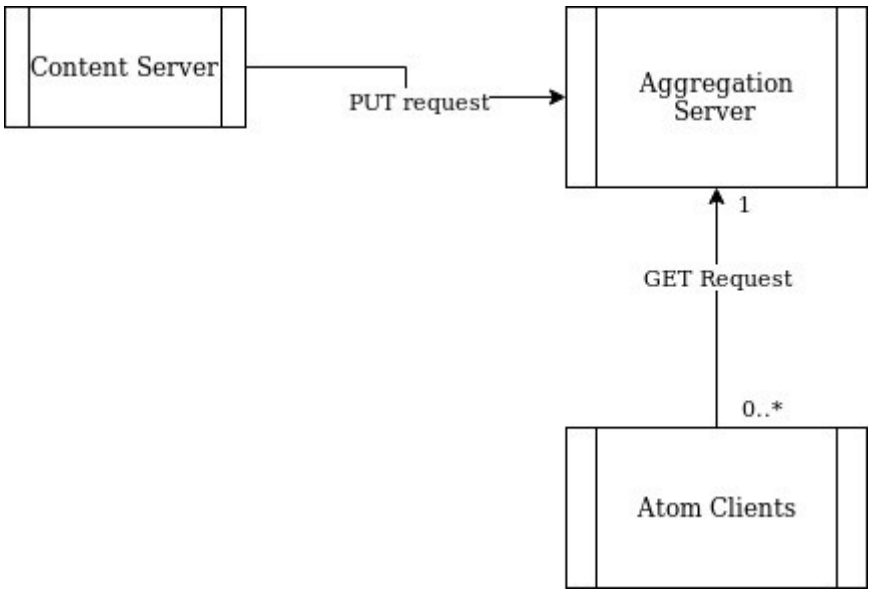
Assignment 2: ATOM syndication Design

By Huy Gia Do Vu (a1742494)

The assignment spec alludes to three main components:

1. Content Server.
2. Aggregation Server.
3. ATOM client.

These components use HTTP requests to send messages to each other in this way:



The assignment spec also provides well-defined interfaces between these components. Thus we can develop each of these components individually without worrying that we will affect the other components. As such, we will structure this design document around these main components.

ATOM client

The Atom Client have three operations:

1. Perform a GET request to the aggregation Server in order to receive the ATOM feed.
2. Print this ATOM feed in human readable format.
3. Maintain a lamport clock synchronized with the Aggregation Server.

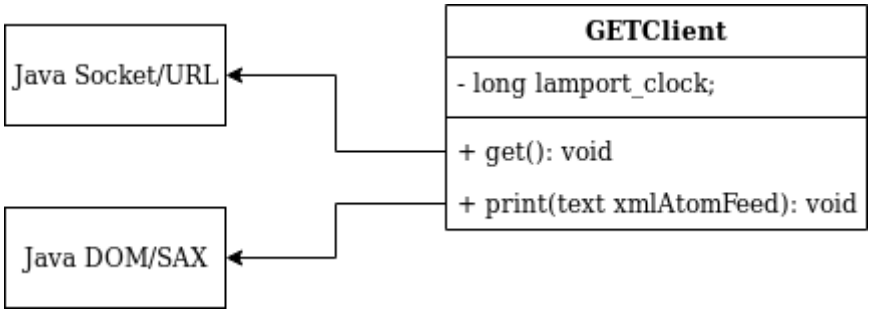
With operations 1 and 2. We have a choice to perform these synchronously or asynchronously. E.g. if we decide on a synchronous solution, our ATOM client can be either performing a GET

request or printing. If we decide on an asynchronous solution, our ATOM client can always be performing GET requests, and create a thread to print. The trade off here is that the asynchronous solution will have more fresh content whereas the synchronous solution will be more simple. Since we don't expect the printing to take that long and it's unlikely that any user expects their ATOM feed to update every couple milliseconds, it's not unreasonable to use the synchronous solution. So we'll go with that.

So let's run through the details of each operation:

1. GET request:
    - We wait for a user to input a url of the aggregation server.
        - We should also let the User know which format our program expects the URL to be in.
        - If we don't get a response, a bad request, or an invalid URL, we should let the user know and let them try again.
        - We can Java Sockets or Java URL for this operation.
    - If the request was successful, we begin printing.
2. Printing:
    - We parse the XML and print out to stdout.
        1. Java has a DOM parser or we can use Java SAX parser.
        2. We print using basic Java print calls.
    - Once finished printing we return to operation 1.
3. Maintain a Lamport Clock.
    - Upon sending a GET request, we embed our local clock in a request header.
    - We assume that the server responds with it's own lamport clock. Wehenever we receive a response we set ours to max(server clock, local clock).
    - We increase our logical clock for every GET request/print operation and for every response we receive.

