {
2140            dataSocketUDP = new DatagramSocket(udpPort);
2141        } catch (SocketException e) {
2142            System.out.println("Impossibile aprire un server UDP, chiusura forzata...");
2143            System.exit(-1);
2144        }
2145        thPool.execute(new UDP_task(dataSocketUDP, users));
2146
2147
2148
2149
2150
2151        // TASK ESECUZIONE RICHIESTE TCP
2152        ServerSocket serverSocket = null;
2153
2154        try {
2155            serverSocket = new ServerSocket(server.getPort());
2156        } catch (IOException e1) {
2157            e1.printStackTrace();
2158            System.out.println("Impossibile aprire un server Socket, chiusura forzata...");
2159            System.exit(-1);
2160
2161        }
2162        System.out.println("Server TCP pronto.");
2163
2164        while (true) {
2165            Socket socket;
2166
2167            try {
2168                socket = serverSocket.accept();
2169                System.out.println("Client rilevato, thread attivato");
2170                Multicast_task multicast =
2171                        new Multicast_task(dataSocketMUL, multAddress, multicastPort);
2172                startingMulticastPort++;
2173                thPool.execute(new ServerTask(socket, users, multicast, serverState));
2174                if (multicastPort > maxMulticastPort) multicastPort = startingMulticastPort;
2175
2176            } catch (IOException e) {
2177                e.printStackTrace();
2178            }
2179        }
2180    }
```

```
}
```

```
package serverPackage;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.rmi.RemoteException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Random;
import java.util.concurrent.LinkedBlockingQueue;


import commonPackage.Multicast_rankings;
import serverPackage.User.userState;

public class ServerTask implements Runnable {

    private Socket socket;
    private UserContainer users;
    private Multicast_task multicastTask;
    private ServerState serverState;

    ServerTask(Socket socket, UserContainer users, Multicast_task multicast, ServerState
    serverState) {
        this.socket = socket;
        this.users = users;
        this.multicastTask = multicast;
        this.serverState = serverState;
    }


    @Override
    public void run() {
        String playerName = null;
        String creator = null;
        ObjectOutputStream writer = null;
        ObjectInputStream reader = null;

        try {

            System.out.println("Thread del server esegue un task di un client");
            writer = new ObjectOutputStream (socket.getOutputStream());
            reader = new ObjectInputStream (socket.getInputStream());

            int actionType = reader.readInt();

            Game game = null;

            /*
             * 0 for answer to: a game request (creating a game):
             *   in a game request, it must first send the total number of invited players,
             *   and after that his name, and all the names of the other players.
             *
             *
             * 1 for answer to: a game invite:
             *   after a game request, other players choose to accept or refuse the invite;
             *   if they refuse, the server must warn all the other players.
             *
             *
```

```
2249              * 2 for answer to: a ranking list request
2250              *
2251              */
2252             switch(actionType) {
2253
2254             case 0:
2255
2256                 createGame(writer, reader, creator, game);
2257
2258                 break;
2259
2260             case 1:
2261
2262                 acceptGame(writer, reader, playerName, game);
2263
2264                 break;
2265
2266             case 2:
2267
2268                 sendRanking(writer, reader);
2269
2270                 break;
2271
2272             default:
2273                 closeTCP(writer, reader, socket);
2274
2275             }
2276         } catch (IOException e) {
2277
2278             System.out.println("Errore di comunicazione col client. Chiusura forzata.");
2279
2280         } finally {
2281
2282             try {
2283                 if (writer != null) writer.close();
2284                 if (reader != null) reader.close();
2285             } catch (IOException e) {
2286             }
2287         }
2288     }
2289
2290     private void createGame(
2291             ObjectOutputStream writer,
2292             ObjectInputStream reader,
2293             String creator,
2294             Game game) {
2295
2296         try {
2297             // reads the number of player other than the creator
2298             int playerNumber = reader.readInt();
2299             ArrayList<String> players = new ArrayList<String>();
2300
2301             // reads the name of the creator of the game
2302             creator = reader.readUTF();
2303             players.add(creator);
2304
2305             int sessionID = users.getUser(creator).getSessionID();
2306             if (sessionID != reader.readInt() || sessionID == 0) {
2307                 // User failed to autenticate
2308                 closeTCP(writer, reader, socket);
2309                 return;
2310             }
2311
2312             for (int i = 0; i < playerNumber; i++) {
2313                 players.add(reader.readUTF());
2314             }
2315
2316             game = users.addGame(players, creator);
2317             if (game == null) {
```

```
2318                // somethings bad happened: some user doesn't exist, or isn't online.
2319                writer.writeBoolean(false);
2320                writer.flush();
2321                closeTCP(writer, reader, socket);
2322                return;
2323            }
2324
2325            writer.writeBoolean(true);
2326            writer.writeInt(game.gameID);
2327            writer.writeInt(multicastTask.getMulticastPort());
2328            writer.flush();
2329
2330            // if it arrives at this point, every user requested exist, and is online.
2331            // theyr data is in the game object, wich is saved inside the userContainer users.
2332            for (User u: game.getGameUsers()) {
2333                try {
2334                    System.out.println("Invio chiamate ai giocatori");
2335                    u.getCallback().gameCall(
2336                            creator,
2337                            game.gameID,
2338                            this.multicastTask.getMulticastPort());
2339                } catch (RemoteException e) {
2340                    System.out.println(
2341                            "Errore: alcuni utenti non possono ricevere una richiesta di
                                partita");
2342                }
2343            }
2344
2345            // Closing TCP connection
2346            closeTCP(writer, reader, socket);
2347
2348            boolean readyGame = false;
2349            // sleeps until the game is ready (everyone accepted) or 7 minutes passed
2350            for (int i = 0; i < 7 * 60; i++) {
2351                try {
2352                    Thread.sleep(1000);
2353                } catch (InterruptedException e) {
2354                    System.out.println("Attesa interrotta inaspettatamente");
2355                    e.printStackTrace();
2356                }
2357                if (game.readyGame()) {
2358                    readyGame = true;
2359                    break;
2360                }
2361                if (game.isClosing()) {
2362                    break;
2363                }
2364            }
2365
2366            try {
2367                if (readyGame) { play(game); }
2368            } catch (InterruptedException e) {
2369                System.out.println("Attesa interrotta inaspettatamente");
2370                e.printStackTrace();
2371            }
2372            // at this point everyone sent his possible permutation or the timeout has been
                reached,
2373            // so the server calculate the scores and close the game
2374            if (game.calculateRanking() == false) {
2375                System.out.println("Fatal Error: Cannot calculate ranking.");
2376                game.close();
2377                return;
2378            }
2379
2380            int[] points = game.getRanking();
2381            LinkedBlockingQueue<User> userQueue = game.getGameUsers();
2382
2383            // Updates ranking and server state file
2384            int c = 0;
```

