ASSIGNMENT 2 – TEAM PROJECT: DISTRIBUTED SCRABBLE GAME

GROUP: FEET OFF THE GROUND & LET DALAO TAKE US FLY

YUNLU WEN

ZIKAI XIE

YIMING ZHANG

FEI ZHOU

OCT 2018

CONTENTS

INTRODUCTION

The objective of this project is to develop a multi-player scrabble game. The final product should contain following features:

- The players are playing on a 20 by 20 grid
- Minimum 2 players
- Players place letters on grid in turns
- The user playing the turn highlight the word and let others vote
- The view is synchronized between all the players
- GUI for client
- Logout
- User should be able to see other players
- The system can host one or more games
- Invite user to join the game
- Advanced features are welcome

DESIGN

ARCHITECTURE

This application uses basic client-server architecture. The client is responsible for presenting the graphical user interface and handle user inputs. The server needs to maintain all the states for clients and games and keep all the clients synchronized.

There are two threads on the client side, one for communication and the other for rendering. The server uses one thread per connection therefore higher concurrency can be handled.

PROTOCOLS

The communication between client and server is implemented using socket, which is built on the top of TCP protocol, so that reliable data transmission can be guaranteed.

The data sent via the sockets are simply json strings. The client generates a json object based on the user operation and converts it to string. When the server receives the request, it simply parse it and extract the method field, and invoke corresponding methods with parameters sent by the client.

The json format of request is given by

```
{
    "method":"methodName",
    "param1":"param1 value",
    "param2":"param2 value",
    ...
}
```

The server sends responses to clients using the exact same mechanism. The base response is shown as follows. The client is responsible for

```
{
    "status":"success",//or "error"
    "type":"message type",
    "other_fields":"..."
}
```

CLIENT DESIGN

DESIGN PATTERNS

MVC

The client uses standard MVC pattern. The user and system states are fetched from the server and stored in memory, which is the Model part. Each single stage of Login, Lobby, Game Interface and Pop-up windows are separate views with corresponding controllers.

LOGIN VIEW

The login view is designed such that the user can specify IP address, port and username. Because there is no database setup on the server side, features like setting password and register permanent users are not supported at this stage. When user click on login button, socket connection is established, then the current user can possess a unique user id for that.

LOBBY VIEW

In the lobby view, all the online users and existing games are displayed in the form of list. At current stage, only one game is allowed for the client side, but the server is capable of hosting multiple games concurrently.

The user can create game, invite other players and start the game here. Once a user goes online, other users can see it in real time. When game invitation received, a pop-up window would be shown. Clicking YES will let the player to join the game.

GAME VIEW

Game view is rendered when one of the players of the game starts the game. It contains a 20 by 20 grid and some player information.

LISTENER

Listener is a separated thread which keeps listening the message from the server side. With listener, the GUI does not need to worry about when to receive the response, it only needs to provide some interfaces to update certain parts for listener to invoke.

SERVER DESIGN

DESIGN PATTERNS

The server utilizes the combination of two simple design patterns, which are the observer pattern and the command pattern.

COMMAND PATTERN

Game manager is introduced in the middle of user and game to avoid messing up relationships between two classes, as shown in Figure 1. This makes the system performing like the command pattern, where the clients send command to the invoker, and the receiver invokes corresponding methods with parameters received. In this case, the client is user thread instance, the invoker is game manager and the receiver is game instance.

OBSERVER PATTERN

After the command is executed from the last phase, the system starts to perform like observer pattern, as shown in Figure 2. The system consists of a subject and several observers assigned to it. Once a new game is started at the server side, a new “subject” is initiated. The players which join the game becomes “observers”. The observers receive notifications once the server has updated its state, for example, when a new character is placed on the grid, or one player loses connection.

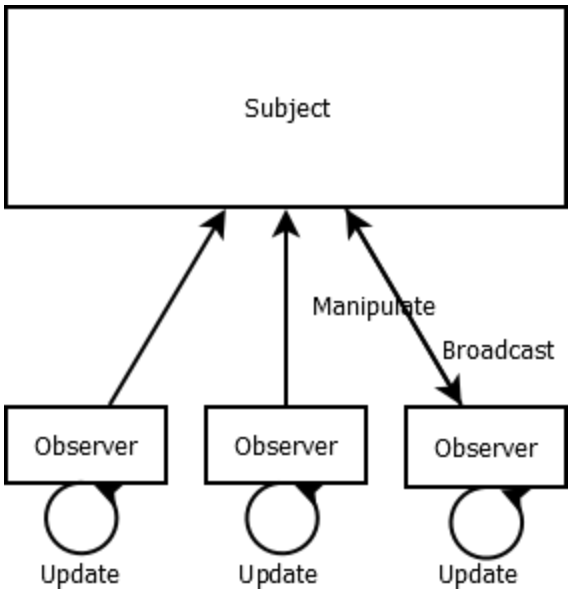

FIGURE 1: OBSERVER PATTERN

To utilize the advantages of observer pattern, the clients implement non-blocking socket communications, that is, they do not have to block to wait for response once they send out messages to the server. Once the messages from server arrive the client side, the clients simply parse the message to figure out which part of the interface should be updated. For example, if one client sees a message saying player 1 has placed a character "c" at the tile with coordinate (1,2), it simply updates that grid.

SINGLETON PATTERN

In the game server, singleton pattern is used to maintain exactly one instance for game manager and user manager. System consistency is considered carefully because corruption is likely to happen in such classes.

