MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DISTRIBUTED SYSTEMS

Assignment 3

Asynchronous Communication using Messaging

A 3.1: The Basics

Ioan Salomie
Marcel Antal
Teodor Petrican

Tudor Cioara
Claudia Daniela Pop

Ionut Anghel
Dorin Moldovan
Ciprian Stan

2018

Contents

1. Introduction

The main disadvantage of the classic request-reply communication paradigm used in the client-server architecture is the fact that it is synchronous communication: *the client process makes a request for the server and it is **blocked** until the server replies.* In the case in which the response from the server comes quickly, this drawback may not be perceived. Otherwise, the client execution is blocked and must wait for the server answer. Furthermore, the server may even be offline or too busy for serving the request, thus keeping the client unresponsive too.

In order to solve this problem, a new communication paradigm was defined. **The message passing inter-process communication paradigm** is an asynchronous communication mechanism that means that the message sending application does not have to wait for a response from the destination process or from any other application.

Message oriented middleware (MOM) is a middleware designed to facilitate the asynchronous inter-process communication. The distributed MOM systems have three major components:

- **Message Sender - process that creates and sends the message**
- **Message Repository – entity that stores the message until it is delivered**
- **Message Receiver – process that is the receiver of the message**

2. Problem Requirements

Suppose we are requested to create a distributed application with the following requirements:

- A client sends a message to a server
- The server may be offline or online when the message is being sent
- After the message was sent, the client must resume its processing without waiting for a response from the server
- The message is processed by the server when it becomes available

Imagine a real world system with the following requirements:

- The administrator of a web site that sells DVDs must access remotely the application that manages the central database where he keeps the information about available stocks
- The administrator adds a new DVD to the database
- Each time a new DVD is added, the application must send automatically notification e-mails to all the subscriber customers to notify them about the new item.

3. Problem Analysis

From the problem requirements we notice the following aspects:

- The information is stored in a central node that must be accessed remotely by clients. Consequently, the **chosen solution** will be a **distributed application** having **client-server architecture**.
- The client must continue its execution as soon as it sends the message, without waiting for the server to respond. The standard client-server architecture with the response-reply mechanism is not enough for fulfilling this requirement.

Consequently, the **chosen solution** will be a **distributed application** based on the **message passing inter-process communication paradigm.** The application will be based on a Message Oriented Middleware. This has 3 major components:

- **Message Sender**
- **Message Repository**
- **Message Receiver**

Each component communicates with the other using the basic **synchronous request-reply mechanism**. However, by introducing **the message repository component** between the **client** and the **server**, an **asynchronous communication** is created from two synchronous communication mechanisms.

4. Designing a Solution

We need to create a distributed application over the network based on the **message passing inter-process communication paradigm.** We start from the **client-server** software architecture and a request-reply communication paradigm and add an intermediate component: **the message repository**. Thus we have two client-server request-reply communications:

- Sender - Message repository
- Message repository – Receiver

An asynchronous communication mechanism is developed:

1. The Sender uses a synchronous request-reply mechanism to send the message to the Message Repository.
2. The Message Repository keeps the messages in the queue
3. The Receiver connects to the Message Repository using a synchronous request-reply mechanism to ask for messages.

