Solution 2

Concurrent Programming

1 Overview

We present here the solution of Exercise 2. We first introduce the design and then explain the different classes we use. We also present the mechanisms used to send/receive messages and to retrieve the list of connected users. The implementation we present corresponds to the case where the clients "pull" the messages from the server. An implementation in which the server would "push" the messages to the clients is also possible.

2 Getting, Compiling and Running the Application

The complete source code is available for download at the exercise page of the course web site. The archive contains an ant (*http://ant.apache.org*) build file that can be used to compile the sources and generate the java documentation. To do this, simply type in the *Ex2* directory[1]:

```
ant
```

During the compilation, ant creates a jar file that can be used to launch the application:

```
java -jar dist/Ex2.jar <nbOfClients>
```

To clean the compiled classes, the javadoc as well as the jar file, simply type in a console:

```
ant clean
```

Finally, you can also compile and run the code using netbeans 4.1 (*http://www.netbeans.org*).

3 Design

The solution we give is splitted in different packages: *client*, *gui*, *serialization*, *server* and *utils*. Their different interactions are shown in Figure 1 and the interactions between the different classes are shown in Figure 2. In the following we give explanation on all the different packages as well as their classes.

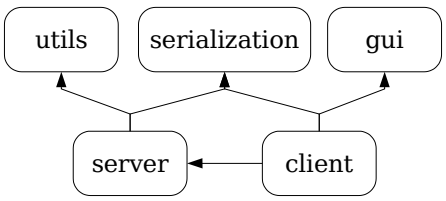

Figure 1: Packages interactions

[1]Please use the latest version of ant (1.6.2).

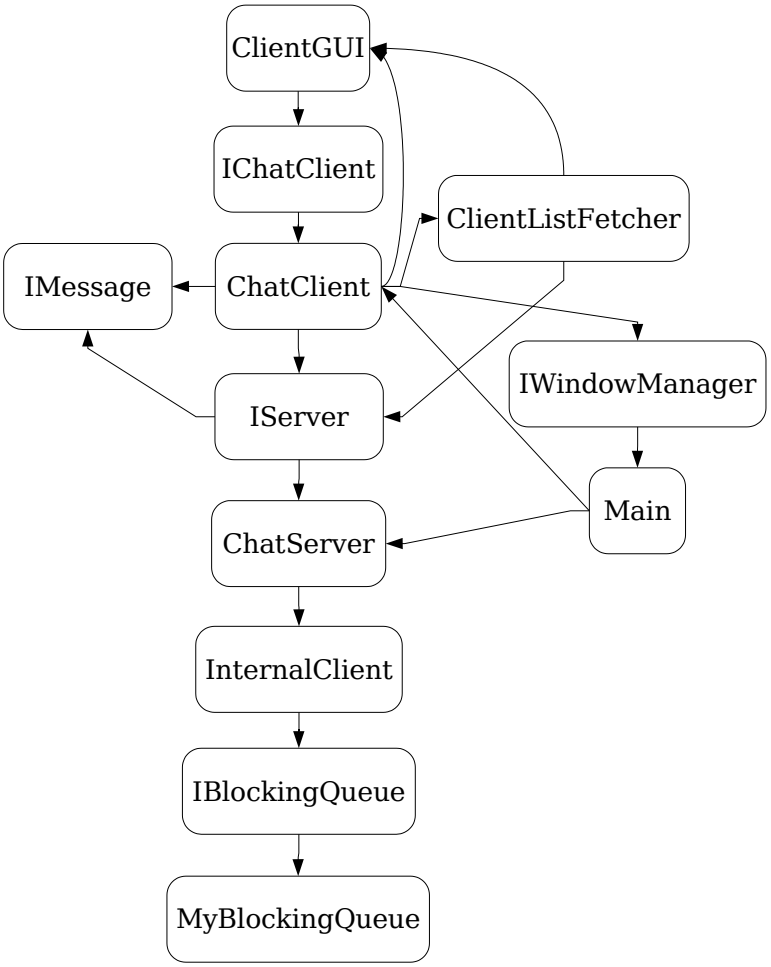

Figure 2: Classes interactions

3.1 Utils package

This package contains one interface and two classes as depicted in Figure 3. The `IBlockingQueue` interface is used by the `ChatServer` to send/receive messages from the different `ChatClient`. This interface represents the specification of a blocking queue. Two implementations of this interface exist: (1) `MyBlockingQueue` and (2) `MyPriorityBlockingQueue`. The first implementation is done by hand while the second one uses the `java.util.concurrent.PriorityBlockingQueue` of Java 1.5.

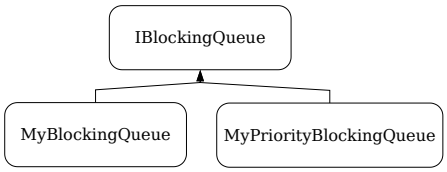

Figure 3: Classes of the *utils* package

3.2 Serialization package

This package contains the `IMessage` interface and its `Message` implementation. The `IMessage` represents the specification of a message that is sent between a client and a server. Its implementation contains a header and data. The header contains the username of the client that sends the message.

3.3 GUI package

This package contains two classes: (1) `ClientGUI` and (2) `IWindowManager`. The first class represents the GUI of the clients and is responsible for letting the user connect/disconnect to/from the server, send messages, show the received messages as well as the list of the connected users. This GUI is updated using the MVC paradigm, by the `ChatClient` (see Section 5).

