```java
package com.nxkundu.server.service;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import java.util.UUID;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.ConcurrentMap;

import com.google.gson.Gson;
import com.nxkundu.server.bo.Client;
import com.nxkundu.server.bo.DataPacket;
import com.nxkundu.server.bo.Server;
/**
 *
 * @author nxkundu
 *
 * @email nxk161830@utdallas.edu
 * @name Nirmallya Kundu
 *
 * ServerService - This service is initialized on the Server Side
 * When the Server is started
 *
 * 1> startServer()
 * - This method starts the server
 * based on the credentials on the Server object
 *
 * 2> sendPacket()
 * - This method runs the thread threadSend and
```

```
40 * continuously send DataPacket that were added to the Queue qSendPacket
41 * to the respective client (toClient Address in the DataPacket)
42 * -
43 *
44 * 3> processReceivedDatagramPacket()
45 * - This method takes the action on the
46 * received DataPacket based on the action field in the DataPacket
47 *
48 *
49 * 4> sendClientStatus()
50 * - This method sends to the client(s)
51 * the list of all the clients
52 * Who are ONLINE and who are OFFLINE
53 *
54 * 5> broadcastClientStatus()
55 * - This method runs the thread threadBroadcastClientStatus and
56 * continuously send Online DataPacket (containing mapAllClients)
57 * to all the clients to notify all that the client who are ONLINE and who are OFFLINE.
58 *
59 * 6> recievePacketUDP()
60 * - This method runs the thread threadReceivePacketUDP
61 * and continuously receives UDP DataPacket from all the logged in client
62 * and process them to find the content of the packet and perform the necessary action
63 *
64 * 7> sendPacketByUDP()
65 * - This methods sends the DataPacket to the
66 * respective client based on UDP DatagramPacket
67 * the DataPacket contains the address of the ToClient
68 *
69 *
70 * 8> resendDataPacketIfNoACKReceived()
71 * - This method runs the
72 * thread threadResendDataPacketIfNoACKReceived
73 * which Resends the data for which no ACK is received
74 * from the respective Client after a predefined amount of time
75 *
76 * 9> updateClientOn()
77 * - When the client Comes ONLINE
78 * read all the messages from the Database
```

```
 * that were sent to the client while they were OFFLINE
 *
 * 10> updateClientOff()
 * - When the client goes OFFLINE
 * write all the messages to the Database
 * that were sent and received by the client
 * while they were ONLINE
 * 11> writeToFile()
 * - This method write to the file
 * all the user credentials that is in the mapUserCred
 *
 * 12> readFromFile()
 * - This method reads from the file
 * all the user credentials that were saved
 *
 */
public class ServerService implements Runnable{

    /*
     * private Server server - holds the information about the server
     *
     * private boolean isService - when the server starts this flag is set to true
     * and as long as the server is in service this holds True
     *
     * private Thread threadService - This is the main thread which runs on the Server Side
     *
     * private Thread threadSend- This thread continuously sends
     * DataPacket to all clients based on the ToClient address on the DataPacket
     *
     * private Thread threadReceivePacketUDP - This thread continuously receives
     * UDP DataPacket from all clients and process them to find the content of the
     * packet and perform the necessary action
     *
     * private Thread threadBroadcastClientStatus - This thread continuously
     * Broadcast the Map of all the clients (ONLINE, OFFLINE) in a
     * DataPacket to all the logged in clients
     *
     * private Thread threadResendDataPacketIfNoACKReceived - This thread Resends the data for which no ACK
     * is received from the client the DataPacket was sent, after a predefined amount of time
```

```
118     *
119     * private ConcurrentMap<String, Client> mapAllClients - This map stores the list of all clients
120     * this is basically from where we the ONLINE clients and OFFLINE clients are stored
121     *
122     * private ConcurrentLinkedQueue<DataPacket> qSendPacket - Whenever the server wants to send or forward
123     * or broadcast a message it adds a DataPacket to this queue
124     * (containing the ToClient and FromClient Address)
125     *
126     * private ConcurrentMap<UUID, DataPacket> mapSentDataPacket - This map stores all the DataPackets
127     * that was sent to the client, and when the server receives the ACK for the DataPacket,
128     * the respective DataPacket is removed from the map
129     *
