Project 6: “The Hangman”

Multiplayer Gaming for Mobile Devices

Fabrizio Demaria, <demaria@kth.se> (900402-T417)
Samia Khalid, <samiak@kth.se> (910323-T248)

1. Task specification

The objective of this project was to develop a simple turn-based multiplayer game for Android devices. We updated the solo version of “The Hangman” app created for the homework 5, turning it into a multiplayer game. As an optional task we also tested cross platform capabilities by implementing a desktop version of the game. We also wrote Java documentation for the most important methods of both the server and the client.

2. Hardware and technologies

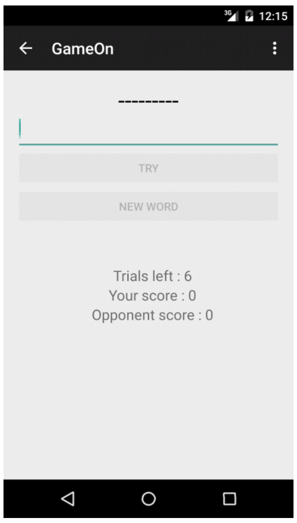
The main working station adopted is a MacBook Pro running OS X Yosemite (version 10.10.2). Both the server and the mobile/desktop versions of the App were developed on the aforementioned machine with Android Studio version 1.0.2 and NetBeans 7.4. The mobile game was tested on both a physical device and the Android emulator program provided by Android Studio. The Android phone used is an Alcatel OneTouch Alpha running Android 4.2.2. The emulator was set to be a Nexus 5 running Android 5.0 (Lollipop).

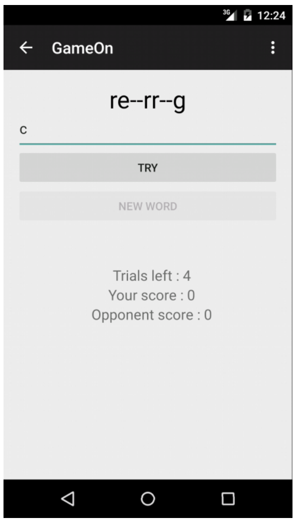
The Android SDK was used for the client side development. On the server side Java Persistence API 2.1 has been adopted to maintain a Java DB database with informations regarding players’ accounts and relative statistics.

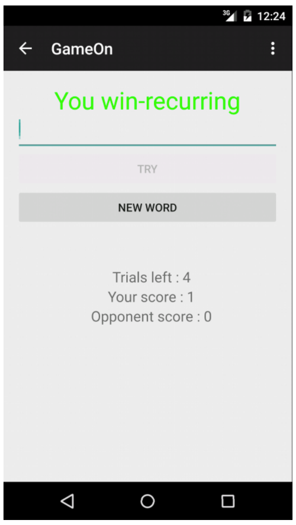
3. The game

3.1 Overview

In “The Hangman” a player is supposed to guess a word chosen randomly from a dictionary by the server. At the beginning of the match, the word is presented to the player as a sequence of dashed lines. During each turn the player can suggest a letter or the whole word. If the letter is present in the word, all of its occurrences in the word are displayed in the corresponding positions. If the letter is not present, the number of available trials decreases (the initial number of trials is 6 for each word). When the counter reaches 0, the game is over and the total score is decremented by 1. In case the player guesses the word, the total score is incremented by 1.





