Laboratory Practices

JMS: Java Message Service (3 sessions)

Concurrency and Distributed Systems

Introduction

This practice aims to develop a simple application using the **Java Message Service** (JMS) messaging service, as well as to install, configure and run a JMS provider. In particular, the communication component of an instant messaging application, which we have called "CSD Messenger", will be developed.

Given the introductory nature of this practice, basic aspects such as connecting to the JMS provider and sending and receiving messages will be covered. Other aspects that would be fundamental for the development of a real application, such as security and fault tolerance, have been intentionally left out for the sake of simplicity.

The estimated duration of this lab practice is three weeks. Each week you should attend a lab session where you will be able to discuss your progress with your teacher. Please note that you will need to employ some time from your personal work to complete the lab practice.

This practice is prepared to be carried out from the **Windows** systems of the DSIC laboratories. If you wish to do it on your personal Windows system, you will need to have installed version 1.8 of Oracle's Java JDK and adapt the names of the directories to your installation.

Activity 1

This activity aims to install, configure, and run a JMS provider. The JMS provider to be installed is Apache ActiveMQ Artemis (simply ***Artemis*** in the rest of this document).

Artemis is a set of libraries and applications developed in Java that is distributed as a compressed archive. For this practice, it is available in the Lessons of practice 5 on the PoliformaT site of the course (file "apache-artemis-1.4.0-bin.zip").

To avoid disk space quota problems, the content of this file is already unzipped in the drive asig(\\fileserver.dsic.upv.es) (M:)

```
M:\ETSINF\csd\jms\artemis
```

As mentioned before, in this activity we are going to install, configure and run the JMS Artemis provider. Thus, we will first create an instance of the messaging broker, define the ports to be used by the broker and launch the broker for execution. To do this, we will follow the instructions below.

NOTE: A brief description of the Artemis commands that will be necessary for our practice is included as an appendix to this document.

IMPORTANT: In all the file modifications indicated below, these modifications must be made "by hand" (writing directly in the file) and not by copy-paste from the practice bulletin (as then non-visible characters could be copied, which would generate errors in the creation and execution of the broker).

Instructions

1. Open a file explorer and go to M:\ETSINF\csd\jms\artemis\bin
2. Create an instance of the messaging broker in a directory on your disk (W:) following these steps:
    - Create the directory structure `W:\csd\jms`
    - From the file explorer opened in step 1, using the "File" menu, open a "command prompt". This will open a new window with the basic Windows command prompt, already located in the working directory `M:\ETSINF\csd\jms\artemis\bin`
    - Write the command:
      ```
      artemis create W:\csd\jms\csdbroker
      ```
    - Reply with **admin**, **admin**, **admin** and **Y** to the four questions asked by the previous command. *(By doing so you will define the user, password and role to be used for anonymous connections, which you will also be enabling. Take into account that nothing is displayed while you type the password)*
3. Since you will do this practice in a computer shared with other users (as it is carried out via the DSIC Remote Desktop), it is necessary to decide in a coordinated way which ports will be used by your broker, so that they are not the same as those used by the brokers launched by other users. The lab professor will indicate you which ports to use. In particular, you will be assigned a port which we will call PORTNUMBER, and your broker will use that port and the next one, which we will call PORTNUMBER+1.

4. Edit the file “W:\csd\jms\csdbroker\etc\broker.xml”, and change the port 61616 to PORTNUMBER in the following line:

```
<acceptor
name="artemis">tcp://0.0.0.0:61616?tcpSendBufferSize=1048576;tcpReceiv
eBufferSize=1048576</acceptor>
```

The broker.xml file is the central server configuration file containing the core element with the configuration of the main server. Among other things, it allows the definition of the so-called ***acceptors***. Each acceptor defines a way in which connections can be made to the Artemis server. In our case, we have defined an acceptor that uses Artemis to listen for connections on port PORTNUMBER, also indicating the sizes of the TCP send buffer and the TCP receive buffer in bytes (1048576bytes = 1Mbyte).

5. In the same file, remove the rest of the lines with elements `<acceptor … </acceptor>`

We remove these lines, as we are only going to use the acceptor defined in the previous step.

