Homework 1

Network Programming, ID1212

Wilhelm Berggren, wbergg@kth.se

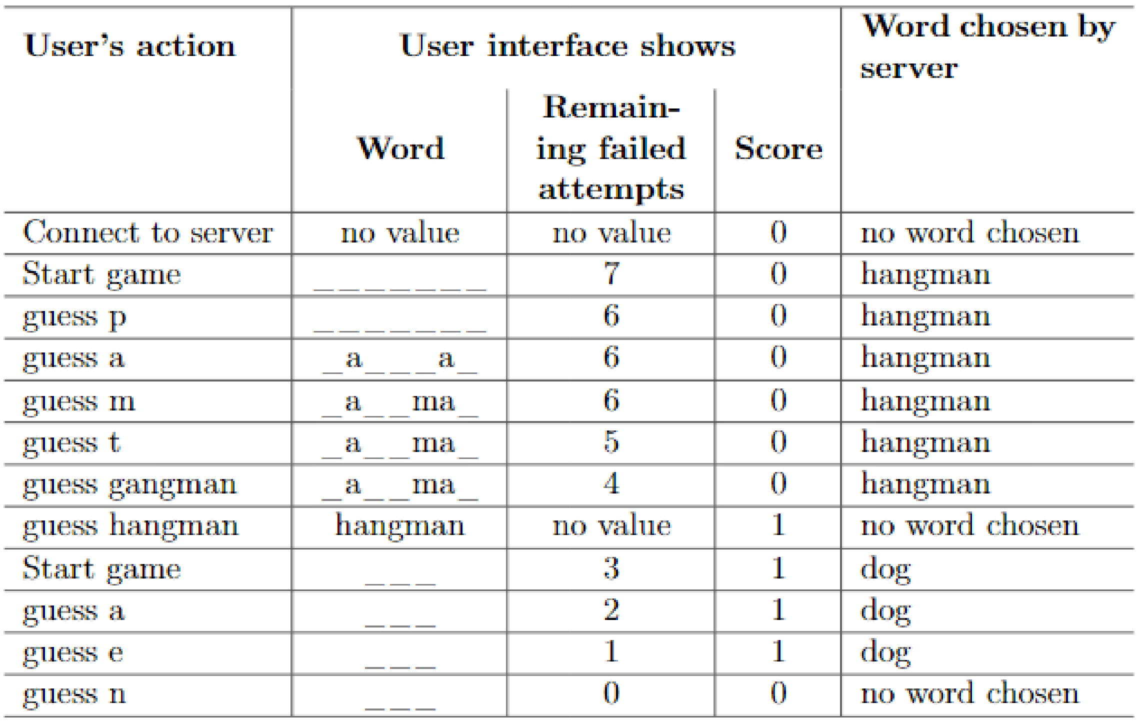
2017-11-17

1 Introduction

This is a report on homework 1 of course ID1212. The task chosen is to create a distributed application in Java for the guessing-game “Hangman”. The requirements are:

1. Client and server communicate over a TCP connection using blocking TCP sockets.
2. Client does not store game data, only receives game data from server.
3. Client has responsive user interface utilizing multithreading, so commands can be issued at any time, even when waiting for server messages.
4. Server must handle multiple clients using multithreading.
5. User interface must be informative, current state visible and next step for user clear.

Sample execution:



User's action	**User interface shows**			**Word chosen by server**
	Word	**Remain-ing failed attempts**	**Score**	
Connect to server	no value	no value	0	no word chosen
Start game	_______	7	0	hangman
guess p	_______	6	0	hangman
guess a	_a___a_	6	0	hangman
guess m	_a__ma_	6	0	hangman
guess t	_a__ma_	5	0	hangman
guess gangman	_a__ma_	4	0	hangman
guess hangman	hangman	no value	1	no word chosen
Start game	___	3	1	dog
guess a	___	2	1	dog
guess e	___	1	1	dog
guess n	___	0	0	no word chosen

2 Literature Study

The knowledge needed to fulfill these requirements was gathered through the online lectures on the course web and code samples on the course GitHub page. Additional knowledge was acquired through years of prior Java experience, the Java documentation and Stack Overflow.

