Project Report
INF431 - Concurrence

Ian Duleba Erik Medeiros

2018

1 - Description of the Project

The project consists of a implementation of a server-client shogi application. Thus, the project has 2 fronts: the server application for managing the players and games; and the client side application where the game happens.

The game is runned at the client side and the server is responsible for matching pairs of opponents and intermediating its communication. The use of parallel programming was necessary on both fronts to render the application more responsive and to better manage all the possible games running on the server side.

The latest code can be found at:

- https://github.com/iduleba/Shogi-Master or
- https://github.com/MedeirosErik/Shogi

2 - Server side

This part was developed using Netbeans for it's simplified process of designs creation. By default, Netbeans uses java.swing, unless specifically configured. The client side was developed using JavaFX a more modern and robust alternative.

Firstly, we created a chat messenger app to test and learn how to deal with server/client communications. A large portion of it's code was reused in the final version of the server. It is available at https://github.com/iduleba/Shogi-Master/tree/master/Chat%20server.

On the server side we have used threads to deal with requests from clients. A main thread will keep listening on port 4444 and it will spawn a new thread to each time a request arrives. Our choice for the port 4444 was made due to the rarity in which it is used by other processes and it's lack of requirement of special privileges.

To show in real time the clients connected to the server a Jframe was created with a few basic functionalities: every five seconds it's table will list all the connected clients with some relevant information about each of them; start and stop buttons to halt and restart the server; and an option to create a client in the form of a chat dialog that enables to manually write the input to send to the server (a great debugging tool!).

Our server application can be deployed to a hosting service such as Openshift or heroku with little modification and it remains as a possible future goal of the project.

2.1 - Overview of the classes:

- **Start**

This is the class to run in order to start the server. It will display a dialog box and after the button is pressed, a thread that will open a Socket and start listening port 4444 is spawned (instance of Listener). Finally we move to Control.java.

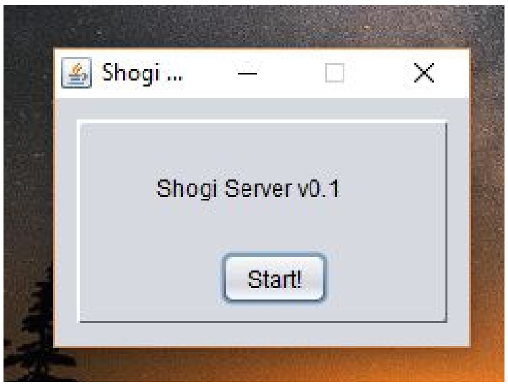


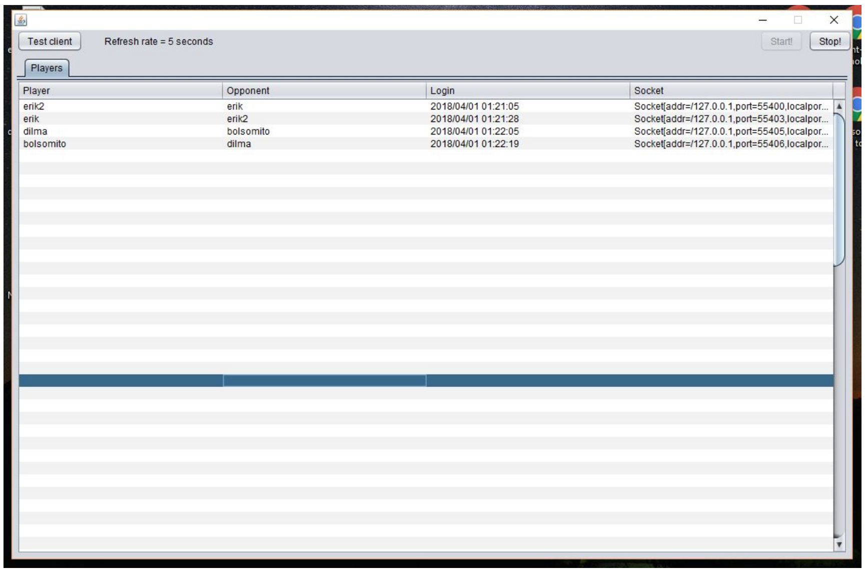
Figure 1 - Start Server

- **Listener**

Listener waits for messages in port 4444 and when a client connects it will start a Thread to handle the connection (instance of clientThread).