Thus our client might look like something like this:



Another thing to note is that we may receive duplicate ATOM entries. In which case we should only print out the new entries since that's probably what users want. Currently, since this is a rough design, I haven't come up with a solution but I will work on it.

I can use real ATOM feeds to test this client independently of the Content Server and Aggregation server. I've had a brief look on most prominent news websites but they mostly use RSS. Hopefully I can find one. Otherwise, I could set up my own server that serves basic ATOM content (perhaps using Express).

Content Server

The Content Servers have 3 main operations:

1. Read in an Input file (formatted according to assigment spec) and parse into XML format.
2. Send XML to Aggregation Server via PUT request.
3. Maintain a lamport clock.

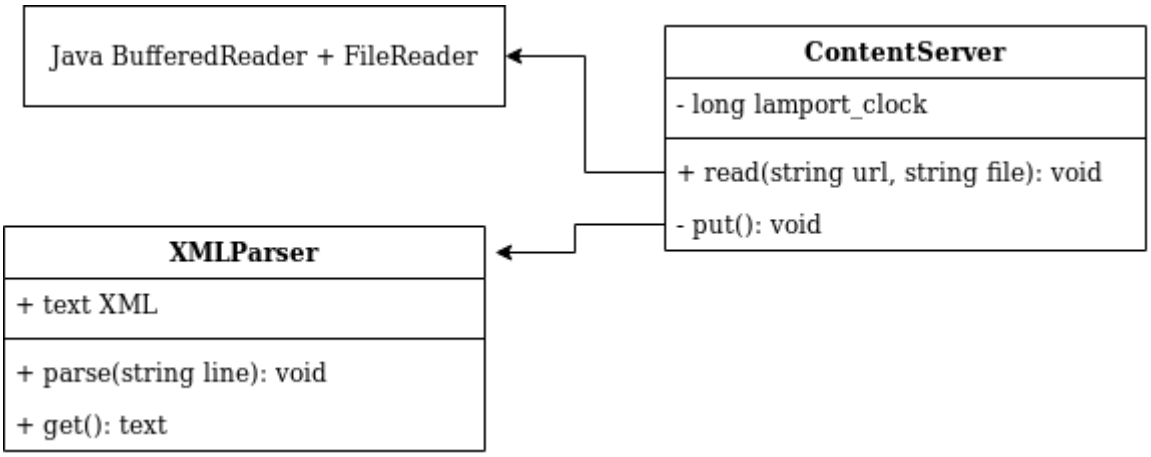
So let's run through the details of each operation:

1. Read in an Input file then parse into XML format.
    - We can use Java BufferedReader and FileReader to read in the file line by line.
    - To parse into XML format, we might use a custom class that takes in a string and parses into XML and also buffers the result.
2. Send XML to Aggregation Server via PUT request
    - We can use Java Sockets / Java URL for this operation.
    - When we get the response, we should print it the status and the body. And then the job's done.
3. Maintain a lamport clock.
    - When we receive a request, we should set lamport_clock = max(request_clock, lamport_clock)
    - We should increment the lamport clock when we Read and Parse into XML format, send a request and receive a request.

Some issues I'm still figuring out:

- The assignment spec implies that the content server should simply run and then terminate. However, is that going to affect synchronization? If we initialize each lamport_clock to zero, then every request to the Aggregation Server is going to be at a clock value of 2 (Read in and then request). Which means we open ourseslves up to a starvation issue. If content servers kept issuing PUT requests, the Aggregation Server is going to prioritize our PUT requests over GET requests (since these will be synchronized to the Aggregation Server after some time) which means our GETClient could be starved of requests. Perhaps the first thing our Content Server could do is send a request for the lamport clock value. But I'm not sure If there's side effects to that. So I'll continue to work on that.

Thus our Content Server might look like something like this:



We can use online XML interpreters or Java's own XML interpreter to test our XML parser to see if our implementation works well. As for the ContentServer, we can set up our own server to see how our PUT requests are looking like and if they conform to HTTP standards. E.g. express.js

Aggregation Server

This is where the most complexity will occur. The Aggregation Server has many operations:

1. Receive multiple requests.
    - We can receive requests using java Sockets.
2. Maintain a lamport clock.
    - Embed own lamport in any request sent.
    - Increment lamport clock on every request, response.
    - Upon receipt of a request, set lamport_clock = max(lamport_clock, request_clock)
3. We must store XML from PUT requests.
    - We can do this using PrintWriter and FileWriter.
    - We should send the correct responses according to the assignment spec.
4. We must send XML when receiving a GET request.
    - We should send the correct responses according to HTTP and the assignment spec.
    - Any requests that are not PUT or GET requests will be sent 404.
5. Maintain logical ordering.
    - Every request should be handled in order of it's respective lamport_clock.
    - If two requests have the same lamport_clock, then we need to define a way to choose between the two. (I'm assuming process ID won't work here since it's possible for two processes to have the same id. Perhaps IP and Port?)
6. Handle any Thread failing.