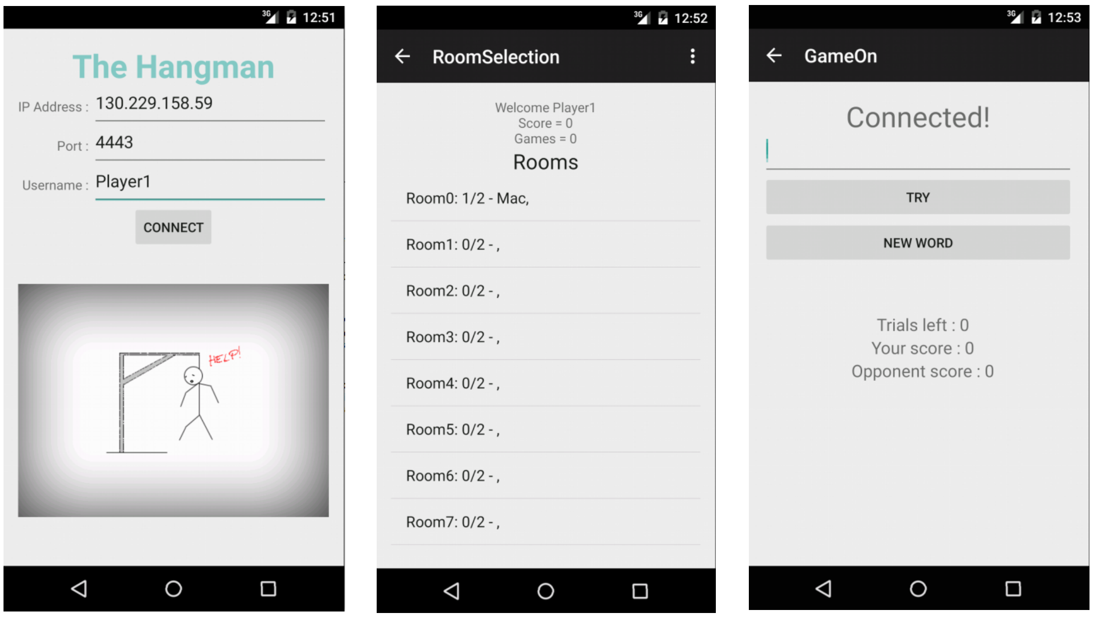


In our multiplayer version of the game, two players try to guess the same word. The two players alternatively try to guess a letter or the complete word. The first one guessing the word wins increasing his/her score. The number of available trials is in common with the two players, and it is possible that none of them guesses the word; in this last case the score is decremented for both.

3.2 How to play

The application is based on the client-server architecture. At first the user has to insert the IP address and port number for the connection setup. Along with these parameters the user also provides the username which is used by the server to lookup and maintain the player's statistics in the database. If the username is not found in the database, a new entry is created.

The username is also used to display who is willing to start a multiplayer game. The server provides a set of 10 "rooms" in which players can register themselves in order to start a match. For our version of the game, each room can host a maximum of two players. After the player logins as previously described, the list of rooms is displayed showing the username of the players already occupying a room. Now it is possible to choose a room: if it is empty, the player waits for an opponent to connect to the same room; if the room was previously occupied by a player, the match starts.



IP Address : 130.229.158.59
Port : 4443
Username : Player1

Welcome Player1
Score = 0
Games = 0
Rooms
Room0: 1/2 - Mac,
Room1: 0/2 - ,
Room2: 0/2 - ,
Room3: 0/2 - ,
Room4: 0/2 - ,
Room5: 0/2 - ,
Room6: 0/2 - ,
Room7: 0/2 - ,

Connected!
Trials left : 0
Your score : 0
Opponent score : 0

When the words is guessed or the number of trials reaches 0, both players have to press "New Word" button to request a new word from the server and continue the match. The scores are synchronised with the server's database.

3.3 Communication protocol

Our single-player version of the game developed for homework 5 was entirely based on TCP. For this reason it was convenient to extend that implementation in order to handle multiplayer matches. The game is rather simple under the point of view of the messages to be exchanged, and we adopted TCP sockets to send strings as commands from client to server and vice versa. We decided not to use any third party Java library and the RMI option was not possible since not supported by Android. For example, in order to start a new match the client has to send the string "*START*" that is interpreted by the server as a command to lookup for a new word in the dictionary and proceed with the game flow.

