```java
package chatrmi.client;

import chatrmi.protocol.ClientInterface;
import chatrmi.protocol.Message;
import chatrmi.protocol.MessageInterface;
import chatrmi.protocol.ServerInterface;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

/**
 * Classe permettant de créer un objet Remote de type client pour un server de
 * chat
 * Implémente l'interface Remote "ClientInterface"
 * @author Nico
 */
public class Client implements ClientInterface{

    /**
     * Adresse du serveur sur lequel le client est connecté
     */
    protected String serverAdress;

    /**
     * Port par lequel le client et le serveur communiquent
     */
    protected int serverPort;

    /**
     * Nom de l'utilisateur
     */
    protected String userName;

    /**
     * Reference sur le serveur (Remote) sur lequel on est connecté
     */
    ServerInterface server;

    /**
     * Reference sur la fenetre du client
     */
    ClientChatView clientChatView;

    /**
     * Constructeur
     */
    public Client() {
        try {
            clientChatView = new ClientChatView(this);
```



```java
            serverAdress = null;
            serverPort = 1099;
            UnicastRemoteObject.exportObject(this, 0);
            clientChatView.setVisible(true);
        } catch (Exception e) {
            System.err.println("Client exception : " + e.toString());
                e.printStackTrace();
        }
    }

    /**
     * Getter
     * @return Le nom de l'utilisateur
     */
    @Override
    public String getUserName() {
        return userName;
    }

    /**
     * Getter
     * @return l'adresse du serveur
     */
    public String getServerAdress() {
        return serverAdress;
    }

    /**
     * Getter
     * @return le port
     */
    public int getServerPort() {
        return serverPort;
    }

    /**
     * Setter
     * @param name nom de l'utilisateur
     */
    public void setUserName(String name) {
        userName = name;
    }

    /**
     * Setter
     * @param adress adress du serveur
     */
    public void setServerAdress(String adress) {
        serverAdress = adress;
    }

    /**
```



```java
     * Setter
     * @param port port de communication
     */
    public void setServerPort(int port) {
        serverPort = port;
    }

    /**
     * Methode permettant d'afficher un message sur la vue
     * @param m message à afficher
     * @throws RemoteException
     */
    @Override
    public void displayMessage(MessageInterface m) throws RemoteException {
        clientChatView.displayMessage(m);
    }

    /**
     * Methode notifiant la connexion d'un autre utilisateur.
     * Ajoute le nom du client à la vue
     * @param c Client
     * @throws RemoteException
     */
    @Override
    public void newUserConnected(ClientInterface c) throws RemoteException {
        clientChatView.addUser(c);
    }

    /**
     * Methode notifiant la déconnexion d'un autre utilisateur.
     * Enlève le nom du client à la vue
     * @param c Client
     * @throws RemoteException
     */
    @Override
    public void userDisconnected(ClientInterface c) throws RemoteException {
        if (c.getUserName().compareTo(userName)==0){
            server = null;
            clientChatView.refresh();
        }
        clientChatView.removeUser(c);
    }

    /**
     * Methode permettant d'envoyer un message à tous le client connecté au
     * serveur
     * On crée un stub pour le message écrit par le client et on l'envoie au
```



```java
     * serveur.
     * @param m texte du message
     */
    public void sendMessageToAll(String m) {
        try {
            Message msg = new Message(new Date(), userName, m);
            MessageInterface stubMsg = (MessageInterface)
            UnicastRemoteObject
                    .exportObject(msg, 0);
            server.sendMessageToAll(stubMsg);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }

    /**
     * Methode permettant de se connecter au serveur.
     * Le client établit une connection au RMI registry crée dans la classe
     * Server.java. Si la tentative de connexion réussi, le client est rajouté
     * à la liste de Client du serveur.
     * @param sAdress adresse du serveur
     * @param sPort port de communication
     * @param name nom du client
     * @return
     */
    public Boolean connect(String sAdress, String sPort, String name) {
        Boolean result = true;
        try {
            serverAdress = sAdress;
            userName = name;
            serverPort = Integer.parseInt(sPort);
            Registry registry = LocateRegistry
                    .getRegistry(serverAdress,serverPort);
