CS2112—Fall 2014
Assignment 7
Distributed and Concurrent Programming

Due: Friday, December 5, 11:59PM
Design Overview due: Friday, November 21, 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 over HTTP, the 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. Approximately 60% of the grades for this assignment will be based on the correctness of the functionality from Assignments 4–6.

Finally, you will implement a small concurrent data structure.

0 Changes

- No longer requiring submission of the commit log. (11/18)
- Added a written problem to implement a critter. (11/16)

1 Instructions

1.1 Grading

Solutions will be graded on design, correctness, and style. A good design makes the implementation easy to understand and maximizes code sharing. A correct program compiles without errors or warnings, and behaves according the requirements given here. A program with good style is clear, concise, and easy to read.

A few suggestions regarding good style may be helpful. You should use brief but mnemonic variable names and proper indentation. Keep your code within an 80-character width. Methods should be accompanied by Javadoc-compliant specifications, and class invariants should be documented. Other comments may be included to explain nonobvious implementation details.

1.2 Final project

This assignment is the fourth and final part of the final project for the course. Read the Project Specification to find out more about the final project and the language you will be working with in this assignment.

1.3 Partners

You will work in a group of two students for this assignment. This should be the same group as in the last assignment.

Remember that the course staff is happy to help with 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.

1.4 Restrictions

Use of any standard Java libraries from the Java 7 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.

2 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. You should also indicate the operating system and the version of Java you use. Feedback on this draft will be given as soon as possible.

3 Version control

You should use a distributed version control system, but we are no longer requiring you to submit your commit log.

4 Introduction

Distributed applications are challenging to build. Many third-party libraries, which may be buggy or badly documented, must be used. 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

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

A good use of the model-view-controller 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 handle HTTP communications.

This assignment contains many hidden corners and surprises, and the tasks may seem daunting to you at first. Avoid last-minute headache by starting early, and attending Lab 12, which will get

you started on servlets.

5 Overview of HTTP

Disclaimer: This overview omits many real-world, low-level specifications that are unimportant for this assignment. Learn more about HTTP in CS4410 and other courses related to networks.

An HTTP web service is a computer program that accepts *HTTP requests* over the network and returns *responses*, which are usually the content 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, and 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 to another computer, known as 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.
  URLs often contain *query parameters*. A query string is the part of a URL containing data to be passed to web applications. For instance, in URL `http://www.google.com/search?client=ubuntu&q=hello&ie=utf-8&oe=utf-8`, the substring after `?` is the query string, and `&` is a delimiter between query parameters in the query string.
  The server interprets query parameters in many ways. For example, the Google server receiving an HTTP request for the above URL 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 both UTF-8 (`ie=utf-8` and `oe=utf8`).
  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, specified with an *HTTP method*. In this assignment, you will use three:
  - GET — Request data from a specified resource.
  - POST — Submit data to be processed to a specified resource.
  - DELETE — Request to remove the specified resource.
- *Body*: Sometimes, query strings alone are not enough to contain all the data we would like to submit to the server. 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?
  Therefore, 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; however, the most convenient format for this assignment is JSON, outlined in Section 6.

- *Content type*: The content type specifies the format of the data contained in the body. Commonly used content types include `text/html`, `application/pdf`, and `application/json`.

A response to an HTTP request contains a content type, a body, and a status code. An HTML document a web browser receives from a website is contained in the response body. A status code is a number representing a certain class of responses. For instance, 200 stands for **OK**, meaning that the request has succeeded. 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.

6 Overview of JSON

JavaScript Object Notation (JSON) is a data-interchange format commonly used on the web. In this assignment, the server and the client will mostly communicate using JSON. This is the format you should use for the body of your HTTP requests and responses. Go to http://www.json.org/ to learn about the JSON format. JSON is one of the simplest data-interchange formats and should therefore serve as a good starting point for reading technical documents.