```
2385            for (User u: userQueue) {
2386                u.addPoints(points[c]);
2387                serverState.updateUser(u);
2388                c++;
2389            }
2390
2391            // sends result to all players with multicast
2392            multicastTask.sendMulticastRanking(points, userQueue);
2393
2394            // delete game
2395            users.removeGame(game);
2396
2397        } catch (IOException e1) {
2398            if (creator != null) users.getUser(creator).setState(userState.OFFLINE);
2399            System.out.println("Errore di comunicazione col server.");
2400        }
2401
2402    }
2403
2404    private void acceptGame(
2405            ObjectOutputStream writer,
2406            ObjectInputStream reader,
2407            String playerName,
2408            Game game) {
2409
2410        try {
2411            // reads the name of the accepting / refusing player, and the gameID
2412            playerName = reader.readUTF();
2413
2414            int sessionID = users.getUser(playerName).getSessionID();
2415            if (reader.readInt() != sessionID || sessionID == 0) {
2416                // User failed to autenticate.
2417                game.close();
2418                closeTCP(writer, reader, socket);
2419                return;
2420            }
2421
2422            int gameID = reader.readInt();
2423
2424            // reads the answer: true for accept, false for refuse
2425            boolean answer = reader.readBoolean();
2426
2427            // reads all the gameID of the games he must refuse to accept this one
2428            int refNumber = reader.readInt();
2429
2430            // closes this games
2431            for (int i = 0; i < refNumber; i++) users.getGameByID(reader.readInt()).close();
2432
2433
2434            game = users.getGameByID(gameID);
2435            if (game == null) {
2436                // if game is null, it doesnt exist (it could be already closed)
2437                // sends an error to the player and breaks (error coded with -1)
2438
2439                writer.writeInt(-1);
2440                writer.flush();
2441                closeTCP(writer, reader, socket);
2442                return;
2443            }
2444
2445            User player = users.getUser(playerName);
2446            if (player == null) {
2447                // this user has invalid name, or went offline.
2448                // reports an error with TCP connection and close.
2449                // (error coded with -2)
2450
2451                writer.writeInt(-2);
2452                writer.flush();
2453                game.close();
```

```
2454                closeTCP(writer, reader, socket);
2455                return;
2456            }
2457
2458            if (!answer) {
2459                // if the player refuses, the server must close the game and warn all the player
                    invited.
2460                // here it closes the game, so every other thread can detect the incongruence and
                    warn the players.
2461                // (coded with -3)
2462
2463                writer.writeInt(-3);
2464                writer.flush();
2465                game.close();
2466                closeTCP(writer, reader, socket);
2467                return;
2468            }
2469
2470            // evetything went fine
2471            writer.writeInt(0);
2472            writer.flush();
2473
2474
2475            if (!game.acceptPlayer(player)) {
2476                // this user already has accepted this game
2477                System.out.println("Inconsistenza nell'accettazione della partita");
2478            }
2479
2480            /*
2481             * since it has accepted, this thread must wait with the connection open;
2482             * it will wait until 7 total minute passed (the main thread wich created the
2483             * game is aware of it and will close the game in the eventuality) or until
2484             * some other thread close it since some player had refused the invite.
2485             */
2486            socket.setSoTimeout(1000);
2487
2488            for (int i = 0; i < 60 * 7; i++) {
2489
2490                try {
2491                    // useless read just to wait the right time and
2492                    // to see if the client is still connected
2493                    reader.readObject();
2494
2495                } catch (SocketTimeoutException e) {
2496                    if (game.isClosing()) {
2497
2498                        // game closed for timeout or for a refused invite
2499                        // the client waiting for game words see the connection
2500                        // get closed and conclude the game has been closed.
2501                        System.out.println("closing game " + game.gameID);
2502                        closeTCP(writer, reader, socket);
2503                        return;
2504                    }
2505
2506                    if (game.isStarted()) break;
2507
2508                } catch (IOException e) {
2509
2510                    System.out.println("Persa la connessione col client, chisura della partita");
2511                    game.close();
2512                    closeTCP(writer, reader, socket);
2513                    return;
2514
2515                } catch (ClassNotFoundException e) {
2516                }
2517            }
2518
2519            if (!socket.isClosed() && game.isStarted()) {
2520
```