6. Edit the file "W:\csd\jms\csdbroker\etc\bootstrap.xml”, and change the port 8161 to PORTNUMBER+1 in the following line:

```
<web bind="http://localhost:8161" path="web">
```

The *bootstrap.xml* file is the default configuration file used to start the server. Among other things, this file indicates the broker configuration file (broker.xml).

7. Now that you have configured your broker, you can run it with the following command at a shell prompt located at W:\csd\jms\csdbroker\bin (*open a "command prompt" from the "File" menu of the file explorer once you are in the directory W:\csd\jms\csdbroker\bin)*

- `artemis run`

You will see the running broker on the screen:

```
     _        _               _
    / \  ____| |_  ___ __  __(_) _____
   / _ \|  _ \ __|/ _ \  \/  | |/  __/
  / ___ \ | \/ |_/  __/ |\/| | |\___ \
 /_/   \_\|   \__\____|_|  |_|_|/___ /
 Apache ActiveMQ Artemis 1.4.0
```

Verification

When you launch the broker, if everything has worked correctly, a message will appear on the screen informing that the Artemis server is running.

If this is not the case, you should repeat the previous process, making sure that everything is written correctly. If you have any doubts, consult your teacher.

Activity 2

This activity aims to test the functioning of the instant messaging system provided in practice, called CSD Messenger.

Previous concepts

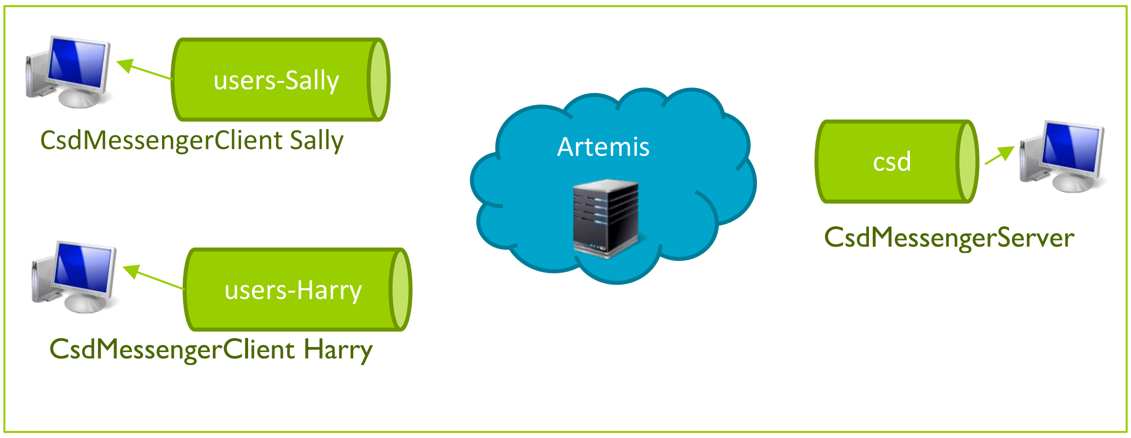
CSD Messenger is basically made up of two applications: a server application (CsdMessengerServer) and a client application (CsdMessengerClient). CsdMessengerServer is in charge of creating the JMS queues for the clients and informing each client of the names of the rest of the clients using the service.

At any given moment, there must be only one active CSDMessengerServer application, but there can be several active CSDMessengerClient applications, one for each user connected to the system. When CSDMessengerClient is executed, the user name must be provided as an argument.

The CSDMessengerServer application consumes messages from the queue named "csd", while each CSDMessengerClient application consumes messages from the queue "users-USERNAME" (where USERNAME is replaced by the corresponding user name).