While JSON objects could be manipulated as strings, this is rather inconvenient. Instead, Java objects should be converted to and fro JSON objects directly. Many libraries provide this functionality; we ask you to pick one. These are two good choices:

- https://code.google.com/p/google-gson/
- http://www.json.org/java/

7 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 in parallel. Java EE 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. Therefore, 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. A *coarse-grained* lock like this, however, may 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`.

8 Application programming interfaces

We have defined a set of application programming interfaces (APIs) that specify how the server and the client must interact. Each item on the APIs can be expanded to give more details.

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 APIs carefully. Extensions to the APIs, 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.

8.1 Incremental updates

The `GET /world` API is the trickiest to implement. When the `update_since` query parameter is specified, the server is supposed to return a *diff*, the difference between the current state of the world and the state at the specified timestamp. This API allows the client to avoid downloading and redrawing the entire world for each time step.

One way to implement this API is to keep track of every version of the world. This could be done easily by deep-cloning the World instance at each time step and saving the clone in an array. You then need an algorithm that takes two World instances and computes their diff, i.e., the content of the grids that have changed between these two instances.

Saving all these worlds is inefficient in space, and comparing them is inefficient in time. A better way to implement this API is to maintain a *log*, a data structure that records a sequence of operations or changes. 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. Subsequent changes to the file, however, 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.

While there are many options to implement this API, full credit will be awarded as long as your implementation works correctly.

Note that on Apiary, this call may return updates since any previous time step no greater than given. In other words, your implementation may return more updates than necessary, but still sufficient, to produce the current world.

9 Project setup

Your project will comprise both client-side code that is a regular JavaFX application and server-side code that runs within a web server. As a starting point, we have provided an example code for a client and a servlet that are configured to talk to each other. They can be found under the `demoClient` and `demoServlet` directories.

Setting up a servlet project within Eclipse is a bit different from setting up a regular Java application. You will need to install the Java Enterprise Edition plugins for Eclipse (J2EE Standard Tools, or JST) and to use the Java EE perspective. Also, you should use the Tomcat 7.0 Server as the servlet container that you run the servlet within.

Follow these steps to set up a servlet project:

- Go to **File** → **New** → **Other** and select **Dynamic Web Project** under **Web**. Click **Next**.
- Click **New Runtime**, and on the pop-up window, select **Apache Tomcat v7.0** under **Apache**. Click **Finish** to create the project.
- Go to the directory where this project is created, and replace everything within **src/** with the `demoServlet` directory. You should now be able to refresh the project on Eclipse and see the code.
- At this point, you should be able to run the project. Open up `src/demoServlet/Servlet.java` and run it. Eclipse should launch the Tomcat server and open up a browser that shows a simple web page. You should also be able to visit `http://localhost:8080/demoServlet/` from a regular web browser and see the page. In fact, if you know your computer's IP address and your computer does not have a firewall turned on, you should even be able to view this web page from another computer on the network by replacing `localhost` with the appropriate IP address.
  You can also access the servlet using the `demoClient` application. If you run this application with three arguments, it will send a `POST` message that changes the message stored on the servlet.

Suggestion: Go to the **Project** menu and select **Build Automatically** to make sure latest changes to your code are always compiled.

Class `demoServlet.Servlet` contains methods `doGet` and `doPost` that handle the respective HTTP requests and generate responses. These methods should be replaced with your own implementation that dispatch to the appropriate code to handle the various API calls. There is also a `doDelete` method that is not shown, but it acts the same way and has the same signature.

10 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 in critters' way 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 2 to the northwest (relative direction 4). Because of the rock wall, however, at least 19 turns or moves are required to reach the food. Food 4 is closer at 7 turns or moves with relative direction 0. This route is faster than an alternative route of length 8 with relative direction 5. Meanwhile, no obstacles stand in the way of Food 2 at distance 5 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 fewest number of moves to the hex containing food, as long as it is no more than `MAX_SMELL_DIST` = 10. 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 4 and 5 are at distance 7. If Food 2 did not exist, any of these could be chosen. To reduce the distance the these food items, the critter could either turn left and move north to get closer to Food 5, or move northeast to get closer to Food 4. The corresponding relative directions are 5 and 0.