            server = (ServerInterface) registry.lookup("server");
            if(server != null && !server.addClient(this)) {
                server = null;
                result = false;
            }
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            try
            {
                clientChatView.
                        displayMessage(new Message(new Date(),
                                "Error", "Impossible to connect."));
            } catch (Exception e2) {
                System.err.println("Client exception: " +
                e2.toString());
            }
```

```
199             server = null;
200             result = false;
201         }
202         if (result){
203             clientChatView.refresh();
204         }
205         return result;
206     }
207 
208     /**
209      * Methode permettant de se deconnecter du serveur.
210      * Le client se déconnecte du serveur. On supprime le lien vers le serveur
211      * et on supprime le client de la liste Clients dans la classe Server.java .
212      */
213     public void disconnect() {
214         try {
215             server.deleteClient(this);
216             server = null;
217             clientChatView.refresh();
218         } catch (Exception e) {
219             System.err.println("Client exception: " + e.toString());
220             e.printStackTrace();
221         }
222     }
223 
224     /**
225      * Methode permettant de savoir si le client est connecté a un serveur.
226      * On vérifie que le client est bien connecté au serveur (l'attribut "
227      * server" qui constitue la réference vers le serveur est non nul).
228      * @return Retourne true si connecté, false sinon
229      */
230     public Boolean isConnected() {
231         return null != server;
232     }
233 }
234 
```



```
1  package chatrmi.protocol;
2  
3  import java.rmi.Remote;
4  import java.rmi.RemoteException;
5  
6  /**
7   * Interface permettant de définir les methodes necessaire pour un client d'un
8   * chat developper en RMI
9   * @author Nico
10  */
11 public interface ClientInterface extends Remote {
12 
13     /**
14      * Methode permettant d'afficher un message.
15      * @param m Message à afficher.
16      * @throws RemoteException
17      */
18     void displayMessage(MessageInterface m) throws RemoteException;
19 
20     /**
21      * Getter
22      * @return Nom de l'utilisateur.
23      * @throws RemoteException
24      */
25     String getUserName() throws RemoteException;
26 
27     /**
28      * Methode notifiant la connexion d'un autre utilisateur.
29      * @param c Client.
30      * @throws RemoteException
31      */
32     void newUserConnected(ClientInterface c) throws RemoteException;
33 
34     /**
35      * Methode notifiant la déconnexion d'un autre utilisateur.
36      * @param c Client.
37      * @throws RemoteException
38      */
39     void userDisconnected(ClientInterface c) throws RemoteException;
40 }
41 
```



```
1  package chatrmi.protocol;
2  
3  import java.text.DateFormat;
4  import java.text.SimpleDateFormat;
5  import java.util.*;
6  
7  /**
8   *
9   * @author Nico
10  */
11 public class Message implements MessageInterface{
12 
13     /**
14      * Date de l'envoi du message.
15      */
16     private final Date date;
17 
18     /**
19      * Nom de l'expediteur.
20      */
21     private final String userName;
22 
23     /**
24      * Texte du message.
25      */
26     private final String textMessage;
27 
28     /**
29      * Constructeur.
30      * @param d Date de l'envoi du message.
31      * @param usr Nom de l'expediteur.
32      * @param msg Texte du message.
33      */
34     public Message(Date d, String usr, String msg) {
35         date = d;
36         userName = usr;
37         textMessage = msg;
38     }
39 
40     /**
41      * Getter.
42      * @return Date de l'envoi du message.
43      */
44     @Override
45     public Date getDate() {
46         return date;
47     }
48 
49     /**
50      * Getter.
51      * @return Date de l'envoi du message en String.
52      */
```



```
53     @Override
54     public String getDateToString() {
55         DateFormat formatter = new SimpleDateFormat("MM/dd/yy h:m:s");
56         return formatter.format(date);
57     }
58 
59     /**
60      * Getter.
61      * @return Nom de l'expediteur.
62      */
63     @Override
64     public String getUserName() {
65         return userName;
66     }
67 
68     /**
69      * Getter
70      * @return Texte du message.
71      */
72     @Override
73     public String getTextMessage() {
74         return textMessage;
75     }
76 }
77 
```

