CS 2112 Fall 2019
Assignment 7
Distributed and Concurrent Programming

Due: Tuesday, December 10, 11:59PM
Design Overview due: Tuesday, November 26, 11:59PM

In this assignment, you will build a web service that simulates the Critter World atop your code from Assignment 5. You will also modify the GUI you built for Assignment 6 to communicate with this server using HTTP (HyperText Transfer Protocol). This separation between user interface and simulation will allow multiple clients to interact with the same Critter World running on a single server.

You are expected to fix problems in your submissions for previous assignments. Part of your score for this assignment will be based on the correctness of the functionality from Assignments 4–6.

1 Changes

- None yet – watch this space!

2 Instructions

2.1 Final project

This assignment is the fourth and final part of the final project for the course. A detailed description can be found in the Project Specification.

2.2 Partners

You will work in groups of two or three for this assignment. This should be the same group as in the last assignment.

Remember that the course staff are available and happy to help with any problems you run into. Read all Piazza posts and ask questions that have not been addressed, attend office hours, or set up meetings with any course staff member for help.

2.3 Restrictions

Use of any standard Java libraries from the Java SDK is permitted. If there is a third-party library you would like to use, please post to Piazza to receive a confirmation before using it, and modify your `build.gradle` file appropriately.

2.4 Design overview document

We require that you submit an early draft of your design overview document before the assignment due date. The Overview Document Specification outlines our expectations. Your design and testing strategy might not be complete at that point, but we would like to see your progress. Please also indicate the operating systems and the versions of Java you are using. Feedback on this draft will be given as soon as possible.

2.5 Version control

As in the last assignment, you must use a version control system and submit a file `log.txt` that lists your commit history from your group. Additionally, you must submit a file showing differences for changes you have made to files you submitted in Assignment 6. Version control systems already provide this functionality.

3 Introduction

Distributed applications are challenging to build. We may be forced to use third-party libraries, which may be buggy or badly documented. The already-complex code for standalone applications must be modified to handle a wide range of requirements.

This assignment helps you learn about various aspects of distributed applications. In particular, you will learn about

- the HyperText Transfer Protocol (HTTP),
- JavaScript Object Notation (JSON),
- the Java servlet framework, and
- implementing thread-safe code.

Good use of the model-view-controller (MVC) design pattern—in particular, a clear separation of the model from the view and controller—will simplify your tasks in this assignment. Nevertheless, you will need to extend both the model and the view to handle HTTP communications.

This assignment contains many hidden corners and surprises, and the tasks may seem daunting to you at first. Avoid last-minute headaches by starting early.

4 User Interface

You must be able to launch your server and client (GUI) from the command line. If your jar file (automatically built by Gradle, and found in `build/libs/`) is called `critterworld.jar`, the following command should launch your GUI:

```
java -jar critterworld.jar
```

This command should launch your server:

```
java -jar critterworld.jar [port] [read password] [write password] [admin password]
```

For example, the following command would launch the server on port 54345 with read, write, and admin passwords of `bilbo`, `frodo`, and `gandalf`, respectively:

```
java -jar critterworld.jar 54345 bilbo frodo gandalf
```

5 Demo Server

To help you with A7, we set up a demo server. It is running at:

```
http://34.66.156.108:8080/
```

If you visit it in a browser, you should get the text `Ok`. The read, write, and admin passwords are `bilbo`, `frodo`, and `gandalf`, respectively.

The server is provided to help you understand A7 and test your client. You can make HTTP calls to it using Postman, CURL, or any other software of your choice, or login with your client. It is meant to fully adhere to the API, but you may discover some bugs or discrepancies. If so, please let us know, and we will try to fix it. But in any case, you should implement your client and server to the API, not to the demo servers' behavior.

Note: The demo server in general gives very bad error messages. Most invalid requests result in a large internal stack trace. Do not model your error messages on the demo server! Please try to make your server fail gracefully, but as long as it does not crash and keeps responding to future requests properly, it is fine.

6 Overview of HTTP

Disclaimer: This overview omits many real-world, low-level specifications that are unimportant for this assignment. Learn more about HTTP in CS 4410 and other courses related to networks.

An HTTP web service is a computer program that accepts **HTTP requests** over the network and returns **responses**, usually in the form of **HTML documents** that web browsers can render to human-viewable web pages. We refer to the entity that sends requests (such as your computer) as the **client**, and the program that returns responses as the **server**.