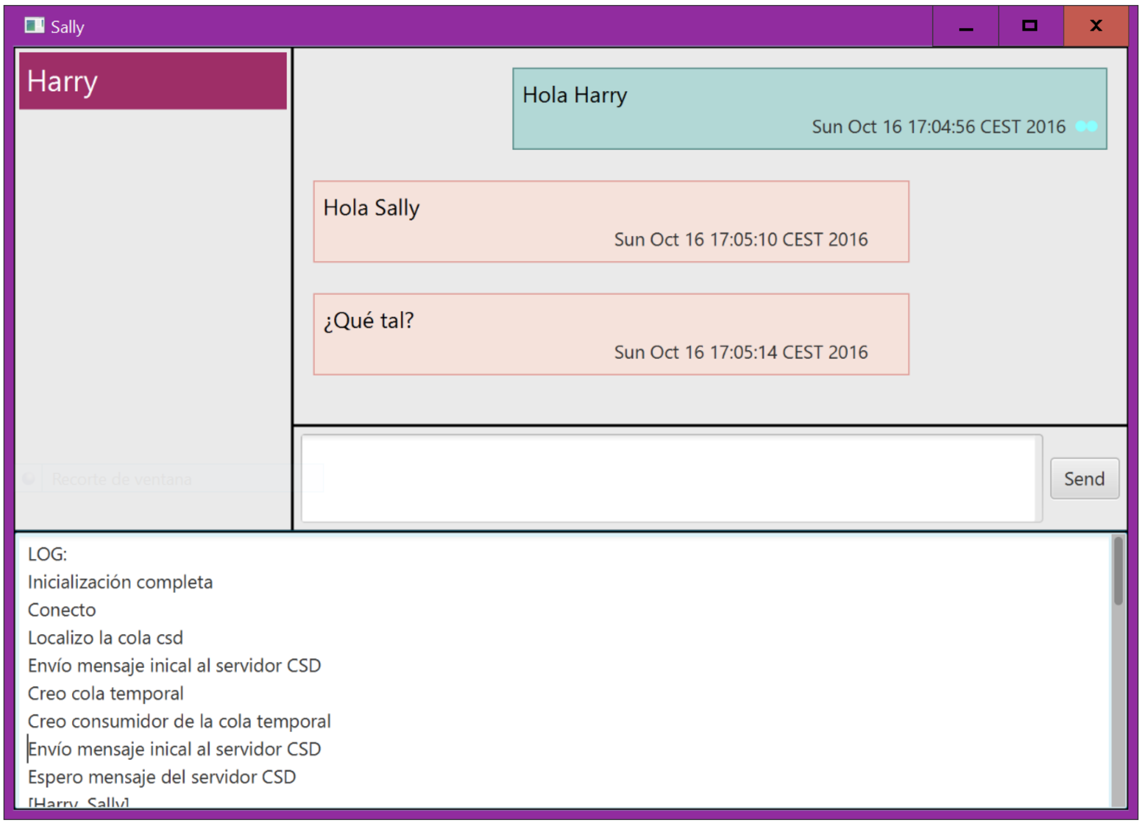
The queue "csd" must exist previously, for which reason it has to be created as part of the Artemis configuration. The queues "users-USERNAME" are created by CSDMessengerServer on demand when a user connects to the system for the first time.

The following figure summarizes the previously presented components, assuming Sally and Harry as users:



CsdMessengerServer is a console-mode Java application that shows on the screen a record of the requests received at the queue "csd".

CsdMessengerClient is a Java application with a graphical user interface, developed with JavaFX. Its look is shown in the following figure.



Hola Harry
Sun Oct 16 17:04:56 CEST 2016

Hola Sally
Sun Oct 16 17:05:10 CEST 2016

¿Qué tal?
Sun Oct 16 17:05:14 CEST 2016

```
LOG:
Inicialización completa
Conecto
Localizo la cola csd
Envío mensaje inical al servidor CSD
Creo cola temporal
Creo consumidor de la cola temporal
Envío mensaje inical al servidor CSD
Espero mensaje del servidor CSD
[Harry, Sally]
```

The elements of this interface are:

- Left panel. It shows the users known to the system. The first user to connect will see this panel empty. At least one other user is required to be logged in for the list of known users to be displayed.
- Right panel. It shows the list of messages in the conversation with the user selected on the left panel. It contains a text area to write a message and a button to send it (it can also be sent by pressing the Enter key).
- Lower panel. It shows a log with the application informative messages.