130     * private ConcurrentMap<UUID, DataPacket> mapBufferedDataPacket- This map stores all the DataPackets
131     * that was to be sent to a client but the client is offline
132     *
133     * private ConcurrentMap<UUID, DataPacket> mapReceivedDataPacket - This map stores all the DataPackets
134     * received from the client so that the server can send ACK to the respective client that it has
135     * successfully received the DataPacket
136     *
137     * private ConcurrentMap<String, String> mapUserCred - This map stores the user credentials
138     * This map is filled when a client sign up
139     * And when a client logs in this map is referred
140     * to verify the user credentials and allow them to login
141     *
142     * private static final String FILENAME_USER_CREDENTIALS = "UserCred.txt"
143     * This variable stores the file name which is used to write the
144     * user credentials to the file system
145     *
146     */
147
148    private Server server;
149    private boolean isService;
150
151    private Thread threadService;
152    private Thread threadSend;
153    private Thread threadReceivePacketUDP;
154    private Thread threadBroadcastClientStatus;
155    private Thread threadResendDataPacketIfNoACKReceived;
156
```

```
157     private ConcurrentMap<String, Client> mapAllClients;
158
159     private ConcurrentLinkedQueue<DataPacket> qSendPacket;
160
161     private ConcurrentMap<UUID, DataPacket> mapSentDataPacket;
162     private ConcurrentMap<UUID, DataPacket> mapBufferedDataPacket;
163     private ConcurrentMap<UUID, DataPacket> mapReceivedDataPacket;
164
165     private ConcurrentMap<String, String> mapUserCred;
166
167     private static final String FILENAME_USER_CREDENTIALS = "UserCred.txt";
168
169     /***************************** Constructors *****************************/
170
171     public ServerService(){
172
173         isService = false;
174
175         mapAllClients = new ConcurrentHashMap<>();
176
177         mapSentDataPacket = new ConcurrentHashMap<>();
178         mapBufferedDataPacket = new ConcurrentHashMap<>();
179         mapReceivedDataPacket = new ConcurrentHashMap<>();
180
181         qSendPacket = new ConcurrentLinkedQueue<>();
182
183         mapUserCred = new ConcurrentHashMap<>();
184     }
185
186     /***************************** Object Methods *****************************/
187
188     /**
189      * startServer() - This method starts the server
190      * based on the credentials on the Server object
191      *
192      */
193     public void startServer() {
194
195         System.out.println("Starting Server ...");
```

```
196
197         /*
198          * The user credentials are read from the
199          * file FILENAME_USER_CREDENTIALS when the server starts
200          */
201         readFromFile();
202
203         System.out.println("List of Regeistered User: ");
204         System.out.println(mapUserCred);
205
206         try {
207
208             server = Server.getInstance();
209             server.startServer();
210         }
211         catch (SocketException e) {
212
213             e.printStackTrace();
214             System.out.println("SocketException in creating the instance of server! Exiting...");
215             System.exit(0);
216         }
217         catch (UnknownHostException e) {
218
219             e.printStackTrace();
220         } catch (IOException e) {
221
222             e.printStackTrace();
223         }
224
225         threadService = new Thread(this,"StartServer");
226
227         System.out.println("Server Started Successfully...");
228
229         threadService.start();
230     }
231
232
233     @Override
234     public void run() {
```

```
235
236         isService = true;
237
238         /*
239          * recievePacketUDP() - This method runs the thread threadReceivePacketUDP
240          * and continuously receives UDP DataPacket from all the logged in client
241          * and process them to find the content of the packet and perform the necessary action
242          */
243         recievePacketUDP();
244
245         /*
246          * sendPacket() - This method runs the thread threadSend and
247          * continuously send DataPacket that were added to the Queue qSendPacket
248          * to the respective client (toClient Address in the DataPacket)
249          */
250         sendPacket();
251
252         /*
253          * broadcastClientStatus() - This method runs the thread threadBroadcastClientStatus and
254          * continuously send Online DataPacket (containing mapAllClients)
255          * to all the clients to notify all that the client who are ONLINE and who are OFFLINE.
