tkv-blackjack

Project Documentation

ELEC-C7241 - Tietokoneverkot

Esa Koskinen, 429788
Felipe González Carceller, 528621

2017-04-09

Table of contents:

1. Introduction

This document describes and analyses the implementation details of a small software project made for the course ELEC-C7241 - Tietokoneverkot, held in Aalto University between 2017-01-03 and 2017-04-27. The project consists of an online multiplayer implementation of the traditional card game Blackjack, and includes both the server and client programs.

The document first mentions how both of the programs are compiled and launched. After that the document describes the general architecture of the project, and the overall structure. After that there are sections on the used communications protocol and testing methodology. After them, the document analyses the various architectural decisions made early in the design process, and discusses how the implementation fares against various unexpected situations. Finally, there is a section on working methodology and amounts, as well as an appendix showing the git log.

The software works for the most part; the networking is well developed and both the client and server are very resilient to errors, exceptions and crashes. Running both the server and multiple clients on one machine works fine. However, the game logic is still somewhat unfinished. Most of the handling of player actions is implemented in both server and client code, and the exchanging of messages over the network works fine, but as some of the parts in the game state's background management are missing, the game is not yet in a playable state. We expect the implementation and testing of these to take about 10 to 20 hours of work by one person, but unfortunately were not able to finish them in time for the project deadline.

2. How to use

We have made several easy-to-use scripts that can be used to build and launch the program. To fire it all up, **make sure that you are in the project's root directory** and run the following commands on your console:

```
. build.sh
. runserver.sh
. runclient.sh
```

The dot is shorthand for using `source`; basically the contents of the files have various commands that have to be executed in order to start the programs. The first command cleans, compiles, and builds the program. Then with the `. runserver.sh` command it starts up the server. In case you want to run the server and the client in the same console, you can utilize terminal jobs and use `. runserver.sh &` instead to run the process in the background. Depending on your setup you may also be able to use `nohup` to get rid of console output.

After you have the server running use the `. runclient.sh` command in the console where you want to play the game. The UI will first ask for the host and port, where the host can either be an IPv4 address or a hostname (just press enter to input the defaults). After that, it tries to connect to the server. If it connects successfully it will notify the user of it and then ask for the player's name.

The software code has various constants for easy tweaking of all sorts of things. Most importantly, there are debug options in the tkv_project.client.BlackjackController and tkv_project.server.ServerConstants classes; turning them on or off will greatly affect the amount of debug messages printed on the console. There are other various constants that one can play with; the idea was to place all of them into a file or to be entered as command line parameters, but unfortunately we did not have time for that. Remember to run build.sh after tweaking the source code to compile the changes.

Closing the server closes all clients gracefully as well.

3. Program structure and implementation

The program consists of two main components. The server and the client. On the server side we have the classes ServerLauncher, ConnectionManager, ServerController and ServerConstants. The server is initialized by running the ServerLauncher class which first creates a new ServerConstants. ServerConstants contains all the constants of the server so it is easier to modify them when needed. The ServerLauncher then creates a new ConnectionManager. The ConnectionManager manages all the client connections and whenever a message is received from a client it is acted accordingly by calling methods in the ServerController class which alter the game's state depending on the message received. After that the server sends the new, updated state of the game, back to the client.

On the client side we have the classes ClientLauncher, BlackjackController, NetworkManager, TextUI and an abstract class UserInterface. Like the server, the client is initialized by running the ClientLauncher which first creates a new BlackjackController and TextUI. The BlackjackController handles the actions regarding the game locally and calls the NetworkManager's methods when something needs to be sent to the server. The TextUI prints everything the user sees from asking the server details to updating the view every time a message regarding the new game state is received from the server. It also notifies the user with an adequate error message when something goes wrong or an exception is encountered. After the BlackjackController and the TextUI are created the client tries to connect to the server and if successful, creates a NetworkManager. The NetworkManager is responsible for sending messages to the server and acting accordingly upon receiving them.

Both the client and server have a fairly strong separation of concerns, which unfortunately had to be blurred slightly due to the final adjustments taking more time than expected. The exception handling is fairly robust, and the server can handle multiple incoming connections at once. In practice, the game runs in real-time; the frequency of updates can be changed in the BlackjackController constants. The system is built to handle this even after the final pieces of game logic are made.