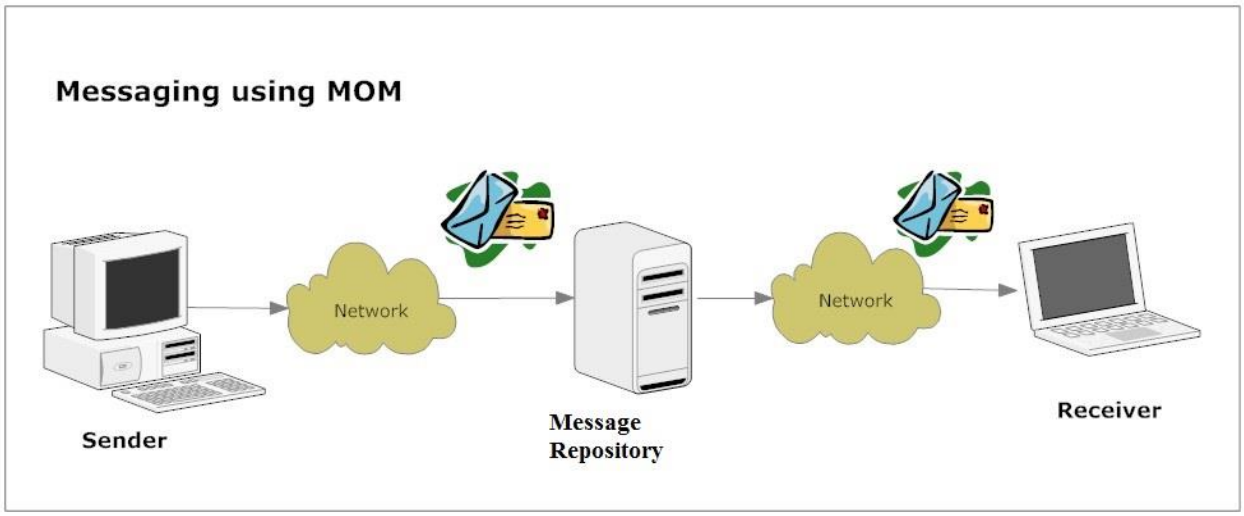

Figure 1. MOM software architecture

For the transport layer of the application we use **sockets** [1] that assure a two-way communication between the client and the server, thus allowing us to implement a synchronous request-reply communication method.

This architecture can be mapped on the requirements from section 2 as follows:

- The sender is the client application used by the administrator to introduce data regarding the DVDs
- The Message Repository is a special application where the sender connects to send the message (the characteristic of the new DVD)
- The receiver is a server that connects to the Message Repository, takes the message (the DVD) and starts sending e-mails to the subscribed customers

5. Implementing the Solution

The solution is implemented in 3 different modules, as seen in the conceptual architecture below:

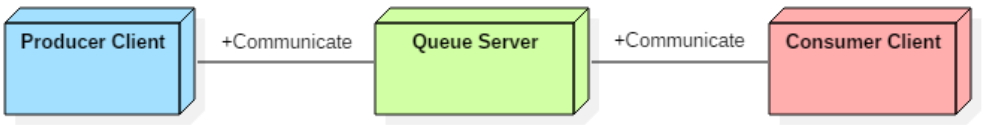

Figure 3. Conceptual architecture

There is a Producer Client, which has some data to process (in this case, message). It sends the data to a Queue Server: this component holds a queue with messages, and can push to/pop from the queue as required. The Consumer Client will request messages from the Queue Server, to process them.

Each module has the following architecture:

- **The Producer Client Application** - contains two classes:
  - **ClientStart** – class which contains the main method. Here, a QueueServerConnection instance is created and a loop sends some messages through it.
  - **QueueServerConnection** – class that contains the sockets connecting the client with the server and methods for communicating with it.
- **The Consumer Client Application –** contains two classes:
  - **ClientStart** – class which contains the main method. Here, a QueueServerConnection instance is created; an infinite loop will run asking for messages from the queue server, and processing them when available.
  - **MailService** – service class which provides a method for sending an email to a specified address.
  - **QueueServerConnection** – class that contains the sockets connecting the client with the server and methods for communicating with it.
- **The Queue Server Application –** contains three packages:
  - **Communication** – contains the classes dealing with the communication of the queue server with clients: Server (waiting for incoming connections), Session (dealing with one request), Message (the class representing the exchange mechanism of the communication).
  - **Queue** – contains the Queue class, which is the underlying mechanism; it is a BlockingQueue which allows for insertion and removal of elements to/from the structure.
  - **Start** – contains the ServerStart class, which is the starting point of the component.

5.2. Producer Client Application

Producer Client Application has two components: Producer Start Component and Producer Connection Component.