Noteworthy concepts I have learned about:

- Threads and concurrency, Runnable
- TCP methods
- Buffers
- Asynchronous tasks
- Component based flow of information

3 Method

My workflow for accomplishing the task was to introduce requirements by iteration and create a minimal working model for each iteration. Boiling down the example code to its essentials allowed me to get a solid overview of the base functionality of a TCP server-client relationship. Fleshing out the package, refactoring and going from for example a simple TCP connection to sending messages and creating threads for new connections. Finally when the requirements for the underlying connection logic was in place, I worked on creating the game logic, that too starting with a bare essential model and introducing more requirements followed by refactoring.

4 Result

Repository: https://github.com/WilhelmBerggren/id1212
Client: https://github.com/WilhelmBerggren/id1212/blob/master/src/main/java/hangman/Client.java
Server: https://github.com/WilhelmBerggren/id1212/blob/master/src/main/java/hangman/Server.java

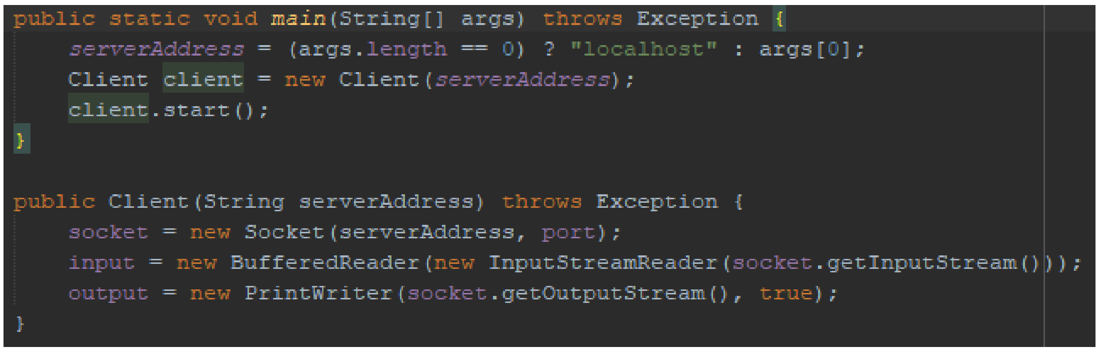
The client was constructed to have two new threads issued from the main thread. One for listening to the user and asynchronously sending input to the server, and one for listening to incoming messages from the server and outputting these to System out.

The server was constructed to run the listening on the main thread, and spawn a new thread for new connections. The game logic was implemented as a normal class, which can be instantiated separately for each connection.

A simple view of how the server operates:
main => serve => new client connection => new game instance => track score and create rounds => client disconnects

Requirement 1 Client and server communicate over a TCP connection using blocking TCP sockets.