With respect to the old single-player version, we had to implement the rooms service to handle multiplayer matches. This part had to be developed from scratch and we decided to optimize the communication part regarding the status of the rooms by sending a serialized object instead of a series of strings. A serializable class called *Room* was designed to contain the following informations for each of the 10 rooms:

- Username of the players
- Sockets for the communication with players
- Number of players in a room

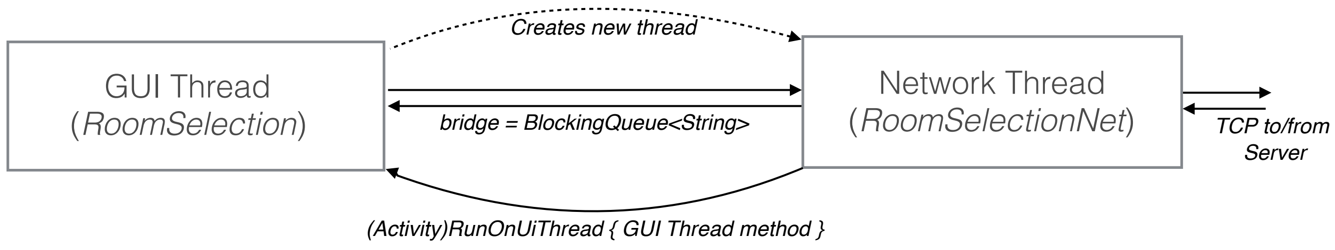
Sockets were declared as *transient* variables since they are not serializable. All the other informations sent to the client are used to construct the table of rooms presented to the player.

We encountered a problem in updating the rooms information when a player disconnects from the game (leaving the room). The method *socket.isClosed()* at server side was not able to detect that the socket has been closed at the client side. In order to correctly update the rooms list every time a player refreshes it, we developed a method inside the *Room* class capable of checking the availability of a socket by performing an I/O operation with a timeout: if reading from a socket returns *null* it means that the socket is closed on the other side, otherwise reading would block waiting for client's input on the active socket. For this reason we set a timeout of 100 ms in case of active socket. With this implementation, we can update the information in the *Room* object before sending it to the client, and the list of rooms with all the connected players is consistent with the actual connected players.

3.4 Structure of the client

The main issues regarding the development of the client were related to the thread management in the Android environment. The main concept we had to deal with is the fact that the GUI related threads cannot handle network operations. This requirement is used in Android to assure the responsiveness of the GUI.

Our architecture works as follow:



A "network thread" is created for each GUI activity. The two threads can exchange strings through the BlockingQueue: for example a user select a room from the list and the corresponding room's index is passed to the network thread through the bridge and this information is provided to the server with the TCP connection. The network thread will receive the answer from the server and it can update the GUI by using *RunOnUiThread* to call methods on the GUI thread.

Android doesn't allow to pass a socket when a new activity is created, so we had a problem in maintaining a single socket when moving through activities (i.e. from the "room selection" activity to the "game" activity). Several options were available to counteract the problem, like adopting the Android Service tool. We opted for a *Singleton* class that was easier to implement. This class can be called from any activity and/or thread and it maintains a unique socket created when the user first carries out the login and connects to the server. When the user presses the "back" navigation item the socket is closed and the initial activity (*MainGame*) is started again.

The mobile app is composed of the following Java classes:

- *MainGame (activity)*: starting point of the application; asks to user for the IP address of the server, port number of the server and the player's username. After connection the activity *RoomSelection* is started;
- *RoomSelection (activity)*: creates the *RoomSelectionNet* thread to initiate the communication with the server. With the informations received (indirectly from *RoomSelectionNet*) it displays the rooms list to the player. It also receives from the server the player's statistics and shows them

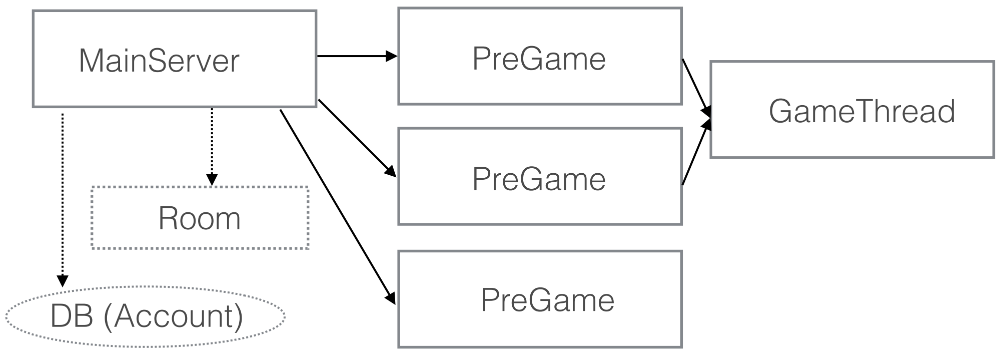
(i.e. the total score of the player and number of played matches). When a room is selected the *GameOn* activity is started;