5.2.1. Producer Start Component

The ClientStart class contains code for creating a connection with the queue server (line 9) using the specified HOST and PORT values and it also contains code for sending messages to the server (lines 12-14). The method *main* sends five messages to be inserted in the queue server.

```
1. public class ClientStart {
2.         private static final String HOST = "localhost";
3.         private static final int PORT = 8888;
4.
5.         private ClientStart() {
6.         }
7.
8.         public static void main(String[] args) {
9.               QueueServerConnection         queue         =         new
   QueueServerConnection(HOST, PORT);
```

```
10.
11.             try {
12.                     for (int i=0;i<5;i++) {
13.                             queue.writeMessage("this is email number
    "+i);
14.                     }
15.             } catch (IOException e) {
16.                     e.printStackTrace();
17.             }
18.     }
19. }
```

Figure 4. ClientStart class Code Snippet

5.2.2. Producer Connection Component

The producer connection component is responsible for serving the connection between the client and the queue server. It contains two methods: (1) the first method (lines 17-36) sends requests to the server in order to insert a message in the queue and (2) the second method (lines 46-67) retrieves a message from the queue of the server.

```
1.  public class QueueServerConnection {
2.          private String host;
3.          private int port;
4.
5.          public QueueServerConnection(String host, int port) {
6.                  this.host = host;
7.                  this.port = port;
8.          }
9.
10.         /**
11.          * Sends a request to the server to insert a message into the
    queue.
12.          *
13.          * @param messageToSend the message to be inserted into the queue
14.          * @return the status of the operation (true == successful)
15.          * @throws IOException thrown if there is a problem with the
    connection
16.          */
17.         public boolean writeMessage(String messageToSend) throws
    IOException {
18.                 Socket clientSocket = new Socket(host, port);
19.                 ObjectOutputStream outToServer = new
    ObjectOutputStream(clientSocket.getOutputStream());
20.                 ObjectInputStream inFromServer = new
    ObjectInputStream(clientSocket.getInputStream());
21.                 outToServer.writeObject(new
    Message("SEND",messageToSend));
22.
23.                 Message response;
24.                 try {
25.                         response = (Message)inFromServer.readObject();
```

```
26.             } catch (ClassNotFoundException e) {
27.                   response = null;
28.                   e.printStackTrace();
29.             }
30.
31.             outToServer.close();
32.             inFromServer.close();
33.             clientSocket.close();
34.
35.             return                  (response!=null                  &&
    response.getType().equals("ACK"));
36.       }
37.
38.       /**
39.        * Retrieves a message from the queue of the server, by sending
    a "READ" request to it.
40.        * If the queue is empty, this method will hang until the queue
    will get a message. In
41.        * other words, this method will wait for the server to provide
    a message.
42.        *
43.        * @return the message from the queue, sent by the server
44.        * @throws IOException thrown if there is a problem with the
    connection
45.        */
46.       public String readMessage() throws IOException {
47.             Socket clientSocket = new Socket(host, port);
48.             ObjectOutputStream        outToServer        =        new
    ObjectOutputStream(clientSocket.getOutputStream());
49.             ObjectInputStream        inFromServer        =        new
    ObjectInputStream(clientSocket.getInputStream());
50.             outToServer.writeObject(new Message("READ",null));
51.
52.             Message response;
53.             try {
54.                   response = (Message)inFromServer.readObject();
55.             } catch (ClassNotFoundException e) {
56.                   response = null;
57.                   e.printStackTrace();
58.             }
59.
60.             outToServer.close();
61.             inFromServer.close();
62.             clientSocket.close();
63.
64.             if (response==null || !response.getType().equals("ACK"))
    return null;
65.
66.             return response.getContent();
67.       }
68. }
```

Figure 5. QueueServerConnection class Code Snippet

5.3. Consumer Client Application

Consumer Client Application has three components: Consumer Start Component, Consumer Service Component and Consumer Connection Component.