```
2521                writer.writeUTF(game.getGameWord());
2522                writer.flush();
2523            }
2524
2525        } catch (IOException e) {
2526
2527            if (playerName != null) users.getUser(playerName).setState(userState.OFFLINE);
2528        }
2529    }
2530
2531    private void sendRanking(
2532            ObjectOutputStream writer,
2533            ObjectInputStream reader) {
2534
2535        // Reuse of MUL_rank class
2536        Multicast_rankings rankings = users.getGlobalRanking();
2537
2538        try {
2539            writer.writeObject(rankings);
2540            writer.flush();
2541        } catch (IOException e) {
2542            System.out.println("Impossibile inviare la classifica");
2543        }
2544
2545        closeTCP(writer, reader, socket);
2546
2547    }
2548
2549    private void closeTCP(ObjectOutputStream o, ObjectInputStream i, Socket s) {
2550
2551        try  {
2552            o.close();
2553            i.close();
2554            s.close();
2555
2556        } catch (IOException e ) {
2557            System.out.println("Impossibile chiudere la connessione o il socket." );
2558        }
2559    }
2560
2561    private void play(Game game) throws IOException, InterruptedException{
2562
2563        HashSet<String> wordsSet = Game.words;
2564
2565        if (wordsSet == null) {
2566            System.out.println("Fatal Error: non existent words file. Cannot choose the words for
                the game");
2567            game.close();
2568            return;
2569        }
2570
2571        Random random = new Random();
2572        String word;
2573
2574        // find a String in the file wich is at least 6 character long
2575        while (true) {
2576            word = "";
2577            int pos = random.nextInt(wordsSet.size());
2578            for (String el: wordsSet) {
2579                if (pos == 0) {
2580                    word = el;
2581                    break;
2582                }
2583                pos--;
2584            }
2585            if (word.length() > 6) break;
2586        }
2587
2588        // shuffle the string
```

```
        ArrayList<String> s = new ArrayList<String>();
        for (int j = 0; j < word.length(); j++) {s.add(word.substring(j, j+1));}
        java.util.Collections.shuffle(s);
        String shuffledWord = "";
        for (String st: s) { shuffledWord += st;}

        game.setGameWord(shuffledWord);
        game.start();

        for (int j = 0; j < 60 * 5; j++) {
            Thread.sleep(1000);
            if (game.everyOnePlayed()) break;
        }

    }
}




package serverPackage;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.net.DatagramPacket;
import java.net.DatagramSocket;

import commonPackage.UDP_words;

public class UDP_task implements Runnable{

    private DatagramSocket udpDataSocket;
    private UserContainer users;

    UDP_task(DatagramSocket udpDataSocket, UserContainer users) {
        this.udpDataSocket = udpDataSocket;
        this.users = users;
    }

    @Override
    public void run() {

        DatagramPacket dataPacketIn = null;
        byte[] incData = new byte[1000000];

        System.out.println("Server UDP pronto.");

        while (true) {

            try {

                dataPacketIn = new DatagramPacket(incData, incData.length);
                udpDataSocket.receive(dataPacketIn);

                ByteArrayInputStream in = new ByteArrayInputStream(dataPacketIn.getData());
                ObjectInputStream is = new ObjectInputStream(in);

                UDP_words words = (UDP_words) is.readObject();

                if (words.sessionID != users.getUser(words.player).getSessionID()) {
                    // Errore nell'identificazione del giocatore
                    System.out.println("Utente entraneo ha provato ad identificarsi");
                    continue;
                }
                Game game = users.getGameByID(words.gameID);
                if (!game.sendWords(words.words, users.getUser(words.player)))
                    System.out.println("Errore: impossibile aggiungere le parole del
```

```
                    giocatore");;
2658
2659
2660            } catch (IOException e) {
2661                System.out.println("Errore di comunicazione");
2662                e.printStackTrace();
2663            } catch (ClassNotFoundException e) {
2664                System.out.println("Errore nella lettura della classe delle parole");
2665                e.printStackTrace();
2666            }
2667        }
2668    }
2669 }
2670
2671
2672
2673
2674
2675 package serverPackage;
2676
2677 import java.net.InetAddress;
2678 import java.net.UnknownHostException;
2679 import java.rmi.RemoteException;
2680 import java.rmi.server.RemoteObject;
2681 import java.util.ArrayList;
2682
2683 import commonPackage.RMI_client_interface;
2684 import commonPackage.RMI_server_interface;
2685 import serverPackage.User.userState;
2686
2687 public class RMI_server extends RemoteObject implements RMI_server_interface{
2688
2689     private static final long serialVersionUID = 7321894133525981176L;
2690
2691     private UserContainer users;
2692     private String serverName;
2693     private int port;
2694     private InetAddress inetServer;
2695     private ServerState stateServer;
2696
2697     RMI_server(String serverName, int port, UserContainer users, ServerState stateServer) {
2698         this.serverName = serverName;
2699         this.port = port;
2700         try {
2701             this.inetServer = InetAddress.getByName(serverName);
2702         } catch (UnknownHostException e) {
2703
2704             e.printStackTrace();
2705         }
2706
2707         this.users = users;
2708         this.stateServer = stateServer;
2709     }
2710
2711     public int getPort() { return this.port; }
2712
2713     public String getName() { return this.serverName; }
2714
2715     public InetAddress getAddress() { return this.inetServer; }
2716
2717     public UserContainer getUser() { return this.users; }
2718
2719
2720     @Override
2721     public boolean register(String name, String password) {
2722
2723         User user = new User(name, password);
2724         if (users.addUser(user)) {
2725             stateServer.addUser(user);
```

```
            return true;
        }
        return false;
    }

    @Override
    public int login(String name, String password, RMI_client_interface clientCallback) {

        User user = users.getUser(name);
        if (user == null) return -2;
        if (!user.verifyPassword(password)) return -1;
        if (user.getState().equals(userState.ONLINE)) {
            // verifies it is still online

            try {
                user.getCallback().isOnline();

