Multiple User Games

By: Jingjing Wang & Chris Weitzel
Goals:

The goal of this project is to create two fully functional multi-player games. The games should be connected by a menu which allows the user to select which game they would like to play. Each game should use a different technology to implement its network communication. Jingjing programmed Gobang using RMI, and Chris programmed checkers using sockets. We worked together on the game selection menu. The original version of the menu was written by Jingjing, and simply contained two buttons for starting the game. Chris rewrote parts of the code in order to make it work with his game. We worked together in order to update the look of the menu in the final version.

The Games:

Gobang:

Jingjing is responsible for this game. It is also known as Five Chess which is an abstract strategy board game. It is traditionally played with go pieces (black and white stones) on a board. Once placed, pieces are not allowed to be moved or removed from the board.

Rules:

The rule of this game is very simple: Black plays first, and players alternate in placing a stone of their color on an empty intersection. The winner is the first player to get an unbroken row of five stones horizontally, vertically, or diagonally, one example is shown below, in this case, the player who holds the black piece is the winner. For example, in figure 1, the player who holds black piece is the winner.
Figure 1

Design:

In this game, it allows two players to play the game with each other. It is designed first two clients who click on the “start” will be players. For others, after they click on “start”, they could watch the game and also could choose to chat with other clients in a public chat room.

Architecture:

Since this is a distributed game, so it must allow multiple clients to communicate with others, to implement this, this game uses RMI technology rather than socket technology. Because socket technology is possible to make a conflict when the server sends the data to multiple clients.

For this game, there is a server that need to be started at first which is the central part of this design. To make the application be easy to run, the developer put the steps that needs to start rmiregistry in the code. So when starting the server (java BorderServer), it will start at first automatically.
After the server being started, it is waiting the clients to finish the setup of the communication. At this time, when the client click on “start” , then it will connect to the server and also will invoke a remote method defined in the server side, which is called

\[
\text{void addCallback(String name, PlayerCallBackInterface CallbackObject)}
\]

The client will submit its name and its object reference to the server which will be stored into a vector in the server side. The reason to do this is that the server needs to invoke the method defined in the client side during the game process.(e.g. player.Redraw() ). When the server needs to call the method in the client side, it can finds the object reference one by one from the vector like the following:

\[
\text{PlayerCallBackInterface player = (PlayerCallBackInterface) callbackObjects.elementAt(i);} \\
\text{player.ReDraw(x, y, mytype);} \\
\]

Then this method will be implemented in PlayerCallBackImpl class in the client side.

Also, after two players join the game, the game will start. The server will first give the player who hold black piece the turn to place a piece, and after that change the turn after each valid move.

Features:

Since mentioned before, this game is able to allow two clients to play with each other and multiple clients to watch the match and chat with others. The figure 2 is Graphic Interface designed in this game:
The Board size in this designed game is 19 rows and 19 lines, and this board is drawn by a class called DrawBoard.

In the game menu, there is three submenus: Start, Restart and Exit. The “start ” menu is used to join the game and add the object reference to the server. And “Restart” menu only can be clicked by two players, not others. When a player clicks on this menu, it waits another player to click on it also. And at the same time, in another player side, there is an information to remind this player that another player asked to restart the game. When both players clicked on “restart” menu, the server will restart the game. The “help” menu is used to provide the rule of this game. If a client would like to chat with other clients rather than just simply watches the game, then he can click on “ChatOnline” menu, then a chatroom will be displayed at the right side (see figure 2), the participants shows the name of the clients that also joined the game.

The bottom side provides the information of players and the piece type that they hold. Also, in this area, it will give some different kinds of information based on the distinct situations during the game. For instance, when the winner appears, it will show the winner information.
For the board part, the developer adds the MouseListener to it. The use of it is that when the player click on the board, the position of the click \((x, y)\) will be got and can translated into the row and line numbers, then the row, line numbers and the piece type will be sent to the server side. The server will store this data and will judge this move which will be discussed late.

**Judgement:**

The server will not only build up the communication between clients, but also should judge every action of the player, and then apply for different situations. First of all, it should check whether the movement is a valid move, that means the player can place on a empty intersection in the board, if not, the server will invoke a method defined in this client to tell this client must move again. Also, after each valid move, the server will judge whether winner appears. The algorithm for the judgement of the winner is very simple, we can look at the figure 3:

\[
\begin{array}{ccc}
2 & 3 & 4 \\
1 & (x, y) & 5 \\
8 & 7 & 6 \\
\end{array}
\]

**Figure 3**

In figure 3, \((x, y)\) will be the position of the piece, and the server will check its 8 directions. If there is a line which has equal or more than 5 continuous same pieces, then the winner will appear. So in this case, to make the algorithm be easy to implement, the developer declared three arrays to store the valid move, one is to store the black piece’s valid move, the second is to store the white piece’s valid move and the third is to record both black and white piece’s valid move. For the judgement, the developer only used the first and the second array.

**Fault Tolerance:**

Sometimes due to the failure of the network, or the problem of the client itself, the client may disconnect with the sever. Since in this design, the client registers itself into the server. And during the game, the server need to remotely invoke the clients’ method based on the vector that stores the client
object reference. So in this case, if there is no crash detection function to delete such kind of object references, then the object reference will still kept in the vector, and thus every time will be still be invoked by the server. The result is that a lot of exceptions will be produced and the performance will be seriously decreased.