5.3.1. Consumer Start Component

This class contains the *main* (line 6) method which starts the Consumer Client application. The application contains an infinite loop that retrieves messages from the queue server and then sends the e-mails as they arrive. The line 16 should be uncommented and completed with the email address, the title of the email and a String message that represents the body of the message.

```
1.  public class ClientStart {
2.
3.          private ClientStart() {
4.          }
5.
6.          public static void main(String[] args) {
7.                  QueueServerConnection          queue          =          new
    QueueServerConnection("localhost",8888);
8.
9.                  MailService           mailService           =           new
    MailService("your_account_here","your_password_here");
10.                 String message;
11.
12.                 while(true) {
13.                         try {
14.                                 message = queue.readMessage();
15.                                 System.out.println("Sending mail "+message);
16.
           //mailService.sendMail("to_mail_address","Dummy                Mail
    Title",message);
17.                         } catch (IOException e) {
18.                                 e.printStackTrace();
19.                         }
20.                 }
21.         }
22. }
```

Figure 6. ClientStart class Code Snippet

5.3.2. Consumer Service Component

This component contains the class that sends the emails. It uses Gmail SMTP by default but the properties can be changed in the constructor if desired. The credentials must be the ones used for the connection to the SMTP server. The constructor (line 14) takes as arguments the *username* and the *password* of the user and the method *sendMail* sends the actual email with the destination, subject and content that are specified as parameters.

```
1.  public class MailService {
2.      final String username;
3.      final String password;
4.      final Properties props;
5.
6.      /**
7.       * Builds a mail service class, used for sending e-mails.
8.       * The credentials provided should be the ones needed to
9.       * autenthicate to the SMTP server (GMail by default).
10.      *
11.      * @param username username to log in to the smtp server
12.      * @param password password to log in to the smtp server
13.      */
14.     public MailService(String username, String password) {
15.         this.username = username;
16.         this.password = password;
17.
18.         props = new Properties();
19.         props.put("mail.smtp.auth", "true");
20.         props.put("mail.smtp.starttls.enable", "true");
21.         props.put("mail.smtp.host", "smtp.gmail.com");
22.         props.put("mail.smtp.port", "587");
23.     }
24.
25.
26.     /**
27.      * Sends an email with the subject and content specified, to
28.      * the address specified.
29.      *
30.      * @param to address to send email to
31.      * @param subject subject of the email
32.      * @param content content of the email
33.      */
34.     public void sendMail(String to, String subject, String content) {
35.         Session session = Session.getInstance(props,
36.                 new javax.mail.Authenticator() {
37.                     protected                    PasswordAuthentication
    getPasswordAuthentication() {
38.                         return   new   PasswordAuthentication(username,
    password);
39.                     }
40.                 });
41.
42.         try {
43.
44.             Message message = new MimeMessage(session);
45.             message.setFrom(new InternetAddress(username));
46.             message.setRecipients(Message.RecipientType.TO,
47.                     InternetAddress.parse(to));
48.             message.setSubject(subject);
49.             message.setText(content);
50.
51.             Transport.send(message);
```

```
52.
53.             System.out.println("Mail sent.");
54.         } catch (MessagingException e) {
55.             e.printStackTrace();
56.         }
57.     }
58.
59.
60. }
```

