CSCI 5105: Introduction to Distributed Computing

Spring 2018

Instructor: Jon Weissman

Project 3: Simple xFS

Due: April 25 12pm (noon)

1. Overview

In this project, you will implement a simple "serverless" file system based on the classic xFS paper [1] in which peer nodes can share their files directly with other peers. Using this system, a user can store files in their local storage (a specific directory up to you, e.g. `~you/5105/share/machID`) to share with other users. The `machID` is a unique name that refers to a unique peer. At any point in time, a file may be stored at a number of peers. When a node wishes to download a file, it learns about the possible locations of the file (from the server) and makes a decision based on the **latency** and **load** (to be described). This system will also support basic **fault tolerance**: any nodes or the server can go down and then recover. You can use any communication protocol UDP, TCP, RPC, or RMI. In this lab, you will learn about how to implement a simple xFS with failure handling. You may reuse any of the code developed in Project 1, 2 or you can start from scratch.

2. Project Details

The project will be comprised of multiple **nodes** and a single **tracking server**. When a node boots (or recovers), it determines its file list by scanning the specific directory (e.g. /`machID`), and then reports this file list, checksums (you may compute hashes or something similar), and the node endpoint information (e.g. IP and Port) to the tracking server. The xFS operations are below:

1. `Find` will returns the list of nodes which store a file
2. `Download` will download a given file; once downloaded it becomes shareable from that peer
3. `GetLoad` returns the load at a given peer requested from another peer
4. `UpdateList` provides the list of files stored at a given peer to the server
5. `More as needed up to you (e.g. to support fault tolerance)`

For simplicity, the **latency** between all possible peer nodes is stored in a configuration file known to all of the peers. This configuration file should be read at boot-time by the peers. You can choose the latency values in this file in the range [100-5000] ms. In addition, each peer maintains a **load** index, which is the number of concurrent downloads or uploads this peer is currently performing. Thus, the peers are multithreaded. Take care to synchronize the threads as needed.

a) `Find (filename)`

This is called on the server and returns the set of peers that store the file and the file checksum. The file name will be unique globally, e.g. there is only one "foo.txt" allowed. A peer can get a node list from the Tracking Server only when they provide exact same input with a file name, e.g. Find("foo.txt") provides

provides the list of nodes which have “foo.txt”.

b) `Download (filename)`

This is called by a peer on another peer to download the file. The file is then placed in the shared directory. When the file is downloaded, you will emulate the latency delay by first running sleep (latency) on either the sending or receiving peer. The peer who downloaded the file should send the new updated file list to the tracking server. A peer could also choose to read the updated file list periodically (assuming files are added or deleted to shared directory out-of-band). This is not required however.

c) `Peer selection`

It is up to you to devise an algorithm for selecting a peer taking both load and latency into account. You can call `GetLoad` to determine the current load of the peer. You already know the latency from the static configuration file. Tell us how you arrived at this algorithm. How does the average file download time change as the number of peers hosting a file increases? You may want to include a simple graph.

Fault Tolerance

You will handle numerous failures in this lab using soft state. Soft state is the state that can be reconstructed after the failure. How you may handle certain failure modes are unspecified and up to you (*noted in italics*).

1) Downloaded files can be corrupted. This is detected by checksum mismatch (recompute the checkpoint after the file is downloaded). You can artificially test this by deliberately adding noise to the download ed file (i.e. change a byte). You can either re-fetch the file from that peer or try another peer.
2) Tracking server crashes (fail-stop).
   When the server crashes and recovers, think about how it will rejoin the network and repopulate file sharing information from the peers. The server should not store this information persistently, but rather it should recover it from the peers themselves.
   The peers should be blocked waiting for the server to come back up. **Optionally**, they could also choose to serve files to other peers, assuming the peers cached some file location information earlier. This is not required, however.
3) Peer crashes (fail-stop).
   A sender node can be down. In this case, the receiver node can try to download from another node which has the next lowest cost from the list returned by `Find`. *When the peer node recovers, think about how it will rejoin the network and supply file sharing information to the server.* **Optional**: a peer could notify the server of a dead node, though this is not required.

You should decide whether a peer will be failed or retry an operation later when it meets the exceptional situations like below.

1) All available peers are down.
2) Fail to find a file.