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package chatrmi.protocol;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

/**
 * Interface permettant de définir les methodes necessaire pour un message d'un
 * chat developper en RMI
 *
 * @author Nico
 */
public interface MessageInterface extends Remote{

    /**
     * Getter
     * @return Date de l'envoi du message
     * @throws RemoteException
     */
    Date getDate() throws RemoteException;

    /**
     * Getter
     * @return Date en String.
     * @throws RemoteException
     */
    String getDateToString() throws RemoteException;

    /**
     * Getter
     * @return Nom de l'expediteur.
     * @throws RemoteException
     */
    String getUserName() throws RemoteException;

    /**
     * Getter
     * @return Text du message .
     * @throws RemoteException
     */
    String getTextMessage() throws RemoteException;
}
```



```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package chatrmi.server;

import chatrmi.protocol.ClientInterface;
import chatrmi.protocol.Message;
import chatrmi.protocol.MessageInterface;
import chatrmi.protocol.ServerInterface;
import java.io.File;
import static java.lang.System.exit;
import java.net.InetAddress;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.LinkedList;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;


/**
 * Classe permettant de créer un objet Remote de type server pour un server de
 * chat
 * Implémente l'interface Remote "ServerInterface"
 * @author Nico
 */
public class Server implements ServerInterface {

    /**
     * Nom du serveur.
     */
    private String serverName;

    /**
     * Port de communication.
     */
```



```java
    private int serverPort;

    /**
     * Chemin d'accès du fichier contenant l'historique des messages.
     */
    private String historicsFile;

    /**
     * Registre remote.
     */
    private Registry registry;

    /**
     * Etat du serveur. (Allumé : true, éteind : false).
     */
    private boolean state;

    /**
     * Liste des clients connectés.
     */
    private LinkedList<ClientInterface> clients;

    /**
     * Historiques de messages.
     */
    private LinkedList<MessageInterface> messages;

    /**
     * Vue serveur.
     */
    private ServerRunningView serverRunningView;

    /**
     * Constructeur. Initialise le nom du serveur à "server", le port de
     * communication à 1099 et le fichier d'historique à "historics.xml".
     */
    public Server() {
       initialize("server",1099,"historics.xml");
    }

    /**
     * Permet d'initiliser les attributs.
     * @param sName Nom du serveur.
     * @param sPort Port de communication.
     * @param hFile Chemin d'accès du fichier d'historique.
     */
    private void initialize(String sName, int sPort, String hFile) {
        serverName = sName;
        serverPort = sPort;
        historicsFile = hFile;
```



```java
        state = false;
        clients = new LinkedList<>();
        messages = new LinkedList<>();
        serverRunningView = new ServerRunningView(this);
        serverRunningView.setLocationRelativeTo(null);
        serverRunningView.setVisible(true);
        loadHistorics();
        try
        {
            UnicastRemoteObject.exportObject(this, 0);
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            exit(0);
        }
    }

    /**
     * Bind le serveur au registre Remote sur le port "serverPort" et avec le
     * nom "serverName".
     * @return True si réussi, false sinon.
     */
    private boolean run() {
        boolean result = true;
        try {
            LocateRegistry.createRegistry(serverPort);
        } catch (Exception e) {
            //System.err.println("Server exception: " + e.toString());
        } finally {
            try {
                registry = LocateRegistry.getRegistry();
                registry.bind(serverName, this);
            } catch (Exception e) {
                System.err.println("Server exception: " +
                e.toString());
                result = false;
            }
        }
        return result;
    }

    /**
     * Permer de lancer le serveur avec comme port de communication "port".
     * Modifie l'état du serveur si reussi ou non (valeur retournée).
     * @param port Port de communication.
     * @return True si réussi, false sinon.
     */
    public boolean start(String port)
    {
        try
        {
```