CLASS DESIGN

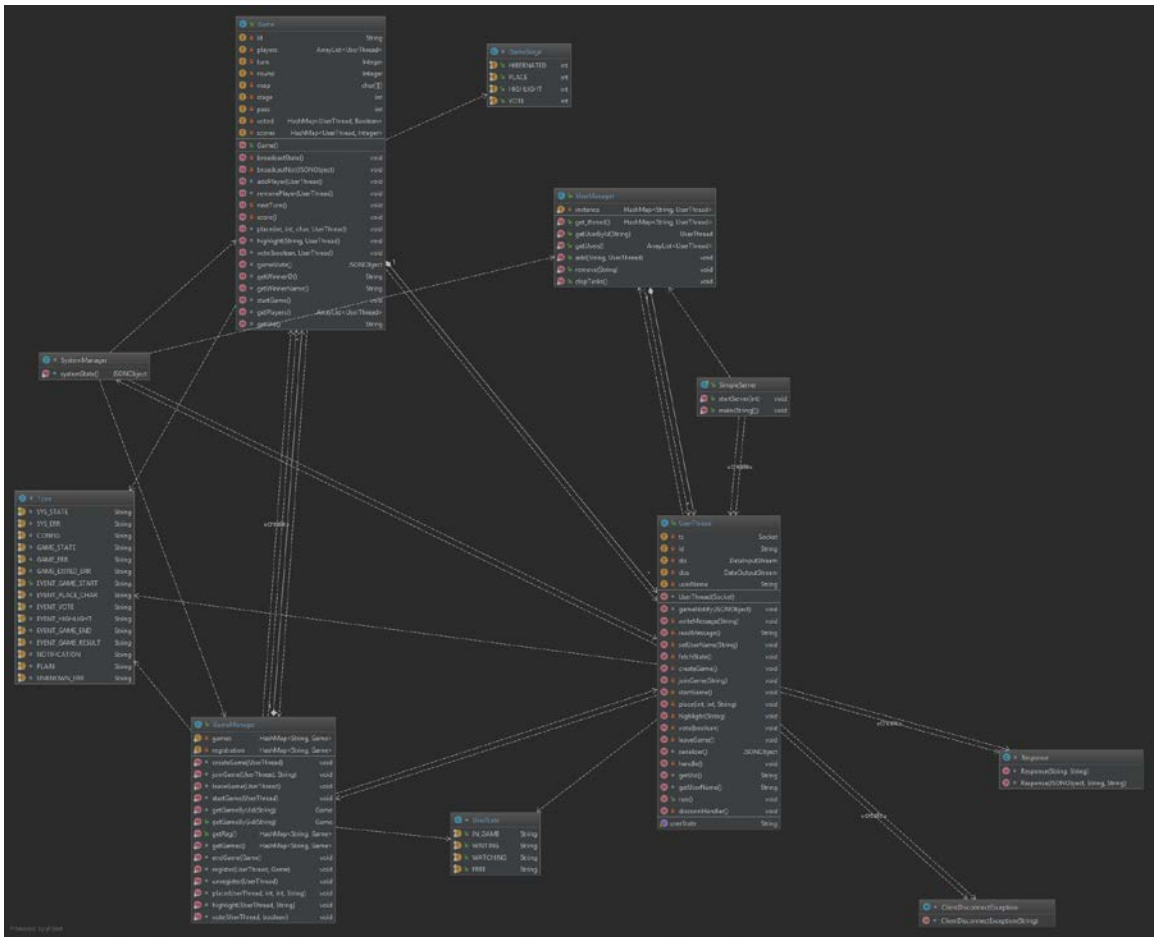

FIGURE 2: UML DIAGRAM OF SERVER

As shown in figure The server consists of following components

- SimpleServer: This is the main class of the server. It starts a daemon process, listens to a configured port and accept incoming connections.
- SystemManager: System Manager is responsible for manipulating system information like online users and games in progress.
- UserManager: The User Manager manages user instances, like adding, removing and fetching user information.
- GameManager: Game Manager manages the lifecycle of game instances. It can create/start/stop games and register users for games they join. It also handles method invoking from users and apply them to Game instances. All the game related operations from users must be processed by Game Manager instead of directly by Game instances.
- UserThread: User Thread represent connections established with clients. Each thread has a socket instance related to it. When a thread is created, a unique id is generated for it to distinguish different users. The user thread is responsible for parsing requests from clients and dispatch the command to the game manager.

- Game: Game simply represents game objects. Each game objects maintains a set game states, including the map, user information, scores, turn, round and game stage. Details of these attributes can be found in Appendix.

The UML diagram of server can be found in Figure 1.

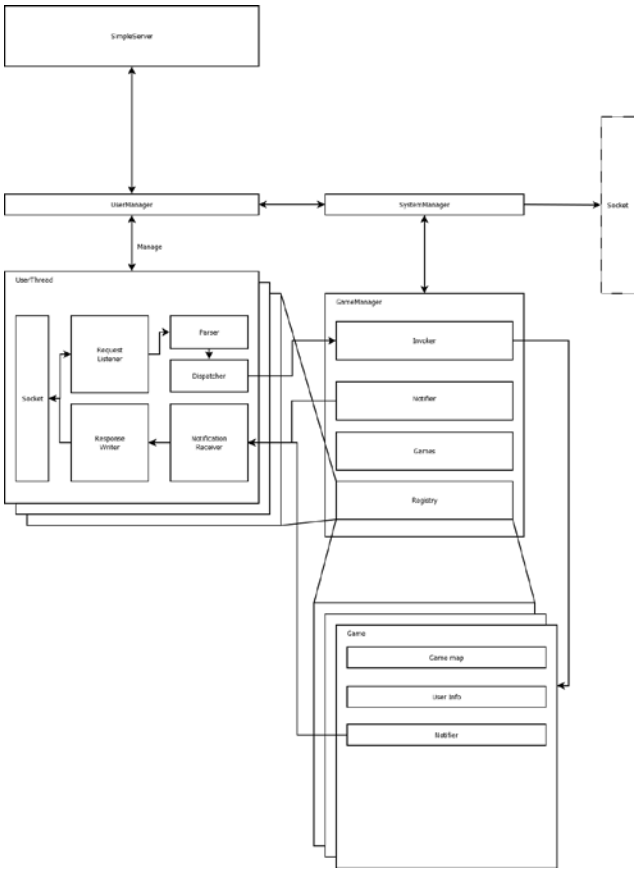

FIGURE 3: THE SYSTEM ARCHITECTURE OF SERVER. THE USER THREAD PARSES AND DISPATCHES THE INCOMING REQUEST. THE GAME MANAGER GETS THE COMMAND AND INVOKE CORRESPONDING METHODS IN THE GAME INSTANCE. THE MESSAGES GENERATED IN THIS PROCEDURE ARE BROADCASTED TO THE USERS.