                // unreacheable if the user is not online
                return -3;
            } catch (RemoteException e) {
                // in this case the user wasn't online, so it can login again.
            }
        }

        user.setState(userState.ONLINE);
        user.setCallback(clientCallback);

        return user.getSessionID();
    }

    @Override
    public int logout(String name, String password) {

        User user = users.getUser(name);
        if (user == null) return -1;
        if (!user.verifyPassword(password)) return -2;

        System.out.println("User " + user.getName() + " settato offline");
        user.setState(userState.OFFLINE);
        user.resetCallback();

        return 0;
    }

    @Override
    public ArrayList<String> requestOnlineUsers() throws RemoteException {

        return users.getOnlineUsers();
    }



}



package serverPackage;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.util.ArrayList;
import java.util.concurrent.LinkedBlockingQueue;

import commonPackage.Multicast_rankings;
```

```
import commonPackage.RankingItem;

public class Multicast_task {


    DatagramSocket dataSocket;
    InetAddress multicastAddress;
    int multicastPort;

    Multicast_task(DatagramSocket dataSocket, InetAddress multicastAddress, int multicastPort) {
        this.multicastAddress = multicastAddress;
        this.dataSocket = dataSocket;
        this.multicastPort = multicastPort;
    }


    public void sendMulticastRanking(int[] userPoints, LinkedBlockingQueue<User> players) {

        int cont = 0;
        byte[] bytes = null;
        Multicast_rankings rankings;
        ArrayList<RankingItem> ranks = new ArrayList<RankingItem>();
        try {

            // Prepares to send a "Multicast_rankings" object to multicast channel
            for (User u: players) {
                RankingItem usRank = new RankingItem(u.getName(), userPoints[cont]);
                ranks.add(usRank);
                cont++;
            }
            rankings = new Multicast_rankings(ranks);
            rankings.orderRanks();

            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            ObjectOutputStream daos = new ObjectOutputStream(baos);
            daos.writeObject(rankings);
            daos.close();
            bytes = baos.toByteArray();

        } catch (IOException e) {
            System.out.println("Impossibile preparare il buffer per l'invio dei dati in
            multicast..." );
            return;
        }

        if (bytes == null) {
            System.out.println("Errore nella creazione del buffer");
            return;
        }

        DatagramPacket dataPacket = new DatagramPacket(bytes, bytes.length, multicastAddress,
        multicastPort);

        System.out.println("Invio dati sul Multicast");
        try {
            this.dataSocket.send(dataPacket);
        } catch (IOException e) {
            System.out.println("Impossibile inviare il messaggio multicast contenente la
            classifica a tutti gli utenti");
            e.printStackTrace();
        }


    }

    public int getMulticastPort() { return this.multicastPort; }

}
```

```
2861
2862
2863
2864
2865 package serverPackage;
2866
2867 import java.io.File;
2868 import java.io.IOException;
2869
2870 import javax.xml.bind.DatatypeConverter;
2871 import javax.xml.parsers.DocumentBuilder;
2872 import javax.xml.parsers.DocumentBuilderFactory;
2873 import javax.xml.parsers.ParserConfigurationException;
2874 import javax.xml.transform.Transformer;
2875 import javax.xml.transform.TransformerException;
2876 import javax.xml.transform.TransformerFactory;
2877 import javax.xml.transform.dom.DOMSource;
2878 import javax.xml.transform.stream.StreamResult;
2879
2880 import org.w3c.dom.DOMException;
2881 import org.w3c.dom.Document;
2882 import org.w3c.dom.Element;
2883 import org.w3c.dom.Node;
2884 import org.w3c.dom.NodeList;
2885 import org.xml.sax.SAXException;
2886
2887 public class ServerState {
2888
2889     String ID = "id";
2890     String stateFileName;
2891     UserContainer users;
2892     Document doc;
2893
2894     ServerState (String stateFileName, UserContainer users) {
2895         this.stateFileName = stateFileName;
2896         this.users = users;
2897
2898         boolean success = false;
2899         // Tries to open the file, otherwise it creates it
2900         try {
2901             if (stateFileName == null || stateFileName.equals("")) {
2902                 stateFileName = new String("ServerState.xml");
2903             }
2904
2905             File stateFile = new File(stateFileName);
2906             DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
2907             DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
2908             doc = dBuilder.parse(stateFile);
2909             doc.getDocumentElement().normalize();
2910
2911             success = true;
2912
2913         } catch (IOException e) {
2914             System.out.println("Errore nell'apertura del file di stato;");
2915         } catch (ParserConfigurationException e) {
2916             System.out.println("Errore nella conversione del file XML;");
2917         } catch (SAXException e) {
2918             System.out.println("Errore nella lettura del file XML;");
2919         }
2920
2921         if (!success) {
2922             System.out.println("Creazione nuovo file");
2923             createNewFile();
2924         }
2925     }
2926
2927
2928     public void recover() {
2929
```

