Warm-up Assignment 4

CS 4500 Software Development

Due: Friday, February 5, 9pm

Submission:

1. At the top-level of Khoury Github, create a directory `A4` with the following:
   a) a directory `src/` containing the source code for the program `a4` from Task 1
   b) the file `traveller-integration-report.md` from Task 2
   c) **optionally**: a PDF `new-project-programming-language.pdf` – if you are switching your programming language for the project
2. Create a new release on GitHub, tagged `a4` and add a ZIP file `a4-exec.zip` as the "binary" containing the `a4` executable and any auxiliary files required by the executable. Details on how to do that are here.
3. Download the *Source code (zip)* from the release and submit it via Handins. If you are storing additional (binary) assets (e.g., compiled libraries, JAR files) in your Github repo, the ZIP file might be too large for submission on Handins. In that case, you will need to remove these third-party files from the ZIP manually before submitting.

Task 1

Implement *a client program* for the TCP Traveller server according to the protocol specification below.

After starting up and connecting to the server via TCP, the client enters an interactive loop, reading JSON requests for the Traveller service from STDIN, processing them and passing the corresponding TCP requests to the server, and rendering responses to STDOUT. The client accepts well-formed JSON requests following Task 3 in Warm-up 3. An interactive session is ended when the user sends a `^D` (Ctrl-D) to STDIN.

The client, `a4`, should take the following arguments, in this order:

1. the IP address of the server,[1]
2. the destination port, and

[1] It does not have to accept a hostname. However, we will be pleased if it does.

3. the user's name.

Example call:

```
$ ./a4 127.0.0.1 8080 ferd
```

If the name is missing, use `Glorifrir Flintshoulder` as default. If the port is missing, use `8000` as the default. If the IP address is also missing, use `127.0.0.1` (localhost). That is, if `a4` is called without arguments, it should connect to `127.0.0.1` on port `8000` and use `Glorifrir Flintshoulder` as the user's name. If only one argument is given, it is the IP address. If two, they are the address and port.

For this task, you only need to implement a client that follows the given protocol. To test your client, you can implement a *mock server*. The Wikipedia page on Mock Objects is a good starting point on this concept. Alternatively, you can simulate a server using `netcat` in *listen mode* and manually entering server responses or preparing a script. See `man netcat` and, e.g., https://linuxize.com/post/netcat-nc-command-with-examples/#creating-a-simple-chat-server.

Rationale

The idea with this task is to further practice implementing to a specification. Unless you decide to write an example server yourself, you won't have an actual server implementation to test against. However, the interactions with the server are few and relatively simple. As such, it should be fairly straightforward to write a script exercising a particular scenario.

Task 2

Within 24 hours of this assignment being published, you will receive an implementation of your specification for a `Traveller` server module (aka, back-end) in the directory `A4/inbox` of your master branch. Alternatively, there might be a memo explaining why the specification could not be implemented.

If you received an implementation, write a short (1-2 pages) memo addressing the following questions:

1. How well did the other team implement your specification? Did they follow it truthfully? If they deviated from it, was it well justified?
2. Were you or would you be able to integrate the received implementation with your client module from Task 3 of Warm-up 3? What was the actual or estimated effort required?

- The implementation might not be in the language you requested. In that case, you think about whether you would be able to integrate the module through a foreign function interface or a similar mechanism. Note, *your* language does not actually have to support FFI – just assume you have a mechanism for calling foreign functions and interpreting foreign values.

3. Based on the artifact you received and the above two questions, how could you improve your specification to make it more amenable for implementation as you intended?

If you received an explanation of why the specification could not be implemented, or why it is incomplete, instead of answering 1 and 2 above, write a reply to the explanation and include an answer to 3.

Optional: Switching Languages for the Semester Project

Complete this task if and only if you wish to switch your chosen programming language.

As this is the last warm-up assignment, you now have the option to choose a different language to use for the remainder of the semester.

1. Confirm (for yourself) that you can comfortably use your language to

(a) process command-line arguments;
(b) work with STDIN and STDOUT;
(c) work with TCP sockets: create and listen for connections on a particular port, as well as connect to a specified IP address and port
(d) write, manage and run unit tests
(e) read, parse, and write JSON

The above points form a minimum, but not an exhaustive list of concepts you will acquire on the side for software system construction. Over the course of the semester, your chosen language will have to support other essential concepts from software systems building.