Instructions

1. Download the files "CsdMessengerServer.jar", "CsdMessengerClient.jar" and "jndi.properties" from Lessons - Practice 5. Save them in: W:\csd\jms

2. “CsdMessengerServer.jar” and “CsdMessengerClient.jar” contain the above-mentioned server and client applications. Before executing them, it is necessary to configure the port in which the JMS broker is listening. To do that, you must:

   2.1. Edit the file “jndi.properties” and change the port 61616 to PORTNUMBER in the line:
   ```
   connectionFactory.ConnectionFactory=tcp://localhost:61616
   ```

   2.2. Add the modified “jndi.properties” file to the .jar archives corresponding to the server and client applications (*open a "command prompt" from the "File" menu of the file explorer once located in the W:\csd\jms directory and type*):
   ```
   jar uvf CsdMessengerServer.jar jndi.properties
   jar uvf CsdMessengerClient.jar jndi.properties
   ```

3. Now, it is necessary to change the broker configuration in order to adapt it to the application that we want to develop. To do that:

   3.1. Stop Artemis by pressing Control-C in the running window.
   3.2. Create the “csd” queue. To do that, edit the file: "W:\csd\jms\csdbroker\etc\broker.xml” and add an element <queue> as shown next:

```
<jms xmlns="urn:activemq:jms">
...
   <queue name="csd"><durable>true</durable></queue>
</jms>
```

   With this configuration, we create a queue named “csd” that persists even when the JMS broker is stopped (property "durable").

   3.3. Disable the automatic creation of queues. To do that, edit the same file as in the previous step, adding an element <auto-create-jms-queues> with the value `false` as shown below:

```
<address-settings>
   <address-setting>
      ...
      <auto-create-jms-queues>false</auto-create-jms-queues>
   </address-setting>
</address-settings>
```

   In Artemis, by default, when a client references a JMS queue, the queue is immediately created automatically. That feature is unsuitable for the messaging service implemented in this practice because the creation of the JMS queues must be governed by CsdMessengerServer. Therefore, we have removed the automatic auto-creation of queues.

   3.4. Execute the broker again:
```
artemis run
```

Verification

Now everything is ready to try the applications. To do that:

1. Execute the CSD Messenger server in another terminal
```
java -jar CsdMessengerServer.jar
```
2. Execute a client in another terminal, e.g. with the user Sally:
```
java -jar CsdMessengerClient.jar Sally
```
3. And execute another client in another terminal, e.g. Harry:
```
java -jar CsdMessengerClient.jar Harry
```
4. You can execute more clients with other names, if you want to.

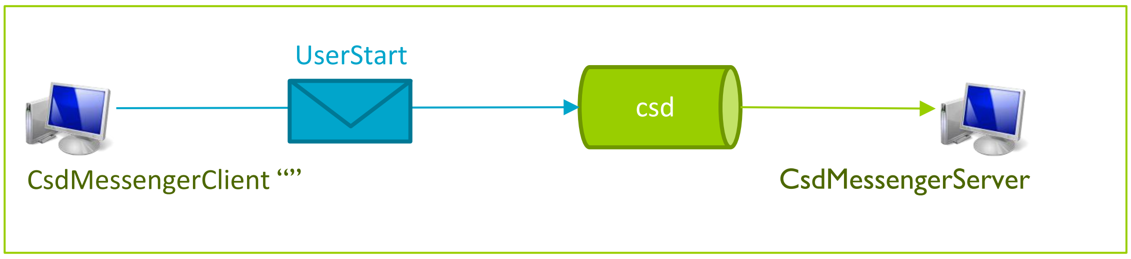
Use the client applications to chat. Check that it is possible to send messages to a user even if the user is not connected at that moment, and that the messages are delivered when the user gets connected later. This is due to the fact that the messages are stored in the queues of the JMS provider. For example, close Sally’s client application and then send some messages from Harry to Sally. When opening Sally’s client application again, you should see that Sally then receives the messages sent while she was not connected.

Activity 3

You have available a partial implementation of the application CsdMessengerClient, named **FuentesCsdMessenger**. In this case, the user interface component is totally implemented, but the communication component is not, thus no sending or reception of messages takes place. This activity is intended to get the FuentesCsdMessenger implementation to send a message to CsdMessengerServer.

