COMP90015: Distributed Systems
Assignment 2: Scrabble - Report

Dean Pakravan: 757389
Qijie Li: 927249
Luis Zapata: 938907
Dongming Li: 1002971

14th October 2018

1 Introduction

This paper is a report on the team project - Scrabble, and the project is for the subject COMP90015 Distributed Systems. The project is a java application that runs on a distributed system which is an altered version of the board game Scrabble.

2 The Project - Scrabble

2.1 System Architecture - Overall Design

In project 1 (Dictionary Client and Server), though we implemented in Java, our distributed systems were implemented with sockets and threads - a simple, yet straightforward system. For this project, we had the idea of only having *one* jar file. Any player can host or join a game in one jar file rather than having a separate server and client jar files. This lead us to implementing Java RMI; when after we designed the GUI and the initial game design, it seemed like the best design choice for us.

The basis of the RMI is this:

- A player who hosts the game is the server.
- Whenever an action is performed (e.g. A turn is passed, a tile is placed or a word is claimed), the client sends the information to the server.
- The information in this case is three core parts. The State of the game (ref. GameState), the board and the list of players.

- Once the server has this new information, it will broadcast the new board, the next player and the new state of the game to all other current players.
- Rinse and repeat until the game is finished.

When players open up the jar file, they are given the option of "Joining a game" or "Hosting a game". Hosting a game will make that player the server of the game. The host will hold all the necessary information about the current game being played. The host will need to enter the following information:

- Username
- Port Number for players to join

Other players who Join a game will need to input the following information:

- Username
- IP Address of the host game
- Port Number of players to join.

2.2 How the game is played

Each player takes turns placing a tile on the board. When it is their turn (indicated on the top north-west corner),

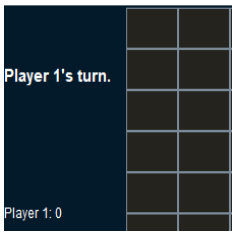

Figure 1: Current Player

they will need to type a letter into the orange box on the east side of the board.

Figure 2: Entering a letter

The player will need to click an empty tile on the board. The tile will become green if a letter can be placed in that spot. Then the player will press "Next Turn" to place the tile and

Figure 3: Next Player Button

change turns to the next player.

If a player wants to place a tile that would create a new word, they will need to apply the same procedure as above but instead of clicking "Next Player", they will need to click "Claim to have a word" on the east north corner. All other players will be greeted with a a voting option

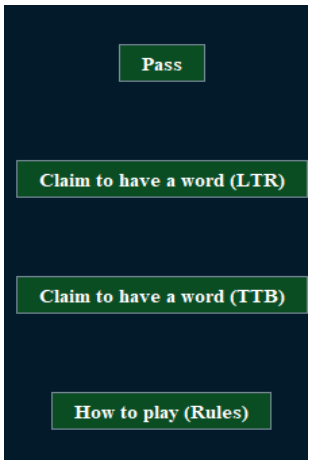

Figure 4: Buttons for each player

("Yes" or "No") if they believe the current player has made a valid word. If at least one player votes "No" then no score is awarded. The scoring is dependant on the length of the word that the player claims. For example the word "scrabble" is worth 8 points.

3 Group Member Contributions

All members contributed in all areas of the whole game design. Each member was given an area of focus for the game and was required to meet deadlines each week for the group. This was a successful effort. Each members contributions are broken into sections below.
For member Dean Pakravan:

- Design of Scrabble GUI.
- Construction of core game design and handling exceptions in the event of player miss-input
- Report write-up

For member Qijie Li:

- Combining RMI, Scrabble GUI and handling of the distributed part of voting system.
- Consulting and providing feedback for the work of all team members
- Requiring understanding all codes and integrating them all together.

For member Luis Zapata:

- Design & Construction of RMI system for Scrabble gameplay
- Implementation of personal RMI template with Dean's scrabble GUI and game design successfully.
- Building from single player scrabble GUI game into a distributed system.

For member Dongming Li:

- Design of Login system interface
- Construction of lobby system with the addition of inviting players.
- Integration of RMI knowledge with Luis Zapata into the Login system interface.

