Assignment 2

Start Assignment

Due No due date **Points** 100 **Submitting** a file upload **File types** pdf

Handin Dates

- **25th of August** at 5:00 pm - Submit a **design sketch (https://myuni.adelaide.edu.au/courses/85272/assignments/344547)** via Canvas (PDF)
- **16th of September** at 5:00 pm - Submit draft revision 1 of your assignment 2. **No grace period allowed.**
- **6th of October** at 5:00 pm - Submit the final version of your assignment 2. **No grace period allowed.**

Design Sketch

The design sketch is a rough architecture of your system for us to be able to provide feedback on early. You may want to consider use cases of your system and the flow of information through it in the sketch, or simply the components you have thought of and where they sit in the system.

Hints:

- Functional analysis is good
- A component view (even if it's extremely coarse: clients, aggregation server, content servers) is required
- Multi-threaded interactions are a good place to focus some design effort. Show how you ensure that your thread interactions are safe (no races or unsafe mutations) and live (no deadlocks).
- Explain how many server replicas you need and why
- UML is a good way of expressing software designs, but it is not mandated.
- It would be useful to know how you will test each part
- Diagrams are awesome

Note: Assignments with no design file will receive a mark of zero.

Preview

We strongly advise that you submit a draft revision/preview of your completed assignment 2 so that we can provide you with feedback.

You will receive feedback within 1 week. The feedback will be detailed but carries no marks. You are given the opportunity to revise and change your work based on the feedback for the final submission so you use it while you can.

Final revision

If you received feedback in the last submission, please add a PDF (**Changes.pdf**) in your final version of submission that includes a discussion of the feedback received and what changes you decided to make and why.

Using Version Control

Working in your repository

As you work on your code you will be adding and committing files to your repository. Git documentation explains and has examples on performing these actions.

It is strongly advised that you:

- Commit **regularly**
- Use **meaningful** commit messages
- Develop your tests **incrementally**

Assignment Submission

You are allowed to commit as many times as you like.

On submission there will be not assigned marks.
Keep an eye on the forums for announcements regarding marks.

Assignment Description

Objective

To gain an understanding of what is required to build a client/server system, you are to build a system that aggregates and distributes weather data in JSON format using a RESTful API.

Introduction

A RESTful API is an interface that computer systems use to exchange information securely over the Internet. Most business applications have to communicate with other internal and third-party applications to perform various tasks. A RESTful APIs support this information exchange because they follow secure, reliable, and efficient software communication standards.

The application programming interface (API) defines the rules that you must follow to communicate with other software systems. Developers expose or create APIs so that other applications can communicate with their applications programmatically. Representational State Transfer (REST) is a software architecture that imposes conditions on how an API should work. REST was initially created as a guideline to manage communication on a complex network like the internet. You can use REST-based architecture to support high-performing and reliable communication at scale.

Two important principles of the REST architectural style are:

- **Uniform interface**: it indicates that the server transfers information in a standard format. The formatted resource is called a representation in REST. This format can be different from the internal representation of the resource on the server application. In this assignment, the format used is the JSON standard.
- **Statelessness**: refers to a communication method in which the server completes every client request independently of all previous requests. Clients can request resources in any order, and every request is stateless or isolated from other requests.

The basic function of a RESTful API is the same as browsing the internet. The client contacts the server by using the API when it requires a resource. API developers explain how the client should use the REST API in the server application API documentation.

It is common now to use the JSON standard and existing HTTP mechanisms to send messages. For this assignment you can use socket-based communication between client and server and do not need to use the Java RMI mechanism to support it - as you would expect as you don't have to use an RMI client to access a web page! So you need to read the input data and convert it into JSON format and then send it to a server. The server will check it and then distribute data to every client who connects and asks for it. When you want to change the data in the server, you overwrite the existing file, which makes the update operation *idempotent* (you can do it as many times as you like and get the same result). The real test of your system will be that you can accept PUT and GET requests from other students on your server and your clients can talk to them. However, don't share the code with the other students :)

JSON standard

JavaScript Object Notation (JSON) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values). It is a common data format with diverse uses in electronic data interchange, including that of web applications with servers.

The following example shows a possible JSON representation describing current weather information.

```
{
    "id" : "IDS60901",
    "name" : "Adelaide (West Terrace /  ngayirdapira)",
    "state" : "SA",
    "time_zone" : "CST",
    "lat": -34.9,
    "lon": 138.6,
    "local_date_time": "15/04:00pm",
    "local_date_time_full": "20230715160000",
    "air_temp": 13.3,
    "apparent_t": 9.5,
    "cloud": "Partly cloudy",
    "dewpt": 5.7,
    "press": 1023.9,
    "rel_hum": 60,
    "wind_dir": "S",
    "wind_spd_kmh": 15,
```

```
    "wind_spd_kt": 8
}
```

The server, once configured, will serve out this JSON formatted file to any client that requests it over HTTP. Usually, this would be part of a web-client but, in this case, you will be writing the aggregation server, the content servers and the read clients. The content server will PUT content on the server, while the read client will GET content from the server.