In this assignment, you will write a server that simulates the Critter World. You will also modify your GUI to act as a client that queries the server for the state of the world. Your world model will become the back end of your server.

An HTTP request has several components, the most important of which are as follows:

- **URL (Uniform Resource Locator)**: An example of a URL is a web address, such as `http://www.google.com/`. Whenever a URL is entered in a browser, an HTTP request is sent from the computer running the browser (the client) to another computer (the server) that processes the request and serves a web page. For example, a computer operated by Google is the server that will receive the request when `http://www.google.com/` is entered in a browser.

- **Query string**: A URL might contain a **query string** contain parameters of the request. The query string is the part of the URL and contains data to be passed to the web applications that will process the request. For instance, in the URL `http://www.google.com/search?client=ubuntu&q=hello&ie=utf-8&oe=utf-8`, the substring after the `?` is the query string, and `&` is a delimiter between query parameters in the query string.
  The server interprets query parameters in various ways. For example, the Google server receiving an HTTP request for the URL above might learn that the user is using the Ubuntu operating system (`client=ubuntu`), that the search keyword is "hello" (`q=hello`), and that the input and output encodings are UTF-8 (`ie=utf-8` and `oe=utf-8`). In this assignment, our client will use query strings to communicate with the server.
- **Method**: In HTTP parlance, a **resource** may be a web page, an image, a video, etc. There are four ways to interact with with a resource, as specified by an **HTTP method**. In this assignment, you will use three methods:
  - GET — Request data from a specified resource.
  - POST — Submit data to be processed by a specified resource.
  - DELETE — Request to remove the specified resource.
- **Body**: Sometimes, query strings alone are not enough to contain all the data needed to process the request. For instance, special characters like `=` and `&` have special meanings. What if the data being submitted (e.g., a critter program) also contains these characters? (Try searching for `a&b=c` in Google and look at the resulting query string.) What if you want to upload a binary file to the server? For these reasons, an HTTP request (particularly, a `POST` request) may also include a **body**, which is a string of arbitrary size and content. This body can contain any data in any format. For this assignment, the most convenient format is JSON, outlined in Section 7.
- **Content type**: The **content type** specifies the format of the data contained in the body. Commonly used content types are `text/html`, `application/pdf`, and `application/json`.

A **response** to an HTTP request contains a content type, a body, and a status code. If the query was for a web page from a website, the response body contains an HTML document that can be rendered by the browser. The **status code** is a number representing a certain class of responses. For instance, 200 stands for **OK**, meaning that the request was successful. Perhaps the most recognizable status code is 404, meaning that the requested resource was **Not found**.

In this assignment, any request that is undefined in the API should receive a 404 response.

7 Overview of JSON

JavaScript Object Notation (JSON) is a simple data-interchange format commonly used on the web. In this assignment, the server and the client will use JSON for the body of HTTP requests and responses. Go to `http://www.json.org/` to learn about the JSON format.

Many libraries provide this functionality; we recommend https://github.com/google/gson/. There are a number of other possibilities listed on the JSON website http://www.json.org/.

Java objects can be converted to and from JSON directly. With the Google library, a Java object can be serialized as a JSON string for transmission using toJson(Object), then reconstituted on the receiving end using fromJson(String, Class).

8 Thread safety

Your server should support connections from multiple clients viewing the same world. If multiple clients are interacting with a single server, but the server only uses one thread to handle all client requests, some clients might have to wait for a long time. Therefore, the server should handle multiple requests concurrently. Java provides a web server with a **thread pool** that runs in the background. A thread pool contains a fixed number of threads that handle requests. Each new request is passed to a thread in the thread pool for processing. Concurrent requests are handled by different threads. To make this work correctly, all shared resources must be accessed in a thread-safe fashion. Your implementation of the server will be tested for thread safety against multiple clients running in parallel.

One way to make the implementation thread-safe is to lock the entire world with a mutex. However, a **coarse-grained** lock like this can reduce concurrency to an unacceptable level. An improvement is to use reader-writer locks such as one provided in the standard Java library: `java.util.concurrent.locks.ReentrantReadWriteLock`.

9 Application programming interfaces

The A7 Application Programming Interface (API) specifies how the server and the client interact. Click on each item in the API to see more detail about it.

Your server must be able to operate with our implementation of the client, and your client must be able to operate with our implementation of the server. To make this possible, you need to follow the given API specs religiously. Extensions to the API, such as by adding new parameters or fields, are possible, but your implementation must still be able to interoperate with other implementations that might not provide or use this extra information.