4 Class Interaction Diagram

We will show and explain how the classes/packages interact with each other in our following project. Figure 5 illustrates a sample interaction when a client places a letter on the board; it attaches its

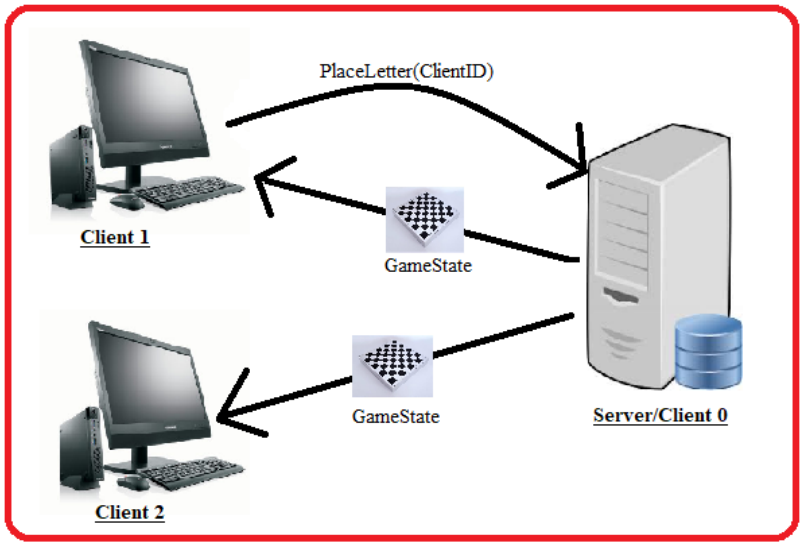

Figure 5: Sample interaction between server and client

personal, unique clientID, which only the server and client know. After the server receives the new placement of the letter, it broadcasts the new GameState to all clients.

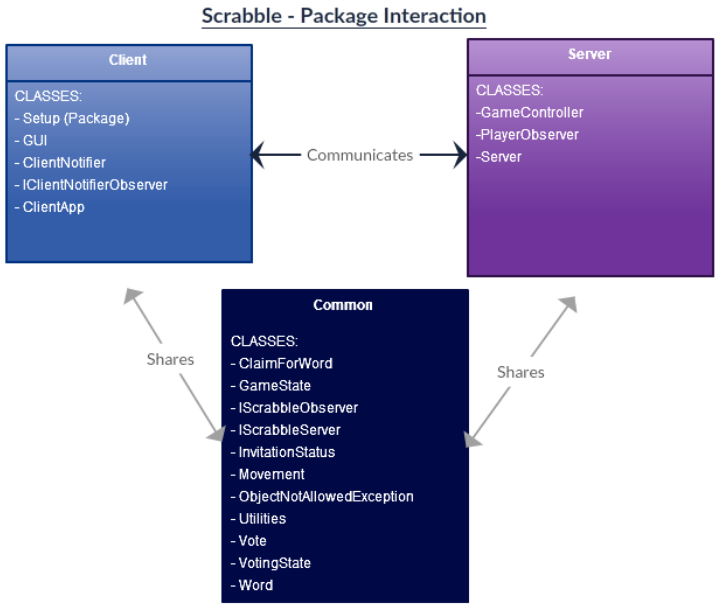

Figure 6: Interaction of Packages

A key point to note is that even the server is considered a client in our design. This is because the server (i.e. the player who hosts the game) will also be playing the game. Hence they are considered a client. We have three **main** packages: client, server and common. The client and server package contain classes that aid with being a client and server respectively, while the common package holds classes that are used by both the server and the client. Communication is still between the server and client as expected.

Displayed in figure 7 is the *key* interaction of the communication between classes displayed in figure 6 that are essential for the client/server communication. The game controller has many IScabbleObserver instances, hence the "*" near the arrow. Each client will have a ClientNotifier which will implement IScrabbleObserver, which deal with making sure the player is active (i.e. connected to the game) and their GameState updates every turn. The ClientApp and GameController are connected to IScrabbleServer (interface) which contain all the methods the client can call to the server. This is further explained in detail in section 5 & 6.

