

Assignment 2 part 2: Othello Networking

Value: 8% of your overall grade.

Due date: December 13th (Sunday, midnight)

Purpose:

We've got the UI sorted out, so let's finish the networking.

We're going to build up a server that can handle any number of clients, and we're going to add networking capabilities to our client. When all is said and done, our Othello client can connect to our central Othello server, making it much easier to get a good game on.

Minimum Requirements

You will need:

- A working networking modal from Assignment 2-1,
- The pink "output area" as a text area,
- A menu system of some sort,
- A text field, and
- A working "Submit" button.

If you've kept up with the assignments to date, you're in a great position. Keep it up.

Changes:

There are always changes, aren't there?

First and foremost, the client should not print a single thing to the console. Remove all `System.out.println` lines. The client is a finished product.

Next, the "disconnect" option and the submit button in your client must start disabled. You know you've done it right when they look sort of grayed out and unresponsive.

Finally, you will need to make a few changes to the modal.

- A new label, "Name:"
- A new textfield to take in name input, and
- "Status:" should now be blank

See the screenshot on the next page.

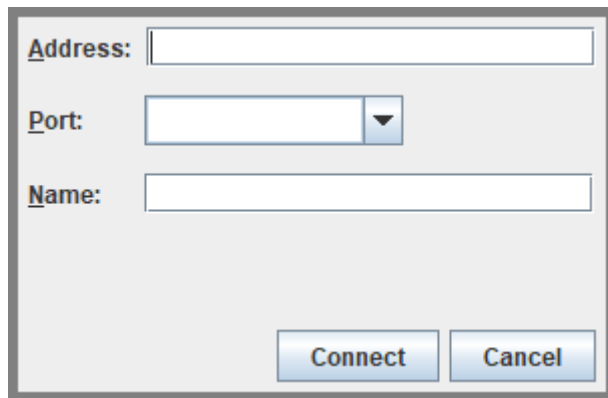


Figure 1: The new Network Connection dialog.

The modal will now act differently:

If the Address is blank, the Port input is invalid, or the Name has less than three characters, make an error message appear on your status label.

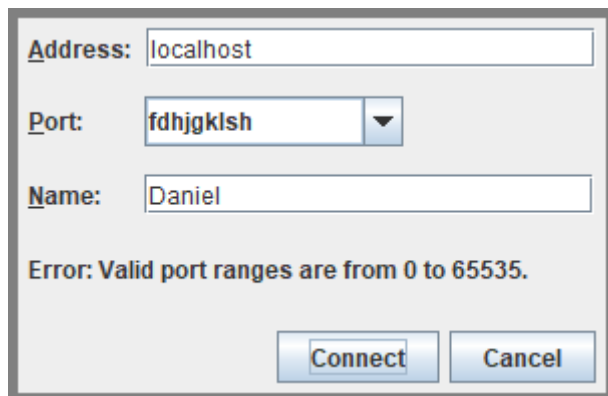


Figure 2: Network Connection dialog error reporting.

The error messages are, “Address must not be blank”, “Valid port ranges are from 0 to 65535” and “Name must be at least three characters long.”

Pressing “connect” will not permit the dialog to close until all input is valid. The user may press Cancel at any time.

(Note: Spelling matters. Plz no spelng mistakes in any messages.)

Be sure to correctly use the “connectPressed()” method already provided in your dialog controller.

Client Implementation

If the user has pressed connect (and you can test with `connectPressed()` in the dialog), the client should attempt to negotiate a connection. If the user has pressed cancel, no message is printed to the output area.

If the connection has failed, you will want to display an appropriate message to the output area.

