CE303 – Advanced Programming

Assignment 1 – Socket Based Game – Report

Student Name: Bradley Rex

Registration Number: 1504843

Table of Contents

1.0 Instructions:

In order to play the game, the GameServer.jar file must be run first to start the server and allow the clients to connect. Then at least two of either GameClient.jar or GameBotClient.jar or a mix must be run in order for the game to begin its ten second countdown until the players can place tiles on the board. The steps are as follows:

1. Double click the GameServer.jar to run the server. A window will appear to allow you to cleanly shutdown the server after the game has ended.
2. Then, double click either GameClient.jar (for a human client) or the GameBotClient.jar (for a bot client). Do this step at least twice to start the ten second countdown until the players can place moves. Note: players can join mid game, but the ten seconds allows up to 3 additional players to join before the game starts. If you want more time to connect clients, then simply run a human client as the first player so that the game waits on your first move.
3. Once the game has ended, simply close all client windows and the server window.

To run the jars from the command line, follow these steps:

1. Open a command prompt window.
2. Change the directory so that you are in the same directory as the GameServer.jar, GameClient.jar, and GameBotClient.jar. This will be in:
    - ~/Submission/Executable_Jars/

    Of the extracted folder.
3. Then type the following to start the server:
    - java -jar GameServer.jar
4. Once the server has started, use either of the following to start a human or bot client respectively:
    - java -jar GameClient.jar
    - java -jar GameBotClient.jar
5. Note: the same rules as above apply. The server must have two connect clients before starting and cannot have more than five. Bot clients will take ~1.5 seconds between moves.

2.0 A Description of the Program:

2.1 Overall Structure:

The approach taken for this assignment was to use Java's Sockets to connect clients to the server and transfer data between them. The server program, GameServer.java, starts by creating a frame for the GUI, creating a new Game object, and creating a new ServerSocket object with the port number: 8080.

The server will then loop until the game is finished, but while looping it will try to establish a connection with a client and will block the thread until a client has connected. It will then continue to increase the player count, create a GameService object and run that service in a new thread. Each thread will process the commands sent by the client program, send responses back to the client, and handle all game updates through the Game object that it was given a reference too.

Once five players have connected, the server will continue to loop, sleeping for 1 second between iterations, until the game has finished.

The Game object created in the server's thread represents the state of the game at any point during the game and provides several methods for changing the state of the game and performing checks to ensure a move is compliant with the games rules.

The Game object keeps track of the state of the board via a 2D int array where each inner array is a row in the game and each element within the inner array is a column which represents the mark of the player who occupies the tile in that row and column. It also stores a list of GameService objects which represent connected players of the game. This allows to the Game object to call methods on the GameService objects which can then e.g. relay data to the client's program. In addition to the list of players, the Game object also maintains a PlayerMark variable which keeps track of the mark of the player whose turn it currently is. By having a shared game object which has access to all the connected players, any update in the games state will cause the game to send out an update to all players so that they can update their client programs e.g. the client's GUI.

The Game object has several methods for updating the game state. One method is the addPlayer() method, this adds a GameService object to the list of GameService objects and when the player count is exactly 2, it handles the starting of the server. It does this by creating a new thread to perform a ten second count before informing the clients that they can move, or that it is someone else's turn.

It also has the method makeMove(), which returns a Boolean true if the move was valid with the card used, at coordinates (x, y) and with the player mark of that player. If this method returns true when it is called by the GameService, then the GameService object which called it will then call the appropriate Game methods such as sendBoard() (which sends the game board to all the players as a flattened array by calling the GameService objects updateBoard() method), checkedBlocked() (which checks if any player was blocked after that move), and nextPlayer() only if the player didn't use their double card (which determines the next player in line to place a move and calls the informPlayersOfTurn() method to inform the players).

Another important method in the Game class is the endGame method. This is called when all players are blocked and will call methods to determine the winner and the scores before sending this information to the clients through the GameService objects method called 'end()'.

Next is the GameService class. This class acts as the line of communication between the client and the server. It runs in a separate thread for each connected client and will continuously await input through its BufferedReader object, 'input'. When a client issues a command, the string received will be split and sent to the parseCommand method where it checks that it is the clients turn before attempting to place a move (Unless the command is END in which case it doesn't need to be that client's turn in order to parse the command).

If it is the clients turn (determined by comparing the Game objects playerMarkTurn with the client's mark), then the appropriate methods of the game object are called in an attempt to make a move on the game board. In all cases, the parseCommand method will then send a response back to the client via its PrintWriter output stream, 'output'.