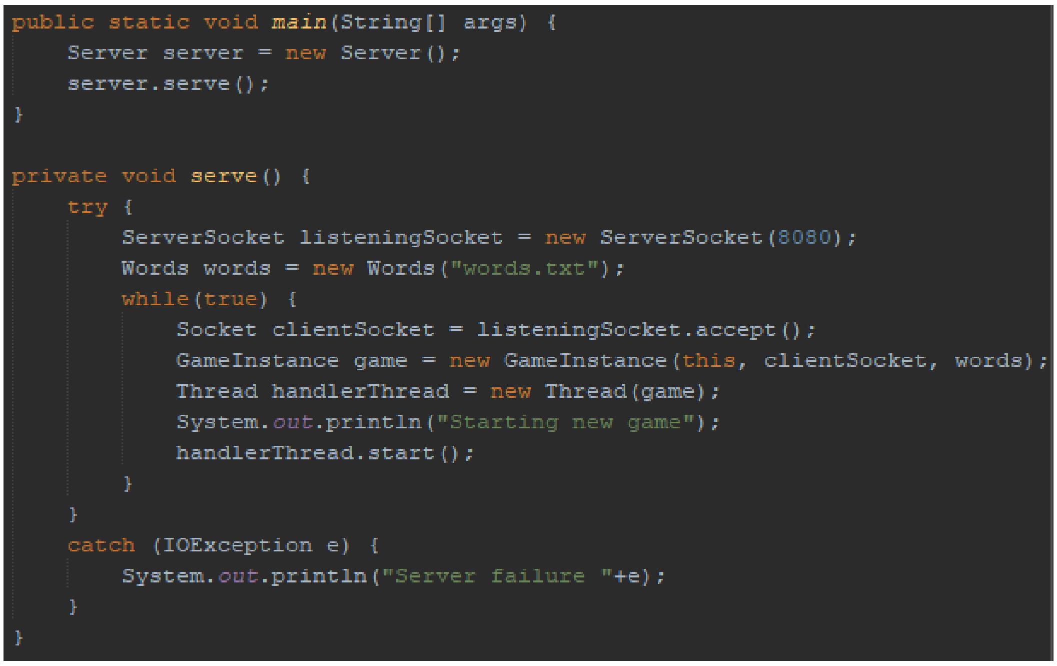
Client:



```java
public static void main(String[] args) throws Exception {
    serverAddress = (args.length == 0) ? "localhost" : args[0];
    Client client = new Client(serverAddress);
    client.start();
}

public Client(String serverAddress) throws Exception {
    socket = new Socket(serverAddress, port);
    input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    output = new PrintWriter(socket.getOutputStream(), true);
}
```

Pictured is the client establishing the socket from which the server is communicated with and assigning input and output of the socket.

Server:



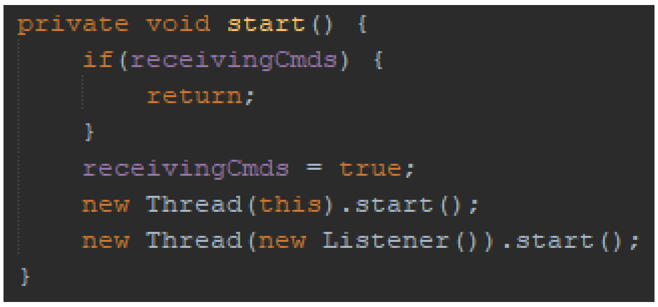
```java
public static void main(String[] args) {
    Server server = new Server();
    server.serve();
}

private void serve() {
    try {
        ServerSocket listeningSocket = new ServerSocket(8080);
        Words words = new Words("words.txt");
        while(true) {
            Socket clientSocket = listeningSocket.accept();
            GameInstance game = new GameInstance(this, clientSocket, words);
            Thread handlerThread = new Thread(game);
            System.out.println("Starting new game");
            handlerThread.start();
        }
    }
    catch (IOException e) {
        System.out.println("Server failure "+e);
    }
}
```

Pictured is the initial server loop which listens for client connections and starts new instances of the game with each connection passed along so as to communicate with it.

Requirement 2 Client does not store game data, only receives game data from server.

No screenshot needed for this requirement. The client simply sends and receives messages from the server.

