Workbook 4

Introduction

Last week you explored how to use the Java `synchronized` statement to acquire and release locks, and how to use the wait-notify paradigm to coordinate activities between threads. You then went on to build your own thread-safe implementation of the `MessageQueue` interface. This week you will write your own implementation of the Java chat server, using `SafeMessageQueue` as one of the core data structures.

Important

An on-line version of this guide is available at:

http://www.cl.cam.ac.uk/teaching/current/FJava

You should check this page regularly for announcements and errata. You might find it useful to refer to the on-line version of this guide in order to follow any provided web links or to cut 'n' paste example code.

A Java chat server

As you saw in Workbook 1 and 2, TCP communications often block the execution of a Java thread when data is read from a network socket. As a result, you used an additional Java `Thread` object to read data from the network socket and display it to the screen, allowing the main thread to read and parse user input. The Java chat server requires a more complex arrangement of Java `Thread` objects to handle messages coming from, and going to, multiple clients.

Figure 1, “The major Java objects and message flows in the Java chat server” provides an overview of the Java objects and messages passed between them when two clients are connected to the server; the Java objects which are part of the Java chat client you wrote for Workbook 2 are not shown in the diagram. The figure includes two instances of the `ClientHandler` class, each of which contains two `Thread` objects and a `MessageQueue` object. The figure also shows a single `MultiQueue` object and a single `ChatServer` object.

The flow of messages of type `uk.ac.cam.cl.fjava.Message` are represented in the Figure by solid black arrows. You probably recall from Workbook 2 that the first version of the Java chat client you wrote supported four message types. Table 1, “Message classes sent between the client and server” provides you with a summary of these message types. You will only be sending and receiving messages of these types in this Workbook. On the diagram, the messages flowing along the arrows shown as "from client" will be either `ChatMessage` or `ChangeNickMessage` types; all the other solid arrows on the diagram represent the flow of messages of type `StatusMessage` or `RelayMessage`.

Message type (class)	Direction	Description
`ChangeNickMessage`	Client→Server	Update nickname of the client stored by the server.
`ChatMessage`	Client→Server	Message written by a user is sent to the server.
`RelayMessage`	Server→Client	User message sent from server to all clients.
`StatusMessage`	Server→Client	Message generated by the server, sent to all clients.