Some other GameService object methods are updateBoard(), which is called by the Game object in order to send the state of the game board to the user. This again works by sending a string via the PrintWriter object. This is similar to informClientOfTurn() which informs the client of whose turn it is. The whole protocol will be detailed in the following section.

Another class is the GameClient class, this class represents the program run by the client to connect to the server and play the game through. It sets up a GUI, which has a grid of Tile objects, each of which have a mouse listener attached to them in order to

handle mouse click events on them. The ‘play’ method of this class contains the main loop which waits for a response from the GameService object it is connected to.

The mouse listeners attached to each of the Tile objects displayed in the GUI allows each tile to be clicked to issue a move command to the server. The GameClient will then receive an appropriate response to issuing that command with the game in that state. The client can also use the JRadioButtons at the bottom of the GUI to select a card to use before placing a tile.

The parseResponse() method of the GameClient and GameBotClient classes is what handles the responses from the GameService object it is connected to. This will then make appropriate changes to the GUI, such as changing the colour of a Tile and updating the turn indicator in the bottom right as well as the local game board. Once the game is over, this object will receive the END response from the GameService class which will make this GameClient close the socket and display a JOptionPane indicating the winner and the scores.

The final class is the GameBotClient class. This class is similar to the GameClient class with the exception that it doesn’t allow a user to click on the tiles and instead makes random moves when it is the bot’s turn. The GameBotClient class works by waiting for a TURN response from the server. When the TURN response is received, a check to see if it is the bot’s turn is made. If so, the bot then enters a loop where it makes random moves until the GameService object sends a LEGAL_MOVE response, in which case it exits and then waits until it is it’s turn again.

Some other small changes were also made, such as removing the mouse listeners from the tiles, changing the radio buttons to JLabels and adding a ‘Last Card:’ indicator to make the GameBotClient class unable to be interacted with by a human player.

Overall, the GameClient and GameBotClient objects allow for commands to be sent to their corresponding GameService object which can then communicate with the Game object setup by the GameServer object, in order to place moves, gets updates and end the game.

2.2 The Protocol:

2.2.1 Client Commands (From Client to Server):

- MOVE <influenceCard> <X> <Y>
  - This command issued by the client is a string with the first word being 'MOVE', this is followed by a space and then the name of the influence card to be used (NONE, DOUBLE, REPLACEMENT, FREEDOM). This is then followed by a space, then an int indicating the row, a space, and then another int indicating the column. This will then attempt to place the players mark using that influence card at those coordinates.
- END
  - This command instructs the GameService object of the client to exit its loop and then attempt to close the socket connection to its client.

2.2.2 Server Responses (From Server to Client):

- MESSAGE <aMessage>
  - This response indicates to the client that a message should be printed in their console. E.g. the server sends a welcome message when the client first connects.
- MARK <aPlayerMark>
  - This response is used to update the clients program with their mark for the game they are currently playing. aPlayerMark is the string representation of a PlayerMark.
- BOARD <M> <M> <M> …
  - This response is used to send the entire game board as a flattened array to the client for them to update their local board with the correct tile colours. <M> represents the int representation of a PlayerMark. E.g. The first 10 <M>'s represent the marks of the 10 tiles on the first row of the board from left to right.
- TURN <aPlayerMark>
  - This response sends the mark of the player whose turn it currently is. <aPlayerMark> is the string representation of the player mark. This is used by the client to indicate when it is their go or whose go it currently is. Also used by the bot to determine when it should play.

- LEGAL_MOVE <influenceCard>
    - This response indicates to the client that the MOVE command that they issued was legal and sends the card that was used for that move back to client. This is so the client can update their available influence cards.
- ILLEGAL_MOVE
    - This response indicates to the client that the MOVE command that they issued was illegal and that they should attempt another move. If the ILLEGAL_MOVE is sent, then the client can assume no changes were made.
- INVALID_MOVE
    - This response indicates to the client that the MOVE command that was issued was not formatted correctly. As the GUI deals with the MOVE commands sent, this response should never be received.
- END <winnerMark> <score> <score> …
    - This response indicates to the client that the game is now over. It also sends the mark of the winner, <winnerMark>, as well as the scores, <score>, for each player in the game in the order in which they joined. This information is used by the client to display an appropriate JOptionPane.

2.3 Handling Concurrency:

This program uses a single Game object shared amongst all the connected clients which are each running in their own threads. This is therefore a cause to ensure that concurrent access to this object is maintained for the correct behaviour of the game.