256          */
257         broadcastClientStatus();
258
259         /*
260          * resendDataPacketIfNoACKReceived() - This method runs the
261          * thread threadResendDataPacketIfNoACKReceived
262          * which Resends the data for which no ACK is received
263          * from the respective Client after a predefined amount of time
264          */
265         resendDataPacketIfNoACKReceived();
266
267     }
268
269     /**
270      * sendPacket() - This method runs the thread threadSend and
271      * continuously send DataPacket that were added to the Queue qSendPacket
272      * to the respective client (toClient Address in the DataPacket)
273      */
```

```
274     private void sendPacket() {
275
276         threadSend = new Thread("SendPacket"){
277
278             @Override
279             public void run() {
280
281                 while(isService) {
282
283                     try {
284
285                         if(!qSendPacket.isEmpty()) {
286
287                             DataPacket dataPacket = qSendPacket.poll();
288
289                             if(dataPacket.getAction().equals(DataPacket.ACTION_TYPE_LOGIN_SUCCESS)
290                                     || dataPacket.getAction().equals(DataPacket.ACTION_TYPE_LOGIN_FAILED)
291                                     || dataPacket.getAction().equals(DataPacket.ACTION_TYPE_SIGNUP_FAILED)) {
292
293                                 System.out.println(dataPacket);
294                                 sendPacket(dataPacket);
295                             }
296                             else {
297
298                                 if((mapAllClients.containsKey(dataPacket.getToClient().getUserName()))
299                                         && (mapAllClients.get(dataPacket.getToClient().getUserName())).isOnline()) {
300
301                                     sendPacket(dataPacket);
302                                 }
303                                 else {
304
305                                     mapBufferedDataPacket.put(dataPacket.getId(), dataPacket);
306                                     System.out.println("Buffered Packets : " + mapBufferedDataPacket);
307                                     //mapSentDataPacket.put(dataPacket.getId(), dataPacket);
308
309                                     //TODO WRITE TO DB
310
311                                     /*
312                                      * When the Client is OFFLINE
```

```
313                                     * Add the Message to the Database
314                                     * So that when the client comes ONLINE the next time
315                                     * All the messages that the client received while offline
316                                     * can be viewed by them
317                                     *
318                                     */
319                                 }
320                             }
321 
322                         }
323                     }
324                     catch (IOException e) {
325 
326                         e.printStackTrace();
327                     }
328 
329                     try {
330 
331                         Thread.sleep(500);
332                     }
333                     catch(Exception e) {
334 
335                         e.printStackTrace();
336                     }
337                 }
338             }
339         };
340 
341         threadSend.start();
342     }
343 
344     /**
345      * processReceivedDatagramPacket() - This method takes the action on the
346      * received DataPacket based on the action field in the DataPacket
347      *
348      * @param dataPacket
349      * @param fromClient
350      */
351     private void processReceivedDatagramPacket(DataPacket dataPacket, Client fromClient) {
```

```
352
353         System.out.println("Received Packet = " + dataPacket);
354
355         switch(dataPacket.getAction()) {
356
357         case DataPacket.ACTION_TYPE_LOGIN:
358
359             fromClient.setLastSeenTimestamp(new Date().getTime());
360
361             String username = dataPacket.getFromClient().getUserName();
362             String password = dataPacket.getFromClient().getPassword();
363
364             boolean isLoginSuccess = false;
365             String loginMessage = "";
366             if(mapAllClients.get(username) != null && mapUserCred.get(username).equals(password)) {
367
368                 isLoginSuccess = true;
369                 loginMessage = "Login Successful";
370             }
371             else if(mapAllClients.get(username) == null) {
372
373                 isLoginSuccess = false;
374                 loginMessage = "Failed! Email Not Registered";
375                 fromClient.setLastSeenTimestamp(0);
376             }
377             else if(mapAllClients.get(username) != null && mapUserCred.get(username).equals(password) == false) {
378
379                 isLoginSuccess = false;
380                 loginMessage = "Failed! Incorrect Password";
381                 fromClient.setLastSeenTimestamp(0);
382             }
383
384             fromClient.setPassword("");
