SI4 Polytech'Nice Sophia – UCA

ECUE « Réseaux avancés et Middleware », Partie « Middleware »

Projet per group of two students

6 April 2020

Deadline on the Moodle around the 10th of May 2020,

Francoise.Baude@univ-cotedazur.fr

A simplified Discord application written in Java RMI

The goal of the project is to reproduce in pure JAVA RMI, a quite similar behaviour of the Discord application we are using during this confinement period. This will allow the students to apply the various RMI concepts. Moreover, even if the rest of the course pertains to the study of the Message Oriented Middleware (MOM) Apache Active MQ exposing the Java Messaging System API, that technology will not be used in the project. However, some functionalities that your Discord application will feature are going to be close to what a MOM should implement in order to offer 1) publish/subscribe on a topic, and 2) send/receive on a message queue features.

The server keeps the credentials and needed information about all the registered users, so to allow to propagate any message it should. All communications among users have to pass through the server. This is a point of failure and can harm scalability of the system. However, it is like this that the Discord (and a lot of other communication-oriented) application works.

Here follow the (sometimes simplified) capabilities you must consider supporting. 1)Ability to write a textual message and read all those exchanged in the textual channel/saloon, including those that arose when the user was disconnected from the system. 2)Ability to know who the registered users are, in order to be able to privately communicate with a given user through private text messages, but in an asynchronous manner. For groups of 3 students only, 3) direct and instant messaging among users, without intermediation of the server.

The user console can be programmed using the proposed last year template that has been provided on the moodle. Of course, any other user interface of your choice can be used instead. But do not spend too much effort on that aspect, as it is not part of the project' objectives.

The capability to exchange audio messages in a vocal saloon is left out of the project specifications, simply because there is not an easy way to record and play audio messages in Java, and this may become out of subject regarding the course. For the same reason, sharing / streaming the screen of a user has not to be addressed.

Table of contents

1) Generalities:

- There is just one single textual channel. No need to keep list of channels, and have users to know to which channel they enter.
- Any printed message shows the pseudo of its author, and the message itself (always a string of characters).
- The server is launched only once, and we assume it is not relaunched at all. This is to ensure that all messages to propagate to future re-connected users are kept in memory, and avoids you to have to consider to save them persistently in files as a real MOM would do.
- The user accounts are created in advance (i.e. in the initialisation phase at server side). Each account, as usual, has the following information: firstname, lastname, pseudo, and an RMI stub, so that the server or any other user, can execute a RMI method at user side. A user is programmed as an RMI object: all methods that can be invoked remotely should be exposed by this RMI object. The console the user runs, in order to deploy and navigate in between the various menus, is handled by the main thread at user side.
- Once a user account has been created, consider the user is by default already interested to consume any message that may further be published on the textual channel. Somehow, the user is automatically subscribed to the textual channel. Moreover, as in the true Discord system, even when the user is not yet connected to the server, messages published on that channel are going to be delivered to the user when connecting to the server again after a successful authentication. This property is known as “Durable subscription” in JMS terms.
- As in MOMs implementing the JMS specification, the notification of messages published on a given topic, does not have to be synchronous with the reception of messages at server / MOM side. In a MOM, it is only needed to feature asynchronous communication of messages between sender and receiver entities through pub/sub or point-to-point queue mechanisms. But, if at the end, your implementation is such that communication is (partly) synchronous, this is not so much a problem. Synchronous communication is a stronger property, so, one that can do the best can do less.
- However, for performance reasons, asynchronous communication support is considered enough and should be privileged by your project as much as possible. I.e. this can give you a bonus, but, if you simply implement synchronous notification of a new message to connected users to the topic, you will not be penalized. E.g. synchronizing the publication of a message on a channel with the immediate propagation of it to all connected subscribers may have as effect to temporarily block the server; depending of how you program it, it could not be able to

handle a new publication, while a previous one is still not fully notified. This can end up with a system that is not very responsive.

2) Textual channel:

- A user is capable to read all messages sent to the textual channel from the very beginning, and there is no way to erase any published message. (Note that this capability does usually not exist in a real MOM, as only not yet read messages are pushed to any subscribed user, but when all subscribed users have received a given message, it can be removed from the topic).

