```
1  /*
2   * To change this license header, choose License Headers in Project
    Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package chatrmi.server;
7
8  import chatrmi.protocol.ClientInterface;
9  import chatrmi.protocol.Message;
10 import chatrmi.protocol.MessageInterface;
11 import chatrmi.protocol.ServerInterface;
12 import java.io.File;
13 import static java.lang.System.exit;
14 import java.net.InetAddress;
15 import java.rmi.RemoteException;
16 import java.rmi.registry.LocateRegistry;
17 import java.rmi.registry.Registry;
18 import java.rmi.server.UnicastRemoteObject;
19 import java.text.DateFormat;
20 import java.text.SimpleDateFormat;
21 import java.util.Date;
22 import java.util.LinkedList;
23 import javax.xml.parsers.DocumentBuilder;
24 import javax.xml.parsers.DocumentBuilderFactory;
25 import javax.xml.transform.OutputKeys;
26 import javax.xml.transform.Transformer;
27 import javax.xml.transform.TransformerFactory;
28 import javax.xml.transform.dom.DOMSource;
29 import javax.xml.transform.stream.StreamResult;
30 import org.w3c.dom.Document;
31 import org.w3c.dom.Element;
32 import org.w3c.dom.NodeList;
33
34
35 /**
36  * Classe permettant de créer un objet Remote de type server pour un
    server de
37  * chat
38  * Implémente l'interface Remote "ServerInterface"
39  * @author Nico
40  */
41 public class Server implements ServerInterface {
42
43     /**
44      * Nom du serveur.
45      */
46     private String serverName;
47
48     /**
49      * Port de communication.
50      */
```

```
51      private int serverPort;
52
53      /**
54       * Chemin d'accès du fichier contenant l'historique des messages.
55       */
56      private String historicsFile;
57
58      /**
59       * Registre remote.
60       */
61      private Registry registry;
62
63      /**
64       * Etat du serveur. (Allumé : true, éteind : false).
65       */
66      private boolean state;
67
68      /**
69       * Liste des clients connectés.
70       */
71      private LinkedList<ClientInterface> clients;
72
73      /**
74       * Historiques de messages.
75       */
76      private LinkedList<MessageInterface> messages;
77
78      /**
79       * Vue serveur.
80       */
81      private ServerRunningView serverRunningView;
82
83      /**
84       * Constructeur. Initialise le nom du serveur à "server", le port de
85       * communication à 1099 et le fichier d'historique à "historics.xml".
86       */
87      public Server() {
88         initialize("server",1099,"historics.xml");
89      }
90
91      /**
92       * Permet d'initiliser les attributs.
93       * @param sName Nom du serveur.
94       * @param sPort Port de communication.
95       * @param hFile Chemin d'accès du fichier d'historique.
96       */
97      private void initialize(String sName, int sPort, String hFile) {
98          serverName = sName;
99          serverPort = sPort;
100         historicsFile = hFile;
```

```
101             state = false;
102             clients = new LinkedList<>();
103             messages = new LinkedList<>();
104             serverRunningView = new ServerRunningView(this);
105             serverRunningView.setLocationRelativeTo(null);
106             serverRunningView.setVisible(true);
107             loadHistorics();
108             try
109             {
110                 UnicastRemoteObject.exportObject(this, 0);
111             } catch (Exception e) {
112                 System.err.println("Server exception: " + e.toString());
113                 exit(0);
114             }
115         }
116 
117         /**
118          * Bind le serveur au registre Remote sur le port "serverPort" et avec le
119          * nom "serverName".
120          * @return True si réussi, false sinon.
121          */
122         private boolean run() {
123             boolean result = true;
124             try {
125                 LocateRegistry.createRegistry(serverPort);
126             } catch (Exception e) {
127                 //System.err.println("Server exception: " + e.toString());
128             } finally {
129                 try {
130                     registry = LocateRegistry.getRegistry();
131                     registry.bind(serverName, this);
132                 } catch (Exception e) {
133                     System.err.println("Server exception: " + e.toString());
134                     result = false;
135                 }
136             }
137             return result;
138         }
139 
140         /**
141          * Permer de lancer le serveur avec comme port de communication "port".
142          * Modifie l'état du serveur si reussi ou non (valeur retournée).
143          * @param port Port de communication.
144          * @return True si réussi, false sinon.
145          */
146         public boolean start(String port)
147         {
148             try
149             {
```

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

```
454         }
455
456     }
457
458
459
460
461
462
```