Elements

The main elements of this assignment are:

- An **aggregation server** that responds to client requests for weather data and also accepts new weather updates from content servers. The aggregation server will store weather information persistently, only removing it when the content server who provided it is no longer in contact, or when the weather data is too old (e.g. not one of the most recent 20 updates).
- A **client** that makes an HTTP GET request to the server and then displays the weather data.
- A **content server** that makes an HTTP PUT request to the server and then uploads new weather data to the server, replacing the old one. This information is assembled into JSON after being read from a file on the content server's local filesystem.

All code elements will be written in the Java programming language. Your clients are expected to have a thorough failure handling mechanism where they behave predictably in the face of failure, maintain consistency, are not prone to race conditions and recover reliably and predictably.

Summary of this assignment

In this assignment, you will build the aggregation system described below, including a failure management system to deal with as many of the possible failure modes that you can think of for this problem. This obviously includes client, server and network failure, but now you must deal with the following additional constraints (come back to these constraints after you read the description below):

1. Multiple clients may attempt to GET simultaneously and are required to GET the aggregated feed that is correct for the Lamport clock adjusted time if interleaved with any PUTs. Hence, if A PUT, a GET, and another PUT arrive in that sequence then the first PUT must be applied and the content server advised, then the GET returns the updated feed to the client then the next PUT is applied. In each case, the participants will be guaranteed that this order is maintained if they are using Lamport clocks.
2. Multiple content servers may attempt to simultaneously PUT. This must be serialised and the order maintained by Lamport clock timestamp.
3. Your aggregation server will expire and remove any content from a content server that it has not communicated within the last 30 seconds. You may choose the mechanism for this but you must consider efficiency and scale.
4. All elements in your assignment must be capable of implementing Lamport clocks, for synchronization and coordination purposes.

Your Aggregation Server

To keep things simple, we will assume that there is one file in your filesystem which contains a list of entries and where are they come from. It does not need to be an JSON format specifically, but it must be able to convert to a standard JSON file when the client sends a GET request. However, this file must survive the server crashing and re-starting, including recovering if the file was being updated when the server crashed! Your server should restore it as was before re-starting or a crash. You should, therefore, be thinking about the PUT as a request to handle the information passed in, possibly to an intermediate storage format, rather than just as overwriting a file. This reflects the subtle nature of PUT - it is not just a file write request! You should check the feed file provided from a PUT request to ensure that it is valid. The file details that you can expect are detailed in the Content Server specification.

All the entities in your system must be capable of maintaining a Lamport clock.

The first time weather data is received and the storage file is created, you should return status 201 - HTTP_CREATED. If later uploads (updates) are successful, you should return status 200. (This means, if a Content Server first connects to the Aggregation Server, then return 201 as succeed code, then before the content server lost connection, all other succeed response should use 200). Any request other than GET or PUT should return status 400 (note: this is not standard but to simplify your task). Sending no content to the server should cause a 204 status code to be returned. Finally, if the JSON data does not make sense (incorrect JSON) you may return status code 500 - Internal server error.

Your server will, by default, start on port 4567 but will accept a single command line argument that gives the starting port number. Your server's main method will reside in a file called `AggregationServer.java`.

Your server is designed to stay current and will remove any items in the JSON that have come from content servers which it has not communicated with for 30 seconds. How you do this is up to you but please be efficient!

Your GET client

Your GET client will start up, read the command line to find the server name and port number (in URL format) and optionally a station ID; it will then send a GET request for the weather data. This data will then be stripped of JSON formatting and displayed, one line at a time, with the attribute and its value. Your GET client's main method will reside in a file called `GETClient.java`. Possible formats for the server name and port number include "http://servername.domain.domain:portnumber", "http://servername:portnumber" (with implicit domain information) and "servername:portnumber" (with implicit domain and protocol information).

You should display the output so that it is easy to read but you do not need to provide active hyperlinks. You should also make this client failure-tolerant and, obviously, you will have to make your client capable of maintaining a Lamport clock.

Your Content Server

Your content server will start up, reading two parameters from the command line, where the first is the server name and port number (as for GET) and the second is the location of a file in the file system local to the Content Server (It is expected that this file located in your project folder). The file will contain a number of fields that are to be assembled into JSON format and then uploaded to the server. You may assume that all fields are text and that there will be no embedded HTML or XHMTL. The list of JSON elements that you need to support are shown in the example above.