The first task the communication component must do is sending a JMS message to CsdMessengerServer, in order to notify that a user is connecting to the system.

For the moment, in order not to complicate this activity excessively, the message will not contain the user name. Thus, CsdMessengerServer will not reply to that message, and you do not need to prepare your code to deal with that reply (that will be done in the next activity).



Instructions

1. Download the files “**FuentesCsdMessenger.jar**” and “**ArtemisLib.jar**” from Lessons - Practice 5
2. If you are using BlueJ, open it and select the menu entry Tools/Preferences/Libraries (Herramientas/ Preferencias/Librerías) and add the library “ArtemisLib.jar”. Restart BlueJ to apply the change. If you use another programming environment, add this library as required in that environment.
3. Open the file “FuentesCsdMessenger.jar” with your Java programming environment. A folder named “FuentesCsdMessenger” will have been created.
4. Use a text editor to edit the file “jndi.properties”, changing the port 61616 to PORTNUMBER, like you did in last activity.
5. Using your Java programming environment (e.g. BlueJ), edit the class **comm.Communication**.
6. In that class, implement the following steps in the method **initialize**():
   (NOTE: the code of the class CsdMessengerServer contains examples that are very similar to what is asked here. Additionally, you have links in brackets to the Java documentation of the method to be used)
   6.1. Initialize a JNDI context (javax.naming.InitialContext()).
   6.2. Set up the connection with the broker (javax.naming.InitialContext.lookup()).
   6.3. Create a JMS context (javax.jms.ConnectionFactory. createContext()).
   6.4. Create a *producer* associated to the previous context:
   (javax.jms.JMSContext.createProducer()).

6.5. Create an object of the class *messageBodies.UserStart*, passing an empty string as the argument of the object constructor. For example:

```
UserStart us=new UserStart("");
```

6.6. Construct a JMS message and assign the previous object to it (javax.jms.JMSContext.createObjectMessage()).

6.7. Create a queue associated to the name "dynamicQueues/csd" (javax.naming.InitialContext.lookup()).

6.8. Send the message created in step 6.6 to the queue created in step 6.7, using the producer created in step 5.4 (javax.jms.JMSProducer.Send())

Verification

Make sure that CsdMessengerServer is running.

Execute the method `ui.CsdMessengerClient.main(),` using the user name that you prefer as the argument.

A window should appear with several warnings and also a client GUI showing only the message “LOG: Inicialización Completa”.

On the other hand, you should be able to see the following line in the terminal where you executed CsdMessengerServer:

```
"He recibido una petición UserStart anónima. No hago nada"
```

Activity 4

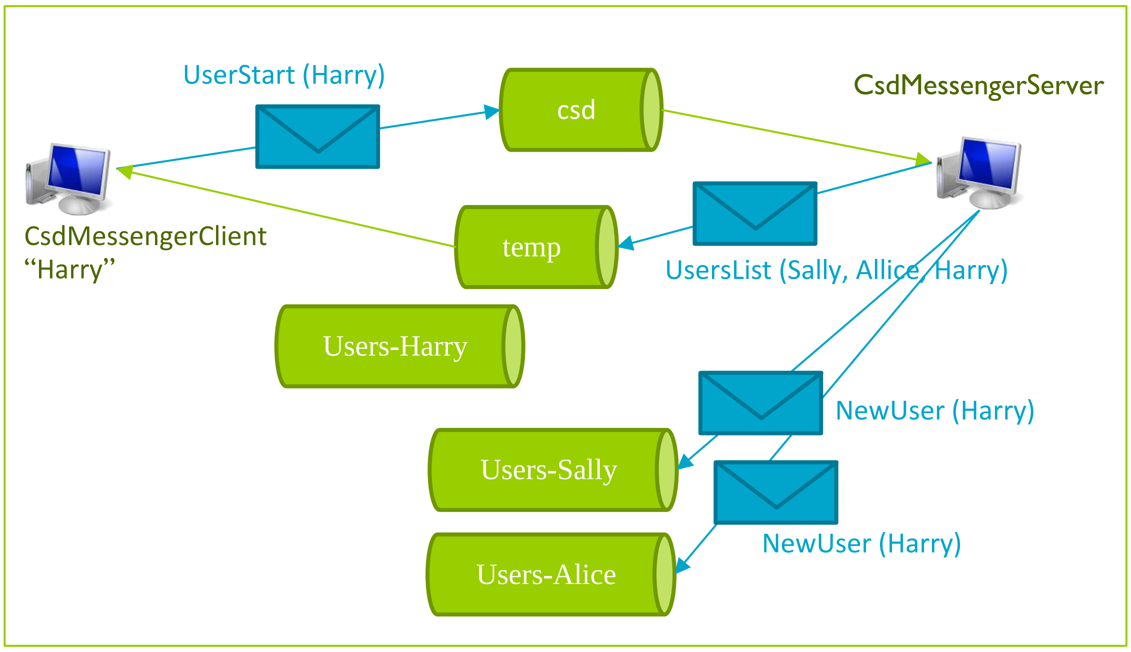
In this activity, the aim is to complete the message exchange protocol started in the previous activity, taking care of the response of CsdMessengerServer to the notification that a user connects.