- When a user is connecting to the server, it should be informed by the server about the fact that some textual messages have been sent on the channel since the previous disconnection. To indicate that fact, the server would send back to the just authenticated user a specific message that should print in the user console (like 'Welcome, there are new messages to read').

- To have access to either old or new messages, the user has to explicitly inform the server that he enters into the textual channel (specific "window" on the user console; i.e. in your case, it means, enter in a specific menu). This means that entering the saloon is done through a method that has to inform the server that the user is now in the saloon. Symmetrically when the user leaves the saloon, the server must be informed.

- Once the user enters the channel, the server should send back (as a return parameter, so in a synchronous manner) an ordered list of all textual messages, by differentiating from which rank in the list, the new yet unread messages start. Once this list has been sent to the user, the server considers that there is not any unread message left for this user. To implement this, you may keep at server side the rank of the latest message exchanged on the textual channel that the user has seen, at the moment the user disconnects. When the user connects again, the server simply computes the difference between that rank and the rank of the latest message published in the channel. The server can infer from this difference if there is a need to inform the user that there have been newest yet unread messages.

- Now as the user has entered the channel, and is reading in the channel, any new message published in the channel should be propagated to the user, and eventually printed in the user console. Including the message written by a given user must eventually be printed (back) on his own console (probably in a color that is a different color than the one used for the standard input).

- As soon as the user leaves the textual channel user interface (in your case, it goes back to the upper level menu), it should not receive newest published messages (they should not be printed in the console, in the part dedicated to the channel content). However, depending of your implementation, there might be a delay for the server to take in consideration the fact that the user has left the channel. During this delay, notification of new messages may still be in progress, so new messages may arrive at the user side, and be printed in the console (even if the user may have already taken the decision to leave the channel).

- In particular, and depending of your implementation, it could be the case that the user leaves the channel before his last written/sent message in the channel has been notified to him. It is then up to the user to decide or not to enter the textual channel in case he left it, in order to receive and read the newest messages, including his own, in their entire length. As when connecting again, the server infers the newest messages are read as soon as it has propagated them to the user.

- Contrary to the case described above, if the user has already entered in the channel since he is connected to the server, only the newest messages should be sent by the server and printed on the user console. (And not all messages sent on the channel from the very beginning)

- While present in the textual channel, the user is able to write a new message, and publish it on the channel, channel which is similar to the JMS topic concept. This means that the server is in charge of receiving the new textual message, storing it in the messages list (storing corresponds to the effective publication), and eventually propagate (i.e. notify) it to subscribed/currently connected users., which may include the author, if the author of the message has not yet left the channel or has not disconnect from the application. Not yet connected users, or users not currently present in the saloon will receive it later, when connecting back (as with JMS durable subscribers), or when entering again the saloon.

- As mandated from the JMS specification, only FIFO send ordering property regarding message reception must be ensured (but if your solution ensures more stronger properties it is not a problem!!). FIFO-order means: 1) only consecutive messages authored by a same given user should be notified by respecting the order of their arrival at the server side (arrival order which then, corresponds to the send order from the user console and most probably to the publication order). 2) If two users send in parallel a message in the channel, the server should propagate them back to all connected users, but, it could be the case given your implementation that these two messages arrive and are printed in a different order on any two users' respective console.

  Illustration of FIFO-order property: suppose U1 sends M1 then M2, U2 sends M3 at some point in real time, in between. At server side, the storage order of these messages in the list of messages of the channel, i.e. the order of publication, could be M1,M2,M3, or, equivalently M1,M3,M2, or even M3,M1,M2. When the server will propagate the messages to users, propagation operation which has not to be mandatorily synchronized with the action of storing published messages in the list, users U1 and U2 can receive and print messages in any order, as soon as M1 prints before M2. It could be M1,M2,M3; or, M1,M3,M2, or even, M3,M1,M2, and even, on U1 and U2, the print order may differ. This is the theory, but, in practice, given your implementation, the ensured property might be stricter. E.g, you may always see M3 after M1 and M2 printed on any user console (a.k.a. Total ordering additionally to FIFO ordering). What will happen in practice will depend when in the implementation you decide that the server should notify subscribers: immediately, when a message is published, that is before returning from the invoked publication method, or later on. The problem with the 'later-on' is to "trigger" that later-on operation: at server side, this means that you may

run an explicitly programmed thread, whose role in the background is to notify each new published message to each subscriber.