- If any Thread fails, throws an exception, then we should ensure any resources given to the thread are reset. E.g. any locks, any buffers.

Mechanisms:

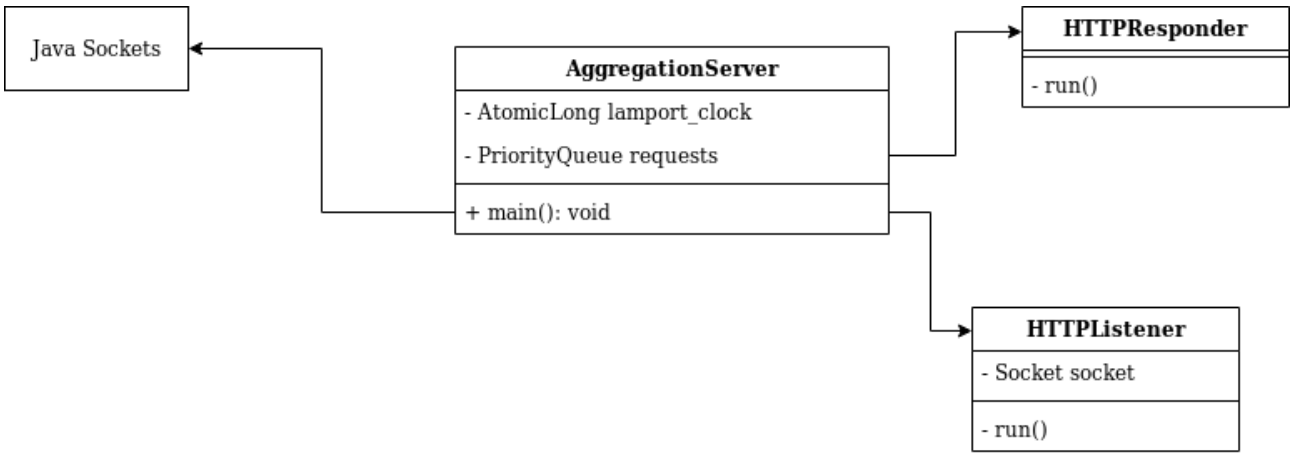
A big design decision will be on how to implement this class. We will run through some approaches and disucss the tradeoffs for each.

Suppose for every request, we create a Thread to handle that request. These threads will handle GET and PUT requests. We can define our lamport clock as an AtomicInteger/AtomicLong which makes it thread safe and allows us to avoid using a lock. However, the most difficult part will be to maintain logical ordering. If we want to maintain logical ordering then we need a mechanism to coordinate which request goes first. We can perform this perhaps by creating a prority queue of a logical clocks, IP, port, and mutext tuple. We acquire the mutext when a thread is created and release it only when it's that threads turn. Thus we need a Thread that manages this. And we need to ensure no two threads access the priority queue at a time. This approach is a bit messy. It might work, but there's a lot of room for error and a lot of effort will be required to ensure that it's live and safe.

One fact we can leverage is the fact that each request should only be performed sequentially. Suppose we only have two threads, one thread to listen to request and place connections in a priority queue, and another thread to process theses requests, then we reduce a lot of complexity. However, I'm unsure of the side effects of leaving open connections without any thread to listen to them.

It might be useful however, to combine the two approaches. Requests don't have to be responded to by the same thread. Suppose we spawn a thread for every request only to listen to the request. If the request is valid, it puts it onto a shared priority queue. However, a separate thread is responsible for responding to the request. This way, we only have to synchronize access to the priority queue and we achieve logical ordering. However, there is one flaw to this approach. Since there's only one thread responding but many threads listening, there's a big bottle-neck. So if our server is given too many requests, we might get flooded with rerquests. I'll continue to work towards a better solution.

If we run with this solution, our Agregation server might look like this:



We can test our server by using multple REST clients that send GET and PUT requests multiple times concurrently. e.g. Insomnia.

That concludes our design.

What's Next?

- Find a better solution to Aggregation Server.
  - Preferably without bottle neck.
  - Currently, our solution also doesn't scale. If we continue to get requests we will continue to create new threads which will eventually crash the server. So perhaps a solution with Thread Pools?
- Come up with methods of automated testing for Aggregation Server. (I believe Insomnia has a unit testing function?)
- Come up with methods of automated testing for Client and Content Server.

Any feedback is great.