385
386             Client serverToClientLoginACK = new Client(Server.SERVER_USERNAME);
387             DataPacket loginACKDataPacket = null;
388
389             if(isLoginSuccess) {
390
```

```
391                 System.out.println("Login Success ... for " + fromClient);
392                 mapAllClients.put(fromClient.getUserName(), fromClient);
393 
394                 loginACKDataPacket = new DataPacket(serverToClientLoginACK, DataPacket.ACTION_TYPE_LOGIN_SUCCESS);
395                 loginACKDataPacket.setToClient(fromClient);
396                 loginACKDataPacket.setMessage(loginMessage);
397             }
398             else {
399 
400                 System.out.println("Login Failed ... for " + fromClient);
401                 loginACKDataPacket = new DataPacket(serverToClientLoginACK, DataPacket.ACTION_TYPE_LOGIN_FAILED);
402                 loginACKDataPacket.setToClient(fromClient);
403                 loginACKDataPacket.setMessage(loginMessage);
404             }
405 
406             qSendPacket.add(loginACKDataPacket);
407 
408             if(isLoginSuccess) {
409 
410                 updateClientOn(fromClient);
411                 sendClientStatus(true, fromClient);
412             }
413             break;
414 
415         case DataPacket.ACTION_TYPE_SIGNUP:
416 
417             fromClient.setLastSeenTimestamp(new Date().getTime());
418 
419             String usernameSignUp = dataPacket.getFromClient().getUserName();
420             String passwordSignUp = dataPacket.getFromClient().getPassword();
421 
422             boolean isSignUpSuccess = false;
423             String signUpMessage = "";
424 
425             if(mapAllClients.get(usernameSignUp) == null) {
426 
427                 isSignUpSuccess = true;
428                 signUpMessage = "Signup Successful! Logging in";
429                 mapAllClients.put(fromClient.getUserName(), fromClient);
```

```
430                 mapUserCred.put(usernameSignUp, passwordSignUp);
431
432                 writeToFile();
433             }
434             else {
435
436                 isSignUpSuccess = false;
437                 signUpMessage = "Failed! Email Exists";
438                 fromClient.setLastSeenTimestamp(0);
439             }
440
441             fromClient.setPassword("");
442
443             Client serverToClientSignUpACK = new Client(Server.SERVER_USERNAME);
444             DataPacket signupACKDataPacket = null;
445
446             if(isSignUpSuccess) {
447
448                 System.out.println("Signup Success ... for " + fromClient + " .. Loggin in..");
449
450                 signupACKDataPacket = new DataPacket(serverToClientSignUpACK, DataPacket.ACTION_TYPE_LOGIN_SUCCESS);
451                 signupACKDataPacket.setToClient(fromClient);
452                 signupACKDataPacket.setMessage(signUpMessage);
453             }
454             else {
455
456                 System.out.println("Signup Failed ... for " + fromClient);
457                 signupACKDataPacket = new DataPacket(serverToClientSignUpACK, DataPacket.ACTION_TYPE_SIGNUP_FAILED);
458                 signupACKDataPacket.setToClient(fromClient);
459                 signupACKDataPacket.setMessage(signUpMessage);
460             }
461
462             qSendPacket.add(signupACKDataPacket);
463
464             if(isSignUpSuccess) {
465
466                 updateClientOn(fromClient);
467                 sendClientStatus(true, fromClient);
468             }
```

```
469             break;
470
471         case DataPacket.ACTION_TYPE_LOGOUT:
472
473             mapAllClients.remove(fromClient.getUserName());
474
475             updateClientOff(fromClient);
476             sendClientStatus(true, fromClient);
477             break;
478
479         case DataPacket.ACTION_TYPE_ONLINE:
480
481             fromClient.setLastSeenTimestamp(new Date().getTime());
482
483             if(mapAllClients.get(fromClient.getUserName()) != null) {
484
485                 mapAllClients.put(fromClient.getUserName(), fromClient);
486             }
487
488             break;
489
490         case DataPacket.ACTION_TYPE_ACK:
491
492
493             UUID dataPacketACKId = UUID.fromString(dataPacket.getMessage());
494
495             if(mapSentDataPacket.containsKey(dataPacketACKId)) {
496
497                 mapSentDataPacket.remove(dataPacketACKId);
498             }
499             else {
500
501                 //Not Possible
502             }
503
504             break;
505
506         case DataPacket.ACTION_TYPE_MESSAGE:
507
```

```
508             Client serverToClientACK = new Client(Server.SERVER_USERNAME);
509             DataPacket dataPacketACK = new DataPacket(serverToClientACK, DataPacket.ACTION_TYPE_ACK);
510             dataPacketACK.setToClient(fromClient);
511             dataPacketACK.setFromClient(fromClient);
512             dataPacketACK.setMessage(dataPacket.getId().toString());
513             qSendPacket.add(dataPacketACK);
514
515             if(mapReceivedDataPacket.containsKey(dataPacket.getId())) {
516
517                 break;
518             }
519
520             mapReceivedDataPacket.put(dataPacket.getId(), dataPacket);