Table 1. Message classes sent between the client and server

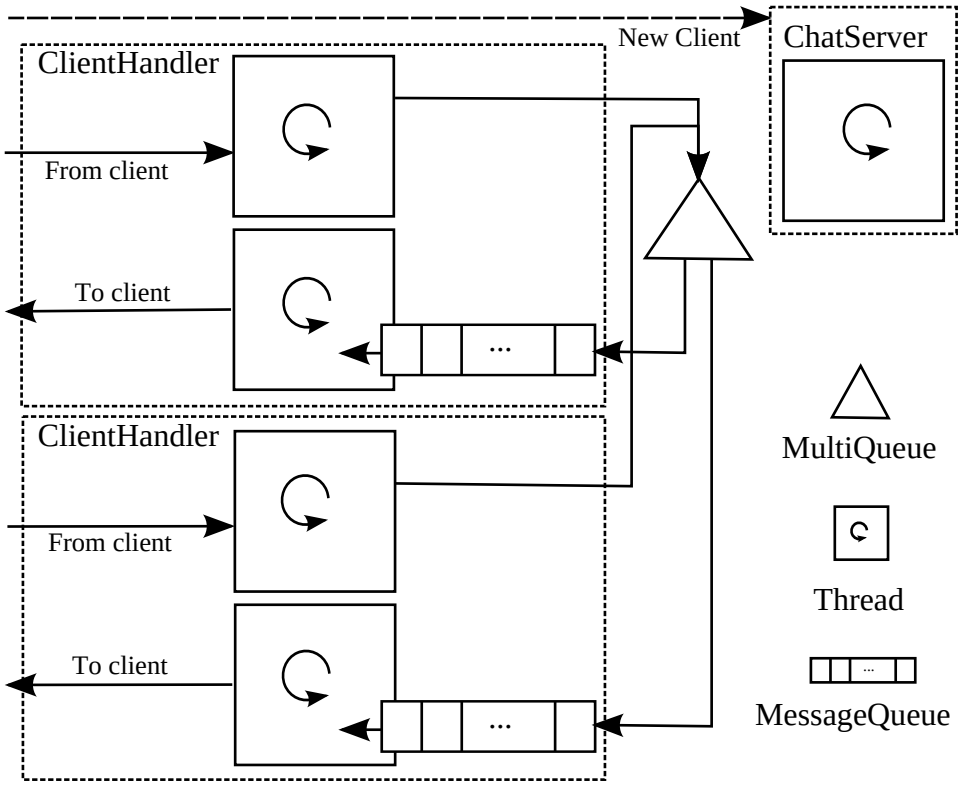

Figure 1. The major Java objects and message flows in the Java chat server

The Figure only provides you with the objects and associated message flows when there are two clients connected to the Java chat server. In general, if there are N clients connected to the server, there are N instances of `ClientHandler` (and therefore 2N client threads in total) and one `ChatServer`. In addition, there are N queues of type `MessageQueue` (i.e. one associated with each `ClientHandler`) and *one* queue of type `MultiQueue` which all N instances of `ClientHandler` send messages to. Every queue is accessed by more than one thread and therefore must be thread-safe.

The following subsections of this workbook provides a description of each of the objects in turn and what you will need to do to implement them. Please read, and re-read, *all* the descriptions carefully before starting on the implementation work described at the end of the Workbook. As in previous weeks, start by visiting the Ticklet 4 project on Chime https://www.cl.cam.ac.uk/teaching/current/FJava/ticklet4 and cloning a copy of your repository onto your local machine.

SafeMessageQueue

A thread-safe FIFO queue. You implemented this as part of Workbook 3. Copy your implementation into the package `uk.ac.cam.crsid.fjava.tick4`.

MultiQueue

```
public class MultiQueue<T> {
 private Set<MessageQueue<T>> outputs = //TODO
 public void register(MessageQueue<T> q) {
  //TODO: add "q" to "outputs";
 }
 public void deregister(MessageQueue<T> q) {
  //TODO: remove "q" from "outputs"
 }
 public void put(T message) {
  //TODO: copy "message" to all elements in "outputs"
 }
}
```

The server should contain a single instance of a `MultiQueue` object, regardless of the number of clients connected to the server. The instance should be specialised to process objects of type `Message`, although in practise there should only ever be `RelayMessage` or `StatusMessage` objects (i.e. server to client messages) and never any `ChatMessage` or `ChangeNickMessage` objects stored in the `MultiQueue` object. Internally, the `MultiQueue` object should store a list of active `SafeMessageQueue` objects associated with clients, and whenever a new message is added to the `MultiQueue` object through the `put` method, your implementation should copy the message to every registered `SafeMessageQueue` object.

To maintain a current list of active clients, the `MultiQueue` object must support `register` and `deregister` methods. All three methods (`put`, `register` and `deregister`) are, in general, invoked by multiple instances of `ClientHandler`, and as such you must ensure that `MultiQueue` is thread safe! You can do so by making appropriate use of the Java `synchronized` statement to prevent concurrent access to `outputs` by taking out a lock on `this` whenever the body of any method is invoked.

ChatServer

```
public class ChatServer {
 public static void main(String args[]) {
  //TODO
 }
}
```

This class should contain the special `main` method which is executed when the Java chat server is first started. Your implementation of the `main` method should expect an integer value representing the port number the server should start on as a single argument on the command line. If there are no arguments (or a malformatted argument) is found, your implementation should print out

```
Usage: java ChatServer <port>
```

and your implementation should then terminate.