2. Write a memo containing two paragraphs: one that explains why you are abandoning your originally chosen language, and another explaining how you have checked that you are familiar with the above concepts in your new language.

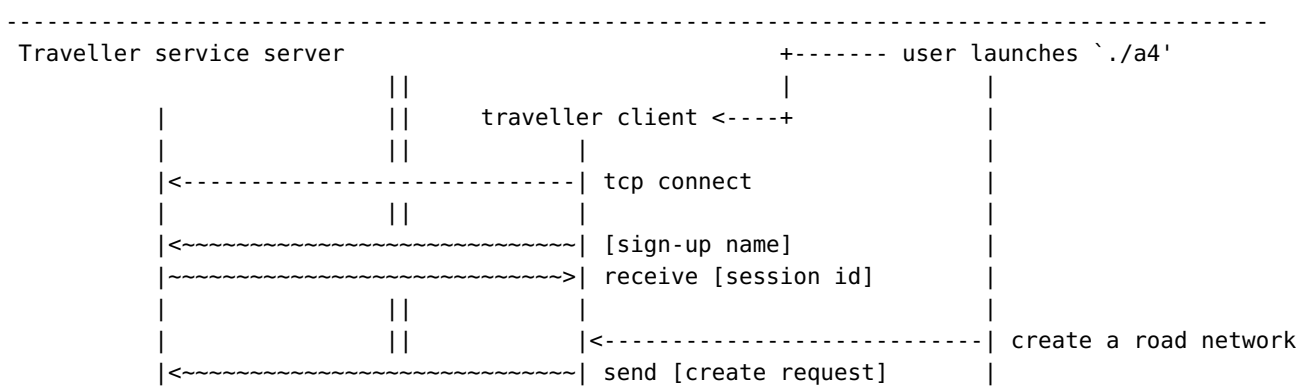
Protocol for Task 1

Protocol for a Traveller Server-Client Interaction

The first three sections explain the interaction arrangements between the user, the client program (`a4`), and the `server` component. The remaining sections specify the format of the messages.

In the protocol specification, the square brackets `[ ]` enclose references to message shapes explained below.