521
522             switch (dataPacket.getMessageType()) {
523
524             case DataPacket.MESSAGE_TYPE_MESSAGE:
525
526                 if(mapAllClients.containsKey((dataPacket.getToClient().getUserName()))) {
527
528
529                     Client toClient = mapAllClients.get(dataPacket.getToClient().getUserName());
530                     dataPacket.setToClient(toClient);
531                     qSendPacket.add(dataPacket);
532
533                 }
534                 break;
535
536             case DataPacket.MESSAGE_TYPE_BROADCAST_MESSAGE:
537
538                 for(String key : mapAllClients.keySet()) {
539
540                     Client toClient = mapAllClients.get(key);
541                     if(toClient.getUserName().equalsIgnoreCase(fromClient.getUserName())) {
542                         continue;
543                     }
544                     DataPacket dataPacketBroadCast;
545                     try {
546
```

```
547                         dataPacketBroadCast = (DataPacket) dataPacket.clone();
548                         dataPacketBroadCast.setFromClient(fromClient);
549                         dataPacketBroadCast.setToClient(toClient);
550                         qSendPacket.add(dataPacketBroadCast);
551                     }
552                     catch (CloneNotSupportedException e) {
553 
554                         e.printStackTrace();
555                     }
556                     catch (Exception e) {
557 
558                         e.printStackTrace();
559                     }
560 
561                 }
562 
563                 break;
564 
565             case DataPacket.MESSAGE_TYPE_IMAGE_MESSAGE:
566 
567                 if(mapAllClients.containsKey((dataPacket.getToClient().getUserName()))) {
568 
569 
570                     Client toClient = mapAllClients.get(dataPacket.getToClient().getUserName());
571                     dataPacket.setToClient(toClient);
572                     qSendPacket.add(dataPacket);
573 
574                 }
575 
576                 break;
577             }
578 
579             break;
580         }
581 
582     }
583 
584     /**
585      *
```

```
586      * updateClientOn() -
587      *
588      * When the client Comes ONLINE
589      * read all the messages from the Database
590      * that were sent to the client while they were OFFLINE
591      *
592      * @param client
593      */
594     private void updateClientOn(Client client) {
595
596         System.out.println("Client Logged In : " + client.toString());
597         //TODO handle when the Client comes ONLINE
598
599         /*
600          * When the client Comes ONLINE
601          * read all the messages from the Database
602          * that were sent to the client while they were OFFLINE
603          */
604     }
605
606     /**
607      *
608      * updateClientOff() -
609      * When the client goes OFFLINE
610      * write all the messages to the Database
611      * that were sent and received by the client
612      * while they were ONLINE
613      *
614      * @param client
615      */
616     private void updateClientOff(Client client) {
617
618         System.out.println("Client Logged Out : " + client.toString());
619         //TODO handle when the Client comes OFFLINE
620
621         /*
622          * When the client goes OFFLINE
623          * write all the messages to the Database
624          * that were sent and received by the client
```

```
625          * while they were ONLINE
626          */
627     }
628
629     /**
630      * sendClientStatus() -
631      * This method sends to the client(s)
632      * the list of all the clients
633      * Who are ONLINE and who are OFFLINE
634      *
635      * @param isSendToAllClient
636      * @param specificClient
637      */
638     public void sendClientStatus(boolean isSendToAllClient, Client specificClient) {
639
640         Client severToClient = new Client(Server.SERVER_USERNAME);
641
642         Set<String> setAllClientEmail = null;
643         if(!isSendToAllClient) {
644             setAllClientEmail = new HashSet<>();
645             setAllClientEmail.add(specificClient.getUserName());
646         }
647         else {
648             setAllClientEmail = mapAllClients.keySet();
649         }
650
651         for(String key : setAllClientEmail) {
652
653             Client toClient = mapAllClients.get(key);
654
655             if(toClient.isOnline()) {
656
657                 DataPacket dataPacket = new DataPacket(severToClient, DataPacket.ACTION_TYPE_ONLINE);
658                 dataPacket.setToClient(toClient);
659
660                 String allClientData = new Gson().toJson(mapAllClients);
661                 dataPacket.setMessage(allClientData);
662
663                 qSendPacket.add(dataPacket);
```

```
664             }
665         }
666 
667     }
668 
669     /**
670      *
671      * broadcastClientStatus() - This method runs the thread threadBroadcastClientStatus and
672      * continuously send Online DataPacket (containing mapAllClients)
673      * to all the clients to notify all that the client who are ONLINE and who are OFFLINE.