```
150             serverPort = Integer.parseInt(port);
151             state = run();
152
                serverRunningView.diplayServerLocalAdress(InetAddress.getL
                ocalHost()
153                     .getHostAddress());
154         }
155         catch(Exception e)
156         {
157             System.err.println("Server exception: " + e.toString());
158         }
159         return state;
160     }
161
162     /**
163      * Permet de stopper le serveur.
164      * Modifie l'état du serveur.
165      * @return True si réussi, false sinon.
166      */
167     public boolean stop()
168     {
169         try {
170
171             registry.unbind(serverName);
172             state = false;
173
174             for (ClientInterface c1 : clients) {
175                 c1.userDisconnected(c1);
176             }
177
178             clients.removeAll(clients);
179             serverRunningView.setClientNumber(clients.size());
180         } catch (Exception e) {
181             System.err.println("Server exception: " + e.toString());
182         }
183         return !state;
184     }
185
186     /**
187      * Méthode permettant de savoir si le serveur est en train de
        tourner ou non
188      *
189      * @return True si le serveur est en train de tourner, false sinon.
190      */
191     public Boolean isRunning() {
192         return state;
193     }
194
195     /**
196      * Methode permettant d'envoyer un message à tous les clients
        connectés au
197      * serveur.
```



```
198      * Affiche le message envoyé au serveur par un client à tous les
         clients de
199      * la liste Clients et rajoute le message à la liste de Messages.
200      * @param m Message à envoyer.
201      * @throws RemoteException
202      */
203     @Override
204     public void sendMessageToAll(MessageInterface m) throws
        RemoteException {
205         // TODO Auto-generated method stub
206         Message msg = new Message(m.getDate(),m.getUserName(),
207                 m.getTextMessage());
208         MessageInterface stubMsg = (MessageInterface)
            UnicastRemoteObject
209                     .exportObject(msg, 0);
210
211         for (ClientInterface client : clients) {
212             client.displayMessage(stubMsg);
213         }
214         messages.addLast(stubMsg);
215     }
216
217     /**
218      * Methode permettant d'ajouter un nouveau client à liste de
         clients
219      * connectés.
220      * Elle permet de rajouter le client qui souhaite se connecter
         au serveur
221      * afin qu'il bénéficie des services offert par celui-ci. Elle
         vérifie bien
222      * que le username choisi par le client n'est pas déjà utilisé.
         Elle le
223      * rajoute à la liste des Clients et lui affiche tous les
         messages existants
224      * dans la liste Messages.
225      * @param c Client
226      * @return True si l'ajout a pu se faire, false sinon.
227      * @throws RemoteException
228      */
229     @Override
230     public boolean addClient(ClientInterface c) throws
        RemoteException {
231         // TODO Auto-generated method stub
232         Boolean result = true;
233         if(c == null){
234             result = false;
235         } else if(contains(c)) {
236             result = false;
237             sendServerMessage(c, "Username \"" + c.getUserName() +
238                     "\" already used.");
239         } else {
240             sendServerMessageToAll(c.getUserName() + " has joined
```



```
                the chat.");
241
242             for (ClientInterface client : clients) {
243                 c.newUserConnected(client);
244             }
245
246             clients.add(c);
247             sendHistorics(c);
248
249             for (ClientInterface client : clients) {
250                 client.newUserConnected(c);
251             }
252
253             serverRunningView.setClientNumber(clients.size());
254             result = true;
255         }
256         return result;
257     }
258
259     /**
260      * Methode permettant d'enlever un client à liste de clients
         connectés.
261      * Le serveur informe tous les clients que l'utilisateur c a
         quitté le chat.
262      * Tous les clients affichent sur le interface graphique : « c
         has left ».
263      * le client c est supprimé de la liste Clients.
264      * @param c Client.
265      * @throws RemoteException
266      */
267     @Override
268     public void deleteClient(ClientInterface c) throws
        RemoteException {
269         // TODO Auto-generated method stub
270
271         for (ClientInterface client : clients) {
272             client.userDisconnected(c);
273         }
274         clients.remove(c);
275
276         sendServerMessage(c, "You have left the chat.");
277         sendServerMessageToAll(c.getUserName() + " has left the
            chat.");
278
279         serverRunningView.setClientNumber(clients.size());
280     }
281
282     /**
283      * Methode permettant de savoir si un client ayant le même non
         d'utilisateur
284      * que "client" est connecté.
285      * @param client Client.
```