```
2930         try {
2931             NodeList els = doc.getElementsByTagName("User");
2932 
2933             for (int i = 0; i < els.getLength(); i++) {
2934                 Node node = els.item(i);
2935 
2936                 if (node.getNodeType() == Node.ELEMENT_NODE) {
2937 
2938                     Element el = (Element) node;
2939 
2940                     // Decode the password and the salt, wich were Coded
2941                     // to be written in the xml file
2942 
2943                     String username = el.getAttribute(ID);
2944                     byte[] codedPassword =
2945                             DatatypeConverter.parseHexBinary(
2946                             el.getElementsByTagName("Password")
2947                             .item(0)
2948                             .getTextContent());
2949 
2950                     byte[] salt =
2951                             DatatypeConverter.parseHexBinary(
2952                             el.getElementsByTagName("Salt")
2953                             .item(0)
2954                             .getTextContent());
2955 
2956 
2957                     int points =
2958                             Integer.parseInt(
2959                                     el.getElementsByTagName("Points").item(0).getTextContent()
2960                                     );
2961 
2962                     if (!users.addUser(new User(username, codedPassword, salt, points)))
2963                         System.out.println("User già esistente");;
2964                 }
2965 
2966             }
2967         } catch (NumberFormatException | DOMException e) {
2968 
2969             System.out.println("Errore nella lettura del file; creazione nuovo file");
2970             createNewFile();
2971         }
2972     }
2973 
2974     private boolean createNewFile() {
2975 
2976         try {
2977 
2978             DocumentBuilderFactory docFactory = DocumentBuilderFactory.newInstance();
2979             DocumentBuilder docBuilder;
2980             docBuilder = docFactory.newDocumentBuilder();
2981 
2982             // root elements
2983             doc = docBuilder.newDocument();
2984 
2985             Element rootElement = doc.createElement("class");
2986             doc.appendChild(rootElement);
2987 
2988             saveFile();
2989 
2990 
2991         } catch (ParserConfigurationException e) {
2992             e.printStackTrace();
2993             return false;
2994         } catch (TransformerException e) {
2995             System.out.println("Impossibile salvere il file");
2996             e.printStackTrace();
2997         }
2998 
```

```
2999         return true;
3000     }
3001 
3002     public void addUser(User user) {
3003 
3004         Element userEl = doc.createElement("User");
3005         doc.getFirstChild().appendChild(userEl);
3006 
3007         userEl.setAttribute("id", user.getName());
3008 
3009         // Encode password and salt, because they can't be stored
3010         // "as is" in an xml file. They'll be Decoded when read
3011 
3012         Element psw = doc.createElement("Password");
3013         String pswCoded = DatatypeConverter.printHexBinary(user.getCodedPassword());
3014         psw.appendChild(doc.createTextNode(pswCoded));
3015         userEl.appendChild(psw);
3016 
3017         Element salt = doc.createElement("Salt");
3018         String saltCoded = DatatypeConverter.printHexBinary(user.getSalt());
3019         salt.appendChild(doc.createTextNode(saltCoded));
3020         userEl.appendChild(salt);
3021 
3022         Element points = doc.createElement("Points");
3023         points.appendChild(doc.createTextNode("0"));
3024         userEl.appendChild(points);
3025         doc.getDocumentElement().normalize();
3026         try {
3027             saveFile();
3028         } catch (TransformerException e) {
3029             System.out.println("Impossibile salvare il file");
3030             e.printStackTrace();
3031         }
3032     }
3033 
3034     public boolean updateUser(User user) {
3035 
3036         NodeList els = doc.getElementsByTagName("User");
3037 
3038         for (int i = 0; i < els.getLength(); i++) {
3039             Node node = els.item(i);
3040 
3041             if (node.getNodeType() == Node.ELEMENT_NODE) {
3042 
3043                 Element el = (Element) node;
3044 
3045                 if (el.getAttribute("id").equals(user.getName())) {
3046                     el.getElementsByTagName("Points")
3047                     .item(0)
3048                     .setTextContent(
3049                             user.getPoints() + "");
3050                     try {
3051                         saveFile();
3052                     } catch (TransformerException e) {
3053                         System.out.println("Impossibile salvere il file");
3054                         e.printStackTrace();
3055                     }
3056                     return true;
3057                 }
3058             }
3059         }
3060 
3061         return false;
3062     }
3063 
3064     private void saveFile() throws TransformerException{
3065 
3066 
3067         // write the content into xml file
```

```
3068         TransformerFactory transformerFactory = TransformerFactory.newInstance();
3069         Transformer transformer = transformerFactory.newTransformer();
3070         DOMSource source = new DOMSource(doc);
3071 
3072         StreamResult result = new StreamResult(new File(stateFileName));
3073 
3074         transformer.transform(source, result);
3075 
3076         System.out.println("File saved!");
3077 
3078     }
3079 }
3080 
3081 
3082 
3083 
3084 
3085 package serverPackage;
3086 
3087 import java.io.BufferedReader;
3088 import java.io.File;
3089 import java.io.FileReader;
3090 import java.io.IOException;
3091 import java.util.ArrayList;
3092 import java.util.HashSet;
3093 import java.util.Iterator;
3094 import java.util.concurrent.LinkedBlockingQueue;
3095 import java.util.concurrent.locks.ReentrantLock;
3096 
3097 public class Game {
3098 
3099     public String creator;
3100     public int gameID;
3101     public static HashSet<String> words;
3102 
3103     private static int IDGenerator = 0;
3104     private LinkedBlockingQueue<User> gameRequestUsers;
3105     private LinkedBlockingQueue<Boolean> accepted;
3106     private String[][] playerWords;
3107     private boolean[] playerHasWords;
3108     private int[] rankingPoints;
3109     private int missingPlayers;
3110     private boolean closing = false;
3111     private boolean started = false;
3112 
3113     // it is assured it can't reach a deadlock because are almost
3114     // always not called in a nested lock
3115     private ReentrantLock closeLock = new ReentrantLock();
3116     private ReentrantLock startLock = new ReentrantLock();
3117     private ReentrantLock missingPlayerLock = new ReentrantLock();
3118 
3119     private String gameWord;
3120 
3121     public Game(LinkedBlockingQueue<User> gameRequestUsers, String creator) {
3122 
3123         this.gameRequestUsers = gameRequestUsers;
3124         this.creator = creator;
3125         accepted = new LinkedBlockingQueue<Boolean>();
3126         int dim = gameRequestUsers.size();
3127         playerWords = new String[dim][];
3128         playerHasWords = new boolean[dim];
3129         rankingPoints = new int[dim];
3130 
3131         for (int i = 0; i < dim; i++) {playerHasWords[i] = false; rankingPoints[i] = 0;}
3132 
3133         Iterator<User> it = gameRequestUsers.iterator();
3134         while (it.hasNext()) {
3135             accepted.add(false);
3136             it.next();
```