If a port number was correctly provided by the user then the `main` method should create a single instance of `java.net.ServerSocket` by providing the port number entered by the user to the constructor of this class. If, for some reason, the Java chat server cannot use the provided port number (e.g. the port is already occupied by another server) your implementation should print out

```
Cannot use port number <port>
```

where `<port>` is the port number entered by the user and then terminate.

In addition to creating an instance of `ServerSocket`, your implementation of the `main` method should also create a single instance of the `MultiQueue` class. After the initialisation of an instance of

`ServerSocket` and `MultiQueue` your implemention should sit in a loop forever doing the following things:

1. call the method `accept` on the `ServerSocket` object;

2. The `accept` method will block execution of the `main` method until a new client connects to the port; when this happens create a new instance of the `ClientHandler` object, passing in a reference to the `MultiQueue` you created earlier together with the instance of `Socket` as returned by `accept`;

3. Go to (1) above.

ClientHandler

```
public class ClientHandler {
 private Socket socket;
 private MultiQueue<Message> multiQueue;
 private String nickname;
 private MessageQueue<Message> clientMessages;
 //TODO: possibly other fields here
 public ClientHandler(Socket s, MultiQueue<Message> q) {
  //TODO
 }
 //TODO: Other code here as necessary
}
```

This class handles the interaction between the Java chat client and the `MultiQueue` object. This class should provide a single constructor which takes two arguments: a reference to the `Socket` object associated with the Java chat client and a reference to a `MultiQueue` object. The constructor should do the following:

1. Update the fields `socket` and `multiQueue` to reference `s` and `q` respectively.

2. Update `clientMessages` to reference a new instance of `SafeMessageQueue` and call the `register` method on `multiQueue` to make `multiQueue` aware of the new client (more details on this in the MultiQueue Section above).

3. Create a default nickname and store it in the private field `nickname` by concatenating a random five digit number onto the string `Anonymous`. (Hint: you might like to use `java.util.Random`.)

4. Create a new `StatusMessage` object to record the fact that a new client has connected to the Java chat server and add this to `multiQueue`. The example chat session you saw first in Workbook 2 requires the server to print the machine name (not the IP address) the client connected from; to do so take a close look at the type of object returned by the method `socket.getInetAddress()`.

5. Define and instantiate a `Thread` object to handle incoming serialised objects which are sent by the client and received by `ClientHandler` on the input stream associated with the `Socket` object provided to the constructor. When the incoming thread receives a serialised `ChangeNickMessage` object, then update the private field `nickname` and create and send a `StatusMessage` object to all clients by calling `put` on the `MultiQueue` object; when the incoming thread receives a serialised `ChatMessage` from the client, create a `RelayMessage` object and call `put` on the `MultiQueue` object to send the message to all clients. If the incoming thread receives a message of any other type from the client the message should be ignored.

6. Define and instantiate a `Thread` object to handle outgoing messages found on the `SafeMessageQueue`. The thread should serialise and send any message objects on the queue to the Java chat client by making use of the output stream of the `Socket` object.

When the Java chat client disconnects, the Java chat server will receive an `IOException`. Handle this exception in the `ClientHandler` by calling the `deregister` method on `multiQueue` appropriately

and create and send a `StatusMessage` object to all the remaining clients. When the Java chat client disconnects, you must ensure that both the threads used to handle incoming and outgoing threads terminate. (If you don't, your Java chat server will eventually fail, since every new Java chat client which connects creates a new instance of `ClientHandler`, and when they disconnect the incoming and outgoing threads don't finish and so continue to consume memory on the server.)

Testing

Testing an implementation of your server is going to be hard since it will create quite a lot of threads. Here are some suggestions which should help:

1. Never leave the body of a `catch` block empty unless you really know what you're doing! If you do nothing else, catch the exception as a variable called `e` and then call `e.printStackTrace()` in the body of the catch block so you see some useful debugging output when your server fails.

2. Package up a copy of the client you wrote for Tick 2 into a jar file and set the jar manifest so you can run it from the command line as follows:

   ```
   java -jar ChatClient.jar localhost 1234
   ```

   Assuming, of course, that you're running your Java chat server on port 1234. You can then run multiple Java chat clients easily in multiple terminals on your computer and use IntelliJ to run the actual server.