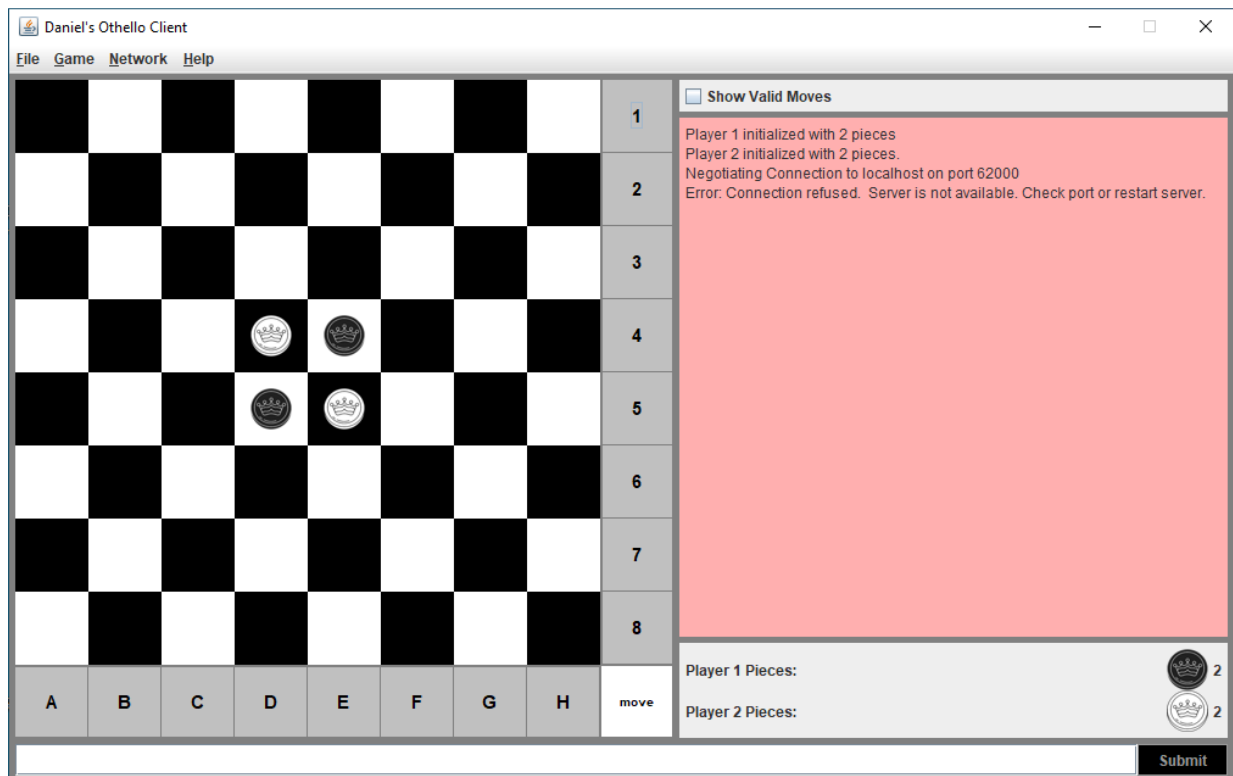


Figure 3: Client reporting a failed connection.

If you do succeed in connecting, a “Connection successful” message is useful. Immediately launch a new thread (details below) to manage the client connection.