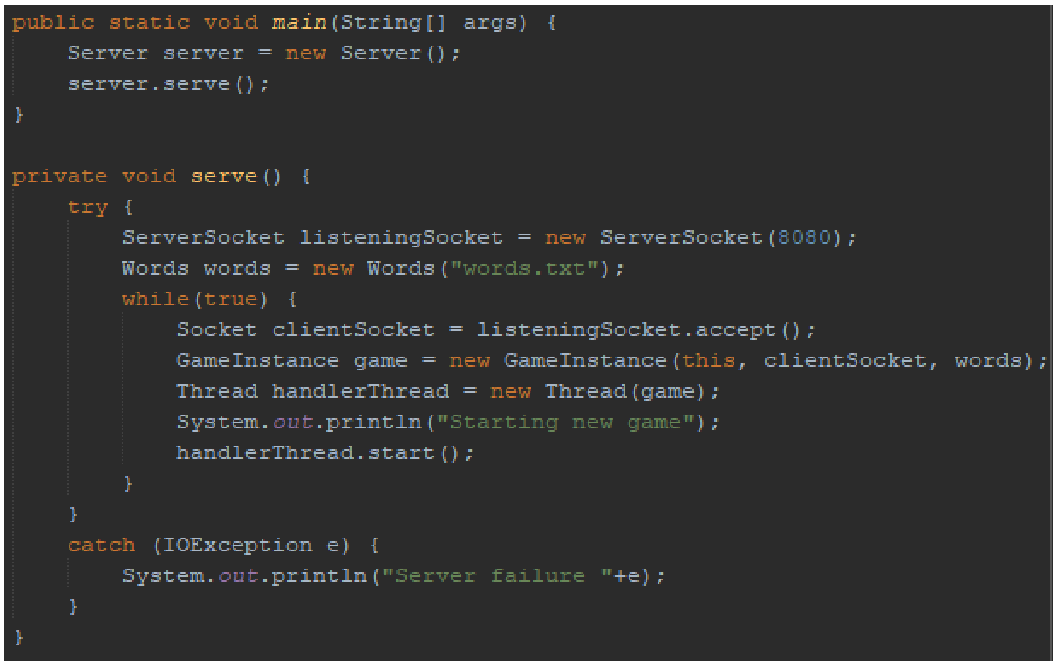
Requirement 3 Client has responsive user interface utilizing multithreading, so commands can be issued at any time, even when waiting for server messages.



```java
private void start() {
    if(receivingCmds) {
        return;
    }
    receivingCmds = true;
    new Thread(this).start();
    new Thread(new Listener()).start();
}
```

Pictured: The client's start method, which creates two separate threads for listening to new commands from the user and the server respectively.

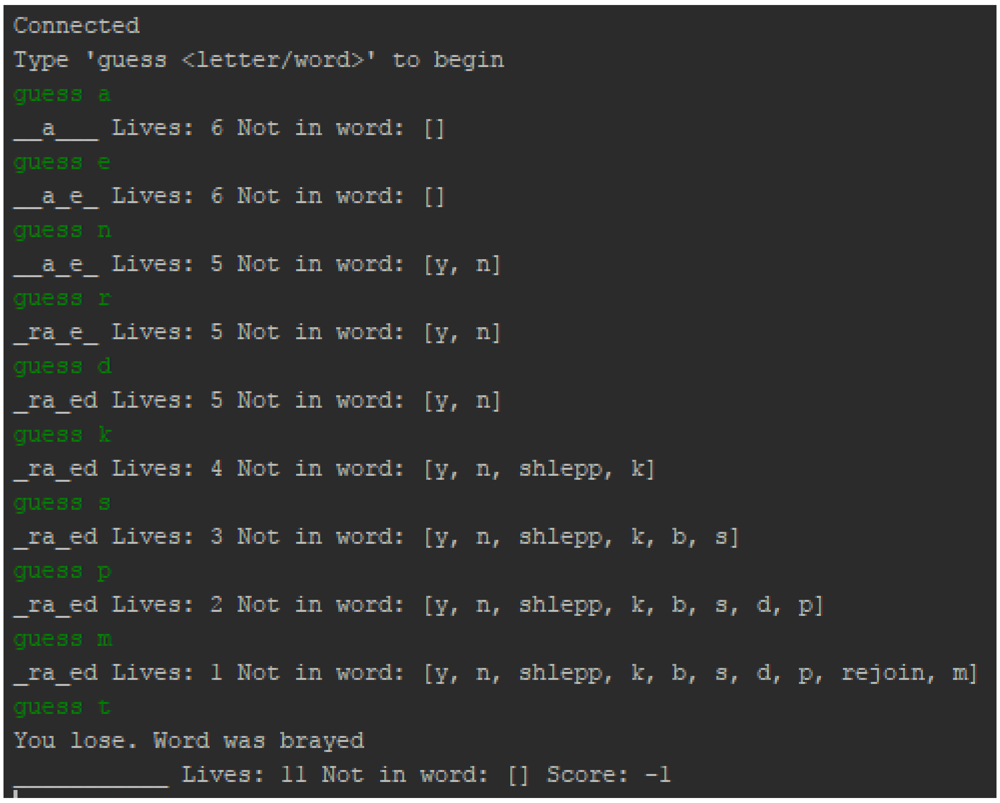
Requirement 4 Server must handle multiple clients using multithreading.



```
public static void main(String[] args) {
    Server server = new Server();
    server.serve();
}

private void serve() {
    try {
        ServerSocket listeningSocket = new ServerSocket(8080);
        Words words = new Words("words.txt");
        while(true) {
            Socket clientSocket = listeningSocket.accept();
            GameInstance game = new GameInstance(this, clientSocket, words);
            Thread handlerThread = new Thread(game);
            System.out.println("Starting new game");
            handlerThread.start();
        }
    }
    catch (IOException e) {
        System.out.println("Server failure "+e);
    }
}
```