```
286      * @return True si oui, false sinon.
287      * @throws RemoteException
288      */
289     private boolean contains(ClientInterface client) throws
        RemoteException
290     {
291         boolean result = false;
292         for(ClientInterface c : clients){
293             if ( client.getUserName().compareTo(c.getUserName()) ==
                0 ){
294                 result = true;
295                 break;
296             }
297         }
298         return result;
299     }
300
301     /**
302      * Methode permettant d'envoyer un message à tous les clients
         connectés au
303      * serveur.
304      * Elle permet d'afficher à tous les clients connectés des
         messages
305      * informatifs qui viennent du serveur et non pas d'autres clients
306      * (exemple : informer de la connexion d'un nouveau client est
         un message
307      * du serveur).
308      * @param m Message à envoyer.
309      * @throws RemoteException
310      */
311     private void sendServerMessageToAll(String m) throws
        RemoteException {
312         Message msg = new Message(new Date(),"Server",m);
313         MessageInterface stubMsg = (MessageInterface)
            UnicastRemoteObject
314                     .exportObject(msg, 0);
315
316         for (ClientInterface client : clients) {
317             client.displayMessage(stubMsg);
318         }
319         messages.addLast(stubMsg); //Tout depend si on enregistre
            les messages
320                                    //du server
321     }
322
323     /**
324      * Methode permettant d'envoyer un message à un client connecté
         au serveur.
325      * @param c Client à qui envoyer le message.
326      * @param m Message à envoyer.
327      * @throws RemoteException
328      */
```

```
329     private void sendServerMessage(ClientInterface c,String m)
330             throws RemoteException {
331         Message msg = new Message(new Date(),"Server",m);
332         MessageInterface stubMsg = (MessageInterface)
            UnicastRemoteObject
333                     .exportObject(msg, 0);
334
335         c.displayMessage(stubMsg);
336     }
337
338     /**
339      * Méthode permettant d'envoyer l'historique des messages à un
         client
340      * spécifique.
341      * @param c Client à qui envoyer le message.
342      * @throws RemoteException
343      */
344     private void sendHistorics(ClientInterface c) throws
        RemoteException {
345         for( MessageInterface m : messages)
346         {
347             c.displayMessage(m);
348         }
349     }
350
351     /**
352      * Methode permettant de charger l'historique des messages
         (provenant du
353      * fichier "historicsFile").
354      * Tous les messages enregistrés dans le document XML sont
         ajoutés dans
355      * l'ordre chronologique à la liste "messages". Ils sont stubés
         pour
356      * pouvoir être envoyés aux différents clients qui se connecterons.
357      */
358     public void loadHistorics() {
359         try {
360             DateFormat formatter = new SimpleDateFormat("MM/dd/yy
                h:m:s");
361             final DocumentBuilderFactory factory =
                DocumentBuilderFactory
362                     .newInstance();
363             final DocumentBuilder builder =
                factory.newDocumentBuilder();
364
365         final Document document= builder.parse(new
            File(historicsFile));
366
367         final Element racine = document.getDocumentElement();
368         final NodeList racineNoeuds =
            racine.getElementsByTagName("Message");
369         final int nbRacineNoeuds = racineNoeuds.getLength();
```



