CS4098 Distributed File System Report

Student No. 13328025

Intro

Our goal was to design and implement a distributed file system exhibiting a range of properties.

These property were:

- Distributed Transparent File access
- Security Service
- Directory Service
- Replication
- Caching
- Transactions
- Locking Service

I will outline in the following pages what my approach and level of implementation was for each of these.

I choose to use Python for building my system as it was sleek and I had experience with libraries that would be needed along the way. The basic skeleton of the system had to be a RESTful server and client. I was able to achieve this by using pythons Flask API. This made it simply to pass http request from client to server and additionally server to server. Along with this I used a mongo database to store the client's information and the files themselves. It also stored Meta data on the overall system.

Distributed Transparent File access

This was the basic underlying implementation that was going to be used throughout the system. I choose to use an upload/download scheme as I thought it was the easiest to implement for what I needed and was easily layered as well as scalable. There are 6 pieces of functionality open to the client. Create an account, login (authorise the client), upload, download, delete, upload an edited file (must have the lock) and unlock a file. I exposed the unlock functionality for two reasons. First is it is not damaging to the system even if it is a client with no lock to unlock to use. If a user is not in possession of a key they can call unlock and be returned with an ambiguous warning. This means a client cannot gain info on the lock. The second reason was that when downloading a client asks for a write lock in the http request but I thought that it would be helpful not to assume when uploading an edit that they were finished and could continue working on the file. This ties in with the continuous working idea I explain in the locking section.

Database Structure

The database is formed with the collections client, public keys, directories, files and servers. Through replication this database structure can be maintained on individual nodes on the DFS.

Authentication and Security Service

Goal: "All interactions between the various servers of your distributed file system should be protected so that third parties cannot spy on the content of messaging crossing the network, or worse still, damage or corrupt this data."

With the security service I concentrated on implementing the 3 key ticketing model outlined in the project descriptor. Upon requesting authentication to start a new session with the system a client provides the server with a username, client id and an encrypted password. The password is encrypted with an arbitrary key derived from the password for just this instance. The project descriptor outlined that the strength of the encryption was not of interest so I decided not to include strict key pairings. When the server receives this request I assume that it can derive the key for the password and decode it to verify the user is indeed who they say they are. Here the actual algorithm kicked in. A random and unique to each user, session key was generated along with an expiry date. The expiry date is stored in the database alongside the client's credentials. The session key is then encrypted with the server's key (We assume all servers know this key). This then becomes the 'ticket'. Next a token is created with the host and port of the server allocated to the client. The ticket is also included but is only of interest to the server. The server will use this ticket to decrypt all communication from this client. This then in turn means the client will have to encrypt all messages to the server with the given session key.

The actual strength of the encryption used to implement the above approach was not to standard but it was outlined in the project descriptor this was not necessary. I used python's Crypto library to encode and decode all data. The type of encryption was 16 bit AES. As outlined this is not overly strong but the model that we were asked to produce is implemented thoroughly throughout the system.

In my system the authentication server is the main server off of which all other servers run off of. This is because flask is initialised here and thus is the hub. However it is only the hub in terms of flask. The actual requests to the directory servers are indeed handled by each directory server. I believe this not to be a hindering design fault and is not a problem in terms of a high level look at the system.

Directory Service

Goal: "The directory service is responsible for mapping human readable, global file names into file identifiers used by the file system itself. A user request to open a particular file X should be passed by the client proxy to the directory server for resolution. The returned file identifier should identify the server actually holding the file. "

For the directory service I decided to implement it in such a way that both files and directories were handled on the same server. To achieve this the database collections for both are configured to integrate each other. For each file the directory it resides on is referenced in each files entry as is the server. The directory collection has a field for the server that it resides on. Thus the illusion of a fully implemented directory service is created. This system works perfectly compared to other but cuts down on spinning out file servers on top of directory servers.

In my system the directory servers are the distributed workers which handle the lion share of client's requests. Both replication and locking are built into and upon the directory servers.

Replication

Goal: “Your task is to implement an appropriate replication model for file access across selected file servers”

For replication I loosely followed a passive replication model. Here when a server receives a file it records it itself and then passes it on to other servers to record. What I did was have a ‘master’ directory server which never received requests from clients but only from servers. This was to cut out the disadvantage brought by passive replication as responses from failed replications are delayed. With my system as long as the server that the client is in contact with records the file successfully the client can continue working with this server. This is because the master server looks after the broadcasting of the file. Unfortunately as of yet there is no rollback procedure only that the master keeps broadcasting until all servers comply.

Caching

The caching I choose to implement was a quite simple client side cache. I used Redis which is a multi-data structure disk cache. I also used zlib to then compress and decompress the data to further enhance speed. I use it as a LRU cache which I think is the most effective cache system for a file system.

Locking

For the locking service I opted for my own implementation. This was in contrast to perhaps using thread locking within each server or between servers. When a client requests to download a file they specify whether they want a read lock (no further writing/editing on the file) or that they do intend to write or modify the file. In the latter instance a write lock is placed on the file for a certain amount of time or until the file is fi ished being modified. My thinking behind this was that this could be scaled up to be something like google docs continuous writing and saving of the file. What I mean by this is that these requests to write are sent continuously while the file is being modified thus other users can see the updates. To achieve this the service needs more extensive work. Intermediary steps between download and an edited upload would have to be added and made autonomous as well as a pool locks are given to clients who are allowed access to the file (At the moment only one person can have a write lock). But I thought it was a step in the right direction and an acceptable feature for this particular system.