Previous concepts

When CsdMessengerServer receives a message of type UserStart and the user name is correctly indicated, it performs the following actions:

- If it is the first time that the indicated user connects:
  - Creates the JMS queue “users-USERNAME”.
  - Notifies the rest of the known users of the existence of a new user.

- It sends a reply to the queue specified in the property “ReplyTo” of the received message, with a list of known users containing all the users that have previously executed the client application.



The purpose of the messages sent by CsdMessengerServer is that the client applications update the list of known users on their left panel.

Using a temporary queue is a common pattern in this type of applications. In this case, this is done because the first time a user connects there is no "users-USERNAME" permanent queue for the user, but a queue is needed to wait for a reply.

Instructions

1. Substitute the lines of code you had written to implement steps 6.5 and 6.6 in the previous activity, to do the following (the rest of the steps should not be modified):
   1.1. Construct a JMS message with an object of class messageBodies.UserStart, using the name of the connecting user (return value from method ui.API.getMyName()) as argument to the object constructor.
   1.2. Create a JMS temporary queue (javax.jms.JMSContext.createTemporaryQueue()())
   1.3. Assign the temporary queue to the ReplyTo property of the JMS message (javax.jms.Message.setJMSReplyTo())
2. After all the lines of code written for the previous activity, implement the following steps:
   2.1. Create a *consumer* associated to the temporary queue (see javax.jms.JMSContext.createConsumer()).
   2.2. Wait for the arrival of a message at the temporary queue (javax.jms.JMSConsumer.receive()). That message will contain an object of class UsersList, containing a string array with the lists of known users, which can be obtained with the method `getUsers()`.
   2.3. Call method `ui.API.updateUserList()`, passing the received array as argument.

Verification

Make sure that CsdMessengerServer is running.

Execute the method `ui.CsdMessengerClient.main()`, using the user name that you prefer as the argument. The left panel will show the list of known users.

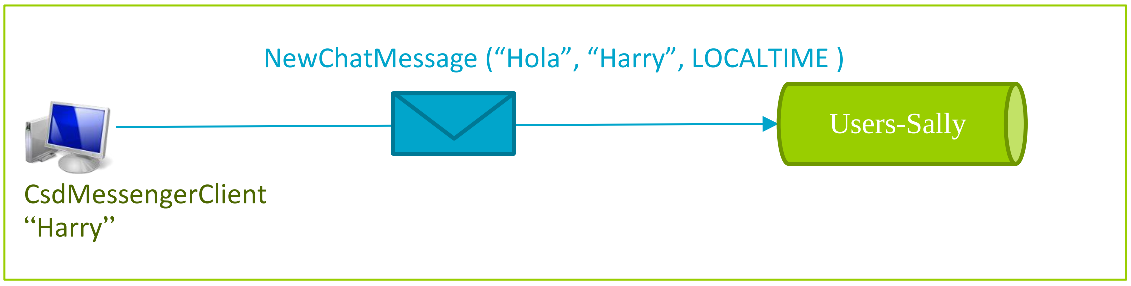
Activity 5

This activity is intended to send a JMS message to the queue of the user who is selected in the left pane when the Send button or the Enter key are pressed.

