

Assignment 2

[Start Assignment](#)

Due 8 Oct by 16:59 **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/75043/assignments/291489>) via Canvas (PDF)
- ~~16th of September~~ **20th of September** at 5:00 pm - Submit draft revision 1 of your assignment 2 via [websubmission Links to an external site.](#) (https://cs.adelaide.edu.au/services/websubmission/?sub_assign=assignment2-draft) --- **No grace period allowed.**
- ~~6th of October~~, **8th of October** at 5:00 pm --- ~~Grace period allowed~~ **Two days grace allowed** - Submit the final version of your assignment 2 via [websubmission Links to an external site.](#) (https://cs.adelaide.edu.au/services/websubmission/?sub_assign=assignment2)

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, Atom 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.

Setting Up Version Control

Getting to know Subversion

This course uses **Subversion (svn)**. Svn is a powerful version control system to help maintain a coherent copy of a project that can be worked on from multiple locations. We will also use svn as the handin mechanism throughout this course. Click [here \(http://www.cs.adelaide.edu.au/docs/svn-instr.pdf\)](http://www.cs.adelaide.edu.au/docs/svn-instr.pdf) to learn more.

Creating the assignment directory in your svn repository

Run the following command in terminal.

```
svn mkdir --parents -m "DS assignment 2" https://version-control.adelaide.edu.au/svn/axxxxxxx/YEAR/s2/ds/assignment2
```

Replace `axxxxxxx` with your student ID number.

This command will create an empty directory named `YEAR/s2/ds/assignment2` in your **svn** repository.

You can access your new assignment directory via **`https://version-control.adelaide.edu.au/svn/axxxxxxx/2021/s2/ds/assignment2`**

Checking out a working version of your assignment

If you are working at home on your personal computer, you can checkout your svn repository running the following command in terminal.

```
svn checkout https://version-control.adelaide.edu.au/svn/axxxxxxx/2021/s2/ds/assignment2 ds-YEAR-s2-assignment2
```

`ds-YEAR-s2-assignment2` is an optional argument that specifies the destination path for your repository on your local machine.

Note that you can have more than one copy of your code checked out, you will need to update it to avoid conflicts.

See the [svn documentation \(http://www.cs.adelaide.edu.au/docs/svn-instr.pdf\)](http://www.cs.adelaide.edu.au/docs/svn-instr.pdf) for details on how this can be done. However, for now, we will assume you have just the one working copy.

Working in your repository

As you work on your code you will be adding and committing files to your repository. The Subversion 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

Use the Computer Science [Web Submission System](https://cs.adelaide.edu.au/services/websubmission/) (<https://cs.adelaide.edu.au/services/websubmission/>) system to submit assignments.

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

The Web Submission System will only perform basic checks for any required files.

On submission there will be not assigned marks.


The assignment will be marked by a teacher who will upload the marks into the Web Submission System. 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, by building a simple system that aggregates and distributes ATOM feeds.


Introduction

Information management and tracking becomes more difficult as the number of things to track increases. For most users, the number of web pages that they wish to keep track of is quite large and, if they had to remember to check everything manually, it's easy to forget a webpage or two when you're tired or busy. Enter *syndication*, a mechanism by which a website can publish summaries as a feed that you can sign up to, so that you can be notified when something new has happened and then, if it interests you, go and look at it. Initial efforts in the world of syndication included the development of the RSS family of protocols but these are, effectively, not standardised. The ATOM syndication protocol is a standards-based approach to try and provide a solid basis for syndication. You can see the ATOM RFC [here](http://tools.ietf.org/html/rfc4287)  (<http://tools.ietf.org/html/rfc4287>) although you won't be implementing all of it!

XML-based formats are easy to transport via Hypertext Transport Protocol (HTTP), the workhorse protocol of the Web, and it is increasingly common to work with a standard format for interchange between clients and servers, rather than develop a special protocol for one small group of clients and servers. Where, twenty years ago, we might have used byte-boundary defined patterns in transmitted

data to communicate, it is far more common to use XML-based standards and existing HTTP mechanisms to shunt things around. This is socket-based communication between client and server and does 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! In this prac, you will take data and convert it into ATOM format and then send it to a server. The server will check it and then distribute a limited form of that 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. As always, don't share code.