- *GameOn (activity):* creates the *GameThread* thread for communication with the server. At this point the game is active and the logic of it is contained in the *GameThread. GameOn* methods are called with *RunOnUiThread* in order to update the view according to the players' moves.
- *Singleton:* this class maintains the socket and makes it available from any thread/activity.

3.5 Structure of the server

The server maintains a database with users' informations and it keeps the current state of all the rooms and ongoing games.

The basic scheme of the server's architecture is as follows:

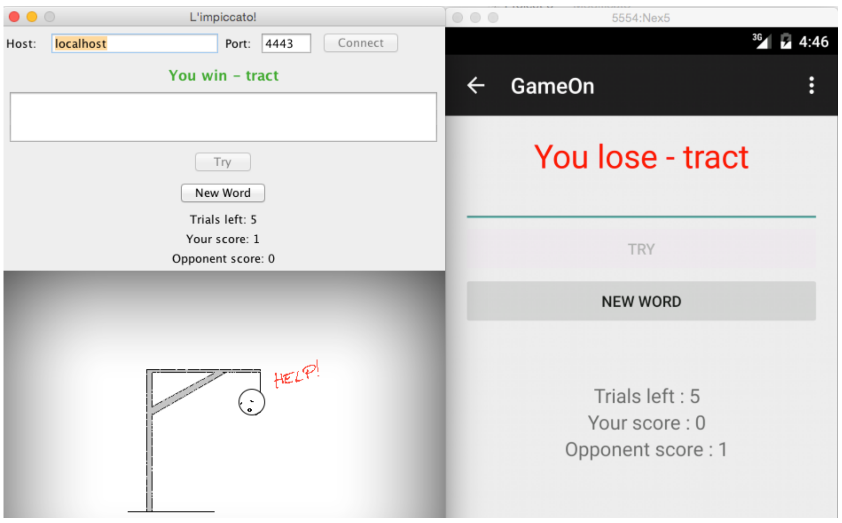


A main thread (*MainServer*) is started to accept new connections from clients. As soon as a connection is established a *PreGame* thread is started to send back informations about the rooms and player's statistics retrieved from the database. Then the thread waits for the room selection from the player. When the selected index is received, the thread checks if the room is empty: in this case the client is indicated to wait for the opponent to connect to the same room; if the room has already a player waiting (this is checked with the *Room* object maintained in the server), a new *GameThread* is created passing the sockets of both players. At this point *GameThread* notifies the player that the game can be started and continues with the game flow.

The database is implemented using Java Persistence API. The database maintains only a single table with: username, total score and total number of matches played. In our implementation no password is required and each account is only identified by the username. If the username is not found at connection time, a new entry is created.

3.6 Crossplatform

As an extra task we modified the desktop game developed in homework 1 to be compatible with the new multiplayer version. The aim of this part of the project was to demonstrate the cross platform compatibility of the game. We had to modify the *GameThread* class to handle the turn-based scenario. Also, small changes in the GUI were required. On the figure on the right on player is playing on the Mac Desktop (left) and one on the Nexus 5 emulator (right).



3.7 Javadoc

We wrote Java documentation for the most important methods of both the server and the client. However, we were only able to export the documentation from NetBeans (server side) while Android Studio's tool for Javadoc generation didn't work properly.

The server's documentation can be retrieved from the following link:

https://drive.google.com/folderview?id=0B03RVIztCHnhb3lmVTNrOVNyOTQ&usp=sharing