Previous concepts

The user interface, on detecting that the Send button or the Enter key have been pressed, requests from the communication component the sending of a message of type NewChatMessage, specifying the message text, the sender's username and the local time for the sender.

That message is delivered to the JMS queue of the user selected on the left panel. The message will be consumed immediately or it will remain in the queue, depending on whether the user is connected or not at that moment.



Instructions

1. Implement the following steps in the method comm.Communication.sendChatMessage:
   1.1. Search for the queue associated to the name "dynamicQueues/users-" + destUser, where destUser is a parameter of the method `sendChatMessage` (javax.naming.InitialContext.lookup()).
   1.2. Create an object of the class messageBodies.NewChatMessage, using as arguments to the object constructor the values text, `ui.API.getMyName()` and timestamp.
   1.3. Construct a JMS message and assign the previous object to it, using the `createObjectMessage` method (javax.jms.JMSContext.createObjectMessage()).
   1.4. Send the message created in the previous step, using the same *producer* used in activity 3 (javax.jms.JMSProducer.Send()).

Verification

Execute the complete client application:

```
      java -jar CsdMessengerClient.jar "Sally"
```

Execute the method `ui.CsdMessengerClient.main()`, using as argument a user name different from Sally. In your application, select Sally and send her a message. In Sally's application you should see the message that you have sent.

If Sally replies, you will not be able to see the message, because your application does not yet deal with incoming messages from other users. You will do that in the next activity.

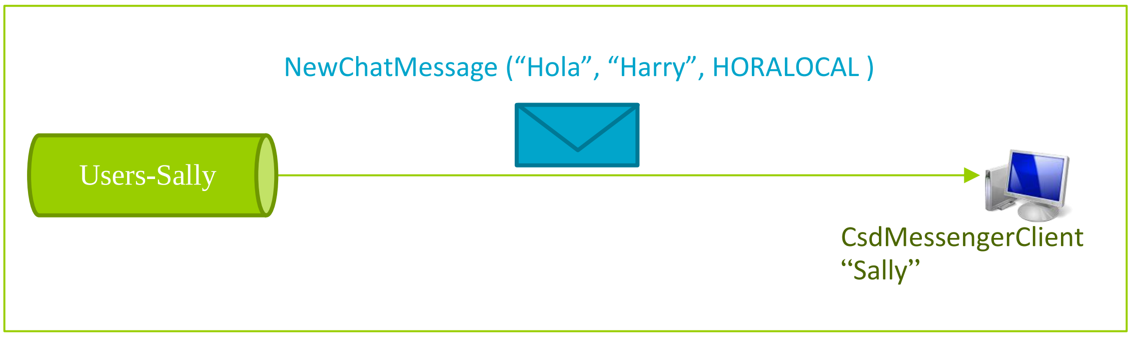
Activity 6

This activity will process incoming chat messages from other users.

Previous concepts

As you could see in the previous activity, a chat message arrives at the JMS queue of the destination user. The application CsdMessengerClient must be permanently listening for incoming messages in that queue, and when a message arrives it must notify the user interface component, so that the message can be shown on the right panel.

Since the application must be permanently listening for incoming messages, while at the same time respond to user interaction, the message receipt must be carried out in a different execution thread from the one in charge of the user interface. Additionally, because it is a different thread, it is necessary to create a new JMS context for that thread, and also a new *producer* and a new *consumer*, given that JMS contexts are designed to be used from a single thread.