Syndication Servers

Syndication servers are web servers that serve XML documents which conform to the RSS or ATOM standards. On receipt of an HTTP GET, the server will respond with an XML response like this (from ["Creating an ATOM feed in PHP"](http://www.ibm.com/developerworks/library/x-phpatomfeed/)  [\(http://www.ibm.com/developerworks/library/x-phpatomfeed/\)](http://www.ibm.com/developerworks/library/x-phpatomfeed/)):

```
<?xml version='1.0' encoding='iso-8859-1' ?>
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
  <title>Fishing Reports</title>
  <subtitle>The latest reports from fishinhole.com</subtitle>
  <link href="http://www.fishinhole.com/reports" rel="self"/>
  <updated>2015-07-03T16:19:54-05:00</updated>
  <author>
    <name>NameOfYourBoss</name>
    <email>nameofyourboss@fishinhole.com</email>
  </author>
  <id>tag:fishinhole.com,2008:http://www.fishinhole.com/reports</id>
  <entry>
    <title>Speckled Trout In Old River</title>
    <link type='text/html' href='http://www.fishinhole.com/reports/report.php?id=4' />
    <id>tag:fishinhole.com,2008:http://www.fishinhole.com/reports/report.php?id=4</id>
    <updated>2009-05-03T04:59:00-05:00</updated>
    <author>
      <name>ReelHooked</name>
    </author>
    <summary>Limited out by noon</summary>
  </entry>
  ...
</feed>
```

The server, once configured, will serve out this ATOM XML 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 ATOM server (or aggregation server) that responds to requests for feeds and also accepts feed updates from clients. The aggregation server will store feed information persistently, only

removing it when the content server who provided it is no longer in contact, or when the feed item is not one of the most recent 20.

- A client that makes an HTTP GET request to the server and then displays the feed data, stripped of its XML information.
- A CONTENT SERVER that makes an HTTP PUT request to the server and then uploads a new version of the feed to the server, replacing the old one. This feed information is assembled into ATOM XML 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 prac

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 12 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 they come from. It does not need to be an ATOM format, but it must be able to convert to a standard ATOM 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 your ATOM feed is created, you should return status 201 - HTTP_CREATED. If later uploads are ok, 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 ATOM XML does not make sense 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 feed that have come from content servers which it has not communicated with for 12 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 will send a GET request for the ATOM feed. This feed will then be stripped of XML 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 from the ATOM format that are to be assembled into an ATOM XML feed and then uploaded to the server. You may assume that all fields are text and that there will be no embedded HTML or XHTML. The list of ATOM elements that you need to support are:

- title
- subtitle
- link
- updated

- author
- name
- id
- entry
- summary

Input file format

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

```

title:My example feed
subtitle:for demonstration purposes
link:www.cs.adelaide.edu.au
updated:2015-08-07T18:30:02Z
author:Santa Claus
id:urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6
entry
title:Nick sets assignment
link:www.cs.adelaide.edu.au/users/third/ds/
id:urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a
updated:2015-08-07T18:30:02Z
summary:here is some plain text. Because I'm not completely evil, you can assume that this wil
l always be less than 1000 characters. And, as I've said before, it will always be plain text.
entry
title:second feed entry
link:www.cs.adelaide.edu.au/users/third/ds/14ds2s1
id:urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6b
updated:2015-08-07T18:29:02Z
summary:here's another summary entry which a reader would normally use to work out if they wan
ted to read some more. It's quite handy.
```

Note that the author field only contains a name and that you will have to convert this into a name element inside an author element. 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 title, link or 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 /atom.xml HTTP/1.1
User-Agent: ATOMClient/1/0
Content-Type: (You should work this one out)
Content-Length: (And this one too)

<?xml version='1.0' encoding='iso-8859-1' ?>
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
(And then your file of data)
...
</feed>
```

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 known working ATOM feeds for testing parts of your system and read all of the relevant spec sections carefully!
- There are many default Java XML 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, ATOM Feed parse function is 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! All work must be submitted to the web submission system and you should always resubmit your work after every commit in SVN. We will not be marking work that is not submitted via the Web Submission system.

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 XML (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
- Atom server expired feeds works (12s)
- 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):

- XML parsing works --> remember that using an existing parser is more than ok!