Start Up Steps

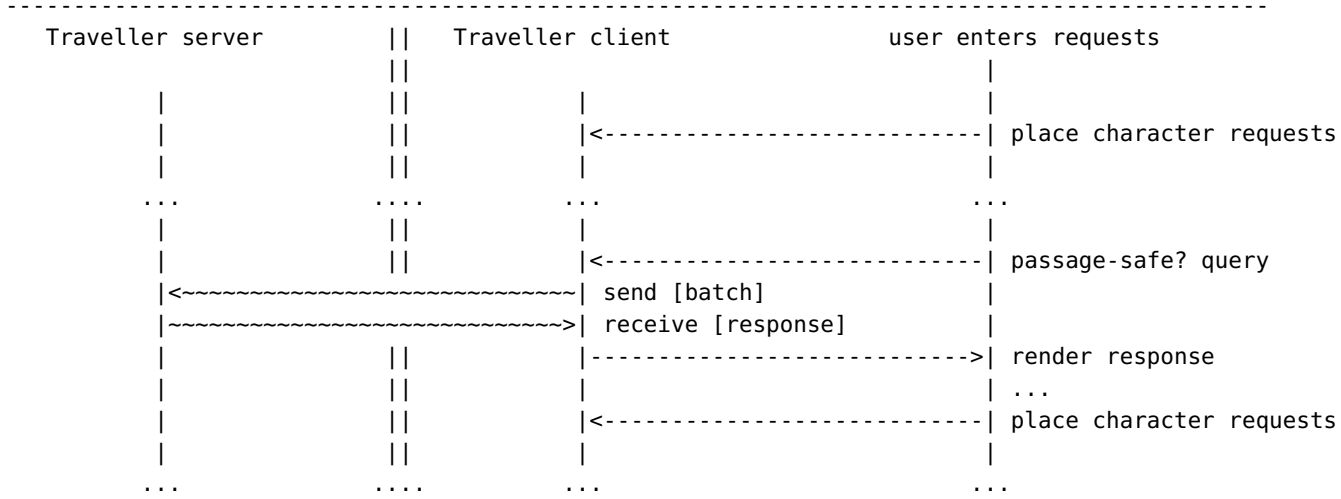


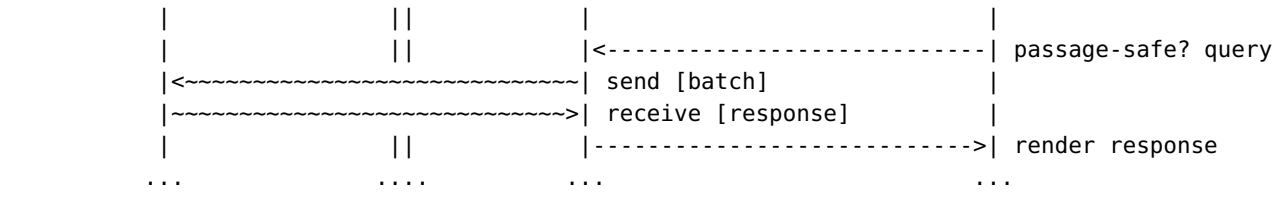
The client program should be prepared to consume up to three arguments:

- the TCP address of the server; default: `127.0.0.1`
- the port number at the server; default: `8000`
- the name of the user; default: `Glorifrir Flintshoulder`

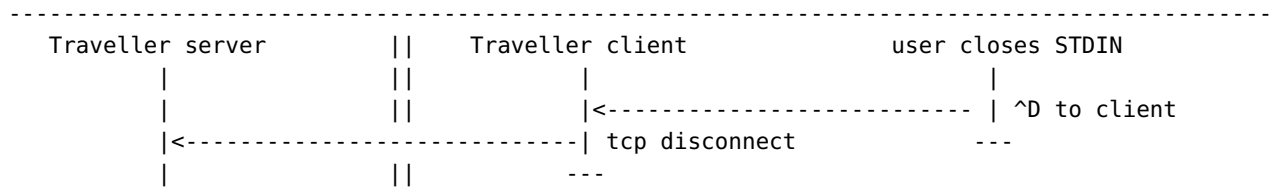
Processing Phase

During this phase the client can send multiple *batch requests*. A batch request consists of a sequence of character placements, terminated by a single query about the safe passage of a character.





Shut Down Steps



User Requests

The user interacts with the client `a4` on STDIN using the exact same JSON commands as specified in Warm-up 3.

TCP Messages

Note the difference between the road network creation JSON request and the protocol message passed through TCP.

- **Sign-up name**

  ```
  String
  ```

  The user uses this name to identify themselves.

- **Session id**

  ```
  String
  ```

  Used for distinguishing connections.

- **Create request**

  ```
  { "towns" : [ String, String, ...],
    "roads" : [ {"from" : String, "to" : String }, ... ] }
  ```

  Sets up a network of towns with roads between them.

- **Batch request**

```
{ "characters" : [ { "name" : String, "town" : String }, ... ],
  "query" : { "character" : String, "destination" : String } }
```

  A batch adds characters to the town network and asks whether a character can safely travel to the given destination town.

- **Response**

```
{ "invalid" : [ { "name" : String, "town" : String }, ... ],
  "response" :  Boolean }
```

  The response to a batch request contains a (possibly) empty list of character placements deemed invalid by the server (see below) and the Boolean response to the query.

From the perspective of the client program, all JSON values that match the above format are well-formed and valid, and can be sent to the server. If the user enters JSON that does not represent a well-formed request, the client program says

```
{ "error" : "not a request",
  "object" : <JSON> }
```

where **<JSON>** stands for the JSON the user entered.

From the perspective of `server`, validity requires the satisfaction of additional constraints:

1. A *Create request* is only valid if each **"from"** and **"to"** element is in the list of towns (the `"towns"` key).
2. A *Batch request* is only valid if
   a) each town has been "created" in an earlier *Create request* and
   b) the character being queried has been placed in a town by the same batch request.

The server will shut down the connection if:

- a batch request is ill-formed,
- the create request is invalid, or
- the query in a batch is invalid.

Any character placement request which is well-formed but invalid gets sent back as a part of the response to a batch.

The client program `a4` renders responses as quasi-English JSON for the user as follows:

- `["the server will call me", String]`
- `["invalid placement", { "name" : String, "town" : String } ]`
- `["the response for", { "character" : String, "destination" : String} , "is", Boolean]`