- **Control**

This is the Jframe that shows the data of the clients and enables the user to control the server.



Refresh rate = 5 seconds

Player	Opponent	Login	Socket
erik2	erik	2018/04/01 01:21:05	Socket[addr=/127.0.0.1,port=55400,localpor...
erik	erik2	2018/04/01 01:21:28	Socket[addr=/127.0.0.1,port=55403,localpor...
dilma	bolsomito	2018/04/01 01:22:05	Socket[addr=/127.0.0.1,port=55405,localpor...
bolsomito	dilma	2018/04/01 01:22:19	Socket[addr=/127.0.0.1,port=55406,localpor...

Figure 2 - Server running

- **ChatClient**

ChatClient was created in the first stages of the project. It is a client of an chat server. It will be instantiated when the user presses the button “Test client” in the Control Jframe.

Using this tool, It is very easy to break the flow of the communication without the care to follow the nominal path, i.e.:

1) A client connects to the server (user does not have control over)
2) Server binds the connection and waits for the name of the client (it is the name that will be displayed in the board during the game). So the first string to be send will be the name.

3) Client sends name and awaits for a response from the server with an confirmation that there is an opponent and the game can start.
4) Server receives name of the client and searches an opponent that is waiting for a match. If no opponent is available, server leaves client on hold for 30 seconds. If no one connects after that period, the server automatically notifies the client and closes the connection.
5) If a match is found, the server sends the name of client1 to client2 and vice-versa, as well as who starts the game.
6) Both clients will now await for moves in each turn. For example: client1 is the first to play, so client2 will stand by awaiting for client1's move. After client1 sends its move the situation reverses until one of three events take place:
    a) either the game has ended by checkmate;
    b) or one player leaves the game;
    c) or the server shuts down.

   So this time we need to send a move to the server. A move will be a string containing six digits, the first two will encode the position of the piece to move (x,y), the second and the third will encode the position that the piece will be positioned, the fifth will be a zero if the move will promote the piece or a one if it doesn't and finally, the last number will be a one if the piece is a captured piece (currently in the graveyard) and a one if the piece if not. The x's and y's range from 1 to 9, the origin being the upper right corner in the board (be it the cemetery or the main board).

At any given moment the server can shutdown, or if the game has already started, one client can leave it by closing it's window (or in the case of this class the window of the chat). The server will make sure that every party involved will terminate the game and close the connections correctly.

- **clientThread**

This is the class that is called by Listener upon the arrival of a client. It handles the communication described above.

Something worth mentioning is that an active approach was chosen to handle the connection, i.e., the server will periodically send messages to the client to make sure it is still connected (KeepAlive).

3 - Client Side

The client side application was done using the javafx API. This API was chosen for its simplicity in creating graphical user interfaces (which can be done using a scene builder software).

The application opens at the login window where it's required for the user to input the server address, when the user clicks at the start button the program connects to the server and waits for an opponent. After that the game starts.