TEST

To test the game logic on server without fully functional client, socket testing software can be used. In this project, the socket test at https://github.com/wylswz/SocketTest.git was used. It is an open-source project originally created by https://github.com/akshath . One of the group member forked this and modified it to add writing/reading UTF support. The new version of the software is shown in Figure 4.

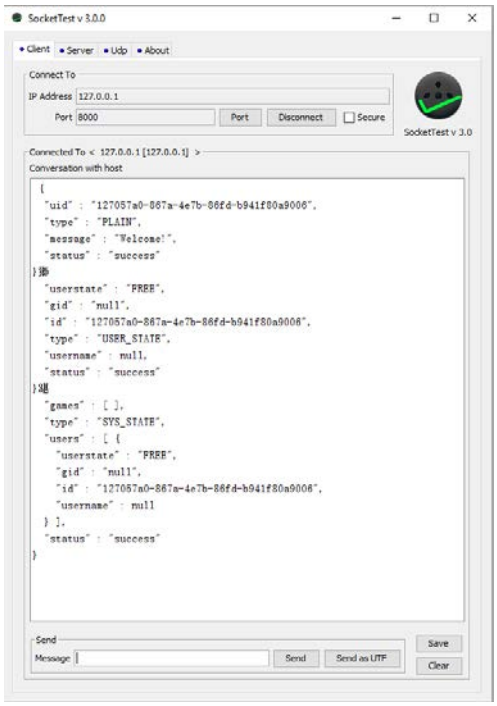

FIGURE 4: SOCKET TESTER

RESULTS

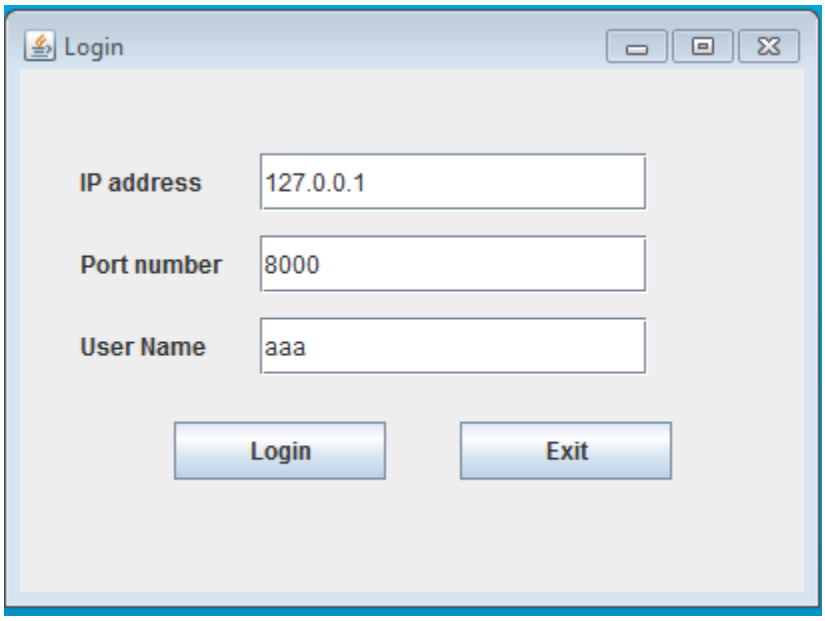

FIGURE 5: LOGIN UI

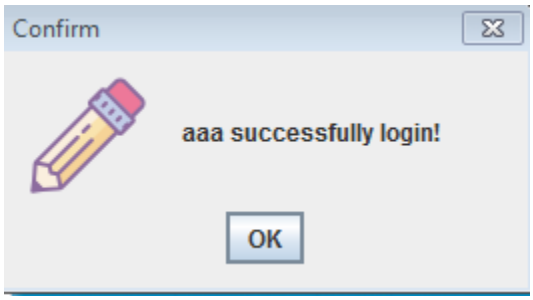

FIGURE 6: LOGIN SUCCESS UI

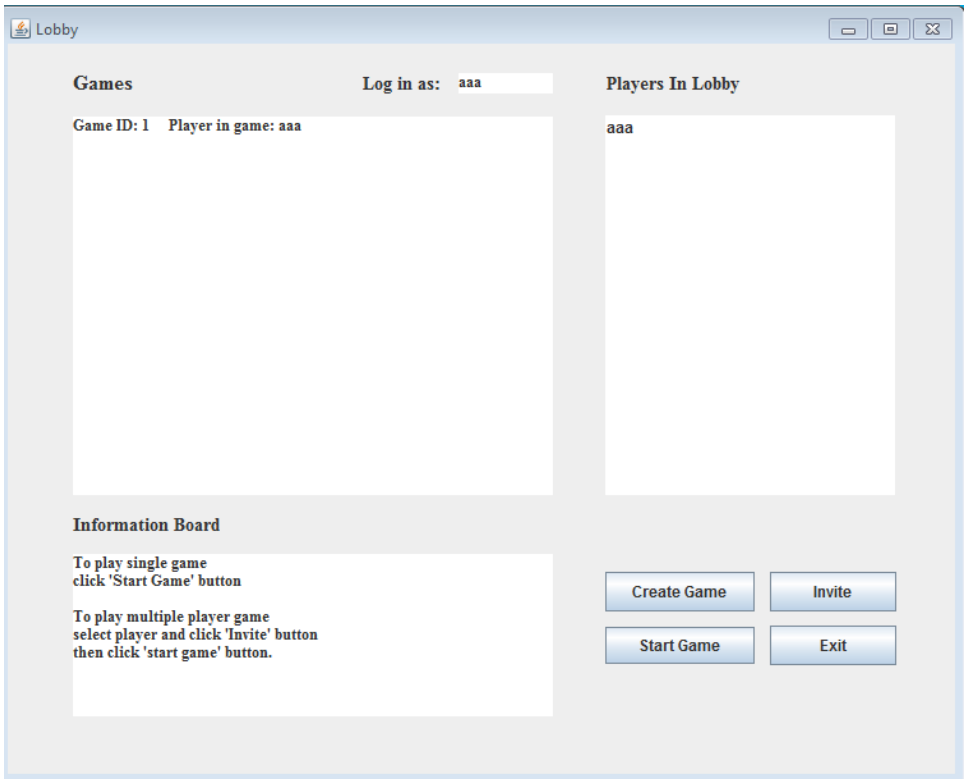

FIGURE 7: LOBBY UI: A USER CREATE A GAME SUCCESSFULLY

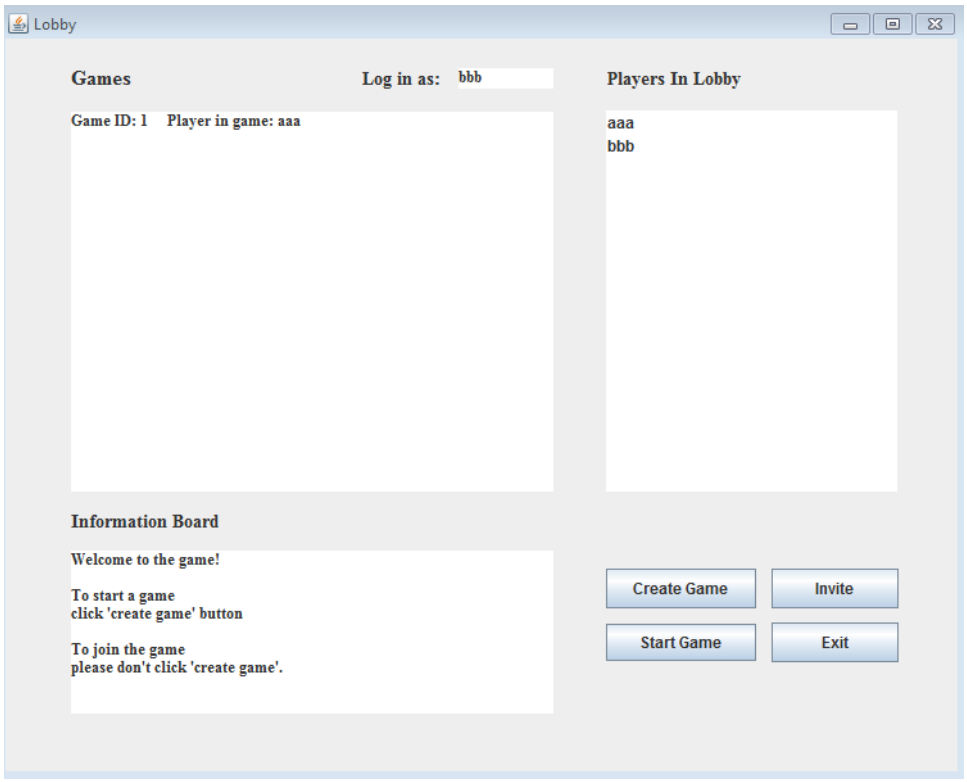

FIGURE 8: LOBBY UI: NEW USER IN THE LOBBY

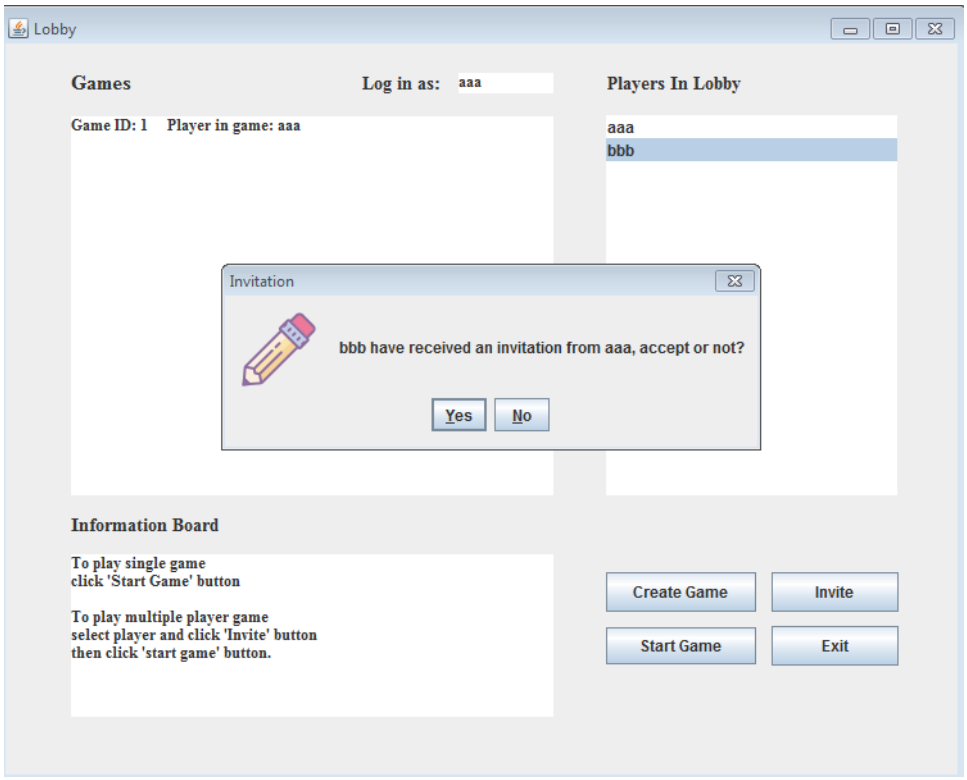

FIGURE 9: LOBBY UI: RECEIVE AN INVITATION

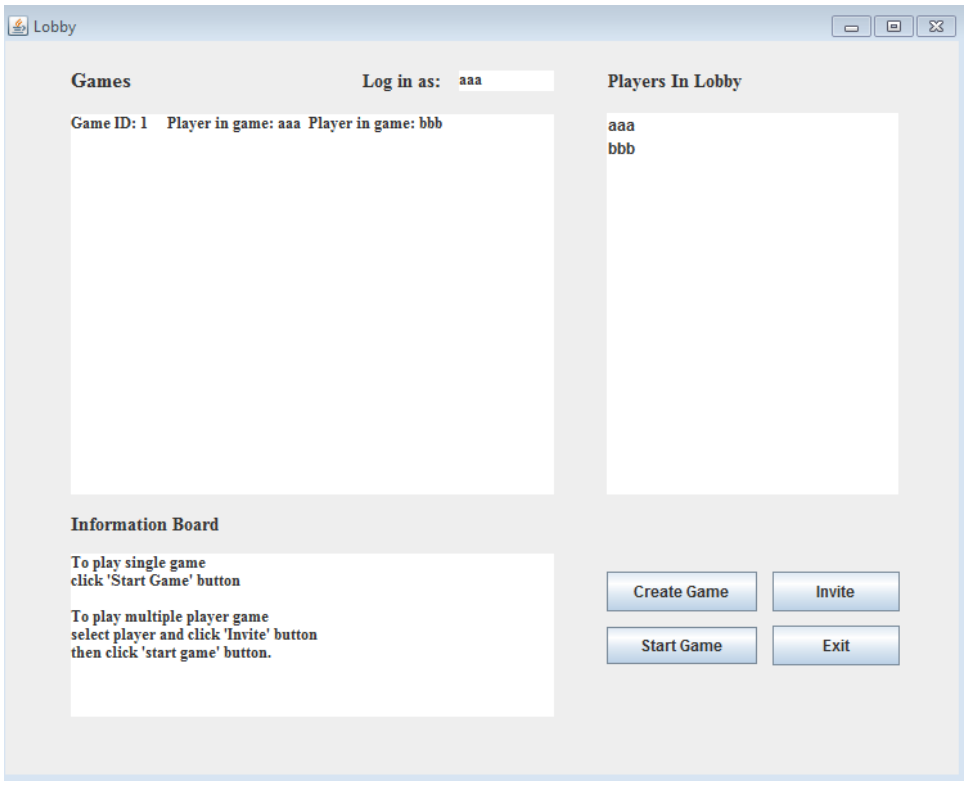

FIGURE 10: LOBBY UI: ACCEPT THE INVITATION AND JOIN THE GAME

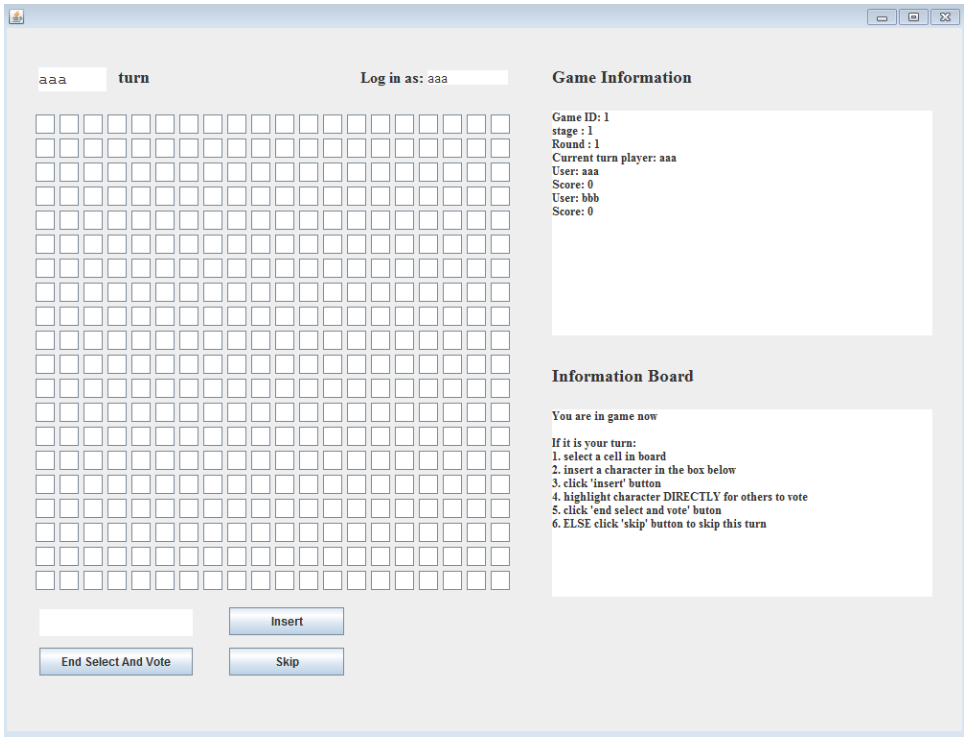

FIGURE 11: GAME UI: USER A