The first and most important approach for ensuring safe concurrent access to this object was through the use of a PlayerMark called playerMarkTurn. This is an instance variable of the Game object which keeps track of the mark of the player whose turn it currently is. This mark is then used by each GameService object connected to the game to determine whether it is their clients turn whenever their client issues a command. Due to this, it is not possible for a player to call the makeMove() method if it is not their turn and as this variable is only changed after the game state has been updated which ensures that the next player cannot make a move on top of the previous player.

However, there is one instance where this variable cannot prevent concurrent access to the Game object and that is when a new player joins the game. This is because when a new player joins, they automatically have a tile placed for them and this move is made even if it is not the players turn. Therefore, in the unlikely event that the current player places a tile in the same spot as the new players initial tile, then one may overwrite the other. To address this, the makeMove method of the Game object has been made synchronised, so that either the current or the new player gets the tile and the other will receive an ILLEGAL_MOVE response and must try again.

2.4 Unit Tests (In GameJUnitTests.java):

2.4.1 The testFreedom() Unit Test:

The testFreedom() test starts by creating a new Game() object which is used by the test to access the makeMove() method which returns a Boolean indicating if a move was legal (true) or illegal (false).

The test places a red tile in the top left corner of the board (0, 0) and then attempts to place a red tile through the use of makeMove() at row 0, column 2, which is not adjacent to another red tile.

When the makeMove() method is called with a NONE influence card type, it returns false as expected. In addition, when called with a FREEDOM influence card, it returns true as expected as the move no longer has to be adjacent to other tiles with the same mark.

The final assertion made in this test is when a tile is placed somewhere that isn't adjacent to a tile with the same mark with a FREEDOM card, but the tile isn't empty. This should return false as freedom does not allow a tile to replace another and, as expected, this is what the method did return.

2.4.2 The testReplacement() Unit Test:

The testReplacement() test starts in the same way; by creating a new Game object. It then continues to place a red tile at row = 0, column = 0, but also a green tile at row = 0, column = 1, which is therefore to the right of the red tile.

The first assertion tests whether a move can be made on top of the green tile with the NONE influence card. As expected, the makeMove() method returns false in this case.

The next assertion then attempts the same move but with a REPLACEMENT influence card. The makeMove() method in this instance should return true as

the green tile is replaced by the red tile, which is what the method does indeed return.

The next assertion ensures that the replacement card doesn't allow the moves to be placed which are not adjacent to tiles with the same mark, as this is what the freedom card does. Sure enough, this call returns false.

The final assertion tests an additional feature of the replacement card. It tests that a replacement move cannot be made to replace a tile that has the same mark as the one being placed; this prevents the card being wasted. As expected, this call returns false when the tiles are the same.

2.4.3 The testCheckAdjacent() Unit Test:

The final test, checks whether the checkAdjacent() method in the Game class behaves as expected and returns the correct Boolean. It begins by creating a new Game object and placing a red tile in the top left corner (0, 0).

The first assertion checks to see that when the checkAdjacent() method is used on a lone red tile in row = 0, column = 2, it returns false as it is only surrounded by empty tiles and the edge of the board. Sure enough, this call returns false.

The next assertion checks to see if a tile at row = 0 and column = 1 has a red tile in one of the 8 tiles surrounding it, which it does, at row = 0, column = 0, and should therefore return true. As anticipated, the method call returns true.

The final assertion in this test provides an extra test to ensure that not only does a tile have to be adjacent to at least one non-empty tile but at least one non-empty tile must have the same mark as the one looking to be placed. It does this by placing a green tile at row = 0, column = 1 and then checking the red tile at row = 0, column = 2 for adjacent tiles. As expected, despite have a non-empty tile adjacent to it, because it does not have the same mark, the method returns false.

3.0 The User Interface:

3.1 The GUI Overview:

The GameClient and GameBotClient classes both use Java Swing to create a GUI that the user can use to either interact with or use just to visualise the game state from that players perspective.

The GameClient has a frame which displays all of the games tiles in their respective colours in the centre of the frame. The human player can then click on these tiles in order to issue the MOVE command to the server. In addition to this, the GUI also has four JRadioButtons located at the bottom of the frame. These can be used to select which of the influence cards should be used when the user next clicks a tile. Once a card has been used, it is disabled and cannot be selected again, except for the NONE card. The GameClient also has a turn indicator in the form of a JLabel in the bottom right of the frame, this displays either the mark of the player whose turn it is or displays that it is that players turn.