Instructions

1. Implement the following steps in the method `comm.Communication.run()`:
   1.1. Create a JMS context, from the JMS context created in activity 3 (javax.jms.JMSContext.createContext(), using the parameter value JMSContext.AUTO_ACKNOWLEDGE)
   1.2. Create a *producer* associated to the previous context (javax.jms.JMSContext.createProducer()).
   1.3. Create a queue associated to the name "dynamicQueues/users-" + ui.API.getMyName() (javax.naming.InitialContext.lookup()).
   1.4. Create a *consumer* for the previous queue, associated to the context created in step 1.1. (javax.jms.JMSContext.createConsumer()).
   1.5. Inside a loop, wait permanently for the arrival of messages at the previously created queue (javax.jms.JMSConsumer.receive()).
   1.6. If the object contained in the received message is an object of the class `messageBodies.NewChatMessage`:
      1.6.1. Notify the user interface that a new message has arrived, with the method chatMessageReceived of our API (`ui.API.chatMessageReceived()`);
   1.7. If the object contained in the received message is an object of another class
      1.7.1. Show the name of that class in the Log

Verification

Execute the complete client application:

```
java -jar CsdMessengerClient.jar Sally
```

Execute the method `ui.CsdMessengerClient.main()`, using as argument a user name different from Sally. Now, not only Sally should see the messages that your application sends to her, but you should also see in your application the messages sent by Sally.

Activity 7

This activity aims to complete the client application. The aspects that have not been implemented yet are:

- Manage the `NewUser` message sent by CsdMessengerServer to the client applications when a new user connects.
- Implement the message receipt acknowledgement, in order to show a second grey circle in the state of each message sent.
- Implement the message reading acknowledgement, in order to show both circles associated to the message in color blue.

The student will have to apply the knowledge acquired throughout this practical to carry out this activity.

APPENDIX

A detailed explanation of the configuration of the Apache ActiveMQ Artemis server is available at:

https://activemq.apache.org/components/artemis/documentation/1.5.3/using-server.html

Here we will describe the most relevant aspects of this server in relation to this practice.

Create an Artemis broker

```
artemis create csdbroker
```

How do I start Artemis?

Artemis can be started with the following command, which will launch the broker in background:

```
artemis-service start
```

Or with the command:

```
artemis run
```

This will launch the broker, displaying all the explanatory configuration messages.

In our practice, it is recommended to start it with the "run" command, because in case the broker has been configured incorrectly, it will show us on the screen those points of the configuration where there are problems. We will then be able to try to solve them.

Therefore, for our practice we will have to use:

```
W:\csd\jms\csdbroker\bin\artemis run
```

How do I stop Artemis?

Artemis stops with the command:

```
artemis-service stop
```

Therefore, in our practice we should use:

```
W:\csd\jms\csdbroker\bin\artemis-service stop
```

How do I restart Artemis?

To delete all the mailboxes created and thus be able to use Artemis again as if it had just been installed, we must stop the Artemis service, delete the files in the "data" directory of the installed broker and reactivate the Artemis service.

Specifically, in our practice we must execute:

```
W:\csd\jms\csdbroker\bin\artemis-service stop
```

We will delete all the files in the folder:

```
W:\csd\jms\csdbroker\data
```

And relaunch Artemis:

```
W:\csd\jms\csdbroker\bin\artemis-service start
```

How do I remove the history of the conversations?

This practice is configured so that conversations are saved in a directory called "csdmessenger". To delete the conversation history, simply delete the files in this directory.

Specifically, in our practice we need to delete all the files in the directory:

```
W:\csdmessenger
```

How do I run and stop CsdMessengerServer?

To run it, open a terminal and run:

```
java -jar CsdMessengerServer.jar
```

To stop it, press Control-C

How do I run and stop my application CsdMessengerClient?

To run it, select the class `ui.CsdMessengerClient` from BlueJ, press the right button and select the method `main()`, then specify as the argument the name of the user that you want to use. To stop it, simply close the window.

How do I manage JMS exceptions?

There are several methods from the JMS API that throw exceptions, so when you use them you should enclose your code in a try-catch sentence like the following:

```
try {
   ...
} catch (NamingException | JMSException e) {
   e.printStackTrace();
}
```

How do I display a message on the information panel of the client application?

```
ui.API.addToLog(0, "message to be displayed");
```

Which of the provided classes and methods do I have to modify?

- Only the methods from the class `comm.Communication`

Which of the provided classes and methods can I invoke?

- All the public static methods from the class `ui.API`
- All the classes and their public methods from the package `messageBodies`

Documentation for these methods is available in the code of each class.
The documentation of those methods is available as part of the source code of each class.