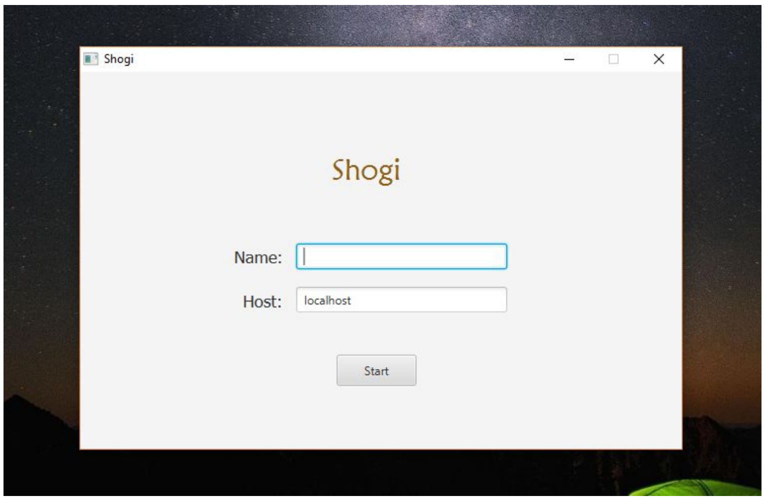


Figure 3 - Login window

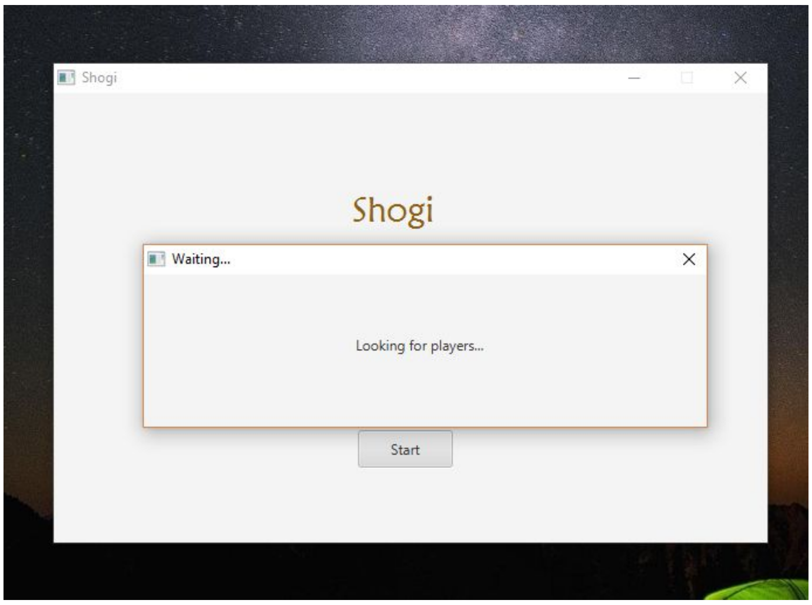


Figure 4 - Waiting for an opponent

The game is played using the mouse to move the pieces, the user first needs to select a piece and then select the square to where he wants to move the piece. All the moves are executed by the javafx mouse event handlers and the mouse events are handled only when it's the local player's turn.

The main aspect of the client application is the simulation of the remote player by reading his encoded moves (in form of String) and executing them on client side. On the other hand, all moves done by the local player are also encoded in String form and sent to the server which is responsible for redirecting them to the opponent.

Since the performing of long tasks in the GUI thread renders the application unresponsive, there are some threads responsibles for communicating with the server and simulating the opponent's moves. We created one thread to keep reading the messages sent by the server and storing them in a buffer, when it's the remote player's turn we spawn a thread that keeps reading the buffer until it finds the move, when the String corresponding to a move is found the program executes it and then pass the turn to the local player.

Classes

The most important classes are:
Game class: responsible for setting the graphical interface and managing the opponent's moves. This class spawn the Thread that simulates the opponent.

LoginControl class: responsible for logging in the server. It creates a thread that logs in the server and waits until an opponent is found.

Connection class: a extending Thread class that keeps communicating with the server and putting all relevant information in a buffer.

Many other classes were created to facilitate the game and GUI implementation such as Board, Piece, Player, MouseEventHandler, PieceEventHandler, BoardEventHandler, etc.

4 - Issues

A significant amount of time was spent learning how events, handlers and sockets work in Java and consequently, an even greater time was dedicated to fixing bugs.

Given the size of the project and the difficulties found, the result at the moment of this report's writing is incomplete. What is still left to do is fix the bugs in the movement of the pieces and their exchange in the network.