As for the GameBotClient, the GUI is very similar with only small changes. The GameBotClient GUI no longer has mouse listeners attached to the tiles, this is because the bot doesn't use the GUI to issue MOVE commands. Also, the influence card radio buttons are no longer radio buttons and are instead labels which are greyed out once they are used, as again, the bot doesn't use the GUI to select the influence cards. Finally, in addition to the turn indicator, there is a JLabel which displays the card that was used last by the bot. Due to all of this, a user is able to easily see what cards have been used, what cards are available and what the last card used by the bot was.

3.2 User Friendliness:

By implementing the client program to use Java Swing to create an interactive GUI, the player can easily make moves and select their influence cards just with simple mouse clicks; there is no need to know the underlying MOVE commands.

Also, by disabling a radio button when a card has been used, the player cannot attempt to use a card they don't have. The player is also able to clearly see when it is their turn and, when it isn't, clearly see whose turn it is. Making for very easy, intuitive gameplay.

As for the bot client, by restricting the user's ability to interact with the GUI, it means a user cannot interfere with the bot's gameplay. Plus, with the addition of the 'Last Card:' indicator, a user can clearly see when a bot has used an influence card.

4.0 The Bot's Logic:

The logic for the bot player is very simple and follows some basic rules:

- When it is the bots turn, the bot will continuously attempt to place tiles at random locations until it receives a LEGAL_MOVE response.
- The bot will always save its replacement influence card until it has no empty adjacent tiles, or the board is full.
- The bot will always save its freedom influence card until it has no empty adjacent tiles and the board is not full, or until half the board is filled up with player tiles, in which case the bot uses the freedom card to establish a presence elsewhere on the board.
- The bot will randomly choose to use a double influence card. The card is used when a random int from 0-19 (inclusive) is below 5, giving the bot a ¼ chance of using a double card. This allows them to use it early in the game, so they don't reach the end without using their double influence card.

5.0 Review of the Project:

At first, this project seemed very challenging and very complicated as it incorporated a large amount of what was taught in the lectures and the labs, but also because it required a good understanding of topics not covered in the lectures such as Java Swing.

One of the most challenging parts of this project was understanding how the client could communicate with the server using just strings passed via output and input streams. However, this became clearer after establishing a basic protocol with simple parsers for the commands and responses, where it then became a case of just adding commands/responses to the protocol as needed to provide some more functionality.

Another challenging aspect of this project was the testing of the server and clients as this often involved having multiple programs running on one machine/screen and it was easy to lose track of what window was what player, etc.

Also, it was quite challenging trying ensure that all the appropriate checks were made on a move before it was considered a legal move as this involved checking different aspects depending on what card was being used, so resulted in a large number of loops and if else statements being used.

On the other hand, one aspect of the project which was easy was the making of the GUI. I have had a fair amount of experience using Java Swing in previous modules, so making a basic GUI was straight forward and simple. Also, the making of the GameServer class was quite simple and straight forward as most of the game's logic and other aspects of the game were delegated between the Game class and the GameService class, so all the GameServer class needed to do was track the number of players and get connections to the clients.

Another aspect that proved easier than expected was the making of the GameBotClient. This was because of the way the GameClient and GameService classes worked, which meant the bot only required some changes to the GUI and then a method which was called to decide which move to make, rather than having a GUI for the user to interact with. The bot's logic was therefore the most difficult part of making the bot as everything else only required small changes from the GameClient class.

One aspect that I'm particularly proud is the UI for the project. I chose to implement a GUI for the client to interact with as opposed to using Putty or the terminal/command line. This makes the game a lot easier to interact with and visualise whilst making it more fun at the same time. I am also happy to have implemented the bot player in such a way that a game of just bots can be played with ease.

However, one feature that I would have liked to have implemented was the ability to play multiple games without having to restart the server after each game. I also would have liked to have improved the efficiency of the bot's logic by allowing it to only attempt moves on tiles which it knows are free as well as prevent it from using a freedom card just to place a tile adjacent to one of their own as this defeats the point of the influence card.

As for managing the project, I feel that I organised myself quite well. I started off by adapting the banking example seen in the lectures to start passing messages related to a game such as the state of the game board and incrementally added to the program to start utilising the sent messages on the client's side as well as using the client's commands to update a shared object on the server.

In hindsight however, I would dedicate more time to improving the efficiency of the game updates as well as the bot client. As it stands, the current solution will send the game board out to all clients after each move has been made. However, this is unnecessary as just a message to update the specific tile that was changed is required after each move, not the entire game board. Also, to increase the user friendliness, next time I would dedicate more time to implementing a feature to allow multiple games to be played without restarting the server as it is rather cumbersome to have to restart the server once a game has finished.