Figure 7. MailService class Code Snippet

5.3.3. Consumer Connection Component

The Consumer Connection Component contains a class that serves the connection between the client and the queue server. It contains two methods: (1) the first method (lines 17-36) allows the requests to be sent to the server and (2) the second method (lines 46-67) retrieves a message from the queue of the server. The constructor (lines 5-8) has two parameters: the host and the port.

```
1.  public class QueueServerConnection {
2.         private String host;
3.         private int port;
4.
5.         public QueueServerConnection(String host, int port) {
6.               this.host = host;
7.               this.port = port;
8.         }
9.
10.        /**
11.         * Sends a request to the server to insert a message into the
    queue.
12.         *
13.         * @param messageToSend the message to be inserted into the queue
14.         * @return the status of the operation (true == successful)
15.         * @throws IOException thrown if there is a problem with the
    connection
16.         */
17.        public boolean writeMessage(String messageToSend) throws
    IOException {
18.               Socket clientSocket = new Socket(host, port);
19.               ObjectOutputStream outToServer = new
    ObjectOutputStream(clientSocket.getOutputStream());
20.               ObjectInputStream inFromServer = new
    ObjectInputStream(clientSocket.getInputStream());
21.               outToServer.writeObject(new
    Message("SEND",messageToSend));
22.
23.               Message response;
24.               try {
25.                     response = (Message)inFromServer.readObject();
26.               } catch (ClassNotFoundException e) {
27.                     response = null;
```

```
28.                     e.printStackTrace();
29.             }
30.
31.             outToServer.close();
32.             inFromServer.close();
33.             clientSocket.close();
34.
35.             return                  (response!=null                 &&
    response.getType().equals("ACK"));
36.       }
37.
38.       /**
39.        * Retrieves a message from the queue of the server, by sending
    a "READ" request to it.
40.        * If the queue is empty, this method will hang until the queue
    will get a message. In
41.        * other words, this method will wait for the server to provide
    a message.
42.        *
43.        * @return the message from the queue, sent by the server
44.        * @throws IOException thrown if there is a problem with the
    connection
45.        */
46.       public String readMessage() throws IOException {
47.             Socket clientSocket = new Socket(host, port);
48.             ObjectOutputStream        outToServer        =        new
    ObjectOutputStream(clientSocket.getOutputStream());
49.             ObjectInputStream        inFromServer        =        new
    ObjectInputStream(clientSocket.getInputStream());
50.             outToServer.writeObject(new Message("READ",null));
51.
52.             Message response;
53.             try {
54.                   response = (Message)inFromServer.readObject();
55.             } catch (ClassNotFoundException e) {
56.                   response = null;
57.                   e.printStackTrace();
58.             }
59.
60.             outToServer.close();
61.             inFromServer.close();
62.             clientSocket.close();
63.
64.             if (response==null || !response.getType().equals("ACK"))
    return null;
65.
66.             return response.getContent();
67.       }
68. }
```

Figure 8. QueueServerConnection class Code Snippet

5.4. Queue Server Application

Consumer Server Application has three components: Queue Start Component, Queue Component and Queue Communication Component.

5.4.1. Queue Start Component

The *ServerStart* class contains a *main* method that creates a new *Server* object at the port that is specified as a parameter. By default the port is 8888.

```
1.  public class ServerStart {
2.
3.          private static final int PORT = 8888;
4.
5.          private ServerStart() {
6.          }
7.
8.          public static void main(String[] args) {
9.                  try {
10.                         new Server(PORT);
11.                         System.out.println("Queue server started.");
12.                 } catch (IOException e) {
13.                         e.printStackTrace();
14.                 }
15.         }
16. }
```

Figure 9. ServerStart class Code Snippet

5.4.2. Queue Component

The Queue class is a wrapper for a queue and the underlying queue is a BlockingQueue. If there are no elements in the queue then this type of queue will block and will wait for elements to retrieve. The first method (lines 14-16) inserts elements in the queue and the second method (lines 18-20) retrieves elements from the queue. The underlying mechanism is FIFO and it works in a push and pop manner.

```
1.  public class Queue {
2.      private static Queue queueInstance;
3.      private BlockingQueue<String> queue;
4.
5.      private Queue() {
6.          queue = new LinkedBlockingDeque<String>();
7.      }
8.
9.      public static Queue getInstance() {
10.         if (queueInstance==null) queueInstance=new Queue();
11.         return queueInstance;
```

```
12.     }
13.
14.     public void put(String message) throws InterruptedException {
15.         queue.put(message);
16.     }
17.
18.     public String get() throws InterruptedException {
19.         return queue.take();
20.     }
21. }
```

Figure 10. Queue class Code Snippet

5.4.3. Queue Communication Component

This component contains three classes: Message, Server and Session and is responsible for the communication with the queue.