674      *
675      */
676     private void broadcastClientStatus() {
677 
678         threadBroadcastClientStatus = new Thread("BroadcastClientStatus"){
679 
680             @Override
681             public void run() {
682 
683                 while(isService) {
684 
685                     Client client = new Client("server@server.com");
686                     sendClientStatus(true, client);
687 
688                     try {
689 
690                         Thread.sleep(5000);
691                     }
692                     catch(Exception e) {
693 
694                         e.printStackTrace();
695                     }
696                 }
697             }
698         };
699 
700         threadBroadcastClientStatus.start();
701     }
702 
```

```
703     /**
704      * recievePacketUDP() - This method runs the thread threadReceivePacketUDP
705      * and continuously receives UDP DataPacket from all the logged in client
706      * and process them to find the content of the packet and perform the necessary action
707      *
708      */
709     private void recievePacketUDP() {
710
711         threadReceivePacketUDP = new Thread("RecievePacketUDP"){
712
713             @Override
714             public void run() {
715
716                 while(isService) {
717
718                     byte[] data = new byte[1024*60];
719                     DatagramPacket datagramPacket = new DatagramPacket(data, data.length);
720
721                     try {
722
723                         server.getDatagramSocket().receive(datagramPacket);
724
725
726                         String received = new String(datagramPacket.getData(), 0, datagramPacket.getLength());
727
728                         DataPacket dataPacket = new Gson().fromJson(received, DataPacket.class);
729                         System.out.println(dataPacket);
730
731                         InetAddress inetAddress = datagramPacket.getAddress();
732                         int port = datagramPacket.getPort();
733
734                         System.out.println(dataPacket);
735                         String userName = dataPacket.getFromClient().getUserName();
736                         String id = "";
737                         String name = "";
738
739                         Client fromClient = new Client(userName, id, name, inetAddress, port);
740
741                         System.out.println(fromClient);
```

```
742                     processReceivedDatagramPacket(dataPacket, fromClient);
743
744                 }
745                 catch (IOException e) {
746
747                     e.printStackTrace();
748                 }
749
750
751                 try {
752
753                     Thread.sleep(500);
754                 }
755                 catch(Exception e) {
756
757                     e.printStackTrace();
758                 }
759             }
760         }
761     };
762
763     threadReceivePacketUDP.start();
764 }
765
766 /**
767  * sendPacketByUDP() - This methods sends the DataPacket to the
768  * respective client based on UDP DatagramPacket
769  * the DataPacket contains the address of the ToClient
770  *
771  * @param dataPacket
772  * @throws IOException
773  */
774 public void sendPacketByUDP(DataPacket dataPacket) throws IOException {
775
776     InetAddress inetAddress = dataPacket.getToClient().getInetAddress();
777     int port = dataPacket.getToClient().getPort();
778     byte[] data = dataPacket.toJSON().getBytes();
779     DatagramPacket datagramPacket = new DatagramPacket(data, data.length, inetAddress, port);
780
```

```
781         server.getDatagramSocket().send(datagramPacket);
782 
783     }
784 
785     /**
786      * sendPacket() - This method decides on
787      * which method to use to send the DataPacket to the client
788      *
789      * @param dataPacket
790      * @throws IOException
791      */
792     public void sendPacket(DataPacket dataPacket) throws IOException {
793 
794         if(dataPacket.getAction().equals(DataPacket.ACTION_TYPE_MESSAGE)) {
795             mapSentDataPacket.put(dataPacket.getId(), dataPacket);
796         }
797 
798         sendPacketByUDP(dataPacket);
799 
800     }
801 
802     /**