What design principles come into play here ?

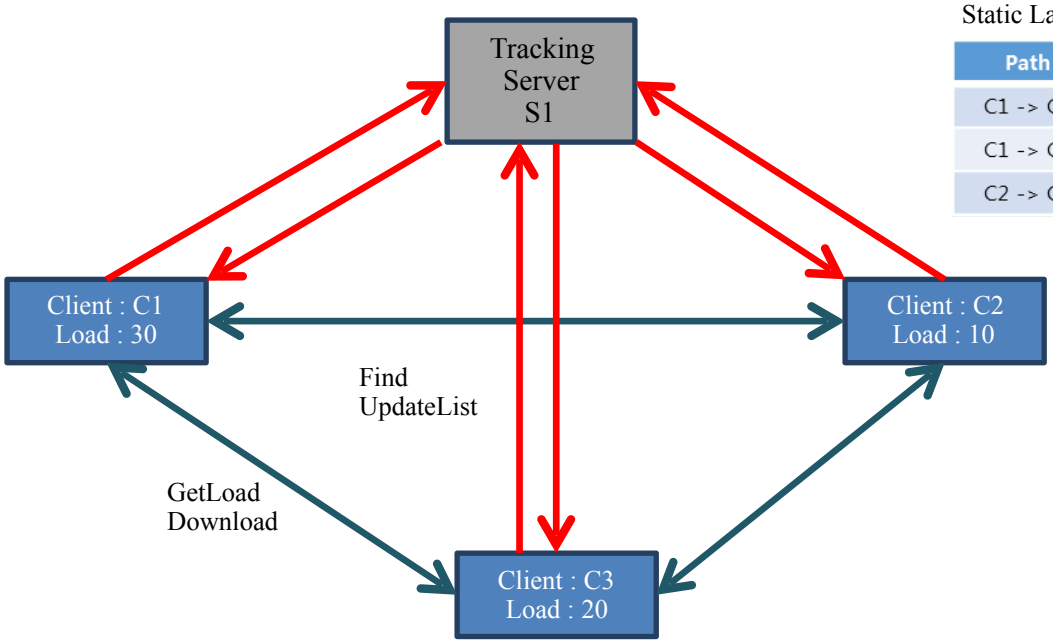
In this project, there is some optional functionality. Though there is no extra credit for the optional implementation.

3. Implementation Details

You may borrow code that you like from Project 1 or 2. Do not use any code found on-line. To make multiple nodes run easily, your nodes may run in a single machine with different port number or within a virtual machine. Note that your nodes are also expected to work well when deployed across different machines.

Static Latency table

Path	(ms)
C1 -> C2	500
C1 -> C3	212
C2 -> C3	330



4. Project Group

All students should work in groups no more than 3.

5. Test Cases

You should also develop your own test cases for all of the components of the architecture, and provide documentation that explains how to run each of your test cases, including the expected results. Also tell us about any possible deadlocks or race conditions. Since it is not easy to run multiple peers and operations, you should consider providing scripts and/or a UI menu which can issue the multiple download and send operations simultaneously. This will allow you (and us!) to test all operations and your algorithm for choosing a peer.

There are many failure modes. Try to catch as many of these as possible. **Finally, you should run your nodes and servers within the CSE firewall – as outside access particularly through UDP and TCP may be blocked.**

6. Deliverables

a. Design document describing each component. Not to exceed 2 pages. Performance graphs and simple analysis.

b. Instructions explaining how to run each component and how to use the service as a whole, including

command line syntax, configuration file particulars, and user input interface.

c. Testing description, including a list of cases attempted (which must include negative cases) and the results.
d. Source code, makefiles and/or a script to start the system, and executables.

Note: a, b, and c should be in a single document file. And please write about how to compile and run your project in detail.

7. Grading

The grade for this assignment will include the following components:

a. 20% - The document you submit – This includes a detailed description of the system design and operation along with the test cases used for the system (must include negative cases)
b. 70% - The functionality and correctness of your own server and clients. This includes your peer selection algorithm and data analysis.
c. 10% - The quality of the source code, in terms of style and in line documentation (exception handling)

8. Resource

[1] Serverless Network File System: http://www.cs.iastate.edu/~cs652/notes/xfs.pdf