CPSC 250L Lab 9
Networking

Fall 2016

1 Introduction

This lab introduces basic network I/O. Specifically, in this lab, you will implement a server that will interact with a client. To do so, you will use *sockets*. You are able to read from and write to sockets the same way you do so with files. Specifically, look up the `ServerSocket` and `Socket` classes in the Java API documentation.

2 Exercises

Fork and clone the `cpsc250l-lab09` repository in the `cpsc250-students` group. Your commit quota for this lab is 3. That is, you must commit 3 times in order to receive full credit for this lab.

2.1 `GuessingServer`

Exercise 1

Imagine a simple game in which you are asked to guess a number between 0 and 300. You

have 9 tries to guess this number, and you are told either "high" or "low" after each guess (meaning that your guess is either higher or lower than the number you are trying to guess). You win the game if you find the number in 9 guesses or less.

Fortunately, we can win the game each time if we use a **binary search** algorithm: choose the value in the middle of a sorted list; if it is higher than the number we are guessing then our number is on the first half of the list; if it is lower, then our number is on the second half of the list. By subdividing the list in two, we are assured to will find our number in at most $\lceil log_2(n) \rceil$ tries (where n is the size of the list).

To see how a binary search works, let us imagine that we are guessing a number between 37 and 337. This is a list with 300 values, which means we can guess the number in 9 tries or less. Following the binary search algorithm, we choose the value in the middle of the list (in this case 187) as our first guess.

We are told that the number is "low" thus we keep searching on the second half of the list. Figure 1 shows the different binary search iterations: iteration 1 has values 37 to 337; we chose the middle value (187) and were told that it was low. This means that for the next iteration (iteration 2), we will choose the second half of the list (making 187 the lower boundary) and select its middle value (262). Iterations continue until identifying that the number to guess is 206.

Iteration	**Lower Bound**	**Upper Bound**	**Midpoint (Guess)**	**Response**
1	37	337	187	"low"
2	**187**	337	262	"high"
3	187	**262**	224	"high"
4	187	**224**	205	"low"
5	**205**	224	214	"high"
6	205	**214**	209	"high"
7	205	**209**	207	"high"
8	205	**207**	206	"won"

Figure 1: Binary search iterations to find a number between 37 and 337.

In this exercise, you will create a server program that guesses numbers via binary search. The game is played each time a client contacts the server. The client will begin the game by sending a message of the form "l h" where l and h represent the initial lower and upper bounds respectively. Afterwards, you will respond with the midpoint of these numbers

$((l + h)/2)$ and the client will tell you if guessed correctly (you receive a "won" message), if your guess is too high (you receive a "high" message), if your guess is too low (you receive a "low" message), or if you ran out of guesses (you receive a "lost" message). If you won or lost, you terminate the game immediately. Otherwise, you set the appropriate bound to the midpoint and guess again. You terminate the server if and only if you receive "SHUT DOWN" as the initial message in any game. This means, that the server should be able to play with any number of sequential clients. A diagram of the game protocol is depicted in Figure 2.

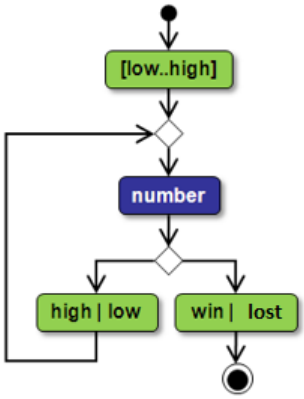

Figure 2: The guessing game message protocol.

Proceed by doing as follows.

1. Create a class called `GuessingServer`. This class should:
   - create a server socket that listens for clients on port 5150;
   - follow the algorithm given in the preceding paragraph.
2. Test your code against `GuessingServerTest`.

Exercise 1 Complete

Run:

```
git add .
git commit -m "Completed exercise 1"
git push origin master
```

---

3 Common Mistakes

The solutions to common mistakes are as follows.

1. Do not shutdown the server until a client sends the "SHUT DOWN" message.
2. Be sure to close the `Socket` object at the end of each game.
3. Open the `PrintWriter` as follows.

   Code:

   ```
   PrintWriter toCli = new PrintWriter(client.getOutputStream(), true)
   ```

   This enables auto flush so that messages can be sent immediately, as opposed to waiting until there is enough data for a "worthwhile" transmission.
4. Do **not** stop the server when debugging or after errors. If you stop it then port 5150 will not be available the next time you run the program. If you want to stop it, you **need** to send the "SHUT DOWN" message.
5. Use many `System.out.println` calls when debugging. This will make it easier to see what error is occurring.