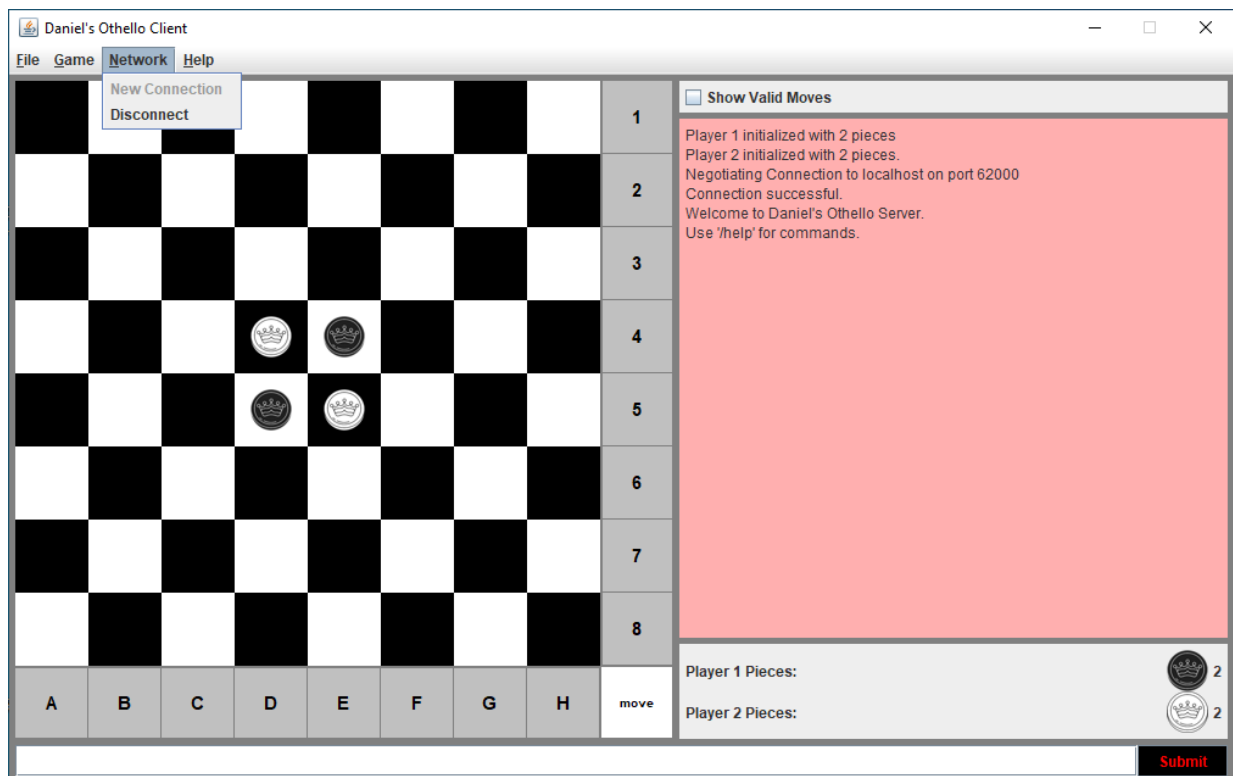


Figure 4: Successful connection.

Note that “New Connection” is now disabled, “Disconnect” and “Submit” are now enabled. We are now live and connected to the server.

Typing messages in the text field at the bottom, and then clicking “Submit”, will transmit messages to the server. Selecting “Disconnect” from the menu should disconnect from the server.

Class OthelloNetworkController

You will need a new class, OthelloNetworkController, which will be a thread. This thread will monitor the socket, and when it detects input, promptly displays it on the output zone.