The Message class contains a constructor (lines 5-8) that takes as parameters the type and the content of the message and it also contains getters and setters. This class is used for communication between the components and it represents an exchange mechanism between the queue server and the clients. There are four types of messages which are illustrated in the table below and the message also has an associated content.

Table I Message Types

Message Type	Description
SEND	Inserts content into the queue.
READ	Retrieves content from the queue.
ACK	The operation is successful on the server side.
ERR	The operation fails on the server side.

```
1. public class Message implements Serializable {
2.     private String type;
3.     private String content;
4.
5.     public Message(String type, String content) {
6.         this.type = type;
7.         this.content = content;
8.     }
9.
10.    public String getType() {
11.        return type;
12.    }
13.
14.    public void setType(String type) {
15.        this.type = type;
16.    }
17.
```

```
18.     public String getContent() {
19.         return content;
20.     }
21.
22.     public void setContent(String content) {
23.         this.content = content;
24.     }
25. }
```

Figure 11. Message class Code Snippet

The Server class has the following properties: (1) it creates socket that accepts connections and (2) it creates a thread which deals with the communication with the client. The constructor (lines 9-12) creates a socket object to listen to and accept connections. The *run* method (lines 17-30) contains a while loop that runs continuously, accepts connections from the clients and it creates and starts a new thread for dealing with the client messages.

```
1.  public class Server implements Runnable {
2.         private ServerSocket serverSocket;
3.
4.         /**
5.          * Create a socket object from the ServerSocket to listen to and
    accept connections
6.          * @param port the port on which the ServerSocket will be bound
    to
7.          * @throws IOException
8.          */
9.         public Server(int port) throws IOException {
10.              serverSocket = new ServerSocket(port);
11.              new Thread(this).start();
12.        }
13.
14.        /**
15.         * Accepts connections from clients and assigns a thread to deal
    with the messages from and to the respective client.
16.         */
17.        public void run() {
18.              while (true) {
19.                   try {
20.                        synchronized (this) {
21.                             Socket clientSocket;
22.                             clientSocket = serverSocket.accept();
23.                             Session      cThread      =      new
    Session(clientSocket);
24.                             cThread.start();
25.                        }
26.                   } catch (IOException e) {
27.                        e.printStackTrace();
28.                   }
29.              }
30.        }
31.
```

```
32. }
```

Figure 12. Sever class Code Snippet

Session class deals with the connection to a client: (1) receiving of messages, (2) decoding of messages and (3) sending of a response. The *run* method (lines 18-55) waits for a message from the client and treats the message according to its type. If the message has the type SEND then it is inserted in the queue and acknowledgement message with null body is sent to the client. Otherwise, if the message has the type READ then a message is retrieved from the queue and an acknowledgement message with the content of that message is sent to the client.

```
1.  public class Session extends Thread {
2.
3.          private Socket clientSocket;
4.          private ObjectInputStream inFromClient;
5.          private ObjectOutputStream outToClient;
6.
7.          public Session(Socket cSocket) {
8.                this.clientSocket = cSocket;
9.                try {
10.                     inFromClient                 =                 new
    ObjectInputStream(clientSocket.getInputStream());
11.                     outToClient                  =                 new
    ObjectOutputStream(clientSocket.getOutputStream());
12.               } catch (IOException e) {
13.                     e.printStackTrace();
14.               }
15.         }
16.
17.         @Override
18.         public void run() {
19.               Message messageReceived;
20.
21.               try {
22.                     // Wait for message from client
23.                     messageReceived            =            (Message)
    inFromClient.readObject();
24.
25.                     //  Treat  messages  according  to  the  type  of  the
    message
26.                     switch (messageReceived.getType()){
27.                           case "SEND":
28.                                 try {
29.                                       //insert  the  message  into  the
    queue
30.
          Queue.getInstance().put(messageReceived.getContent());
31.                                       sendMessageToClient(new
    Message("ACK", null));
32.                                 } catch (InterruptedException e) {
33.                                       e.printStackTrace();
34.                                       sendMessageToClient(new
    Message("ERR", null));
```

```
35.                                     }
36.                                     break;
37.                               case "READ":
38.                                     try {
39.                                           //retrieve a message from the
   queue
40.                                           //since the underlying queue is
   a BlockingQueue, this method call will wait if the queue is empty
41.                                           String        content        =
   Queue.getInstance().get();
42.                                           sendMessageToClient(new
   Message("ACK",content));
43.                                     } catch (InterruptedException e) {
44.                                           e.printStackTrace();
45.                                           sendMessageToClient(new
   Message("ERR",null));
46.                                     }
47.                                     break;
48.                         }
49.
50.                   } catch (ClassNotFoundException | IOException e) {
51.                         e.printStackTrace();
52.                   }
53.
54.                   closeAll();
55.             }
56.
57.             public void sendMessageToClient(Message messageToSend) {
58.                   try {
59.                         outToClient.writeObject(messageToSend);
60.                   } catch (IOException e) {
61.                         e.printStackTrace();
62.                   }
63.
64.             }
65.
66.             public void closeAll() {
67.                   try {
68.                         // Close the input stream
69.                         if (inFromClient != null) {
70.                               inFromClient.close();
71.                         }
72.                         // Close the output stream
73.                         if (outToClient != null) {
74.                               outToClient.close();
75.                         }
76.                         // Close the socket
77.                         if (clientSocket != null) {
78.                               clientSocket.close();
79.                         }
80.                   } catch (IOException e) {
81.                         e.printStackTrace();
82.                   } finally {
83.                         inFromClient = null;
84.                         outToClient = null;
```

```
85.                     clientSocket = null;
86.                 }
87.         }
88. }
```