```
370             System.out.println(nbRacineNoeuds);
371
372         for (int i = 0; i<nbRacineNoeuds; i++) {
373                 System.out.println(i);
374             final Element message = (Element) racineNoeuds.item(i);
375
376                 final Element date = (Element)
377                         message.getElementsByTagName("Date").item(0);
378                 final Element username = (Element)
379                         message.getElementsByTagName("Username").item(
                        0);
380                 final Element text = (Element)
381                         message.getElementsByTagName("Text").item(0);
382
383                 Message msg = new Message(formatter.parse(
384                         date.getTextContent()),username.getTextContent
                        (),
385                         text.getTextContent());
386                 MessageInterface stubMsg = (MessageInterface)
387                         UnicastRemoteObject.exportObject(msg, 0);
388
389                 messages.addLast(stubMsg);
390
391         }
392
393         } catch(Exception e) {
394
395         }
396     }
397
398     /**
399      * Methode permettant d'enregistrer l'historique des messages
400      * (dans le fichier "historicsFile").
401      * Tous les messages enregistrés dans la liste "messages" sont
402      * stockés à la suite dans un document XML avec précision de la
         date
403      * ainsi que le nom d'utilisateur pour chaque message dans
         l'ordre d'ajout
404      * à la liste.
405      */
406     public void saveHistorics() {
407         try {
408             DateFormat formatter = new SimpleDateFormat("MM/dd/yy
                h:m:s");
409             final DocumentBuilderFactory factory =
                DocumentBuilderFactory
410                     .newInstance();
411         final DocumentBuilder builder = factory.newDocumentBuilder();
412         final Document document= builder.newDocument();
413         final Element racine = document.createElement("Messages");
```



```
414         document.appendChild(racine);
415
416             for(MessageInterface m : messages){
417                 final Element message =
                    document.createElement("Message");
418                 racine.appendChild(message);
419
420                 final Element date = document.createElement("Date");
421
422
423                 date.appendChild(document.createTextNode(formatter.for
                    mat(
424                         m.getDate())));
425
426                 final Element username =
                    document.createElement("Username");
427
428                 username.appendChild(document.createTextNode(m.getUser
                    Name()));
429
430                 final Element text = document.createElement("Text");
431
432                 text.appendChild(document.createTextNode(m.getTextMess
                    age()));
433
434                 message.appendChild(date);
435                 message.appendChild(username);
436                 message.appendChild(text);
437             }
438
439         final TransformerFactory transformerFactory =
            TransformerFactory
440                     .newInstance();
441         final Transformer transformer =
            transformerFactory.newTransformer();
442         final DOMSource source = new DOMSource(document);
443         final StreamResult sortie = new StreamResult(new
            File(historicsFile));
444
445         transformer.setOutputProperty(OutputKeys.VERSION, "1.0");
446         transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
447         transformer.setOutputProperty(OutputKeys.STANDALONE,
            "yes");
448
449         transformer.setOutputProperty(OutputKeys.INDENT, "yes");
450         transformer.setOutputProperty(
451                     "{http://xml.apache.org/xslt}indent-amount", "2");
452
453         transformer.transform(source, sortie);
454         } catch(Exception e) {
455
456         }
```



```
457     }
458
459 }
460
461
462
```

```java
package chatrmi.protocol;


import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * Interface permettant de définir les methodes necessaire pour un server d'un
 * chat developper en RMI
 * @author Nico
 */
public interface ServerInterface extends Remote {

    /**
     * Methode permettant d'envoyer un message à tous les clients connectés au
     * serveur.
     * @param m Message à envoyer.
     * @throws RemoteException
     */
    void sendMessageToAll(MessageInterface m) throws RemoteException;

    /**
     * Methode permettant d'ajouter un nouveau client à liste de clients
     * connectés. (Connecter le client)
     * @param c Client
     * @return True si l'ajout a pu se faire, false sinon.
     * @throws RemoteException
     */
    boolean addClient(ClientInterface c) throws RemoteException;

    /**
     * Methode permettant d'enlever un client à liste de clients connectés.
     * (Deconnecter le client)
     * @param c Client.
     * @throws RemoteException
     */
    void deleteClient(ClientInterface c) throws RemoteException;
}
```