Leveraging multithreading, threadpools and network programming to implement an order book in Java

Project structure

```
├── Client
│   ├── MainClient.java
│   ├── NotificationReceiver.java
│   ├── RequestMaker.java
│   └── ResponseManager.java
├── Misc
│   ├── client.properties
│   ├── gson-2.11.0.jar
│   ├── MANIFEST_CLIENT.MF
│   ├── MANIFEST_SERVER.MF
│   ├── server.properties
│   └── storicoOrdini.json
├── Server
│   ├── ClientHandler.java
│   ├── MainServer.java
│   ├── NotificationSender.java
│   ├── OrderBook.java
│   ├── OrderManager.java
│   ├── RequestManager.java
│   ├── ResponseMaker.java
│   ├── User.java
│   └── UserManager.java
└── Shared
   ├── MessageSerializator.java
   ├── NotifyPort.java
   ├── OperationResponse.java
   ├── Order.java
   ├── OrderResponse.java
   ├── Pair.java
   ├── Request.java
   └── Tuple.jav
```

Building

****All commands must be executed from inside the root directory Project/**

1. Create output directories

```
mkdir -p out/client out/server
```

2. Compile java files

```
#Client
javac -cp Misc/gson-2.11.0.jar -d out/client \
    Client/*.java Shared/*.java

#Server
javac -cp Misc/gson-2.11.0.jar -d out/server \
    Server/*.java Shared/*.java
```

3. Create JAR files

```
#Client
jar cfm ClientApp.jar Misc/MANIFEST_CLIENT.MF -C out/client .

#Server
jar cfm ServerApp.jar Misc/MANIFEST_SERVER.MF -C out/server .
```

4. Execute

```
#Client
java -cp "ClientApp.jar:Misc/gson-2.11.0.jar" Client.MainClient

#Server
java -cp "ServerApp.jar:Misc/gson-2.11.0.jar" Server.MainServer
```

Key implementation decisions

User’s database has been implemented as a HashMap instead of as a concurrent one.
Synchronization happens thanks to synchronized methods over the single record that is being managed.
A concurrent hashmap would have required, to be useful, atomic methods like **compute** which would have required many edits in the login and registration phase.

User registration

Using atomic method for checking and creating the user record if absent, successively sending the response to the client.

User login

As written before, we first set the pointer userTuple to point at the record of our interest, which exists (owing to checks made before).

Inside the sync block, we firstly check if user is online and, if it is not, the status is set as true, together with the IP address and the UDP port for notifications.

A response is then sent back to the client.

Adding an order

Synchronized method that allows the server to forward orders to the book.
Made this decision to avoid any possible form of conflict with book records, orders and so on.
It also sends notifications to the user the order belongs to.

JSON file management

Even using File Streaming API it could not have been possible to reopen a file after closing it.
In this project, JSON syntax is “forced” manually in both reading and writing.

Since we have a different order log file for each server session, receiving a filename in input with properties has not been made possible.
For tests, inside the code is kept fixed the file “storicoOrdini.json” given with the project guidelines.

Users and book persistence: owing to bad performance that performing a linear scan of the file for each user operation like login would have meant, users are kept on memory once withdrawn from the file.

A different way to manage this situation should have involved a database management system to perform key searches on a table.

Users and orders persistence is managed by two different modules: UserManager and OrderManager.

Notifications

Two modules: a sender and a receiver, respectively implemented in the server and client packages.

Sender: a blockingqueue has been set up.
The module consists of two main methods: one adds a notification in the queue, the second withdraws a notification from the queue and sends it.

Receiver: a byte-type buffer has been provided together with a linked list to store incoming notifications.
Notifications are withdrawn from the buffer and put into the linked list.
From the linked list they can be withdrawn by the client.

IP and UDP port are provided by the users’s hashmap, where are put as soon as they succesfully log in.

Server / client shutdown management

For both client and server a shutdownhook has been provided, to succesfully capture SIGINT signals.

Server: if triggered, the procedure sends notifications to all the online clients, before performing general cleaning such as sockets and files closure.

Client: if triggered, it sends a notification to the server, which interrupts immediately whatever operation is being performed before closing the connection in a safe manner.
Client shutdown can be triggered also by a server shutdown in which case a notification is received, triggering the shutdown hook with the call of system.exit() by the notificstion receiver.

Requests and messages

ResponseManager: manages the two different types of responses from the server: after an order or after any other operation.
It simply extract the message and code.

RequestMaker: given a number from user input, requestmaker asks for all the data necessary to complete the operation linked to that number.
An auxiliary method to get the request in json format is also provided.

RequestManager: allows the server to extract information obtained by the request maker in a json format.
An auxiliary method to obtain the operation is also provided.

ResponseMaker: a method used to get a response to an operation. With method overriding is possible to set response code and operation type if required. Constructors are used to build the response object.

Message serializator: built separately just to serialize notifications.