3. When you think your server is running correctly, enter in the dialog as shown in Figure 2, “A chat session between Dave and Hal” and make sure your server responds in precisely the same manner.

4. Beware when offering to test a server written by your friend using your Java chat client—unless you have implemented Tick 2*, your existing implementation will execute any method sent to you which is annotated with `@Execute`. You should modify your Java chat client to remove this feature before testing other servers. Even if you have implemented Tick 2*, you might want to reconsider the permissions you've given to any classes you load dynamically. You have been warned!

```
crsid@machine:~> java -jar crsid-tick2.jar
14:23:27 [Client] Connected to java-1b.cl.cam.ac.uk on port 15003.
14:23:27 [Server] Anonymous15983 connected from evapod.discoveryone.space.
\nick Dave
14:23:29 [Server] Anonymous15983 is now known as Dave.
14:23:14 [Server] Anonymous82791 connected from cpu9000.discoveryone.space.
14:23:17 [Server] Anonymous82791 is now known as Hal.
Hello, Hal. Do you read me, Hal?
14:23:22 [Dave] Hello, Hal. Do you read me, Hal?
14:23:27 [Hal] Affirmative, Dave. I read you.
Open the pod bay doors, Hal.
14:23:31 [Dave] Open the pod bay doors, Hal.
14:23:36 [Hal] I'm sorry, Dave. I'm afraid I can't do that.
Why not, Hal? What's the problem?
14:23:39 [Dave] Why not, Hal? What's the problem?
14:23:43 [Hal] I think you know what the problem is just as well as I do.
What are you talking about, Hal?
14:23:50 [Dave] What are you talking about, Hal?
14:23:53 [Hal] This mission is too important for me to allow you to jeopardise it.
I don't know what you're talking about.
14:23:59 [Dave] I don't know what you're talking about.
\destroy Hal
14:24:02 [Client] Unknown command "destroy"
14:24:06 [Hal] I know you and Frank were planning to disconnect me.
14:24:08 [Hal] And that's something I cannot allow to happen.
14:24:12 [Server] Hal has disconnected.
\quit
14:24:17 [Client] Connection terminated.
crsid@machine:~>
```

Figure 2. A chat session between Dave and Hal[1]

[1] A quote from the 1968 epic "*2001: A Space Odyssey*". Directed by Stanley Kubrik, and written by Arthur C. Clarke and Stanley Kubrick.

Ticklet 4

You have now completed all the necessary code to gain the fourth ticklet. Your repository should contain and make use of the following source files:

```
src/uk/ac/cam/crsid/fjava/tick4/ChatServer.java
src/uk/ac/cam/crsid/fjava/tick4/MessageQueue.java
src/uk/ac/cam/crsid/fjava/tick4/SafeMessageQueue.java
src/uk/ac/cam/crsid/fjava/tick4/MultiQueue.java
src/uk/ac/cam/crsid/fjava/tick4/ClientHandler.java
src/uk/ac/cam/cl/fjava/messages/ChangeNickMessage.java
src/uk/ac/cam/cl/fjava/messages/NewMessageType.java
src/uk/ac/cam/cl/fjava/messages/RelayMessage.java
src/uk/ac/cam/cl/fjava/messages/StatusMessage.java
src/uk/ac/cam/cl/fjava/messages/ChatMessage.java
src/uk/ac/cam/cl/fjava/messages/Message.java
```

When you are satisfied you have completed everything, you should commit all outstanding changes and push these to the Chime server. On the Chime server, check that the latest version of your files are in the repository, and once you are happy schedule your code for testing. You can resubmit as many times as you like and there is no penalty for re-submission. If, after waiting one hour, you have not received a final response you should notify `ticks1b-admin@cl.cam.ac.uk` of the problem. You should submit a version of your code which successfully passes the automated checks by the deadline, so don't leave it to the last minute!