```
3137         }
3138
3139         missingPlayers = dim;
3140         gameID = IDGenerator;
3141         IDGenerator++;
3142     }
3143
3144     /*
3145      * Adds a player wich accepted the game. If the missingPlayer
3146      * reach 0 it means that everyone accepted the game. The same
3147      * player can't accept the same game twice
3148      *
3149      */
3150     public synchronized boolean acceptPlayer(User user) {
3151
3152         closeLock.lock();
3153         if (closing) {
3154             closeLock.unlock();
3155             return false;
3156         }
3157         closeLock.unlock();
3158
3159         Iterator<User> uItr = gameRequestUsers.iterator();
3160         Iterator<Boolean> bItr = accepted.iterator();
3161
3162         while (uItr.hasNext()) {
3163
3164             if (uItr.next() == user) {
3165                 Boolean isAlreadyAccepted = bItr.next();
3166                 if (isAlreadyAccepted) return false;
3167
3168                 isAlreadyAccepted = true;
3169
3170                 // with this, threads doesn't have to wait
3171                 // when needs to see if the game started
3172                 missingPlayerLock.lock();
3173                 missingPlayers--;
3174                 missingPlayerLock.unlock();
3175                 return true;
3176             }
3177             bItr.next();
3178         }
3179
3180         return false;
3181     }
3182
3183     /*
3184      * Sets a words array to a specific player
3185      */
3186     public synchronized boolean sendWords(String[] words, User player) {
3187
3188         int playerPosition = 0;
3189         for (User u: gameRequestUsers) {
3190             if (u == player) {
3191                 playerWords[playerPosition] = words;
3192                 playerHasWords[playerPosition] = true;
3193                 return true;
3194             }
3195             playerPosition++;
3196         }
3197         return false;
3198     }
3199
3200     /*
3201      * Get the specific player words sent
3202      */
3203     public synchronized String[] getPlayerWords(User player) {
3204
3205         int playerPosition = 0;
```

```
3206         for (User u: gameRequestUsers) {
3207             if (u == player) {
3208                 return playerWords[playerPosition];
3209             }
3210             playerPosition++;
3211         }
3212         return null;
3213 
3214     }
3215 
3216     /*
3217      * Returns true iff every player has sent his words
3218      */
3219     public synchronized boolean everyOnePlayed() {
3220         for (int i = 0; i < playerHasWords.length; i++) {
3221             if (!playerHasWords[i]) return false;
3222         }
3223         return true;
3224     }
3225 
3226     /*
3227      * Calculates the rankings of this game and place it in rankingPoints.
3228      * Automatically update scores of single players.
3229      * It MUST be called last, since this method is not synchronized.
3230      */
3231     public boolean calculateRanking() {
3232 
3233         if (words == null) return false;
3234 
3235         int currPlayer = 0;
3236         for (User u: gameRequestUsers) {
3237             if (playerWords[currPlayer] != null) {
3238 
3239                 String[] remPW = removeDuplicates(playerWords[currPlayer]);
3240                 for (int i = 0; i < remPW.length; i++) {
3241 
3242                     String pW = remPW[i];
3243                     if (pW == null) continue;
3244                     if (isValid(pW, gameWord) && isInFile(pW)) {
3245 
3246                         rankingPoints[currPlayer] += pW.length();
3247                         u.addPoints(pW.length());
3248                     }
3249                 }
3250             }
3251             currPlayer++;
3252         }
3253         return true;
3254     }
3255 
3256     private String[] removeDuplicates(String[] word) {
3257         String[] remPW = new String[word.length];
3258 
3259         int remDimension = 0;
3260 
3261         for (int i = 0; i < word.length; i++) {
3262             if (word[i] == null) continue;
3263             boolean alreadyCont = false;
3264             int j ;
3265             for (j = 0; j < remDimension; j++) {
3266                 if (remPW[j].equals(word[i])) { alreadyCont = true; break; }
3267             }
3268             if (!alreadyCont) {
3269                 remPW[j] = word[i];
3270                 remDimension++;
3271             }
3272         }
3273 
3274         return remPW;
```

```
3275     }
3276
3277     /*
3278      * Assure a word is Valid:
3279      *  - it does have only letters contained in the original word
3280      *  - it's contained in the words file
3281      */
3282     private boolean isValid(String word, String base) {
3283
3284         ArrayList<String> w = new ArrayList<String>(), b = new ArrayList<String>();
3285         for (int j = 0; j < word.length(); j++) w.add(word.substring(j, j+1));
3286         for (int j = 0; j < base.length(); j++) b.add(base.substring(j, j+1));
3287
3288         // Strings implements comparable
3289         b.sort(null);
3290         w.sort(null);
3291
3292         if (b.size() < w.size()) return false;
3293
3294         /*
3295          * the words single letters group must be a subset of the base single letter group so,
3296          * when searching for a letter in a position i in the first array, it can already start
3297          * at position i in the second array.
3298          *
3299          */
3300         int j = 0, i = 0;
3301
3302         while (i < w.size() && j < b.size()) {
3303             if (w.get(i).equals(b.get(j))) {
3304                 i++;
3305                 j++;
3306             } else j++;
3307         }
3308         return (i == w.size());
3309     }
3310
3311     private boolean isInFile(String word) {
3312         return (words.contains(word));
3313     }
3314
3315     /*
3316      * Must be called before any game is created.
3317      */
3318     public static void createWordList(File wordsFile){
3319         Game.words = new HashSet<String>();
3320
3321         try (BufferedReader br = new BufferedReader(new FileReader(wordsFile))) {
3322             String line;
3323             while ((line = br.readLine()) != null) {
3324
3325                 Game.words.add(line);
3326             }
3327         } catch (IOException e) {
3328             e.printStackTrace();
3329         }
3330
3331     }
3332
3333     public int[] getRanking() { return this.rankingPoints; }
3334
3335     public boolean isClosing() { return this.closing; }
3336
3337     public void close() { this.closing = true; }
3338
3339     public LinkedBlockingQueue<User> getGameUsers() { return this.gameRequestUsers; }
3340
3341     public boolean readyGame() {
3342         missingPlayerLock.lock();
3343         boolean mP = missingPlayers == 0;
```