The `IWindowManager` is used by the `Main` class and lets the different instances of the `ChatClient` call methods on it in order to correctly stop the application.

3.4 Client package

This package contains two classes and one interface. The `IChatClient` represents the specification of a chat client and its implementation is `ChatClient`. The `IChatClient` is necessary to make the link between the `ChatGUI` and the `ChatClient`. The `ChatGUI` manipulates only instances of `IChatClient`. This preserves the encapsulation of the `ChatClient` and makes the `ChatGUI` being independent of the implementation of the `IChatClient`.

The `ClientListFetcher` represents a thread that is used for retrieving the list of connected clients. This class is detailed in Section 7.

3.5 Server package

This package contains the `IServer` interface and its `ChatServer` implementation. Instances of `ChatClient` manipulate only `IServer`. As stated before, this makes the user independent of the implementation of a server and preserves the encapsulation of the `ChatServer`.

The `ChatServer` does not have any reference to `IChatClient` objects. Instead, the `ChatServer` contains a list of its connected clients (i.e., a list of `InternalClient`) and is responsible for managing the *connection*, *disconnection* of the clients as well as the *reception* of the messages and the *sent* of those messages to its clients. Finally, it gives to its clients the list of its connected users via the `getListOfClients()` method.

3.6 Main Class

This class belongs to the core package and is responsible for creating the different instances of `ChatClient` and the `ChatServer`. It also stops the application cleanly when no more `ClientGUI` are available (with the help of the `IWindowManager`).

4 Connection, Disconnection

When the user clicks on the *Connect* button, the `connect()` method of the `IChatClient` is called which, in turn, calls the `connect()` method of the `IServer`. When the `ChatServer` receives this method call, it puts the new client into its list of `InternalClient`. On the other hand, the `ChatClient` starts a new thread that is used for retrieving the new messages and a new `ClientListFetcher` thread that is responsible for getting the list of the connected users from the server every 3 seconds.

Upon *disconnection*, the `disconnect()` method is called on the `IChatClient` which stops the different threads running (i.e., `ChatClient` and `ClientListFetcher`) and calls the `disconnect()` method on the `IServer`. The `ChatServer` removes the client from its list of `InternalClient`.

5 MVC: Model, View and Controller Paradigm

In the MVC paradigm, the user inputs, the graphical feedback to the user and the behavior of the application are separated. In this scheme, the *View* represents the graphical interface and/or the textual representation of the application. The *Controller* is used to handle the user inputs and to command the *Model*. The *Model* defines the behavior of the application, manages data, responds to request information about its state (from the *View*) and to instructions to change its state (from the *Controller*).

In Java, this paradigm is implemented via the `java.util.Observable` class and the `java.util.Observer` interface. The `java.util.Observable` class represents the *Model* and the `java.util.Observer` represents the *View*. In Java, there is no class or interface for the *Controller*. This is because the *View* and *Controller* are usually merged in the same class: *java.awt.Frame* (or *javax.swing.JFrame*).

In the *ChatApplication*, a `ClientGUI` implements the `java.util.Observer` interface and, as such, has to define the `update()` method. This method is called by the `ChatClient`, that extends the `java.util.Observerable` class, whenever either a new `IMessage` is received or when a new list of users is received. When the `ClientGUI` receives the new updates, it changes its GUI accordingly.

To link an *Observer* to an *Observable*, the `addObserver()` method has to be called. This method adds the new *Observer* into the list of *Observers* that is stored by the *Observable*. Finally, to notify all the *Observers* of a new update, the method *setChanged()* and *notifyObservers()* have to be called.

6 Sending, Receiving Messages

When the user clicks on the *Send* button of the GUI, the `sendMessage()` method of the `IChatClient` instance is called. In turn, the `ChatClient` constructs a new `IMessage` and calls the `sendMessage()` method of the `IServer`. Finally, when the `ChatServer` receives the message, it puts it into each instance of `InternalClient` it has.

Putting a new message in an `InternalClient` means giving the message to an `IBlockingQueue`. In this case, the `IBlockingQueue` notifies all the different threads that are

waiting for this queue.[2] When a thread (i.e., `ChatClient`) is notified that a new message is available, it gets it and notifies its graphical interface (using the MVC paradigm as presented in Section 5).

The implementation of the reception of the messages is called *passive waiting* (because the `ChatClient` waits for a new message and does not constantly poll the server).

7 Retrieving the List of Connected Users

As seen in Section 4, the `ClientListFetcher` is responsible for retrieving the list of the clients from the server. This thread simply calls the `getListOfClients()` method on the server every 3 seconds and notifies the graphical interface of the new list.

The implementation of the `ClientListFetcher` is called *active waiting* (because the `ClientListFetcher` constantly polls the server even if no new clients became connected/disconnected). Please avoid doing *active waiting* in your program, this paradigm is not very efficient and can usually be replaced by *passive waiting* methods.

[2] As stated previously, you can find two different implementations of `IBlockingQueue`: (1) `MyBlockingQueue` and (2) `MyPriorityBlockingQueue` that uses `java.util.concurrent.PriorityBlockingQueue`. The first implementation is done by hand and shows you how to program a blocking queue. The second implementation is the native implementation of Java of a blocking queue. This implementation is only part of Java 1.5.