Demonstration of our Server against clients

our client:
Everything worked as specified by protocol.

other clients:
Everything worked as specified by protocol.

Demonstration of our Client against servers
our servers:
Everything worked as specified by protocol.

other servers:
Everything worked as specified by protocol.

A short description of the chosen design

The most important thing to note about or design is the listener pattern that the gui implements. The listener pattern is very usefull when multiple objects need to be notified in changes that happen in another object. This happens in this way:

Objects A and B implement an interface that gives them a method (lets call it messageArrived(data)). Objects A and B can register as listeners of object C, by calling a method (lets say registerListener(this)) on C. C adds them to a List of listeners, and when needed can iterate through it and call their messageArrived(data) method.

In our program we have only one listener (gui listens to teh ServerClient) but still the method removeListener(l) was kept to make obvious that we are using this pattern

State behaviour
The protocol is very strict on what the client can receive at certain stages. More specifically, the client cant recieve any other command than ONLINE right after he sends a CONNECT, and the client shoudl discard all other commands than CLOSE after sending a CLOSE.

The first problem is solved on the server side. A thread safe synchronised method of each ClientHandler instantiates a CommandParser object. The CommandParser handles all commnds that come to the ClientHandlers. Then this object, in the case of a CONNECT, adds the user to the HashMap of online users, and sends a message to all online users. Since the whole CommandParser is thread-safe, only one ClientHandler is using it at a time, so no other message can interfere and be sent before the ONLINE reply. Extensive commenting also before object instantiation in ClientHandler class.

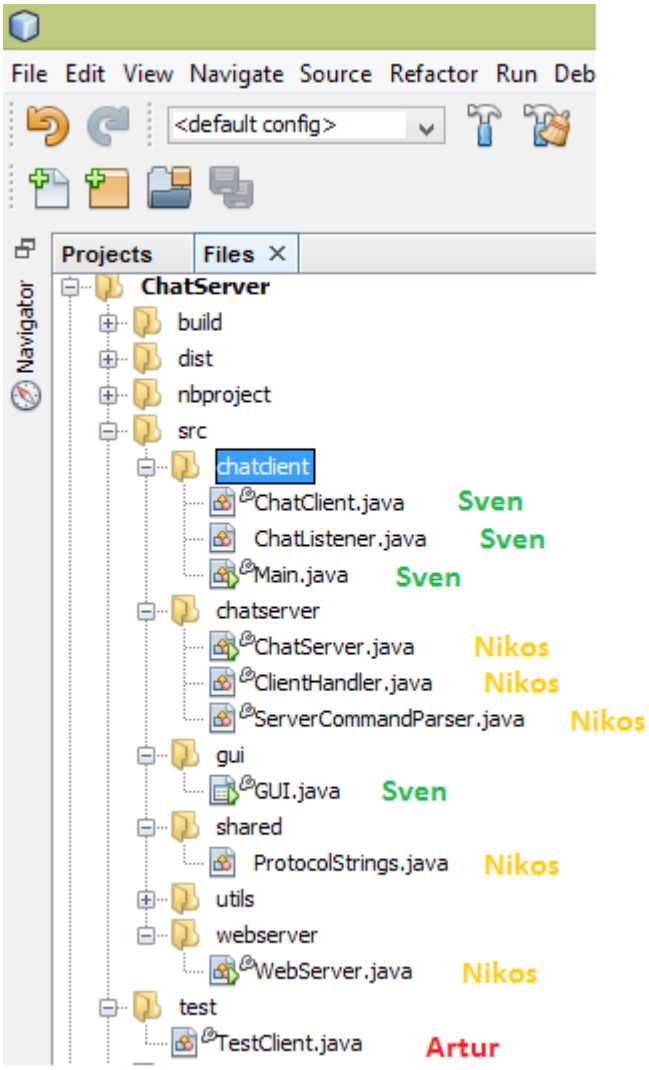
The second problem is solved in the client side. A client can be in one of these 3 states at any given moment: CONNECTING, ISONLINE and CLOSING. A String "state" is used for that. State becomes CLOSING right after a CLOSE command is sent. Any incomming message is filtered through a switch filter:

```
switch (state) {

   case "CONNECTING":
         Discards everyhting other than ONLINE. Unnesessary since server takes care of
         not sending anyhting other than that (see above).
      break;
   case "ISONLINE":
         Does nothing
      break;
   case "DISCONNECTING":
         Discards everyhting other than CLOSE.
      break;
   default:
      throw new IllegalArgumentException("Invalid command: " + serverCommand);
}
```

As it is clear, the first 2 cases of the switch "filter" are unnecessary, but are there to make obvious that we do use a client state approach.

A short description of who did what



Wireshark sample of a complete tcp scenario

No.	Time	Source	Destination	Protocol	Length	Info
7	5.280968000	10.50.130.212	23.101.54.180	TCP	66	54744→9090 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256
8	5.317113000	23.101.54.180	10.50.130.212	TCP	66	9090→54744 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1
9	5.317221000	10.50.130.212	23.101.54.180	TCP	54	54744→9090 [ACK] Seq=1 Ack=1 Win=66048 Len=0
10	5.346841000	10.50.130.212	23.101.54.180	TCP	69	54744→9090 [PSH, ACK] Seq=1 Ack=1 Win=66048 Len=15
11	5.386879000	23.101.54.180	10.50.130.212	TCP	68	9090→54744 [PSH, ACK] Seq=1 Ack=16 Win=131328 Len=14
12	5.444817000	10.50.130.212	23.101.54.180	TCP	54	54744→9090 [ACK] Seq=16 Ack=15 Win=66048 Len=0
14	8.721855000	10.50.130.212	23.101.54.180	TCP	68	54744→9090 [PSH, ACK] Seq=16 Ack=15 Win=66048 Len=14
15	8.761813000	23.101.54.180	10.50.130.212	TCP	75	9090→54744 [PSH, ACK] Seq=15 Ack=30 Win=131328 Len=21
16	8.813081000	10.50.130.212	23.101.54.180	TCP	54	54744→9090 [ACK] Seq=30 Ack=36 Win=66048 Len=0
19	10.08184800	10.50.130.212	23.101.54.180	TCP	62	54744→9090 [PSH, ACK] Seq=30 Ack=36 Win=66048 Len=8
20	10.08201300	10.50.130.212	23.101.54.180	TCP	54	54744→9090 [FIN, ACK] Seq=38 Ack=36 Win=66048 Len=0
21	10.11582700	23.101.54.180	10.50.130.212	TCP	60	9090→54744 [ACK] Seq=36 Ack=39 Win=131328 Len=0
22	10.11829400	23.101.54.180	10.50.130.212	TCP	62	9090→54744 [PSH, ACK] Seq=36 Ack=39 Win=131328 Len=8
23	10.12337800	23.101.54.180	10.50.130.212	TCP	60	9090→54744 [FIN, ACK] Seq=44 Ack=39 Win=131328 Len=0
24	10.12342300	10.50.130.212	23.101.54.180	TCP	54	54744→9090 [ACK] Seq=39 Ack=45 Win=66048 Len=0

1. Client asks for a connection
2. Server acknowledges the connection request
3. Client acknowledges the receiving of the servers acknowledgment
   - Three way handshake done-
4. Client sends his first message to the server
5. Server replies with appropriate message
6. Client sends receipt of received message

Flow Chart