```
3344         missingPlayerLock.unlock();
3345         return mP;
3346     }
3347 
3348     public void start() { startLock.lock(); this.started = true; startLock.unlock(); }
3349 
3350     public boolean isStarted() {
3351         startLock.lock();
3352         Boolean s = this.started;
3353         startLock.unlock();
3354         return s;
3355     }
3356 
3357     public void setGameWord(String word) { this.gameWord = word; }
3358 
3359     public String getGameWord() { return this.gameWord; }
3360 
3361 
3362 }
3363 
3364 
3365 
3366 
3367 
3368 package serverPackage;
3369 
3370 import java.util.ArrayList;
3371 import java.util.Random;
3372 import java.util.concurrent.LinkedBlockingQueue;
3373 
3374 import commonPackage.Multicast_rankings;
3375 import commonPackage.RankingItem;
3376 import serverPackage.User.userState;
3377 
3378 public class UserContainer {
3379 
3380     private LinkedBlockingQueue<User> users;
3381     private LinkedBlockingQueue<Game> gamesRequest;
3382 
3383 
3384     UserContainer() {
3385         users = new LinkedBlockingQueue<User>();
3386         gamesRequest = new LinkedBlockingQueue<Game>();
3387     }
3388 
3389 
3390     public synchronized boolean exist(String name) {
3391 
3392         for (User u: users) {
3393 
3394             if (u.getName().equals(name)) return true;
3395         }
3396         return false;
3397     }
3398 
3399 
3400     public synchronized User getUser(String name) {
3401 
3402         for (User u:users) {
3403             if (u.getName().equals(name)) {
3404                 return u;
3405             }
3406         }
3407         return null;
3408     }
3409 
3410     /*
3411      * Adds a new User and set's a new sessionID.
3412      * This number is used to be sure of the Client identity
```

```
3413       * and to not let any other user to do action for him
3414       * after he logged in. Theres a small chance that 2 users
3415       * have the same sessionID, but this doens't preclude the
3416       * success of the methods wich use it.
3417       */
3418      public synchronized boolean addUser(User user) {
3419
3420          if (!exist(user.getName())) {
3421              users.add(user);
3422              user.setSessionID((new Random()).nextInt(10000000));
3423              return true;
3424          }
3425          return false;
3426      }
3427
3428
3429      /*
3430       * returns the game if every user exist and is online
3431       * returns null otherwise
3432       */
3433      public synchronized Game addGame(ArrayList<String> usersSend, String creator) {
3434
3435          for (String u: usersSend) {
3436              if (!exist(u)) return null;
3437          }
3438
3439          LinkedBlockingQueue<User> gameUsers = new LinkedBlockingQueue<User>();
3440
3441          for (String u:usersSend) {
3442              User el = getUser(u);
3443              if (!el.getState().equals(userState.ONLINE)) { return null;}
3444              gameUsers.add(el);
3445          }
3446
3447          Game game = new Game(gameUsers, creator);
3448          this.gamesRequest.add(game);
3449
3450          return game;
3451      }
3452
3453      public synchronized boolean removeGame(Game game) {
3454          return this.users.remove(game);
3455      }
3456
3457      public synchronized Game getGameByID(int gameID) {
3458
3459          for (Game g: gamesRequest) {
3460              if (g.gameID == gameID) return g;
3461          }
3462          return null;
3463      }
3464
3465      public synchronized String[] getUsersName() {
3466          String[] userNames = new String[users.size()];
3467
3468          int c = 0;
3469          for (User us: users) {
3470              userNames[c] = us.getName();
3471              c++;
3472          }
3473          return userNames;
3474      }
3475
3476      public synchronized Multicast_rankings getGlobalRanking() {
3477
3478          ArrayList<RankingItem> ranks = new ArrayList<RankingItem>();
3479          for (User u: users) ranks.add(new RankingItem(u.getName(), u.getPoints()));
3480          Multicast_rankings rankings = new Multicast_rankings(ranks);
3481          rankings.orderRanks();
```

```
3482
3483         return rankings;
3484     }
3485
3486     public synchronized ArrayList<String> getOnlineUsers() {
3487         ArrayList<String> userOnline = new ArrayList<String>();
3488
3489         for (User u: users) if (u.getState().equals(userState.ONLINE))
             userOnline.add(u.getName());
3490         return userOnline;
3491     }
3492
3493
3494 }
3495
3496
3497
3498
3499 package serverPackage;
3500
3501 import commonPackage.RMI_client_interface;
3502
3503 public class User {
3504
3505     public enum userState {
3506         ONLINE, OFFLINE, PLAYING, UNACTIVE, DEACTIVATED
3507     }
3508
3509     private userState state;
3510     private String userName;
3511     private byte[] userCodedPassword;
3512     private byte[] passwordSalt;
3513     private RMI_client_interface clientCallback;
3514
3515     // For login controls
3516     private int sessionID = 0;
3517
3518     private int totalPoints;
3519
3520     User(String userName, String userPassword) {
3521
3522         this.userName = userName;
3523         this.passwordSalt = Passwords.getNextSalt();
3524
3525         char[] charPSW = new char[userPassword.length()];
3526         userPassword.getChars(0, userPassword.length(), charPSW, 0);
3527         this.userCodedPassword =
3528                 Passwords.hash(charPSW, this.passwordSalt);
3529
3530         totalPoints = 0;
3531         state = userState.OFFLINE;
3532     }
3533
3534     // second costructor, used to re-create an user using server state file
3535     User(String userName, byte[] userCodedPassword, byte[] salt, int points) {
3536
3537         this.userName = userName;
3538         this.userCodedPassword = userCodedPassword;
3539         this.passwordSalt = salt;
3540         this.totalPoints = points;
3541         state = userState.OFFLINE;
3542     }
3543
3544     public int getPoints() { return totalPoints; }
3545     public synchronized void addPoints(int points) { this.totalPoints += points; }
3546     public synchronized void resetPoints() { this.totalPoints = 0; }
3547
3548     public String getName() { return this.userName; }
3549
```