10 Project setup

Your project will comprise both client-side code that is a regular JavaFX application and server-side code that runs on a web server. As a starting point, we have provided sample code for a client and a servlet that are configured to talk to each other. They can be found on Github.

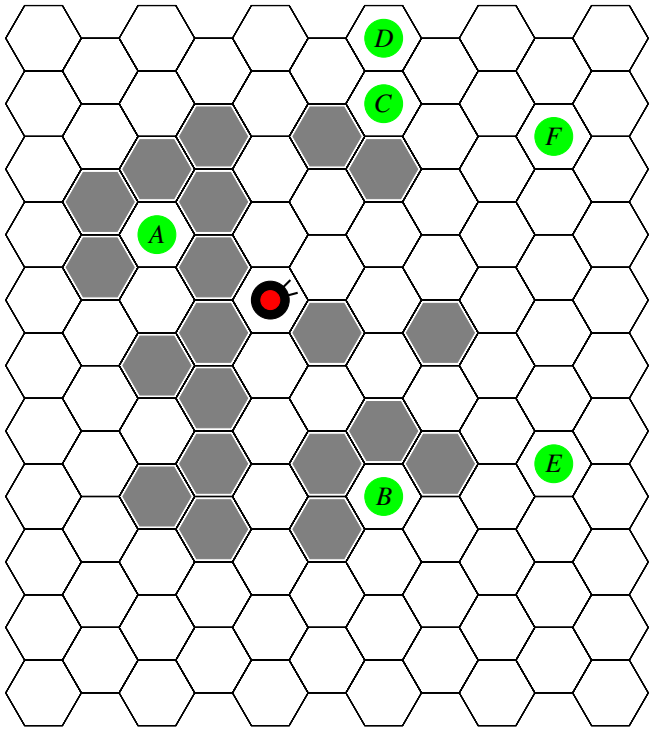

Figure 1: Finding food in a challenging environment

11 Improved food sensing

Critters are now given a sense of smell so that they can find food more easily. The sensor expression `smell`, which has only returned 0 so far, now returns information about the closest food, taking into account obstacles and the need for the critter to turn, including any initial turn(s) to start heading in the right direction. For instance, Figure 1 illustrates an environment in which the critter is heading northeast. Without obstacles, the closest food would be at distance 3 to the northwest (direction 4 relative to the critter's current orientation). Because of the rock wall, however, at least 18 turns and moves are required to reach the food. Food C is closer at 6 turns and moves with relative direction 0. This route is faster than an alternative route of length 7 with relative direction 5. Meanwhile, no obstacles stand in the way of Food F at distance 4 with relative direction 0. Figure 2 shows the distance from the critter to various hexes, taking obstacles into account.

The result of the `smell` expression is based on two parameters: *distance* and *direction*. The distance is the smallest number of forward moves and turns to a hex adjacent to the food with the critter facing toward the food, provided it is no more than `MAX_SMELL_DIST` = 10. This is the smallest number of forward moves and turns for the critter to be in a position to eat food. The direction is relative to the critter's current orientation and is toward a hex that decreases the minimum number of turns or moves to food.

In Figure 1, Foods-C and D are at distance 6. If Food-F did not exist, either of these could be chosen. To reduce the distance to these food items, the critter could either turn

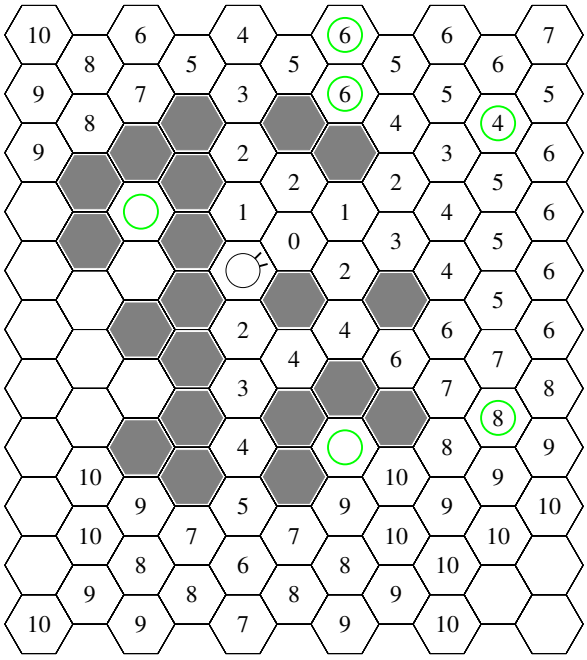

Figure 2: The least number of turns and moves from the critter to various hexes

left and move north to get closer to Food-D, or move northeast to get closer to Food-C. The corresponding relative directions are 5 and 0.