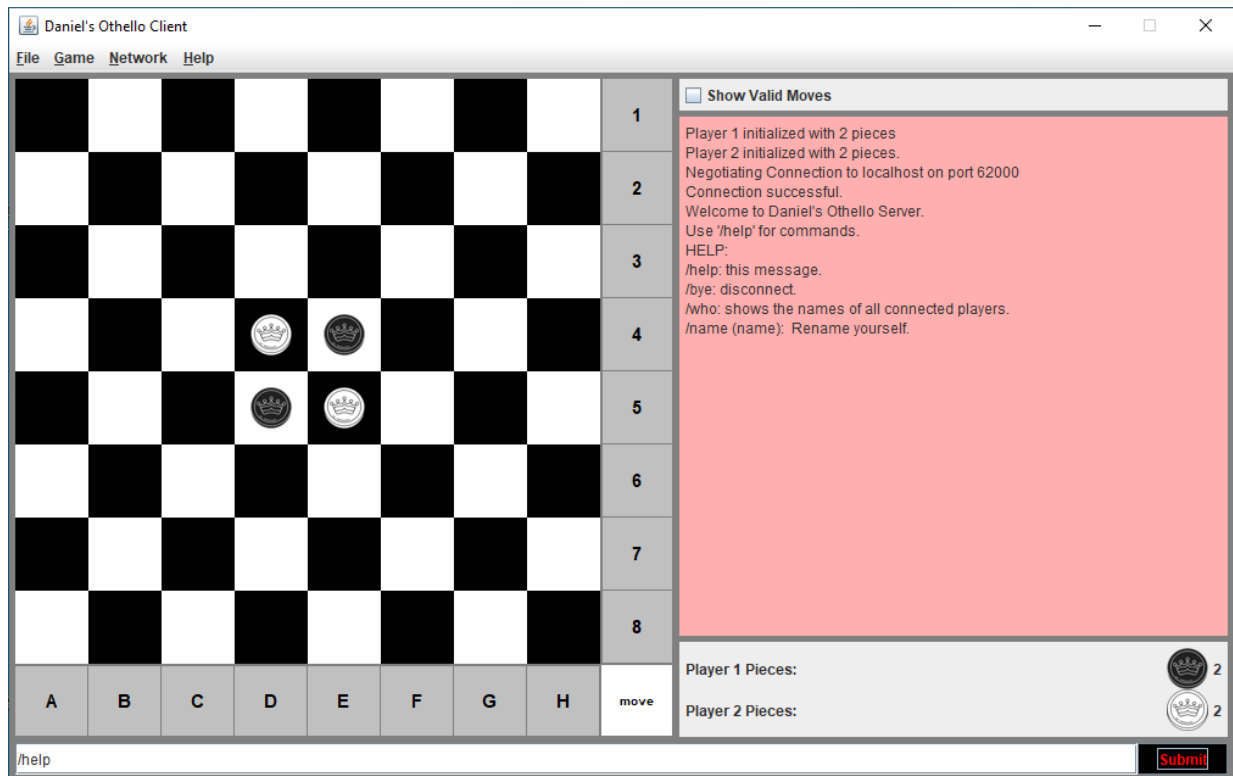


Figure 5: Sent a command to the server and received a response.

Upon connection, the thread will pass the name you've provided to the server. You can use “/name” for this—details to follow in the “Server” section of this document.

If the socket should close abruptly, or any other error should occur, it will close itself down gracefully. “New Connection,” should be enabled, “Disconnect,” and “Submit” should be disabled.

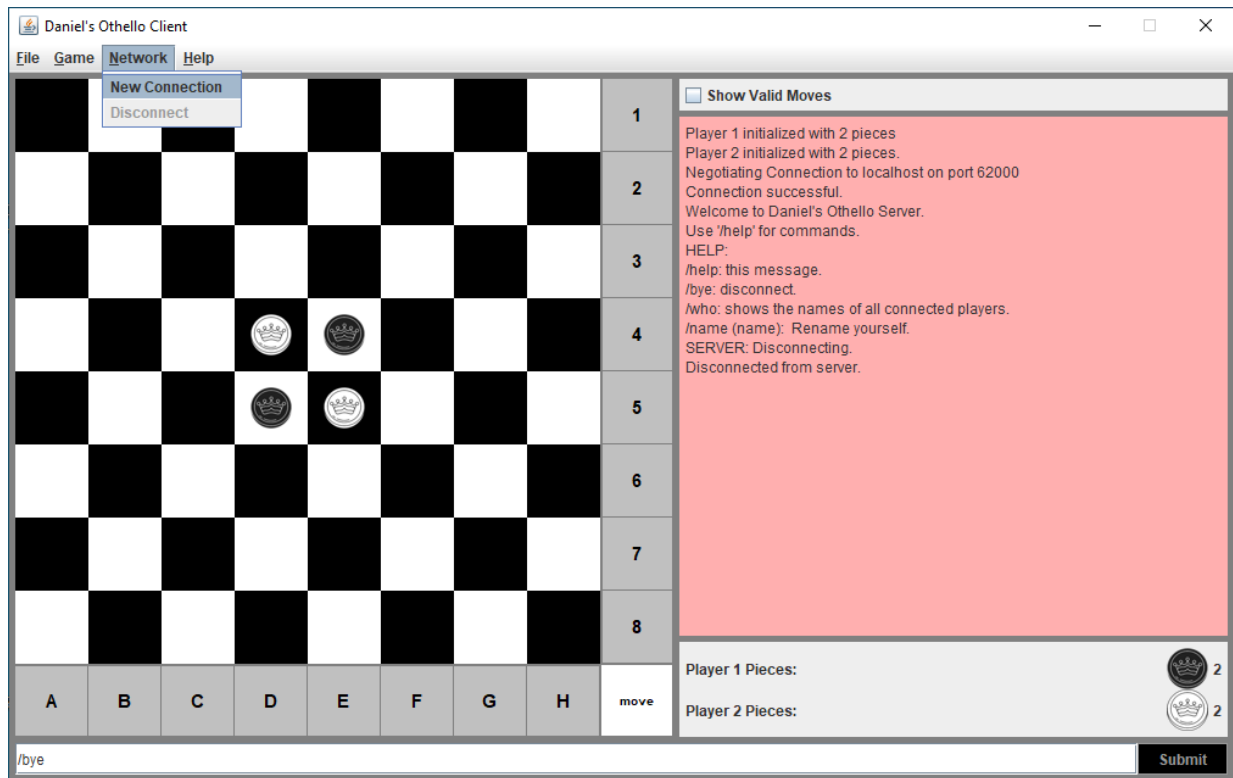


Figure 6: Disconnected.