In general, take care at server side of synchronizing access to shared variables: list of messages, list of authenticated users, list of users connected in the channel, unread messages counters per user, etc. In your final report, you must explain how your implementation features the FIFO ordering property.

- Ideally, subscribers connected to the channel should be notified of a published message in parallel. Calling a (even with void as return parameter) RMI method at user side to receive and print a message on the console implies the caller is blocked until the method execution finishes. This implies that the server iterating on the set of users present in the saloonb to notify them all of this new message, can be a quite long operation as it will call each user reception method one after the other. A more efficient way would be at server side, to implement a parallel (multi-threaded) solution. Some ideas towards a working solution: relying upon programmed threads (either explicitly, or implicitly by using a parallel stream on a collection), each thread in charge of one single user notification, blocking the possibility of handling the next publication until all the users have been notified. Remark: the goal in this project is not to notify many published messages in parallel; but, only to notify a single published message to all users connected in the channel, in parallel. (as a side note: in case parallel notification of many messages would have to be implemented, a simple way to enforce FIFO ordering of messages written by a same user could rely upon a sequence numbering of messages per author: a hole in the sequence is sufficient to detect if some message is missing before printing any other message from this author on a user console). . In your final report, you must explain how your implementation features any parallel notification if any.

3) Private messaging

- As in the Discord system, all registered users can be the destination of a private message sent from any other user. The goal is to again simulate how a MOM would offer asynchronous message exchange, this time not through a topic, but through the concept of a point-to-point queue. Here, you will have to use a per user private queue of messages, stored at the server side (in memory, not on the file system however). It is private in the only sense that only one consumer exists: it is the user who is the destination of the message(s). This is a bit different from the Discord system, where a user can see on the user interface, a distinct conversation he has taken part in, per specific user. Consequently, you will not have to remember for each user to which other user(s) he has sent messages.

- Each time a user writes a message for a specific user, identified through his (unique) pseudo, the message is sent and stored in the user specific private message queue on the server.

- If the user is currently connected, then, the server can immediately trigger the reception method exposed by the user, so that the message gets printed on the target user console. The print could happen anytime, including if the user is currently working in the textual channel

user interface part. So the print may pop up anywhere on the standard output, including in the "middle" of the list of the messages exchanged in the saloon.

- If on the contrary, the user is not currently connected to the server, the message stays in the queue. As soon as the user connects, the server will send back an information that there are some private messages to be received. In that case, the user has to run a specific command in order to trigger the reception of all messages he has not yet consumed from the queue. Contrary to textual channel messages, as soon as the server has propagated all messages from the queue to the user, the queue becomes empty. While emptying the queue, if a new message is produced, a question arises: should it be either stored in the queue and delivered later when the user will reconnect, or immediately propagated to the user with the risk that it prints on user console by breaking the effective FIFO send order? In any case, take care of not loosing it!

- To more easily distinguish the private messages from the textual channel messages, print them in a different format/color on the user interface.

4) The menus and their items

- From the description above, you may have this first level menu:
    - Authenticate, giving pseudo and password (and the user RMI stub)
        - A welcome message should appear on the output stream if authentication has succeeded, otherwise, should print a sort of "please try again" message .
        - Automatically, the user is seeing the second level menu content: it makes him enter the second level menu
    - Quit/Disconnect
        - A bye bye message should appear
    - Help, or any predefined character (like ?)
        - Should list all possible commands and their syntax
- The second level menu:
    - Back, or any predefined word or predefined character (like TAB)
        - Should leave that menu and go back to the enclosing menu
    - Enter the textual channel/saloon
        - Automatically, the user is seeing the appropriate third-level menu content: it makes him enter this third-level menu