So to avoid this situation, in this game design, the server will detect whether each client connects or not in every six seconds. This detail is the server will invoke a remote method defined in the client side:

```
try {
    if (player.Check()==1)
    {
        ........
    }
}
```

```
catch (Exception e)
{
    ........
}
```

In the client side, there is a defined variable, which value is one. `player.Check()` just returns this variable. If a client does not connect with the server, there is an exception will be thrown from this method. And the catch hander will deal with this exception to remove this client object reference from the vector. Also it can check whether this disconcerted client is one of the players. If so, then the game will be over, and the server will tell other clients who is the winner.

**Problems:**

First, when the server sends the winner information to all the clients. Originally, the developer would like there is a `JOptionPane` to show this message, but the developer found that sometimes it has
something wrong with the display of JOptionPane, and the last piece could not be displayed until the developer click on “ok” button. What the developer wonder is that the last piece and the message should be displayed simultaneously. The solution is that the developer gave up the use of JOptionPane to show the winner information. Instead, the developer used swing to draw this information at the bottom of the board.

Second, the board is kind of difficult to draw, because it requires the developer to do the translation between position(x, y) of piece and line or row number of the board. The developer should find the relationship between them. Also the developer needs to find proper black and white piece images that can fit the distance between each line and row. After taking some effort, this problem was solved.

Checkers:

Chris is responsible for this game. It should enforce all of the standard rules of checkers, present the user with an acceptable GUI, use socket based communication, and allow private chat between the two players.

Rules:

Each player is assigned a color (red or black) and 12 pieces of that color. These pieces are placed in the light squares on the closest three rows of the checkers board. On their turn, a player may select one of their pieces to move. Pieces may only move diagonally onto one of the light squares touching the square they are on. Regular pieces may only move towards the far side of the board from the player. Once they reach the last row they become “kings” and can move towards or away from the player. Capturing an opponent’s piece is accomplished by moving your piece over their piece (i.e. “jumping” it) and landing on the square behind it. If a jump is available to a player they are required to take it. It is also possible for a player to make multiple jumps in a single turn as long as it is with the same piece. The game ends when a player either has no remaining pieces, or has no valid moves when their turn starts.

Features:
The games user interface displays all the information available to a player. The checkers board accurately reflects the state of the game. The status panel along the top displays the number of pieces each player has remaining as well as notifying the player if it is their turn. The chat box along the bottom of the window displays the conversation between players as well as any notifications by the server. When the game ends, both players are notified of who won as well as the reason they won and asked if they would like to play again. The server then resets and waits for two new connections. If one or both of the players chooses to play again their user interface resets and attempts to reconnect to the server.

**Design and Implementation:**

On the server side a socket is created and the server waits for two connections. When a connection is accepted, the server creates a new thread for the player, notifies it of which color it controls, and waits for the second player. Once it has both players it notifies the red player that they may move. As the game progresses, the player threads handle receiving and sending messages to and from the players. These threads will parse messages (an array of strings) and decide what the player is requesting. All move validation and chat handling is done in the server. Move validation is done using an integer array that represents the board. The integer in each cell represents the state of that cell (i.e. 0 is empty, 1 is red, etc.). The server checks the requested move to see if it is allowable. The server grants permission to move to the proper player by means of the yourTurn() method. This method sends a simple message to the client program telling it to change the variable that allows the player to request a move. However, before the message is sent, the server checks to see if the game is complete. It checks the count of red and black pieces to see if either player has no pieces left, and then scans the board. If the player who is about to move has no valid moves, then it is a stalemate and that player loses. When the server decides that the game is over, a command is sent to the clients telling them to end the game as well as why the game is over and who won. The server then closes the connections, the player threads end, all variables reset, and the server waits for two new connections.

On the client side, the user selects to play checkers and a menu appears. The player may input their name (To be shown in the chat log before their messages), and the server’s ip/port number. Next the connection is made and the GUI is drawn. At this point the player can see the board and status panel and can chat (though if there is no other player they will simply get an error message). The board is made of 64 JPanels. Each JPanel represents the status of a square (Different icons exist to show the different states of each square). Thirty-two of the squares are dark squares and are not really used for
the game. The other thirty-two will have a mouse-listener added to them. When the player clicks a piece, the GUI decides if they are allowed/able to move that piece. If so, it is activated and the GUI waits for the player to select a destination. Once a second click is received the client uses the requestMove() method to send a move request to the server and change its turn variable to false. The server will validate the move and, if it was a jump, check if a double jump is possible. If it was invalid, the server responds only to the client that sent the request and the client is allowed to try another move. If it was valid, the response is sent to both clients and the server informs the appropriate player that it is their turn. When the command to end the game is received, the client creates a text message based on the reason the game ended, and displays a JOptionPane to the player. If they player selects to play again, they will reconnect to the server. If they choose not to play again, the program exits.

**Fault Tolerance:**

The majority of possible errors are caused by network connectivity and the array that the server uses to represent the board. If a player disconnects from the server, both the client and server must know that it has happened. The client side will display an error message and exit. The server side will declare victory for the other player and end the game. During move validation, when the server is checking to see if a player can make a jump it must check the possible destination squares. In order to make sure that it does not cause an arrayOutOfBoundsException or a logic error the server must see which column and row the piece is in before it checks for possible moves. If it is near the edge of the board, the appropriate square (which does not exist) is not checked, and the exception/logic error is avoided.

**Problems:**

The largest problem with the Checkers program is actually caused by the design goals. Socket communication turned out to be a bit annoying. The part of the code used to parse through the messages in order to determine which method to call became quite large. The only reason that RMI was not used instead was that we wished to use two separate communication technologies.
Future Development:

I would like to reimplement the code and change a few things. First, it will use RMI. Second, it will use an interface that lets the player pick up and drag the pieces instead of clicking on the square. Finally, I would like it to have a parallel server, so that multiple games may be played at once.