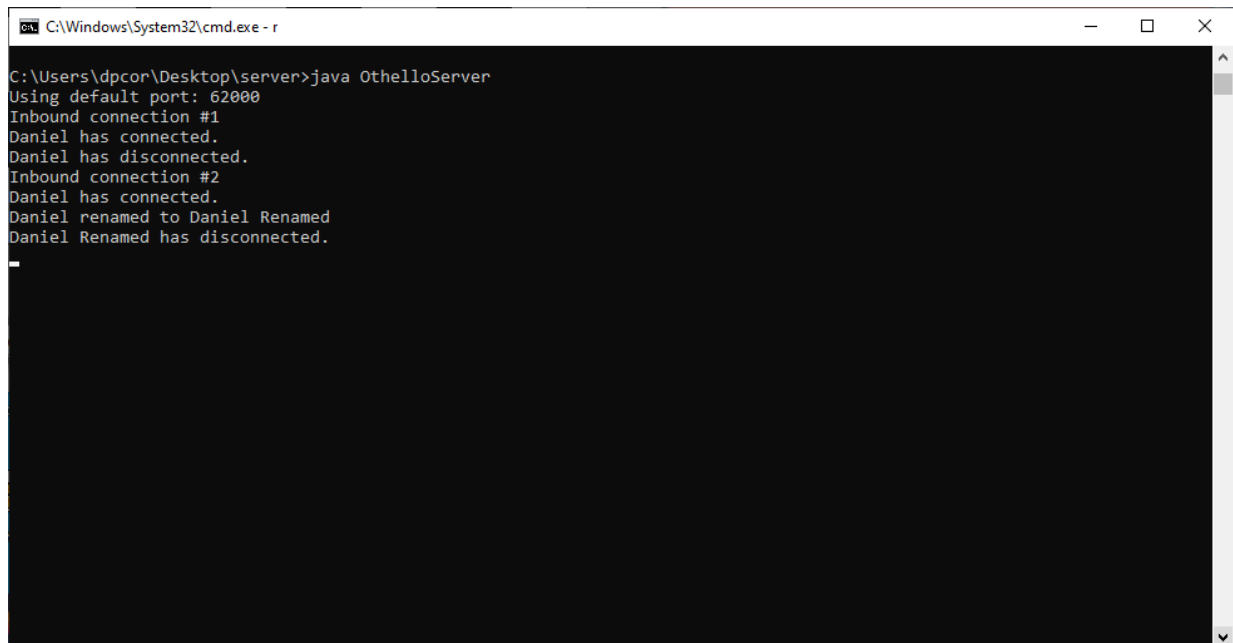
The Server

The server will do much of the work in this assignment. It will have no UI and it operates strictly from the command line. It will print status updates to the console as it goes along.

The server will be in its own package, "server," with a lowercase s.

The server should produce useful output as it goes along. Specifically:

- When a socket is first created, number it sequentially and print to console.
- When the thread is created and the user is first named, print that to console as "(username) has connected."
- If a user has disconnected from whatever means, a disconnection message is required.
- Finally, if a user renames themselves, the old and new name should appear: "(old name) has renamed themselves to (new name)".

