CS 2112 Fall 2025

Assignment 6

Distributed and Concurrent Programming

Due: Monday, December 8, 11:59PM
Draft Design Overview due: Tuesday, November 25, 11:59PM
Project Presentations: December 10–14

In this assignment, you will build a web service that simulates the Critter World atop your code from Assignment 4. You will also modify the GUI you built for Assignment 5 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 3–5.

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 Ed 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, as is Spark and GSON. If there are other third-party libraries you would like to use, please post to Ed to receive confirmation before using them, 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. 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. This can be generated from the root of your repository by running the

command `git log > log.txt`. This generates a file called `log.txt` in the root of your repository with the full commit log.

Before you do anything else, we highly recommend you create a tag in your GitHub repo with the state of your final A5 submission, so that you can easily reference it in the future. From the right-hand panel of your GitHub repo, choose the "Create a new release" link. Then, give the release a title, and click the dropdown above that says "Choose a tag". Give your tag a memorable name like `a5` and choose "Publish release". This will mark this location in time on your repo with the tag name for easy reference.

You must submit a file named `a6.diff` showing differences for changes you have made to files you submitted in Assignment 5. To generate the diff file, you can then use `git diff a5 main > a6.diff` to generate a file named `a6.diff` with all the changes between your `a5` tag and the current state of your `main` branch.

2.6 Release

There is no release code for this assignment. However, if you choose to use our recommended libraries, you will need to add them to your `build.gradle` file. Under the **dependencies** block, add the two libraries above the existing JUnit dependencies, like so:

```
dependencies {
    // the spark webserver
    implementation 'com.sparkjava:spark-core:2.9.4'
    // GSON, for parsing and generating JSON
    implementation 'com.google.code.gson:gson:2.13.2'
    // ...
    // JUnit dependencies can stay here
}
```

Your project will comprise both client-side code that is a regular JavaFX application and server-side code that runs on a web server. If you are using Spark and GSON, the lab demo code is a good starting point, demonstrating a client and a servlet that are configured to talk to each other. They can be found on the course website.

We recommend downloading a fresh copy of the lab demo, as some updates have been made since the lab, with the client code now demonstrating how to send the `Content-Type` header, and the server sending more CORS headers.

3 Introduction

Distributed applications are challenging to build. 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]
```

For example, the following command would launch the server on port 8080:

```
java -jar critterworld.jar 8080
```

Note that this means your `main()` method must be capable of starting either your client or your server, depending on the arguments provided. This can be accomplished by either modifying your `main()` method from A5 to parse the arguments, or by creating a new class with a `main()` method and updating your `build.gradle` to point to it.

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 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 have a **query string** that contains parameters of the request. The query string is 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 both query strings and body content to communicate with the server.
- **Method**: In HTTP parlance, a **resource** may be a web page, an image, a video, etc. There are many 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, the data being sent may be too long to fit in a URL, like uploading a binary file

to the server. What if the data is meant to be secret (like a password) and should not be saved in the browser's history? To solve these problems, some HTTP requests (particularly `POST` requests) 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 6.

- **Content type**: The **content type** is a header that specifies the format of the data contained in the body. Commonly used content types include `text/html`, `application/pdf`, and, for requests with body content in this assignment, `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.

6 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).

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 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`.

8 Application programming interfaces

The A6 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 must follow the given API specs precisely. 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.

9 Demo Server and Client

To help you with A6, we have set up demo versions of the server and client. These reference implementations can be useful to test your server and client against, ensuring they correctly adhere to the API specification.

9.1 Demo Server

The demo server is running at:

```
http://critterworld.azurewebsites.net/
```

If you visit it in a browser, you should get the text `Ok`.

The server is provided to help you test your client. You can make HTTP calls to it using Bruno, 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.

9.2 Demo Client

The demo client is located at:

```
http://www.cs.cornell.edu/courses/cs2112/2025fa/client
```

You can visit it in your browser and enter the server address you wish to connect to.

The client is provided to help you test your server. Note that you will need to set your server to append HTTP CORS headers to all responses and respond to preflight requests to allow it to be reached by our webpage, or else your browser will refuse to connect. This can be done by adding an `options` endpoint and calling the `Spark.after()` method with an appropriate lambda to add the header, like so:

```
Spark.after((request, response) -> {
  response.header("Access-Control-Allow-Origin", "*");
});
Spark.options("/*", (request, response) -> {
  response.header("Access-Control-Allow-Methods", "GET, POST, DELETE");
  response.header("Access-Control-Allow-Headers", "Content-Type");
  response.status(200);
  return "OK";
});
```

This can be inserted after your call to `Spark.port()`. If you're using the demo server from the lab as your starting point, this is already present.

The demo client is very slow—indeed, it runs at only 1 frame per second. In addition, its UI is relatively crude. Do not model your client after the very poor user experience of the demo client.

Like the demo server, the demo client is meant to adhere to the API, but bugs are possible, so if you discover any, please report them.

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 easiest food to reach, taking into account the amount of energy required to turn around and traverse obstacles, including any initial turn(s)

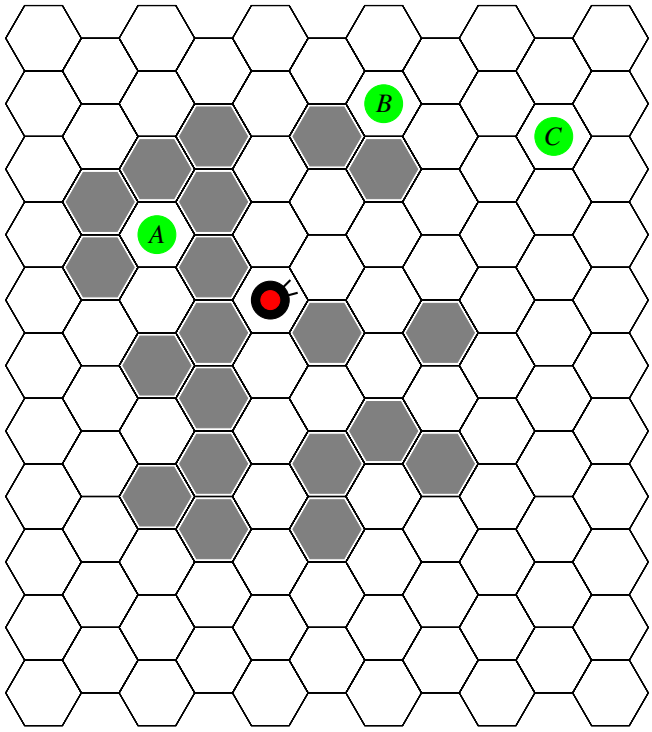

Figure 1: Finding food in a challenging environment

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 11 moves and 7 turns are required to reach the food, costing 40 times the critter's size in energy. Food B is easier to reach, at 4 moves and 2 turns (14 times critter size energy) with relative direction 0. This route is cheaper than an alternative route of 4 moves and 3 turns with relative direction 5, costing 15 times the critter size in energy. Meanwhile, no obstacles stand in the way of Food C at 4 moves with relative direction 0, which only requires 12 times the critter size in energy to reach.

The result of the `smell` expression is based on three parameters: *energycost*, *crittersize* and *move*. The energy cost is the total energy cost required to traverse the cheapest route to a hex adjacent to the food with the critter facing toward the food, provided it is no more than a radius of `MAX_SMELL_DIST` (= 10) hexes away. Otherwise, `smell` evaluates to `-1`. The specifics of the final return value are detailed in the project spec. Additional worked examples are also available in the project spec.

Describe your implementation approach in the overview document.

11 Written problems

11.1 Graphs

Use the graph in Figure 2 in these problems:

1. Starting from vertex a, give the sequence of vertices visited when doing a breadth-first traversal, assuming children are visited in alphabetical order.
2. Do the same for a depth-first traversal.
3. Recall edge classification. Based on the traversal starting from a, label each edge in the graph as a tree edge, forward edge, back edge, or cross edge.

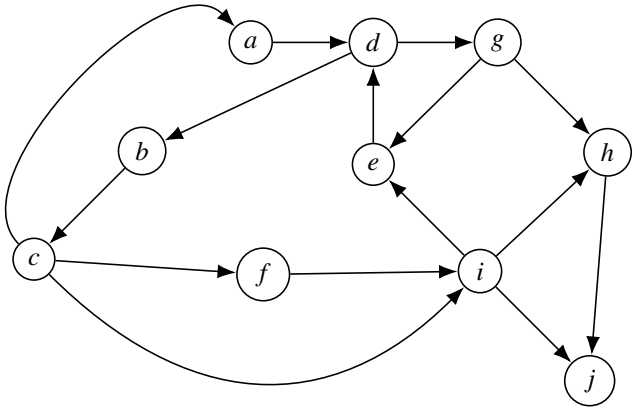

Figure 2: The graph for the written problems

11.2 Critter Programs

4. Write a forager critter program that finds and eats the food that it can reach using the minimum energy possible. If the energy required to reach the food that costs the least energy exceeds the energy the critter currently has, the critter should wait. The critter should always take the path that will cost the least amount of energy, not the fewest number of actions.

   Be sure to test your program works even when the food on the world changes; at every point in time, the critter should be moving in the most energy-efficient way. Include sufficient comments in your program or accompanying your program explaining the algorithm you developed.

12 Project Presentation

Before or 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. Some preparation is recommended—be prepared for your group to talk for about 10 minutes and take questions for 5. You don't have to prepare slides but plan out what you will show about your project and what be said (and who will say it). Giving a practice presentation is highly recommended. Everyone in the project group should attend and be ready to answer questions about the project: design, implementation, and testing. We will set up times on CMSX for students to sign up for presentation slots.

13 KARMA: 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.

13.1 Authentication

KARMA:

One useful extension implemented in previous years is authentication. To implement this extension, the server can accept three additional command line parameters, which constitute the read, write, and administrator passwords for the server.

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

For example, the following command would launch the server on port 8080 with read, write, and admin passwords of `bilbo`, `frodo`, and `gandalf`, respectively:

```
java -jar critterworld.jar 8080 bilbo frodo gandalf
```

If launched in this mode, an additional endpoint is exposed to clients. Clients attempt to login by sending a request to the `POST /login` endpoint with a body of the form:

```
{
  "level": [level],
  "password": [password]
}
```

Here, `[level]` can be either the string `"read"`, for reading general world state (excluding critter programs), `"write"`, for changing the world state and reading back programs of critters that this user created, or `"admin"`, for creating new worlds and having control over all critters. The `[password]` is the password string.

If the password is invalid, the endpoint should respond with status `401 Unauthorized`.

If it matches the password from the command line, the server responds with a status `200` and a body of the following form:

```
{
  "session_id": [session_id_num]
}
```

The `[session_id_num]` is an integer, such as `123`. While you can generate these sequentially, it may provide better security to add some randomization (otherwise session ID 0 is always a valid user and is likely an admin).

This serves as the user's authentication token. The `session_id` must be included as a query parameter with key `session_id` for all subsequent requests this client makes to the server. If it is not present, or invalid, or does not grant the permissions necessary for a given endpoint, all endpoints should return status code `401`.

Note that if you implement this extension, your server must still support the original unauthenticated mode required by the spec if no additional command line arguments are given.

13.2 Incremental updates

HARM: 

Another extension required 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` query 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, some version control systems are implemented this way. For example, Subversion (an older version control software) stores the entire content of a file when the file is committed. Previous versions of the file are then erased and only the changes made are left in a log. Subversion 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.

14 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:

- 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.
- Solving the written problems.
- Preparing your presentation for the course staff

15 Tips and tricks

You have been working hard on your project, and with the end of the semester rapidly approaching, it will be tempting to put this assignment off until closer to the due date. 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.

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.

Since your group members are likely already working on separate computers, one of your computers can be the server and the other, the client. As long as both members are on the same wifi network, this is relatively easy to setup. 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 run `ifconfig` in your terminal on macOS or Linux, or `ipconfig` in PowerShell on Windows. Look for a field titled `inet` or `IPv4 Address` followed by an IP address. If the server is then run on `[port]`, the client can connect to this computer's server by setting the URL to `http://[ip]:[port]`.

If your computers are on different networks, setup is significantly more complicated. You might need to expose 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. Many major cloud providers offer free tiers for students, including Amazon Web Services, Microsoft Azure, and Google Cloud Platform.

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.

You are not required to work on this project during Thanksgiving break, but some teams choose to use the break to get ahead. If you choose to do this, note that you and your group may not see each other for multiple few days. This is where Git—with useful, detailed commit messages—and **lots of communication** will be essential.

16 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.
- 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.
- `a6.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.