Figure 13. Session class Code Snippet

5.5. Application Sequence Diagram

In the following section we will present the functionality of the client and server applications with the help of sequence diagrams. There are 2 major components that are running asynchronously with regards to one another (the producer client and the consumer client); the queue server is the means of communication between them.

Therefore, two sequence diagrams will be presented, one with the workflow of the producer client and the queue server and one with the workflow of the consumer client and the queue server.

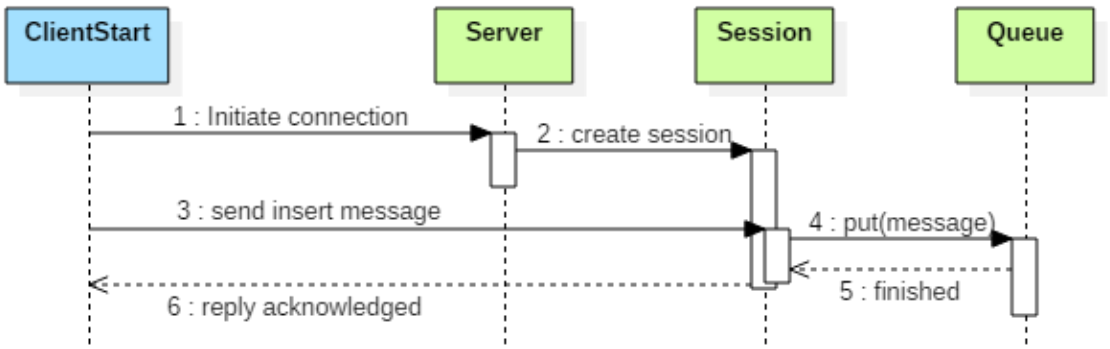

Figure 14. Sequence diagram for message insertion in queue (Producer Client)

The sequence diagram in the figure above comprises of the following steps:

1. The client initiates a connection with the queue server.
2. The server creates a Session (thread) to handle the client.
3. Client sends the actual request and the message to be inserted into the queue.
4. Server actually inserts the message into the queue.
5. Insertion executed successfully.
6. Return an acknowledged message, notifying that the operation was successful.