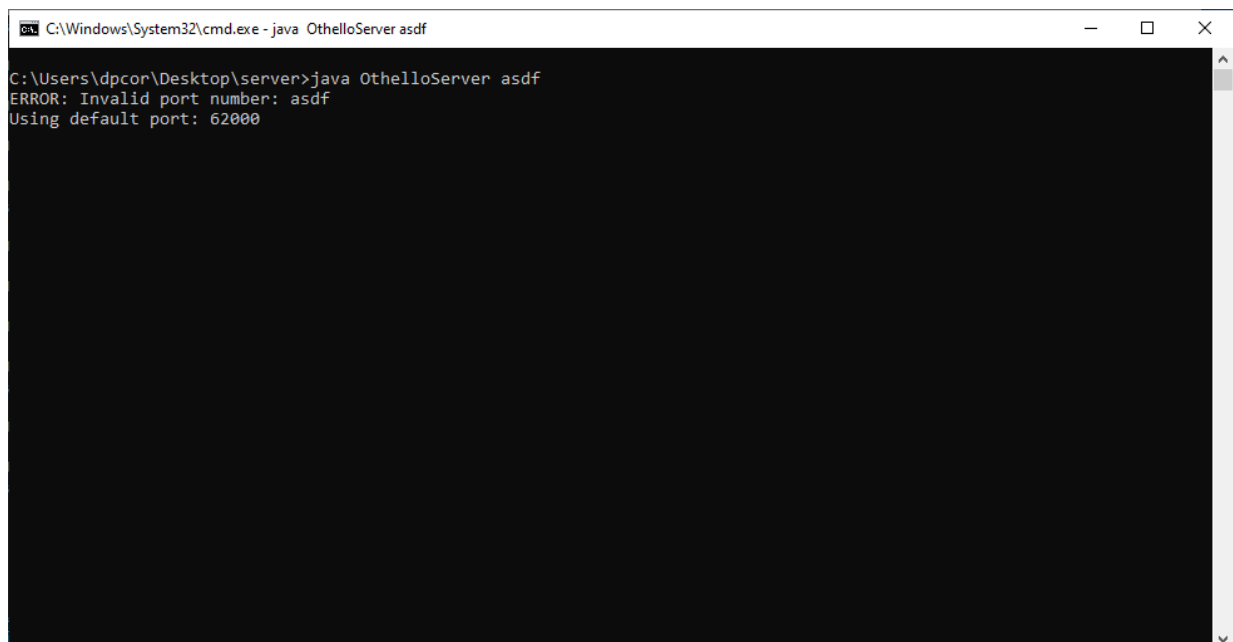


```
C:\Windows\System32\cmd.exe - r
C:\Users\dpcon\Desktop\server>java OthelloServer
Using default port: 62000
Inbound connection #1
Daniel has connected.
Daniel has disconnected.
Inbound connection #2
Daniel has connected.
Daniel renamed to Daniel Renamed
Daniel Renamed has disconnected.
```

Figure 7: Server in operation with sample messages.

Class: OthelloServer

This will serve as your entry point. It will accept a command line argument for a port and parse it. I should be able to run this server on any valid port. If the input is invalid, it will default to port 62000



```
C:\Windows\System32\cmd.exe - java OthelloServer asdf
C:\Users\dpcon\Desktop\server>java OthelloServer asdf
ERROR: Invalid port number: asdf
Using default port: 62000
```

Figure 8: Server launch

The server should create a Server socket, monitor it on a loop, and create an *OthelloServerThread* to manage that specific connection. It should also keep track of that *OthelloServerThread* in a threadsafe manner.

It must also announce to the console when a new user has connected,

Some suggested methods:

`broadcast(String)`: Broadcasts the given message to all existing threads.

`remove(OthelloServerThread)`: removes this thread from the list. Useful when a given thread has disconnected one way or another.

`who()`: returns a list of the names of all currently connected clients.

This list is not exhaustive. You will need to make a method that broadcasts to all connections save the one that issued that event. (ie: for `/name` functionality)

Class: OthelloServerThread

This class is responsible for monitoring the actual socket. If the user enters certain commands (they all start with the slash—`/`) then a special command will occur. Otherwise, it is a text message to be broadcast to all users who are connected.

There are four commands that must be implemented:

- `/help` will display a help message. See Figure 5, above.
- `/bye` will make the server disconnect the client
- `/who` will return a list of all connected users to the client, and
- `/name (newname)` will permit you to change your name.