Given distance and direction as defined, the result of the `smell` expression is:

$$distance * 1,000 + direction$$

Since food-F is the closest in the example, the sensor should have value 4000. If food-F were not present, then directions to either food-C or food-D could be returned, which are 6000 and 6005 respectively. If there is no food within a 10-hex walk, `smell` evaluates to 1,000,000.

Describe your implementation approach in the overview document.

12 Written problems

Use the graph in Figure 3 in these problems:

1. Starting from vertex *a*, give the sequence of vertices visited when doing a depth-first traversal, assuming children are visited in alphabetical order.

2. Do the same for a breadth-first traversal.

3. Draw the quotient graph obtained by collapsing the strongly connected components. Label each vertex of the quotient graph with the vertices of the corresponding strongly connected component in the original graph.

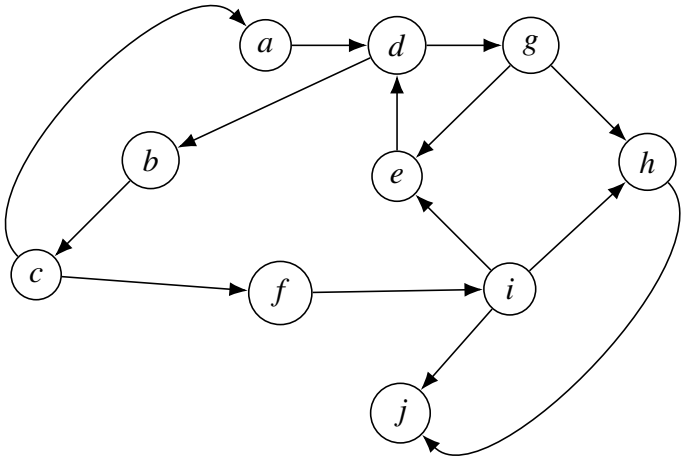

Figure 3: The graph for the written problems

13 Ring buffers

A **ring buffer** is a fixed-size FIFO queue implemented using an array and two integer indices called the **head** and **tail**, along with some number of locks or condition variables. It can be used to implement the thread pool introduced in Section 8.

This data structure is commonly used in distributed systems with limited memory. For insertion, the item inserted at the tail end of the array, and the tail index is incremented. If there is insufficient space in the array, as determined by the two indices, then the insertion fails. For removal, the item at the head is returned, and the head index is incremented. If a **producer** thread tries to insert an item into the queue but the array is full, the thread should block until there is space. Conversely, a **consumer** thread that tries to remove an item from an empty queue should block until a producer thread pushes a new item onto the queue.

13.1 Implementation

We have provided you with a partially broken ring buffer that is not thread-safe. You must update the `put` and `take` methods to be thread safe.

Any implementation of concurrent code is likely to be wrong unless all project members have examined the implementation carefully together. While it might be tempting to delegate the job of implementing the ring buffer to one partner, we recommend that all project partners work together closely on this implementation.

13.2 Testing

Testing concurrent code is challenging because it is **nondeterministic**, meaning that events do not always occur in the same order. The observed behavior depends on thread-scheduling choices that are made by the runtime system and are out of the control of the application programmer. To test the ring buffer, develop a test harness that causes several threads to issue method calls concurrently.

One effective trick for catching concurrency bugs is to inject random delays throughout your code, conditioned on a boolean constant flag so that they can be turned off when not debugging. Random delays will cause your program to explore a wider variety of thread schedules. Generous use of assertions is also highly recommended as a way to catch race conditions and other bugs.

Along with the broken RingBuffer, the release code contains a suite of test cases to test its concurrent and non-concurrent functionality. It is recommended, although not required, that you add some test cases of your own to practice testing concurrent programs.

14 Project Presentation

After you submit the assignment, you will have to demonstrate your distributed application to a member of the course staff. These meetings will take place around the due date of the assignment. Only a little preparation is necessary, but you are expected to know your code inside and out, to talk knowledgeably about your design and implementation strategies, and to answer any questions of the staff. Your grader will reach out to you before the assignment is due to discuss mutually agreeable times and places for the meeting.