803      * resendDataPacketIfNoACKReceived() - This method runs the
804      * thread threadResendDataPacketIfNoACKReceived
805      * which Resends the data for which no ACK is received
806      * from the respective Client after a predefined amount of time
807      *
808      */
809     public void resendDataPacketIfNoACKReceived() {
810 
811         threadResendDataPacketIfNoACKReceived = new Thread("ResendDataPacketIfNoACKReceived"){
812 
813             @Override
814             public void run() {
815 
816                 while(isService) {
817 
818                     if(mapSentDataPacket.size() > 0) {
819 
```

```
820                         for(UUID sentDataPacketId : mapSentDataPacket.keySet()) {
821
822                             DataPacket sentDataPacket = mapSentDataPacket.get(sentDataPacketId);
823
824                             if(sentDataPacket.getTimestamp() - new Date().getTime() > 5000) {
825
826                                 sentDataPacket.setTimestamp(new Date().getTime());
827                                 sentDataPacket.incrementTimesResentDataPacket();
828                                 mapSentDataPacket.put(sentDataPacketId, sentDataPacket);
829
830                                 qSendPacket.add(sentDataPacket);
831                             }
832                         }
833                     }
834
835                     try {
836
837                         Thread.sleep(500);
838                     }
839                     catch(Exception e) {
840
841                         e.printStackTrace();
842                     }
843                 }
844             }
845         };
846
847         threadResendDataPacketIfNoACKReceived.start();
848     }
849
850     /**
851      * readFromFile() -
852      * This method write to the file
853      * all the user credentials that is in the mapUserCred
854      */
855     private synchronized void writeToFile() {
856
857         try (BufferedWriter bwBufferedWriter = new BufferedWriter(new FileWriter(FILENAME_USER_CREDENTIALS))) {
858
```

```
859             for(String username : mapUserCred.keySet()) {
860 
861                 String content = username + "\t" + mapUserCred.get(username);
862 
863                 bwBufferedWriter.write(content);
864                 bwBufferedWriter.newLine();
865             }
866             System.out.println("Write Successfully Completed");
867 
868         } catch (IOException e) {
869 
870             e.printStackTrace();
871 
872         }
873 
874     }
875 
876     /**
877      * readFromFile() -
878      * This method reads from the file
879      * all the user credentials that were saved
880      */
881     private synchronized void readFromFile() {
882 
883         try (BufferedReader bufferedReader = new BufferedReader(new FileReader(FILENAME_USER_CREDENTIALS))) {
884 
885             String strCurrentLine;
886 
887             while ((strCurrentLine = bufferedReader.readLine()) != null) {
888 
889                 String[] arrCurrentLine = strCurrentLine.split("\t");
890 
891                 String userName = arrCurrentLine[0];
892                 String password = arrCurrentLine[1];
893 
894                 Client savedClient = new Client(userName);
895                 mapAllClients.put(userName, savedClient);
896                 mapUserCred.put(userName, password);
897             }
```

```
898
899         } catch (IOException e) {
900             e.printStackTrace();
901         }
902
903     }
904
905     /***************************** Getters Setters *****************************/
906
907     public ConcurrentMap<UUID, DataPacket> getMapSentDataPacket() {
908         return mapSentDataPacket;
909     }
910
911     public void setMapSentDataPacket(ConcurrentMap<UUID, DataPacket> mapSentDataPacket) {
912         this.mapSentDataPacket = mapSentDataPacket;
913     }
914
915     public ConcurrentMap<UUID, DataPacket> getMapReceivedDataPacket() {
916         return mapReceivedDataPacket;
917     }
918
919     public void setMapReceivedDataPacket(ConcurrentMap<UUID, DataPacket> mapReceivedDataPacket) {
920         this.mapReceivedDataPacket = mapReceivedDataPacket;
921     }
922 }
923
```