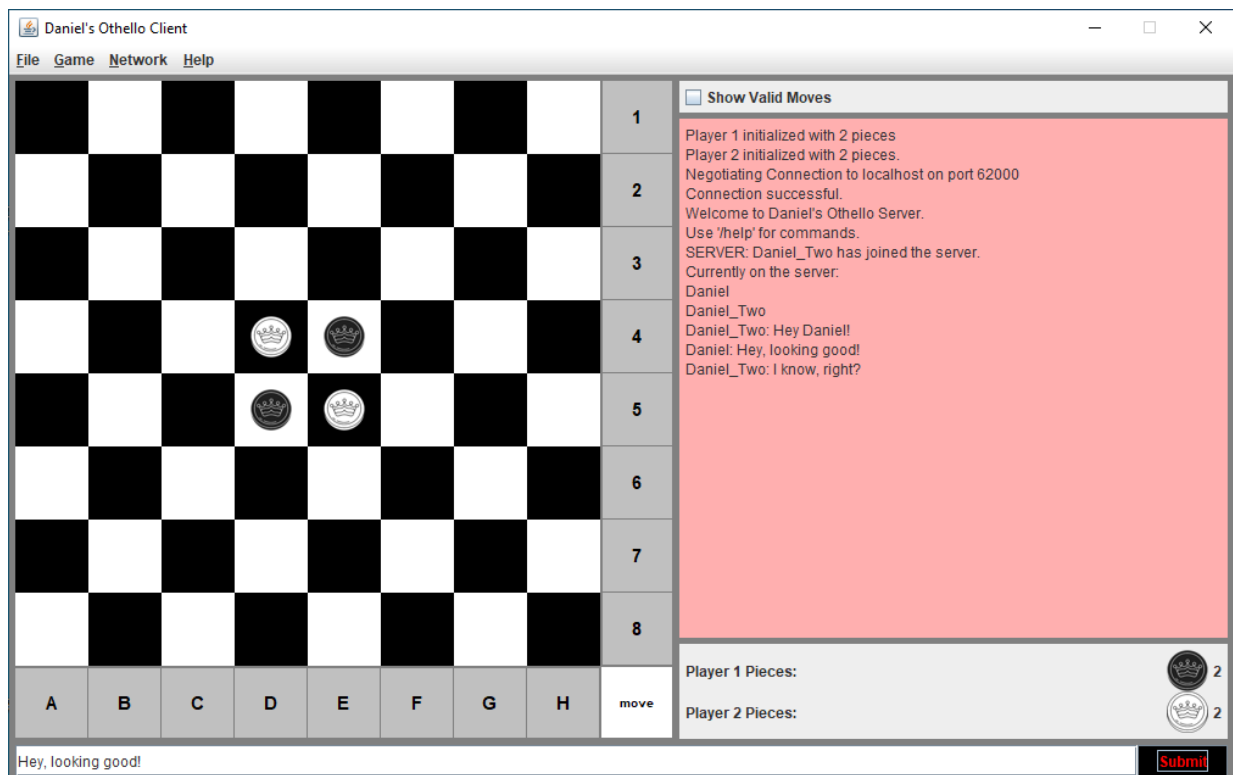


Figure 8: Who is on my server? I am! Twice!