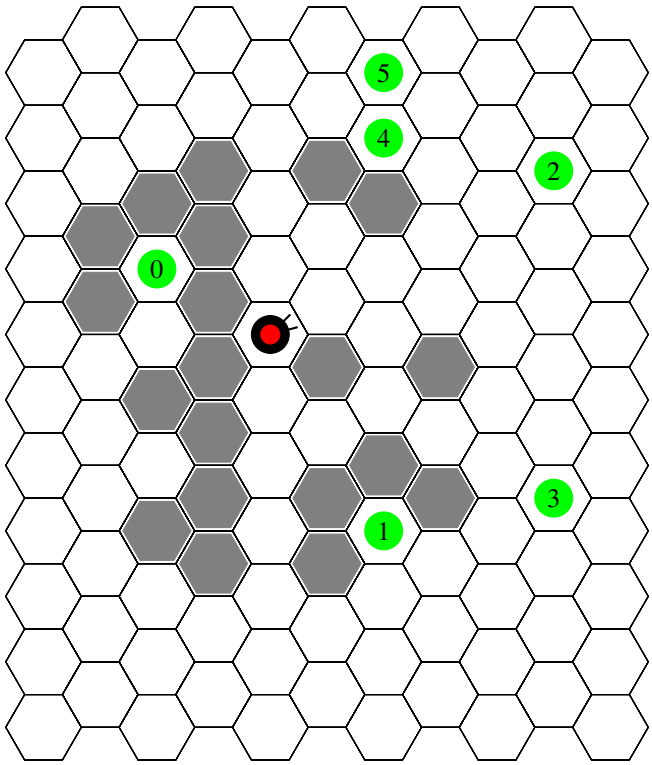

Figure 1: Finding food in a challenging environment

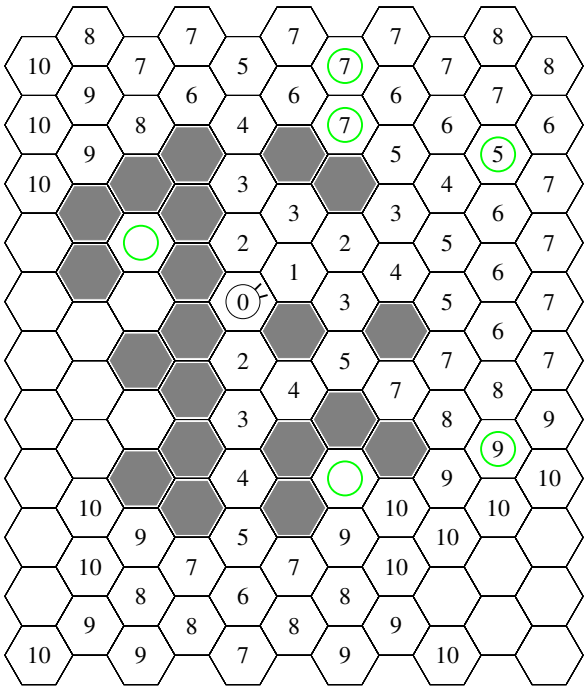

Figure 2: The least number of turns or moves from the critter to various hexes

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

$$distance * 1,000 + direction$$

Therefore, the sensor should have value 5000 in the example above. If there is no food within a 10-hex walk, `smell` evaluates to 1,000,000.

Describe your implementation approach and any specification changes you made in the overview document.

11 Ring buffers

As a separate implementation task, we ask that you implement a thread-safe *ring buffer*, one possible implementation of a thread pool introduced in Section 7. Ring buffers need not be used in your project implementation.

A ring buffer is a fixed-size queue implemented using an array and two integer indices called *head* and *tail*, along with some number of locks or condition variables. This data structure is commonly used in distributed systems with limited memory. For insertion, the item inserted to 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 index 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.

11.1 Implementation

Your thread-safe version of the ring buffer must implement all of the methods from interface `java.util.concurrent.BlockingQueue`, which describes a thread-safe queue abstraction. Any other methods should throw an `UnsupportedOperationException`. You must use an array to implement the ring buffer, and use of any classes from the `java.util` package or subpackages are disallowed. The provided class `util.RingBufferFactory` should be updated to use your implementation.