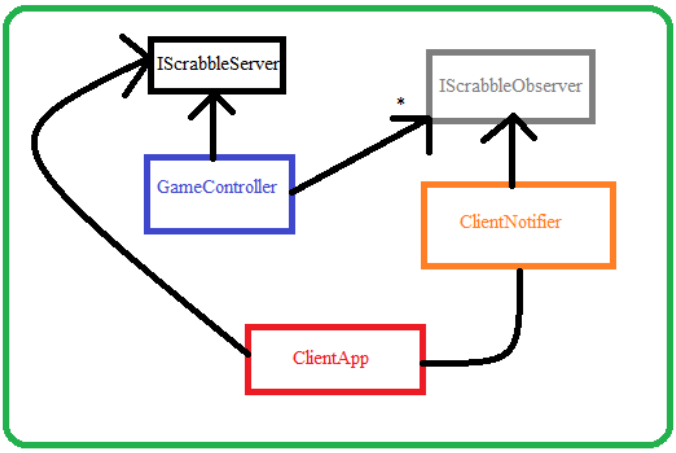

Figure 7: Interaction of Classes

5 Class explanation and interaction

We have several classes in the three packages.
The following is a list of each class from the client package along with a short description.

- GUI:
  This is the board game user interface for the players of scrabble.
- ClientApp:
  This class implements IClientNotifierObserver (found in the same package). This class is essentially the driver function for the client. It is linked to the setup package which has all the initial dialog for the client to join a game (Or the server to host a game), and also is used in server start-up (i.e. entering port number) or client joining (i.e. IP address and port number). This class is key for communicating with the server package and the classes within.
- ClientNotifier:
  Class extends UnicastRemoteObject. The extension is critical in the functionality of RMI. This is the class in the client side that will communicate with the server. A class on the server side will need to extend UnicastRemoteObject as well in order to communicate with the client. Implements the IScrabbleObserver which is an interface in the common package. The main objective of this class is to make sure all players are still active every turn; returning a true statement if they are, as well as reciving the update from the server. This is in the event a client has abruptly ended the game via internet connection error or computer shutdown.
- IClientNotifierObserver:
  This is an interface that holds a method to update the GameState. This is crucial for the ClientApp as it informs the client each time there is an updat to the GameState. Even though the ClientNotifier recieves the update itself, it needs to communicate to the ClientApp that it has the update, which is done through this interface.
- Setup (Package):
  This inner package occurs for the client when they start up the game. This occurs before the board GUI is displayed (Before the game starts).
  - JoinGameDialog:
    Dialog box to illustrate when a client is joining a game
  - NewGameDialog:
    Dialog box when a server wants to create a new game. They will need to enter a *valid* port number in order to start a game. (And username?)
  - StartPageDialog:
    All users are greeted with this dialog page. They are given the option to host a game (server) or join a game (client).
  - WaitForPlayersDialog:
    Once players have joined a game or the server has created a game and entered a port number, they are now waiting in the lobby. This class handles the dialog box for this situation. Players can invite each other into the game. There can be many games on

a single IP address but only one game on the same IP address & port number. (last sentence needed?)

The following is the list of classes and an explanation for the server package:

- GameController:
  This class extended UnicastRemoteObject. This is the class that will communicate with the client; in our case it is the class "ClientNotifier". This class handles some exceptions such as duplicate usernames (in order to handle uniqueness), and not allowing invitations if the game has already begun. The communication with the clients is done in several aspects:
  1. Starting the game
  2. Ending the game (Both by all players passing or a player exiting the game).
  3. Placing a letter on the board
  4. When a player claims a word (Left to Right or Top to Bottom).
  5. Deciding who is the next player (It goes in a rotation according to game design).
- PlayerObserver:
  The server holds all the necessary information about each player such as the username, scores, their votes and so forth.
- Server:
  This is the "Driver" class for the server.

Lastly for the common package, here is a list and explanation of the classes:

- GameState:
  This class holds the current state of the game for all players. Every turn that is made, this state is updated, even in the event of a pass. The basis of this class is mutator methods (informally "getters" and "setters"). For example, this class has a setter for the board of the game; a 20×20 char array, and also has a getter for it. The setter is used when a player places a tile on the board then immediately after all clients call to get the new, updated board. The server (which class?) handles the distribution of the board.
- ClaimForWord:
  Simply contains a public enumerator for when a player claims a word.
- IScrabbleObserver:
  An *Interface* that is implemented in the class ClientNotifer. It holds the methods Update, to update the game state and IsActive, to check if all players have not left the game after each turn is played. This interface extends the interface "Remote" from java RMI. Since the game state is a remote object held on all clients. (needs rephrasing? addition?)
- IScrabbleServer:
  Very similar to IScrabbleObserver, this is an interface which also extends the interface remote. It contains all the following methods that the client can call to the server. The method names are generally self explanatory.