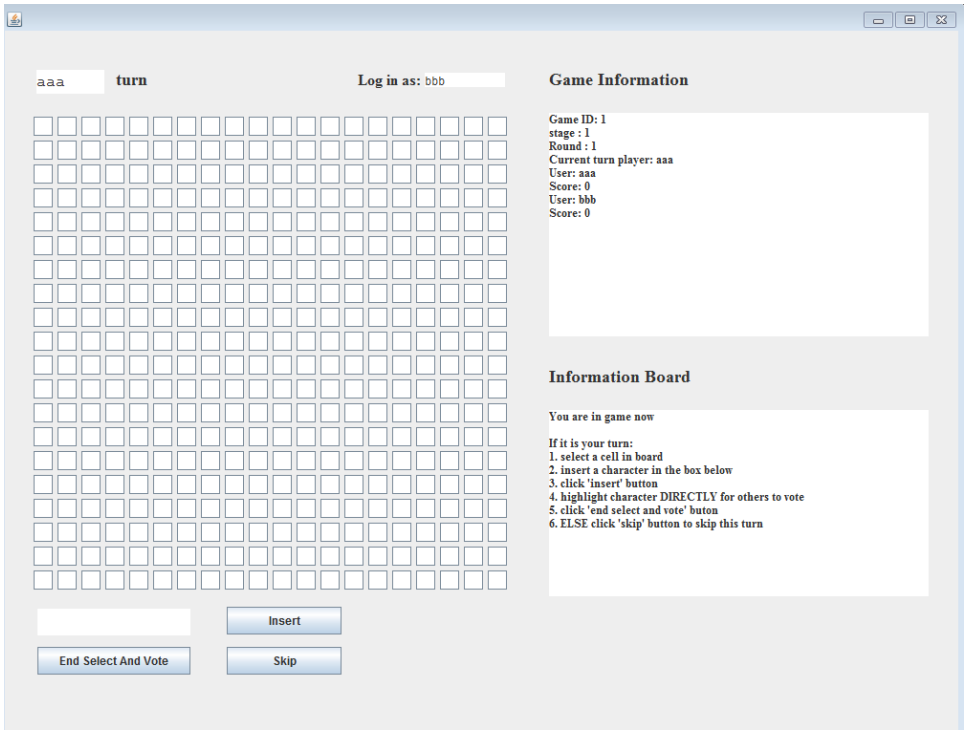

FIGURE 11: GAME UI: USER B

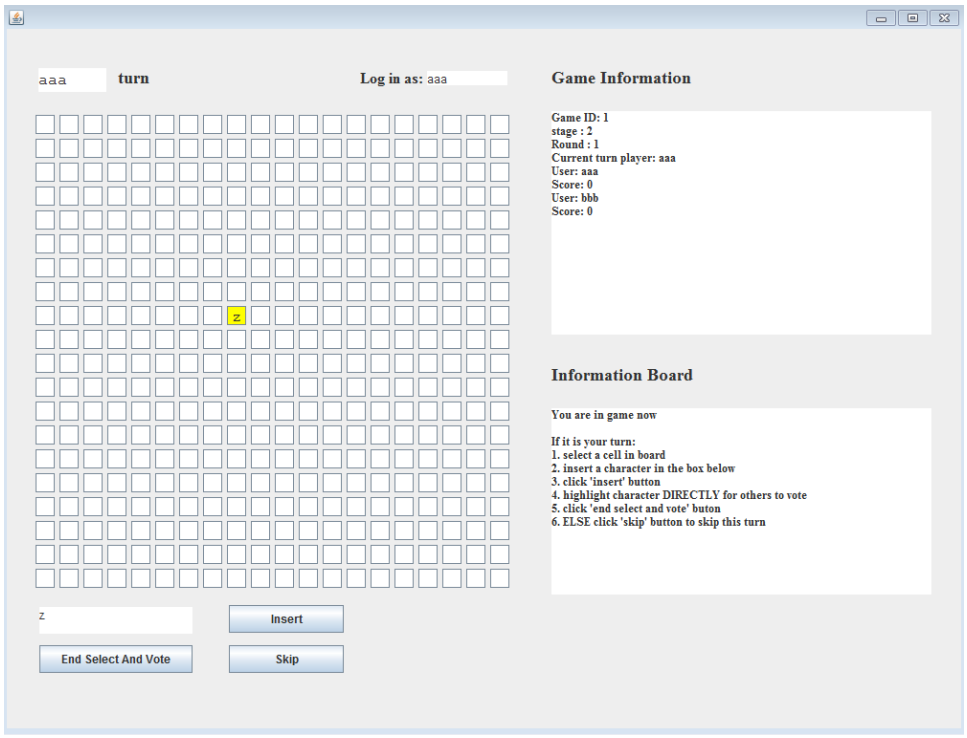

FIGURE 12: GAME UI: USER A INSERT A CHARACTER

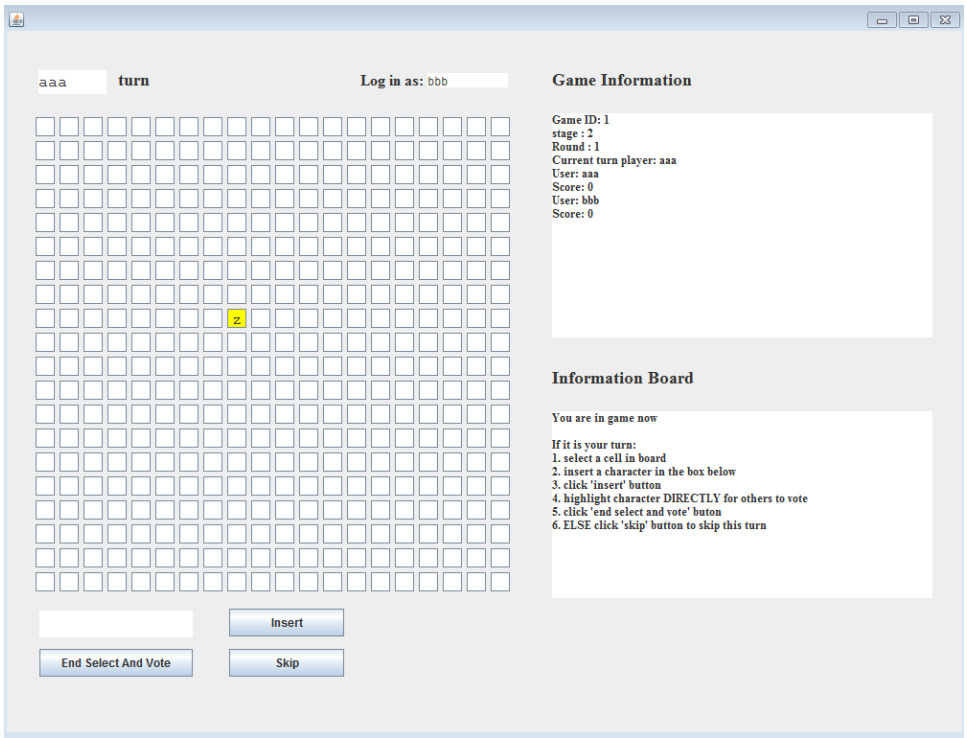

FIGURE 13: GAME UI: USER B CAN SEE USER A INSERT A CHARACTER

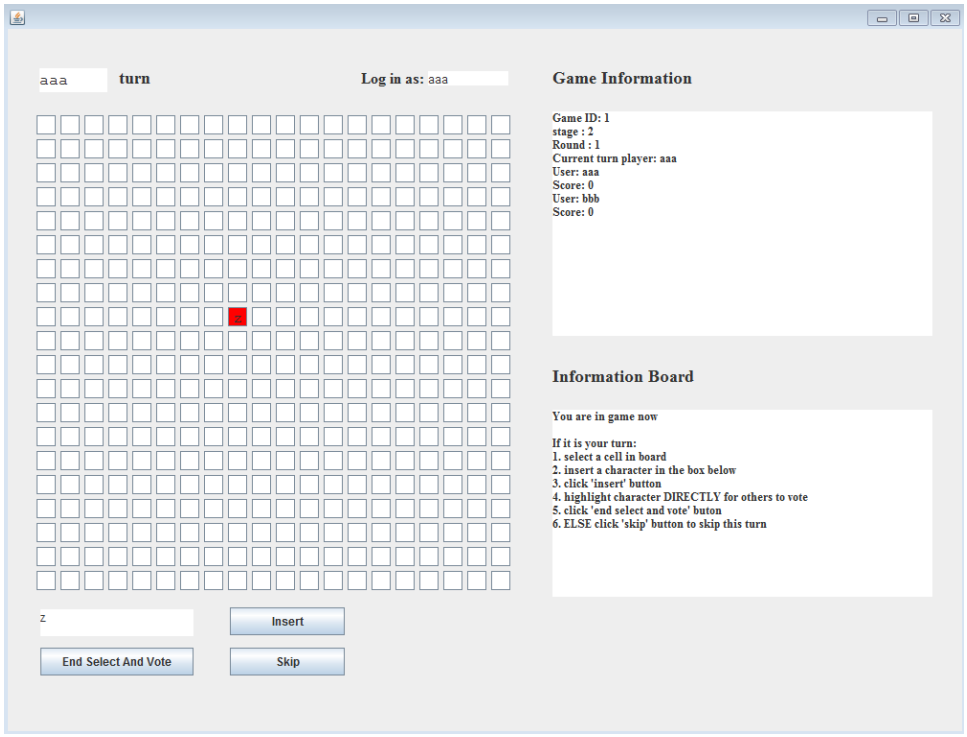

FIGURE 14: GAME UI: USER A HIGHLIGHT A WORD

FIGURE 15: GAME UI: USER B RECEIVES A VOTE REQUEST

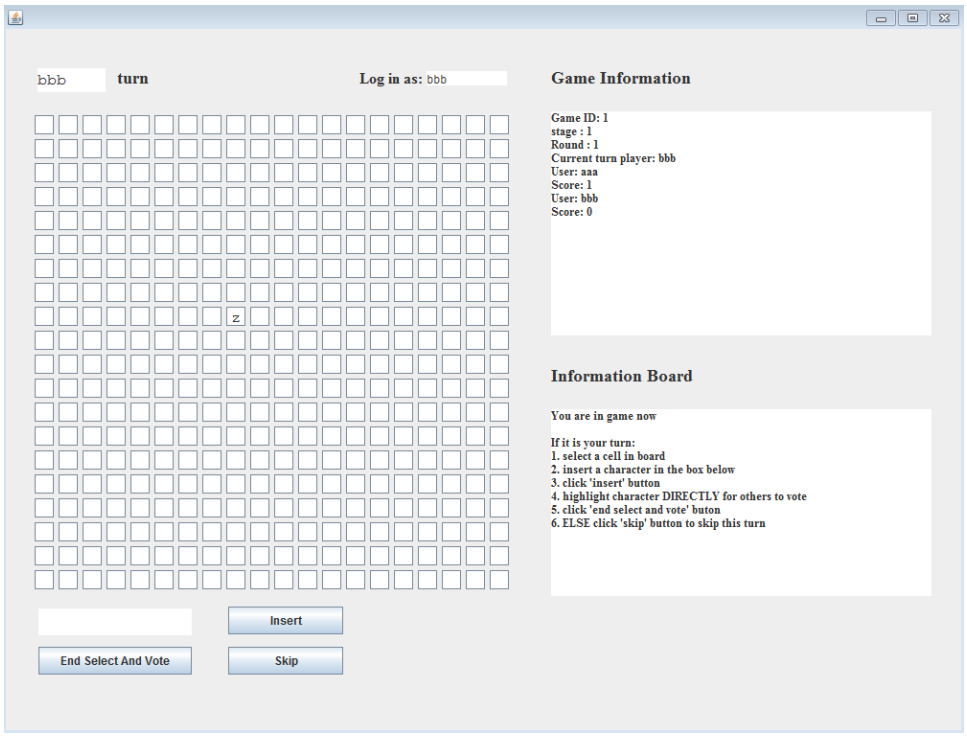

FIGURE 16: GAME UI: USER B'S TURN, USER A SCORE HAS BEEN UPDATED

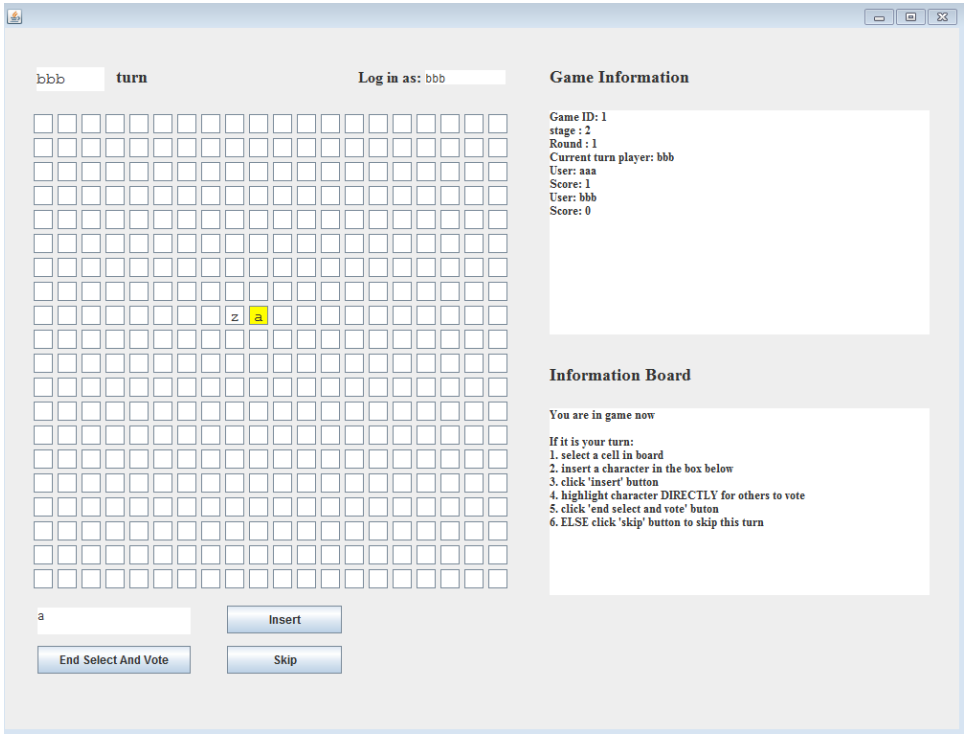

FIGURE 17: GAME UI: USER B INSERT A CHARACTER IN TURN 2

CONCLUSION

In this project, a distributed scrabble game was developed using client-server architecture with all the basic features successfully implemented. Due to limited time, pursue of quality of code, careful design of system architecture and the complexity of communication methodology, the process of client development is slightly slower than the server, therefore some advanced features like multiple game feature, default timeout operation and synced highlight word may not be feasible before the demo day.