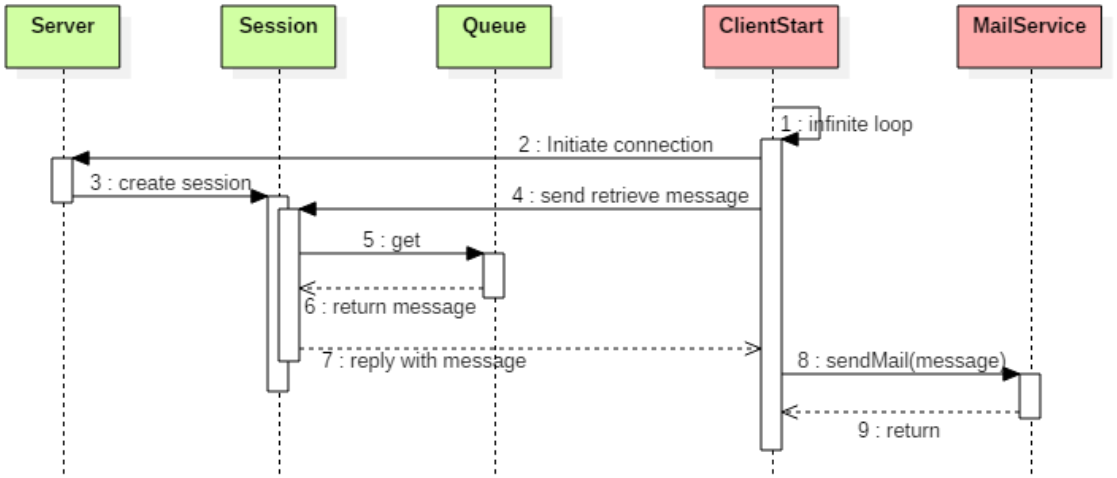

Figure 15. Sequence diagram for message retrieval from queue (Consumer Client)

The next sequence diagram refers to the flow of the consumer client:

1. The client has an infinite loop which executes readMessage (steps 1-7) -> processMessage (steps 8-9).
2. The client initiates a connection with the queue server.
3. The server creates a Session (thread) to handle the client.
4. Client sends the actual request (to retrieve a message from the queue).
5. Message is retrieved from the queue (popped, i.e. eliminated from the queue).
6. Return the message.
7. Reply to the client with the respective message.
8. Call the sendMail() function, to send a mail with the message (content) got from the queue.
9. Return to the client main function.

6. Running the Solution

1. Setup GIT and download the project from https://bitbucket.org/utcn_dsrl/ds.handson.assignment3.git
   - Create an empty local folder in the workspace on your computer
   - Right-click in the folder and select **Git Bash**
   - Write commands:
     - git init
     - git remote add origin https://bitbucket.org/utcn_dsrl/ds.handson.assignment3.git
     - git pull origin master
2. Import the project into Eclipse: FILE-> Import->Maven-> Existing Maven Projects-> Browse for project in the folder created at step 1
3. Run the project:
   - Run the ServerStart class from the queue server module, package start
   - Run the ClientStart class from the consumer client module, package start

- Run the ClientStart class from the producer client module, package start

4. The application will perform the following: the consumer client, once it receives elements from the queue server, will process them (i.e. send mails with the messages). The producer client will send messages to the queue server to be inserted in the queue (i.e. to be processed).
5. By default, consumer client will only print to STDOUT the messages. In order to actually send mails, a valid Gmail username and password will need to be specified in ClientStart class from consumer client module.
6. Access for less secure apps must be switched to on for the Gmail account: https://www.google.com/settings/security/lesssecureapps

7. Reinforcement Learning

1. Run and study the project and identify the following features:
   - Message Sending
   - Message Receiving
   - Message Queue
2. Complete the project by adding the following functionalities:
   - Create the class DVD:
     - Title: string
     - Year: int
     - Price: double
   - The producer sends a DVD to the queue
   - The Consumer sends an email with each new DVD added
   - Add another message consumer that creates a text file with for each message and prints the content of the message in the text file

8. Bibliography

1. Java Sockets: https://docs.oracle.com/javase/tutorial/networking/sockets/
2. JMS: https://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html