4. Communication protocols

The software uses version 1 of the TBP (tkv-blackjack protocol), which we created for the purposes of this project. It is assumed that the reader is familiar with the basic blackjack rules. Version 1 of the protocol does not support doubling or surrendering, whereas the as-yet unspecified Version 2 is planned to include those.

The client can send the following recognized messages (parameters in angle brackets):

```
name:<playername> — this sets the player's name and adds it to the
server's player list. Should only be sent once (but the server
survives receiving many of them from the same client).

hit — Tells the server that the current player would like to hit (=
draw a new card).

stand — Tells the server that the current player would like to stand
(= stop playing cards this round).

update — Tells the server that the current player would like to
receive an update in the gameState. An update can be asked for at
most around MAX_UPDATES_PER_SECOND times per second, as is specified
in the server's constants. The minimum for MAX_UPDATES_PER_SECOND is
specified to be 2; clients may assume that it is at least 2, although
it is usually somewhat higher.

quit — Tells the server that the current client would like to quit.
```

The server will in turn send various status messages that are safe to discard. On an update, it will send a single line-break terminated row containing the two-dimensional gameState array's data, with each player separated by a hash symbol (#), and each player's every parameter separated by an ampersand (&).

5. Implementation analysis

5.1. Transport layer protocol

The protocol used is TCP; this is simply because UDP would not offer significant advantages but on the other hand may increase the amount of problems. We did not consider other transport layer protocols, as we felt that choosing between these two would be sufficient.

5.2. IPv6 compatibility

The software is in theory compatible with IPv6, but this is untested and most likely will fail without minor modifications to the used classes. Who needs IPv6 to play Blackjack when you have a system that can parse both IPv4 and hostnames?

5.3. Client disconnection and exception handling

Everything related to this works fine; the server handles client disconnections appropriately and nearly all unexpected states are handled in a sensible way. Players can re-join after quitting and new players can join as expected. However, it should be noted that the gameState in the server does not yet remove the players who quit, which erroneously keeps them in the client’s screen. Closing of the server is handled gracefully in the clients.

5.4. Handling of multiple simultaneous incoming connections

This is no problem. There is a crude locking mechanism in place for handling multiple incoming players, and the server socket one handles one incoming connection at a time, keeping the others waiting until the previous ones are processed (thanks to java.net magic).

6. Working methods and time usage

6.1. Team practices

We decided to use git for development since it lets us code efficiently on our own. The project started by us meeting and discussing the programming language that we were going to use and the basic layout of the program. We ended up using Java since one of the team members didn't have any previous experience with it and wanted to get to know it. In the first meeting we also distributed the work areas between the team members. During the following weeks both of the team members went on their own and produced some code. Whenever someone encountered a problem in the code we just asked each other to clarify the situation. We would also meet and do pair programming whenever our schedules allowed it.

6.2. Time usage chart

Name:	**Esa**	**Felipe**
Time used (h):	roughly 40-50	around 40

A log of git commits has been included with the project, in the git_log.txt file.

6.3. Testing

We mostly utilized manual user testing, as that is a typical way of holistic testing used in game development. It revealed all problem points fairly quickly, as should be expected from a simple game like this. We had a few minor regressions, but don't think that creating a test suite for covering them would have been worth the time used. Debug logging is used extensively; it has multiple levels which can be controlled with the constants in BlackjackController and

ServerConstants classes. We think that it might be interesting to see at least the server side debug messages, but for the time being we have turned the debug off in both the server and the client.

While manual testing was prevalent, one form of automatic testing was employed. To stress-test the server, we created a script that launches numerous clients at once in separate threads, all attempting to connect to the default server. It can be reproduced with the following steps, but please note that we are not responsible if your computer experiences unexpected problems, hanging, crashing, melting, nuclear fusion or souring of the milk as a result of running the following procedure. GNU Screen required.

1) Launch the server as usual; run
   `. build.sh` and then
   `. runserver.sh`

2) Run the following command:

```
yes "screen java -cp bin/ tkv_project.client.ClientLauncher




&" | bash
```

This caused a computer to hang, so please don’t try it on any computers which may have more than one user. The results were promising though; our server handled all the incoming connections very well until there were too many clients for the computer to handle. Of course, this was only a very rudimentary test and should not be taken too seriously. But seriously, don’t try it in shared environments.