Input file format

To make parsing easier, you may assume that input files will follow this format:

```
id:IDS60901
name:Adelaide (West Terrace /  ngayirdapira)
state: SA
time_zone:CST
lat:-34.9
lon:138.6
local_date_time:15/04:00pm
local_date_time_full:20230715160000
air_temp:13.3
apparent_t:9.5
cloud:Partly cloudy
dewpt:5.7
press:1023.9
rel_hum:60
wind_dir:S
wind_spd_kmh:15
wind_spd_kt:8
```

An entry is terminated by either another entry keyword, or by the end of file, which also terminates the feed. You may reject any feed or entry with no id as being in error. You may ignore any markup in a text field and just print it as is.

PUT message format

Your PUT message should take the format:

```
PUT /weather.json HTTP/1.1
User-Agent: ATOMClient/1/0
Content-Type: (You should work this one out)
Content-Length: (And this one too)

{
    "id" : "IDS60901",
...
(data)
...
    "wind_spd_kt": 8
}
```

Your content server will need to confirm that it has received the correct acknowledgment from the server and then check to make sure that the information is in the feed as it was expecting. It must also support Lamport clocks.

Some basic suggestions

The following would be a good approach to solving this problem:

- Think about how you will test this and how you are going to build each piece. What are the individual steps?
- Write a simple version of your servers and client to make sure that you can communicate between them.
- Use smaller JSON data for testing parts of your system and read all of the relevant spec sections carefully!
- There are many default Java JSON parsers out there, learn how to use them rather than write your own. Both options are acceptable, but we have found that it does save time to use existing ones (if not for anything, you have a ton of tutorials out there!)
- We strongly recommend that you implement this assignment using Sockets rather than HttpServer
- Try modularising your code; for example, JSON parser functions are required in all places, so it is better to have all those functions in one class, then reused in other places.

Notes on Lamport Clocks

Please note that you will have to implement Lamport clocks and the update mechanisms in your entire system. This implies that each entity will keep a local Lamport clock and that this clock will get updated as the entity communicates with other entities or processes events. It is up to you to determine which events (such as send, receive or processing) the entity will consider in the Lamport clock update (for example, a System.out.println might not be interesting). This granularity will influence the performance of your implementation. The local Lamport clocks will need to be sent through to other entities with every message/request (like in the request header) - you are responsible for ensuring that this tagging occurs and for the local update of Lamport clocks once messages/requests are received. Towards this, follow the algorithm discussed in class and/or in the Lamport clocks paper accessible from the forum. As part of this requirement, we are aware that your method for embedding Lamport clock information in your communications may mean that you lose interoperability with other clients and servers. This is an acceptable outcome for this assignment but, usually, we would take a standards-based approach to ensure that we maintain interoperability.

And lastly,

START EARLY!

Don't get caught out at the last minute trying to do the entire assignment at once - it is easy to misjudge the complexity and hours required for this assignment.

Contact the course coordinator, lectures or tutors if you need help getting started.

You are encouraged to post questions on the forums.

Assessment

The allocation of marks for this assignment is as follows:

- 60% - Software solution
- 40% - Automated testing

The assessment of your software solution will be allocated as follows:

- 10% - Code quality, following the checklist in Appendix A (below)
- 20% - Architecture design decisions
- 30% - Support for basic functionality, following the checklist in Appendix B (below)
- 40% - Support for full functionality and quality of design, following the checklist in Appendix B (below)

The assessment of your testing will be allocated as follows:

- The range of test cases considered
  rather than focus on the number of tests, are you identifying the most important test cases with a good spread across possible cases?
- The clarity of your test cases
  your test harness should be verbose enough to ensure that we understand both what you have tested and the outcome of the tests

Your testing architecture, ideally captured in a testing document should become an important part of your development process!

Final Words

Don't forget to commit your work frequently and to submit before the due date!

Appendix A

Code Quality Checklist

Do

- Write comments above the header of each of your methods, describing what the method is doing, what are the inputs and expected outputs
- describe in the comments any special cases
- create modular code, following cohesion and coupling principles

Don't

- use magic numbers
- use comments as structural elements
- mis-spell your comments
- use incomprehensible variable names
- have methods longer than 80 lines
- allow TODO blocks

Appendix B

Assignment 2 Checklist

Basic functionality refers to:

- Text sending works - please send text strings instead of fully formatted JSON (see below for bonus)
- client, Atom server and content server processes start up and communicate
- PUT operation works for one content server
- GET operation works for many read clients
- Aggregation server expunging expired data works (30s)
- Retry on errors (server not available etc) works

Full functionality refers to:

- Lamport clocks are implemented
- All error codes are implemented
- Content servers are replicated and fault tolerant

Bonus functionality (10 points):

- JSON parsing using your own code --> remember that using an existing parser is ok (but no bonus)!