Pictured: When a new connection to a user is made, a game instance spawns in a new thread. The server continues to look for threads even after the first client has connected, allowing more than one client to connect.

Requirement 5 User interface must be informative, current state visible and next step for user clear.



```
Connected
Type 'guess <letter/word>' to begin
guess a
__a___ Lives: 6 Not in word: []
guess e
__a_e_ Lives: 6 Not in word: []
guess n
__a_e_ Lives: 5 Not in word: [y, n]
guess r
_ra_e_ Lives: 5 Not in word: [y, n]
guess d
_ra_ed Lives: 5 Not in word: [y, n]
guess k
_ra_ed Lives: 4 Not in word: [y, n, shlepp, k]
guess s
_ra_ed Lives: 3 Not in word: [y, n, shlepp, k, b, s]
guess p
_ra_ed Lives: 2 Not in word: [y, n, shlepp, k, b, s, d, p]
guess m
_ra_ed Lives: 1 Not in word: [y, n, shlepp, k, b, s, d, p, rejoin, m]
guess t
You lose. Word was brayed
___________ Lives: 11 Not in word: [] Score: -1
```

Pictured: when the client connects, the server sends messages directing the user of what to do and the progress of the game. When one round ends, a new one begins.

5 Discussion

I consider each requirement to have been met. I learned that using such a simple model for sending and relaying data, while fitting in this simple program, came with downsides in how messy some parts of the code were. The main problem in creating this program was not to have a working version, but to separate everything into classes, methods and sections in order to have each part be responsible for its own thing in terms of modularity.

I could have done things better by focusing more on the modeling of the project and keeping things clean. I always try to create the most structured implementation I can, but for one homework assignment I could not be a perfectionist. Quality code projects are written over a much longer time span than one week.

6. Comments About the Course

Time spent on this course so far includes the total time of the introductory and module 1 lectures plus an additional scrub through the first lectures about the text program to get an idea of how to structure my own project. It also includes around 7-10 hours of writing the program and writing the report.

Something that I thought could be presented better was the huge chat program. Perhaps the complicated data flow model could be better explained by the simplest possible implementation, rather than having bits and pieces scattered across many files. I personally learn the best by seeing an idea being built or implemented, rather than being introduced to a huge project. This I think was done well in the latter part of the introductory lectures.