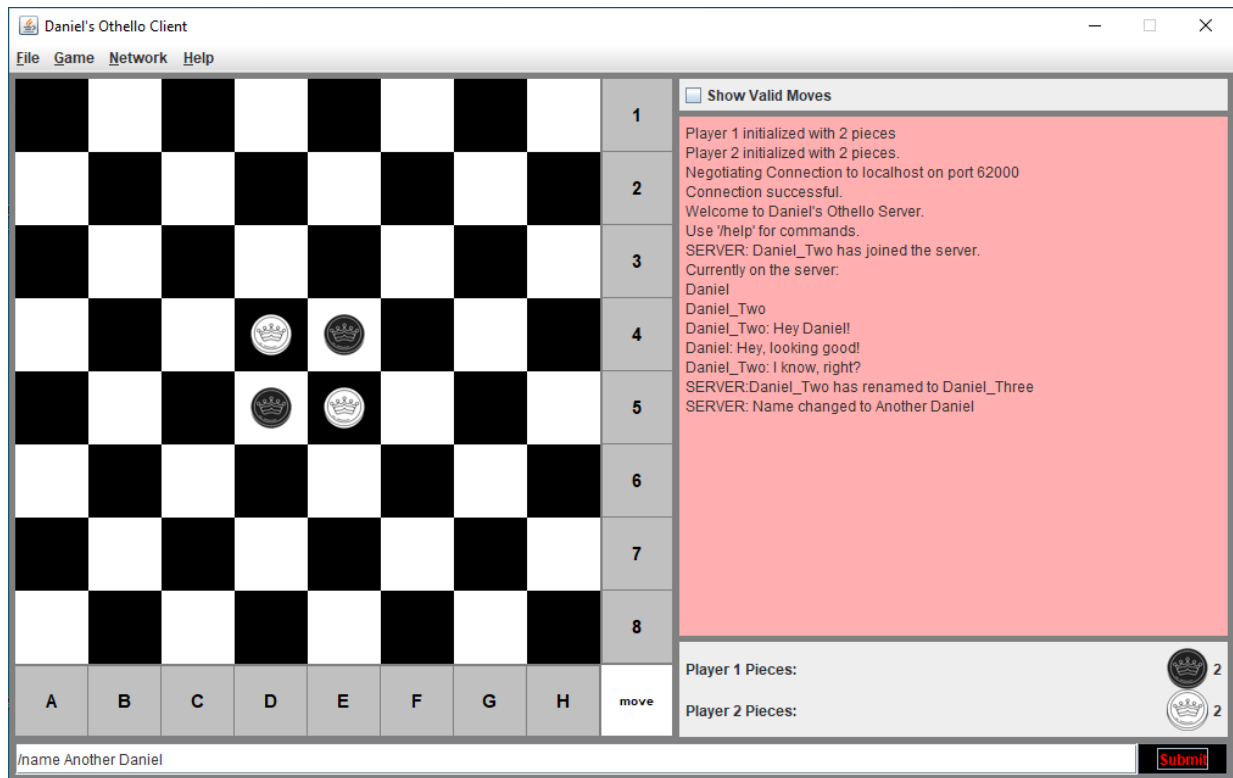


Figure 9: Name changing is getting out of hand.

Some specifics on `/name`: The user that initiates a new name change should get a server message: "Name changed to (new name)". Every other connected client should see, "(old name) has renamed to (new name)".

Likewise, both `/help` and `/who` should only report to the user that sent the command, not to all clients.

Helpful hint: `setName()` and `getName()` are reserved methods in a thread. You'll have to call them something else. While you could override them, I don't suggest it for this assignment.

Final advice:

Plan your code. While the server code isn't as intricate as `OthelloModel` was, there are a few complexities involved, and having a plan will help.

Your client and your server should not be throwing exceptions. Several network operations will require you trap specific exceptions; this will let you handle abrupt disconnections gracefully. Have your Client report useful error messages to the user wherever possible.

Test your code. Connect with multiple clients. Close the server abruptly with CTRL+C, or just shut close a client without disconnecting. If either the client or server throws an exception when you do this, read the stack trace, find where the exception occurred, and add a catch statement to handle it gracefully.

I will be running my version of the server from time to time, generally during my lab hours (refer to my schedule in Course Information). You can test your client against it if your router and firewall permit it. (IP: 69.196.152.225, port 21438) Because of how my

server works, telnet will not connect to the server properly and your connection will be dropped.

Connecting to my server is not strictly necessary. Connecting to your own is, of course. Roughly 60% of the marks in this assignment marks are in the server.

Useful material for this assignment will be covered in lectures. Do attend. You'll find it useful.

Finally, there can be no extensions on this assignment, it is literally as pushed-back as I can make it. A half-working assignment that's late is worth some marks, but I can not accept a late submission at all. Get on it early.

BONUS MARKS

There are a chance to earn a few bonus marks. These will only be available if your server and client work well—if I can connect to a server, transfer messages, execute server commands successfully and have no exceptions thrown under normal circumstances, then you'll qualify.

As these are bonuses, they will be held to a higher standard of marking. In all cases, provide a readme.txt file detailing what bonuses you are attempting, and a bit of implementation detail. You don't need an essay, but I should have a clear idea what you're doing. A paragraph or two will generally do. You may need to update the /help command as well.

Bonus 1: Emote messages (easy)

If a user types `"/emote (message)"`, it should appear without the colon. For example, if I do `"/emote is marking assignments"`, everybody connected will see, "Daniel is marking assignments" in their clients, as opposed to "Daniel: is marking clients".

Bonus 2: Unique Names on Server. (medium)

There's nothing stopping three people from having the same name. You should enforce unique names. Specifically how this is enforced is up to you. An already-connected user gets priority. (ie: you can't be Daniel on my server if I'm already Daniel).

Bonus 3: private messaging (hard)

If a user types `"/pvt (user) (message)"`, the user named (and only that user) should receive a private message (message). If the user can't be found, the user will get a useful error message. You may need to restrict user names and name changes in some manner. Detail those specifics in your document as well. It doesn't need to be an essay or anything.

Have fun with this assignment. There's still a lot of implementation and expansion that can be done. Make this truly your own piece of code, and remember:

"There is no tennis without a serve, and no client without a server." –Anonymous Swing Programmer.