        - All (old and) new messages received on the channel must be (immediately) printed, with a specific mark to distinguish new messages (messages not yet read) from old ones.
        - Of course, while connected in the saloon, newest messages should also appear
    - Read all pending messages from the private message queue
        - All not yet read messages still in the private queue should be printed. It could be immediate (ie, these messages are sent back as a return value of the command), or it could be deferred (ie these new messages will be pushed asynchronously, one by one, to the user, by the server. One by one, so that the user side is able to print any received message before it receives the next one).
    - Who is registered?
        - Receives back a list of all user pseudos. As users are supposed not to unregister from the system, nor, no new user is added to the system dynamically, this list is going to be always the same
    - Send a private message to *pseudo*
        - Allows to write a message, whose destination user is identified by the provided pseudo. If this pseudo effectively exists, the command should returns back an "OK, message sent to user" alert, or if not, an "unknow destination" alert. These alerts immediately show up in the console. At some point, a message sent to the user should appear in the console if the user is connected to the system.
    - Help, or ?
        - Should list all possible commands and their syntax of this second level menu
- The third level menu:
    - Back, or any predefined word or predefined character (like TAB)
        - Allows to go back to the second level menu=(temporarily) leave the textual channel
    - Write a new message
        - At some point, this message should appear on any user console if the user is still in the saloon.
    - Help, or ?
        - Should list all possible commands and their syntax of this third-level menu

5) The additional functionality (for groups of 3 students): instant direct messaging

For the group(s) of students that have 3 people: you must additionally support the following feature:

- Allow a user to get in contact directly with another user, without the intermediation of the server for sending him a message. The message should also hold a meta data: at least the pseudo of the author of this message.
- The message can only be received successfully if the user is connected (i.e. his stub exists).
- This means that the emitter should first get that stub from the server, given the pseudo, in order to be able to use that stub to directly invoke a method for pushing the instant message to the destination user identified by his pseudo. As an optimization, you could imagine that the stub of the destination is in a local cache, supposing this is not the first time that an instant message is sent to that destination.
- If in the meantime, the destination user has disconnected, then an error message should be printed on the emitter side console. If a local cache is used, perhaps the cache should be refreshed before claiming the user has disconnected.
- To simplify: there is absolutely no history; a message sent and not successfully received is definitively lost.
- VARIANT 1: If the message is successfully received by the destination user, there is no automatic way to reply to the message by sending back an instantaneous message. If the user wants to do so, he has to also use the same functionality. The stub of the emitter must first be looked up at the server side thanks to his corresponding pseudo (or in a local cache).
- VARIANT 2, in case there is yet another group of 3 students: there exists a sort of more automatic way to reply by sending back an instantaneous message. The stub of the user to which to reply, is included as meta data in the received message and can be locally kept for further usage.

6) Suggested methodology

- From the above specification description sections, list all various methods user and server sides should expose: clearly list methods that will be called remotely (not forgetting that the method will run in a new thread). Clearly define their respective call and return parameters.
- As usual in programming, split all complex actions in more elementary ones. Do not hesitate to have many methods that may be needed to be called by the methods that are exposed as remote. Do not forget that even if the signature of a remote method has void as return type, this method will not return the hand to the caller before it reaches the last instruction.

- Once this done, sketch on paper if all the designed interfaces and methods can suit the specifications/the various scenarios before going to the implementation

  For the implementation:.

- Do not hesitate to use the possibility to extend a class (in particular, message class, to distinguish the various sorts of messages to send, to receive and manipulate at both side)

- Make sure all shared data manipulations on both user and server sides are correctly synchronized. Do not forget that, the synchronized key word permits to synchronize concurrent access but on a **same** object. If there are two instances of the same class, even with method declared synchronized, if a thread runs code on one instance, it is not blocking the access for another thread in the other instance. In particular, if on the user side, you want to synchronize the access to the standard output (this is a single instance), you may associate the use (println actions) of that stream to a single object, through which all threads wishing to access to the standard output must synchronize. Do not forget too that you can use the Java syntax, even if a method is not declared synchronized:

```
yyy method_xxx ( parameters) {
        Some code;
        synchronized(object to act as synchronization tool){
// I'm not sure here if you could write synchronized(System.out) {//
                some code: where the guarantee that at most one thread at a time is running
code or access to attributes of that object
        }
        Some code;
}
```

- Take care of risks of deadlocks… on a single side but not only

- Test many scenarios, and convince yourself that you are exposing your code to the tricky distribution and concurrent related problems exposed in the preceding sections.

- Prepare a written report to accompany the code you will deliver. It must clearly show that you have taken in consideration these tricky interleaving of actions.

- Good luck !! and do not hesitate to raise questions by email !