Specifically, your implementation must implement the following methods:

- From `Collections`: `add`, `contains`, `equals`, `isEmpty`, `iterator`, `size`.
- From `Queue`: all methods.
- From `BlockingQueue`: `offer`, `poll`, `put`, `take`. However, you need not implement the versions of `offer` and `poll` that take a timeout value.

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 group member, we recommend all project partners to work together closely on this implementation.

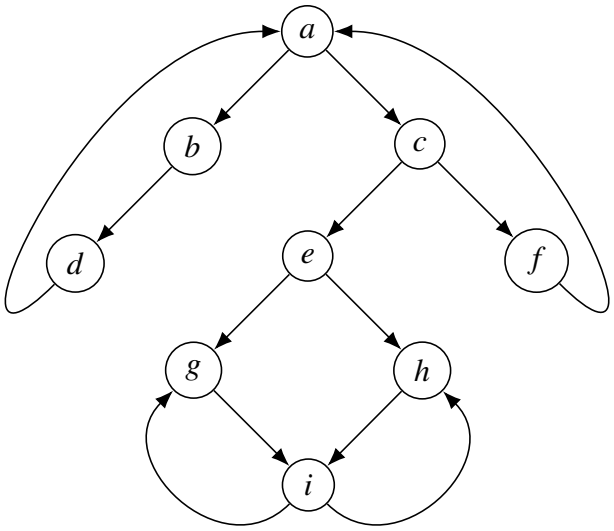

Figure 3: The graph for the written problems

11.2 Testing tips

Testing concurrent code is challenging because it never runs the same way twice. Develop a test harness for the ring buffer such that at least several threads issue method calls concurrently.

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

12 Written problems

12.1 Graphs

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.

12.2 Critter programs

4. Write a critter program that walks in a growing spiral that, on an infinite world without any rocks, would eventually hit every hex. When it comes to food, it should eat the food.

13 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 parser 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.

14 Project demos

After you submit the assignment, you will have to demonstrate your distributed application to one of the course staff members. These meetings will take place the week before the final exam. No preparation is necessary, but you are expected to know your code inside and out, to talk about your design and implementation strategies effectively, and to answer any questions the TA has. Your grader will reach out to you before the assignment is due to discuss mutually agreeable times and places for the meeting.

15 HARMA

HARMA questions do not affect your raw score for any assignment. They are given as interesting problems that present a challenge.

15.1 Extensions to the final project

There are many ways to go beyond what is required in this assignment!

- Add user management and authentication, so many users can be given more limited access to a world, and only a small number of users have administrative privileges.
- Make the client application into a Java applet that runs inside the web browser.
- Make the world able to grow indefinitely as critters explore, so that a large number of users could be supported without colliding with each other.
- Build a parser and serializer in Java for Doge Serialized Object Notation (DSON).

Finally, you probably have many good ideas of your own.

16 Tips

Server/Client 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 supposedly already working on separate computers, one of your computers should be the server, the other one the client. While you can definitely run your code on one machine, and therefore have server/client communicate through IP address `127.0.0.1`, or `localhost`, which will work even if you have no internet connection, this is less fun than having 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 the other to implement the client, but this might not be the best practice. Each partner should know the entire project inside and out. Furthermore, two heads are better than one, especially when it comes to implementation.

If you are going 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.

17 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 and of any **HARMA** problems you attempted. You should also indicate the operating system and the version of Java you use.
- A `zip` file containing these items:
  - *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.
  - *Other Files*: You should include all other files needed for your project. For example, you might use image files or other data files to control appearance. Be sure to include these.

  Do not include any files ending in `.class`. Git users: to save space, exclude the hidden `.git` folder when zipping.

- `a7.diff`: A text file showing diff of changes to files that were submitted in the last assignment, obtained from the version control system.
- `written_problems.txt/pdf`: This file should include your response to the written problems.