  - JoinGame
  - InvitePlayers
  - AcceptInvitation
  - LeaveGameRoom
  - StartGame (Even though the server is the only player who can start the game, they are still considered a client).
  - PlaceLetter
  - VoteForWord
  - PassTurn
  - EndGame

- InvitationStatus:
  Just a public enumerator that handles the invitation process. The following instances are handled:
  - Joined a room
  - Invited
  - Invitation Accepted
  - Invitation Rejected

- Movement:
  A serializable class that tracks the movement of the letters that are placed on the board. This is essentially for when a player claims a word on the board, making sure any claimed words have the tiles adjacent to each other, vertical or horizontal.

- OperationNotAllowedException:
  A class that contains a super constructor for when a player makes an operation that should not. For example when a player attempts to call the server when it is not their turn.

- Player:
  Data transfer object of the PlayerObserver class. This is simply so we can transfer the object from the server easily and each client can view the player information (e.g. scores, current player and so forth..)

- Utilities:
  Just a straightforward class holding the server name. This could be implemented into another class, but was initially held in the event of scalability.

- Vote:
  Class to track what players vote whenever a player claims a word. Contains getters and setters for the playernames and boolean value of their vote.

- VotingState:
  Separate class to handle after all players have submitted their vote. Also calculates the score in the event the majority votes are "Yes". This class creates an instance of the class "Word".

- Word:
  This class handles the location of the word.

6 Communication Protocols & Message Formats

Our communication protocol for this project was Java RMI. We will now explain how communication is achieved between the client and server throughout the game playing process and what message formats are being used.

Communication protocol is done in binary format. Whenever the server wants to send some information, for example the GameState (which is a serializable class), it will transfer the data into binary. Then all the clients receive this and translate it back into its near original format. This is handled by the Java RMI easily. Similarly when the client calls any of the methods (mentioned below), it will also translate into binary and transmit to the server.

Some of the classes in the Class interaction section of this report have already mentioned how they communicate with each other. The classes that **extend** the java RMI class "remote" are strictly designed to communicate between the server and client. The message formats are clearly displayed in IScrabbleObserver interface and the IScrabbleServer interface.
For IScrabbleServer, the message format is:

- JoinGame(String playerName, IScrabbleObserver newClient)
- InvitePlayers(String clientId, String[] playerNames)
- AcceptInvitation(String clientId, boolean accepted)
- LeaveGameRoom(String clientId)
- StartGame(String clientId)
- PlaceLetter(String clientId, Movement movement, ClaimForWord claimForWord)
- VoteForWord(String clientId, boolean rightWord)
- PassTurn(String clientId)
- EndGame(String clientId)

And the message formats for IScrabbleObserver is:

- Update(GameState state)
- IsActive() (Returns Boolean)

7 Advantages and Disadvantages of Design choice

Using java swing for the GUI was very simple and straightforward to implement and code. However, in terms of appearance and functionality, it is very bland and unappealing. This was a minor

disadvantage in terms of appearance. JavaFX would have been a better design choice, however due to time restrictions it was not feasible to learn how to write JavaFX when we were already familiar with swing.

An advantage that came about of using RMI was the messaging parsing. Unlike project 1 with sockets where it was a huge effort to send more than a coupe strings to the clients, RMI made it easy to send primitive types and any class that is serializable. As stated in the Communication Protocols section, the GameState class (which was made serializable) is sent to all the clients which contains the board, player information, voting count and so forth.

Another advantage of the game design choice, which was mentioned in the system architecture, was the idea of only constructing one jar file so the server was also a client. This makes for very easy setup if someone new wants to play the game, you can just send the single jar file and they can host/join a game. This is also a disadvantage in the event that the client who hosts the game; if they abruptly crash (i.e. computer crashes or internet crashes) then connection is lost.