FURTHER IMPROVEMENTS

- Start the game only if all the players click start.
- Detailed game information view (Players can see all the information of a game instance, including turn, stage, players and their scores)
- Users watching the game

CONTRIBUTIONS:

Member	Contributions
Yunlu Wen	Server design / regulation
Zikai Xie	Word grid module design
Yiming Zhang	Main logic design at client side
Fei Zhou	Client-server communication handling, message parsing

APPENDIX

A1: SERVER API REFERENCES

- Connection Establishment

```
{
  "message":"welcome!",
  "uid":"someuserid"
}
```

- Setting username

```
// request
{
  "method":"setUserName",
  "userName":"asd"
}
// response
{
  "userName":"yourusername"
}
```

- Create a game

```
// request
{
    "method":"createGame",
}
// Response
{
  "gid":"game id, default 1 for demo",
  "stage":"current stage of this turn",
  "turnPlayerName":"Who's turn",
  "turnPlayerId":"ID",
  "turn":"Server side turn, which is array index",
  "map":"20 by 20 array",
  "players":[{
    "id":"id",
    "username":"username"
  }]
}
```

- Join a game

```
// request
{
  "method":"joinGame",
  "gid":"1"
}
{
  "gid":"game id, default 1 for demo",
  "stage":"current stage of this turn",
  "turnPlayerName":"Who's turn",
  "turnPlayerId":"ID",
  "turn":"Server side turn, which is array index",
  "map":"20 by 20 array",
  "players":[{
    "id":"id",
    "username":"username"
  }]
}
{
  "message":"Player xxx has joined the game",
}
```

- Start a game

```
  // request
{
  "method":"startGame",
}
// response
{
  //here goes the game state
}
```

- Placing character

```
// request
{
  "method":"place",
  "x":"1",
  "y":"2",
  "c":"c"
}
//response
{
  "x":"x index",
  "y":"y index",
  "c":"the char",
  "type":"EVENT_PLACE_CHAR"
}
```

- Highlight

```
{
  "method":"highlight",
  "highlight":"hltest"
}
// response
{
  "highlights":"hightlights",
  "type":"EVENT_HIGHLIGHT",

  // the client is responsible for parsing the string and render
the map
}
```

- Vote

```
// request
{
  "method":"vote",
  "agree":true
}