15 Extensions to the final project

If you are feeling ambitious, there are many ways to go beyond what is required in this assignment. You are welcome to add on functionality as you please, so long as your program meets the required specifications. This is **not** required and will not affect your grade.

15.1 Incremental updates

One useful extension implemented in previous years is incremental updating. The `GET /world` request reports the current state of the world. You can expand on this request by allowing the user to specify an optional `update_since` parameter that supports incremental updates about the world state. When this parameter is specified, the server should return a **diff**, the difference between the current state of the world and the state at the specified timestamp. This request allows the client to update the world view without having to download the entire world at each time step.

A good way to implement this API request is to maintain a **log**, a data structure that records the sequence of all changes to the world. In fact, version control systems like Git are implemented this way. For example, Git stores the entire content of a file when the file is committed for the first time. However, subsequent changes to the file are recorded in a log. Git can use the log to reconstruct any past revision of the file without having to store every version entirely.

There are many ways to implement this API request, as there are many possible sequences of updates that could correctly reconstruct the current state of the world from the specified earlier state.

16 Overview of tasks

Determine with your partner how to break up the work involved in this assignment. Here is a list of the major tasks involved:

- Learning JSON and choosing a JSON library.
- Implementing a thread-safe web server that runs the simulation and responds to HTTP requests from multiple clients according to the given API.
- Updating your GUI to be an HTTP client that talks to your server.
- Implementing a thread-safe ring buffer.
- Solving the written problems.
- Preparing your presentation for the course staff

17 Tips and tricks

You have been working hard on your project and it will be tempting to take a long break before starting on this assignment. You should resist the temptation. Even though the amount of code you need to write for this assignment is less than previous assignments, it is more likely to take much longer than you expect to refactor your code and get everything working. Get started early and make sure you understand all the component technologies such as servlets and JSON parsing.

Client/server interaction is the epitome of the model-view-controller design pattern. Not only should your model and view practically not know the other one exists, they should also be able to operate on completely separate machines, with HTTP as the only interface between them. Since you and your partner are likely already working on separate computers, one of your computers can be the server and the other the client. You can run your code on one machine with client/server communication through IP address `127.0.0.1` or `localhost`, which will work even if you have no internet connection, but it is more fun to have two computers talk to each other.

You can access each other's computers by finding out what **hostname** or IP address the server runs under. The simplest way to do this is to open up a terminal and look at the string before the directory and username on the prompt. The client can connect to this computer by setting the URL to `http://[hostname/IP]:8080`.

The seemingly easiest way to divide up work for this assignment is for one partner to implement the server and another to implement the client. This would not be a good idea! To be successful, each partner should know the entire project inside and out. Furthermore, two or three heads are better than one, especially when it comes to implementation.

If you intend to work on this project during Thanksgiving break, you and your partner will not see each other for a few days. This is where Git—with useful, detailed commit messages—and **lots of communication** will be essential.

In order to test servers on different machines, you might consider exposing one port of the server computer to the internet, so that the client computer can talk to it over any distance. For this to work, a port on the NAT box at your residence might need to be exposed. This can introduce security vulnerabilities, so be sure to turn off this port when you are done. Before tweaking the NAT box, you should ask its owner for permission.

Another way for you to test your code is to get server space via a third-party provider and run your code on it. One option is Openshift.

18 Submission

You should submit these items on CMS:

- `overview.txt/pdf`: Your final design overview for the assignment. It should also include descriptions of any extensions you implemented. You should also indicate the operating system(s) and the version(s) of Java you use.
- Zip and submit your `src` directory. This directory contains:
  - **Source code**: You should include all source code required to compile and run the project. All source code should reside in the `src` directory with an appropriate package structure.
  - **Resources**: All your resources for your GUI should be under `src/main/resources`, and thus should be included by simply zipping up `src`.
  - **Tests**: You should include code for all your test cases in `src/test/java`. You may create subpackages to keep your tests organized.

  Do not include any files ending in `.class`.
- `build.gradle`: Since we let you add external dependencies for this project and create a new main class, you must submit a `build.gradle` which contains both the correct main class name as well as any dependencies you require.
- `log.txt`: A dump of your commit log from your version control system.
- `a7.diff`: A text file showing a diff of changes to files that were submitted in the last assignment. This file can be obtained from the version control system.
- `written_problems.txt/pdf`: Your answers to the written problems.
- Zip and submit the source code for your `RingBuffer`, including any tests you have written