CSCI 5105

Introduction to Distributed Systems

Spring - 2020

Design Document

XFS

Date: 05/03/2020

Team:

Soumya Agrawal (agraw184)
Garima Mehta (mehta250)
Bhaargav Sriraman (srira048)

TABLE OF CONTENTS

Project Implementation:

The aim of the project was to build a “serverless” file system based on xFS paper, wherein the peer node can share their files directly with other peers without involving the server. The GitHub repo link: https://github.umn.edu/agraw184/xfs

Client:

The Client-side consists of 4 major files: Client.java, ClientHelper.java, ClientThread.java, ClientToClientResponder.java. All these files help in managing the connection between Client and TrackingServer, and Client to other Clients. Client.java has the functionality for the client to connect with the Server/Client to carry out several functionalities that include, Find(ing) a file, which creates a connection to the Server to get the list of peers that have the file. Download(ing) a file, which connects to the ideal peer using the load[1] and latency algorithm[2]. The client creates a Socket (TCP Connection) using the class SocketConnection.java which also creates ObjectInputStream, ObjectOutputStream for the same socket. ClientHelper.java has helper functions to facilitate the processing of the message sent from a client, the message received by a client, the connection to the server (Sending and receiving messages), the downloading of a file and validating the checksum

We are assuming that there are no updates to a file once it enters a peer, that is there is no update to the contents of the file nor is there deletion of the file.

Tracking Server:

The Server side code consists of TrackingServer.java, ServerHelper.java, ServerHealth.java, ServerToClientResponder.java. The server does not have a lot of functionality apart from managing the list of files each peer has, which will be presented when the client tries to find a file using the ServerToClientResponder class for effective communication between the Server and Client.

Latency.properties file has the latency values in milliseconds, which is the expected latency between two clients when they are communicating with each other to download a file. peerList.properties file has the list of peers that are present which is used by the server to populate the list of peers for gathering file information.

[1] *Load* is the number of requests processed by the peer.
[2] *Min(Load*Latency)* is used to obtain the ideal peer.

How to run:

1. *make all*
    a. Creates the java classes required to run the project
2. *java TrackingServer*
    a. Runs the TrackingServer in port 8000 of the running system
3. *java Client 8001*
4. *java Client 8002*
5. *java Client 8003*
6. *java Client 8004*
7. *java Client 8005*
8. *java Client 8006*
    a. Runs 6 Clients at the port provided by the argument.
    b. By default, use these ports. Different ports can be used to run the clients, but in order to do so, subsequent changes are to be made in peerList.properties and latency.properties file.
    c. If more clients are wished to be added, add the latency of each pair in latency.properties. This is used while contacting the peer to download a file.
9. *make clean*

Note:
The file system contents are sent to the server on a periodic basis. (20 seconds). After downloading a new file in a peer, the server will get to know about the new host when it asks for an update.
As a result, please wait for at least 20 seconds when trying a **Find** command after **Download**.

Test Client :
If you wish to run a concurrent download and upload scenario, run the TrackingServer (*java TrackingServer)* and wait for it to run. Then run TestClientMultiple.java class for all the clients to communicate with each other for Find and Download functionalities. **Only requirement** is that two dummy files (hello1.txt) should be added to 8001 and 8002 folders; after running this program, 8003, 8004 and 8005 will also have this file.

About the Classes:

1. TrackingServer:
- Main Class of the server.
- This creates a new thread of the server when contacted by a peer.
- It is contacted in 2 scenarios: *Find and UpdateList*

2. ServerHelper:

- Contains helper methods for the server’s functionality.

3. FileSyncThread:
- Periodically updates the list in the tracking server by contacting the peers for its files.
- As of now, there are no deletion ot updates to the files in the system.

4. ClientToClientResponder:
- Thread of client that responds to request from other clients. Although this thread is also used to send the filelist to the server periodically.
- Contacted for : *GetLoad, Download and SendAll*.

5. ServerToClientResponder:
- Thread of server that responds to client’s request.
- Contacted for : *UpdateList, Find and Ping*

6. ServerHealth:
- Thread to check if server is running, invoked when a Find request is made and client cannot connect to the server.

7. Client:
- Main class of client and runs the terminal UI.
- Spawns ClientToClientResponder when contacted by other entities.

8. ClientHelper:
- Contains helper methods for the client’s functionality.

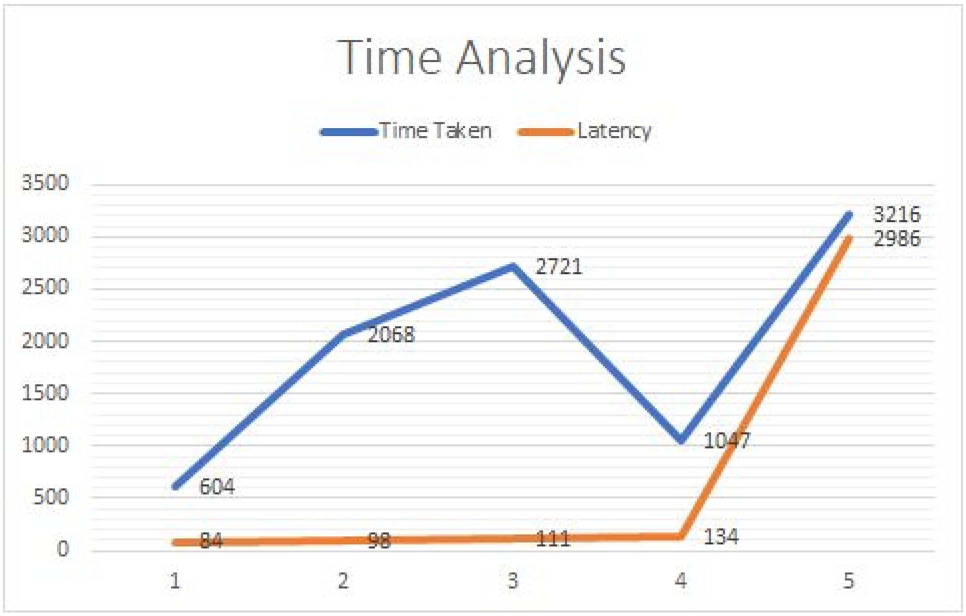
9. SocketConnection:
- This is the class which binds TCP connection between different entities.

10. TestClient:
- This class is used to run clients as separate thread

11. TestServer:
- This class is used to run server as separate thread

12. TestClientMultiple:
- This class tests multiple download and upload scenarios which can be difficult with the UI.

Analysis:



The above graph plots the average time taken for a download request to be processed in the serverless file system. The X-Axis plots the number of peers that hosts the required file and the Y-Axis denotes the latency in ms.

The blue line denotes the time taken with the simulated latency between the ideal host and the requesting peer. The orange line denotes the actual latency in the network, which is obtained after subtracting the predetermined latency from the blue line.

It can be seen that the actual latency increases as the number of peers increases. Even though the availability of the file increases, the requesting peer has to perform more calculations. It needs to getLoad of all the hosts and calculate the min(load*(simulated)latency).
As a result the actual latency also increases.