// response
{
  "user1":true,
  "user2":false,...,
  "type":"EVENT_VOTE",

}
```

A2: SYSTEM TYPE REFERENCES

```
class Type{
    public static final String SYS_STATE = "SYS_STATE"; // System
state
    public static final String SYS_ERR = "SYS_ERR"; // System
Error
    public static final String CONFIG = "CONFIG"; // Change
configuration

    public static final String GAME_STATE = "GAME_STATE"; // Game
state
    public static final String GAME_ERR = "GAME_ERR"; // Game
error
    public static final String GAME_EXITED_ERR =
"GAME_EXITED_ERR";
    public static final String EVENT_GAME_START =
"EVENT_GAME_START"; // Game start event
    public static final String EVENT_PLACE_CHAR =
"EVENT_PLACE_CHAR"; // Any player places a char
    public static final String EVENT_VOTE = "EVENT_VOTE"; // Any
player votes
    public static final String EVENT_HIGHLIGHT =
"EVENT_HIGHLIGHT"; // Any player highlights
    public static final String EVENT_GAME_END = "EVENT_GAME_END";
    static final String EVENT_GAME_RESULT = "EVENT_GAME_RESULT";
// Result of the game
    public static final String NOTIFICATION = "NOTIFICATION";
    public static final String PLAIN = "PLAIN"; // Plain text
    public static final String UNKNOWN_ERR = "UNKNOWN_ERR";
}
```

```
class UserState {
    public static final String IN_GAME = "IN_GAME"; // Playing
the game
    public static final String WAITING = "WAITING"; // Waiting
for the game to start
    public static final String WATCHING = "WATCHING"; // Watching
the game
    public static final String FREE = "FREE"; // Wandering around
}

class GameStage {
    public static final int HIBERNATED = 0;
    public static final int PLACE = 1;
    public static final int HIGHLIGHT = 2;
    public static final int VOTE = 3;

}
```

A3: GAME AND USER STATE REFERENCE

GAME

- map: An array which indicates char in each cell
- turn: Indicates which player's turn
- turnPlayerId: The id of the player playing this turn
- turnPlayerName: The name of the player playing this turn
- gid: id of the game
- players: An array of serialized player object
- stage: The stage of this game, as illustrated in A2.