```
3550     public boolean verifyPassword(String password) {
3551         char[] charPSW = new char[password.length()];
3552         password.getChars(0, password.length(), charPSW, 0);
3553         return Passwords.isExpectedPassword(charPSW, passwordSalt, userCodedPassword);
3554         }
3555 
3556     public userState getState() { return this.state; }
3557     public synchronized void setState(userState state) { this.state = state; }
3558 
3559     public synchronized void resetCallback() { this.clientCallback = null; }
3560     public RMI_client_interface getCallback() { return this.clientCallback; }
3561     public synchronized void setCallback(RMI_client_interface clientCallback) {
3562         this.clientCallback = clientCallback; }
3563 
3564     public synchronized void setSessionID(int id) { this.sessionID = id; }
3565     public synchronized void resetSessionID () { this.sessionID = 0; }
3566     public synchronized int getSessionID() { return this.sessionID; }
3567 
3568     // necessary to save the server state
3569     public byte[] getCodedPassword() { return this.userCodedPassword; }
3570     public byte[] getSalt() { return this.passwordSalt; }
3571 
3572     public String toString() {return userName; }
3573 
3574 }
3575 
3576 
3577 
3578 
3579 
3580 package serverPackage;
3581 
3582 import java.security.NoSuchAlgorithmException;
3583 import java.security.SecureRandom;
3584 import java.security.spec.InvalidKeySpecException;
3585 import java.util.Arrays;
3586 import java.util.Random;
3587 
3588 import javax.crypto.SecretKeyFactory;
3589 import javax.crypto.spec.PBEKeySpec;
3590 
3591 /**
3592  * A utility class to hash passwords and check passwords vs hashed values. It uses a combination
      of hashing and unique
3593  * salt. The algorithm used is PBKDF2WithHmacSHA1, and the hashed value has 256 bits.
3594  */
3595 public class Passwords {
3596 
3597   private static final Random RANDOM = new SecureRandom();
3598   private static final int ITERATIONS = 10000;
3599   private static final int KEY_LENGTH = 256;
3600 
3601   /**
3602    * static utility class
3603    */
3604   private Passwords() { }
3605 
3606   /**
3607    * Returns a random salt to be used to hash a password.
3608    *
3609    * @return a 16 bytes random salt
3610    */
3611   public static byte[] getNextSalt() {
3612     byte[] salt = new byte[16];
3613     RANDOM.nextBytes(salt);
3614     return salt;
3615   }
3616 
3617   /**
```

```
3618     * Returns a salted and hashed password using the provided hash.<br>
3619     * Note - side effect: the password is destroyed (the char[] is filled with zeros)
3620     *
3621     * @param password the password to be hashed
3622     * @param salt     a 16 bytes salt, ideally obtained with the getNextSalt method
3623     *
3624     * @return the hashed password with a pinch of salt
3625     */
3626    public static byte[] hash(char[] password, byte[] salt) {
3627      PBEKeySpec spec = new PBEKeySpec(password, salt, ITERATIONS, KEY_LENGTH);
3628      Arrays.fill(password, Character.MIN_VALUE);
3629      try {
3630        SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
3631        return skf.generateSecret(spec).getEncoded();
3632      } catch (NoSuchAlgorithmException | InvalidKeySpecException e) {
3633        throw new AssertionError("Error while hashing a password: " + e.getMessage(), e);
3634      } finally {
3635        spec.clearPassword();
3636      }
3637    }
3638
3639    /**
3640     * Returns true if the given password and salt match the hashed value, false otherwise.<br>
3641     * Note - side effect: the password is destroyed (the char[] is filled with zeros)
3642     *
3643     * @param password     the password to check
3644     * @param salt         the salt used to hash the password
3645     * @param expectedHash the expected hashed value of the password
3646     *
3647     * @return true if the given password and salt match the hashed value, false otherwise
3648     */
3649    public static boolean isExpectedPassword(char[] password, byte[] salt, byte[] expectedHash) {
3650      byte[] pwdHash = hash(password, salt);
3651      Arrays.fill(password, Character.MIN_VALUE);
3652      if (pwdHash.length != expectedHash.length) return false;
3653      for (int i = 0; i < pwdHash.length; i++) {
3654        if (pwdHash[i] != expectedHash[i]) return false;
3655      }
3656      return true;
3657    }
3658
3659    /**
3660     * Generates a random password of a given length, using letters and digits.
3661     *
3662     * @param length the length of the password
3663     *
3664     * @return a random password
3665     */
3666    public static String generateRandomPassword(int length) {
3667      StringBuilder sb = new StringBuilder(length);
3668      for (int i = 0; i < length; i++) {
3669        int c = RANDOM.nextInt(62);
3670        if (c <= 9) {
3671          sb.append(String.valueOf(c));
3672        } else if (c < 36) {
3673          sb.append((char) ('a' + c - 10));
3674        } else {
3675          sb.append((char) ('A' + c - 36));
3676        }
3677      }
3678      return sb.toString();
3679    }
3680  }
3681
```