CS4032: Distributed File System Project

Aaron Christopher Joyce
12326755

Introduction

The solution I have designed and implemented has been given careful design consideration. The project has been implemented in the programming language *Python*. The project makes use of a number of third-party Python modules, including *Flask* (a Python Web framework - the RESTful API layer), *Pycrypto* (a cryptography module), *MongoDB* (an open-source document-management database framework), *Docker*, and *Pyfscache* (a file system cache). The design and implementation of the distributed system is discussed in the proceeding section. I successfully achieved my objective of constructing a scalable distributed file system, satisfying all of the major properties as set out in the rubric of this assignment; i.e., distributed transparent file access; security service, directory service; replication; caching; transactions; and a lock service.

Design and Implementation

Distributed Transparent File Access

I have written a language-specific interface to distributed file system. The access layer has been written in the programming language *Python*, powered by the RESTful framework *Flask.*

Having authenticated with the authentication server, a client passes a ticket as a parameter in their RESTful requests. This provides the client with access to the server, restricting the files they have access to.

For simplicity, a single “directory server” is appointed to be the master, and holds a master copy of all files uploaded. The client’s request is then simply relayed to all other non-master servers by the master server, asynchronously.

The distributed transparent file access provides an upload, retrieval and deletion interface. The implementation described in this report conforms to a layered application design pattern. The central layers are: users; authentication presentation layer, in the form of a RESTful API interface; business layer, which is responsible for handling important actions, such as file replication; and the data layer, whose responsibility is to persist data on a distributed set of servers.

Security Service

I implemented a trusted authenticated server, which can be used to create a client. It is also used to authenticate an existing client of the service. When creating a client, a client identifier, password and the client’s public key are required. Using this data, a client record is created, against which the server can authenticate, enabling a client to read or write to the distributed file system.

To authenticate with the server, a client must provide their password and client identifier. This password is encrypted using the client's public key, and the encrypted password derived from the client's unencrypted password and public key, is compared with the encrypted password stored on record for the client. If the two encrypted passwords match, the client's authentication request is considered to be valid.

Upon successful validation, the client receives a ticket and session key, along with the host and port of the master server to which they will read and write. The client uses this ticket to prove its authenticity. It ensures that a client can only access those files they uploaded, and cannot upload files on other clients' behalf.

The authentication server storage key is a shared secret between the authentication server and the directory servers. Upon making a successful authentication request, encrypted data is returned to the client. The client must decrypt the data received using their public key. From the decrypted data, the client accesses the session key, the ticket (which will be used to communicate with the master directory server), and the host and port of the master server, to which the client will make upload and retrieval requests to the distributed system.

The client must uses the session key to encrypt the files' data being transmitted over the network. The client may upload an encrypted file to the server, along with the ticket it received from the authentication server. The ticket is used to derive the session key, and the session key is used to decipher the data transmitted to the directory server (i.e., server-side decryption).

This ensures that both the file data transmitted to the server as well as the filename and directory in which it should be placed are encrypted. The session key is generated dynamically, and is not exposed to network traffic in a "plaintext" or unencrypted form; i.e., it is transmitted to the client using an encryption mechanism which means it can only be decrypted using the client's public key.

As a simplifying assumption, a single shared secret is shared by all of the directory servers and the authentication server.

Directory Service

For simplicity, I implemented a "horizontal" or "flat" file system. I successfully achieved this by maintaining a mapping (using MongoDB records), containing a given file's name, directory/location, and a hash digest of the client identifier and a filename appended to the client-defined directory. The hash digest was used as the

file name within the horizontal file system. It was necessary to include the identifier of the client to which a file belongs, as failure to do so could result in two or more clients' files having the same name and being stored in a directory with an identical path, which would result in such files having the same hash digest in the horizontal file system.

Replication

I implemented replication using a primary copy design paradigm. File changes are replicated asynchronously; i.e., the master copy server processes and acknowledges the client's request, and updates the file in its data store. Then, the master copy server updates the non-master copies in the background, which dramatically improve server response times. If replication were done synchronously, a quorum (the minimum number of servers which must successfully process received replication requests) would have to be satisfied by the server prior to a response being provided by the master server to the client. Synchronous replication would result in clients of the system having to wait significantly longer. The asynchronous replication approach, which I have successfully implemented, enables the master copy server to proceed with replication in the background, with possibly some replication lag.

When a client uploads a file to the RESTful *upload* endpoint, the data transmitted is first cached. A transaction thread is then created to store the file on disk on the master server. Also, a separate thread is spawned to replicate the file by the master copy server to the non-master replicas. The spawning of a separate thread ensures that this occurs in the background, parallelly, replicating the files to a set of replicas. The set of available replicas is maintained using MongoDB, where each record refers to a given replica's host and port. The master copy makes RESTful requests to the replicas, just as the client makes to the master copy, essentially retransmitting the original client's request. I used the RESTful end-point available while replicating, as it represents a simple interface and demonstrates excellent application of code reusability principles.

It is important to note that a single instance of MongoDB is shared by all servers; i.e., master and replicas, essentially providing global, atomic access.

Caching

I implemented a caching layer using Python's *pyfscache* library. The cache is specific to each directory server; i.e., a single cache instance is not shared among all servers. Initially, I implemented a shared cache, running on a specific host and port, using Redis. However, there were performance concerns due to the time cost associated with network access; hence, I opted for a *pyfscache.*

I wrote a layer of abstraction around the *pyfscache* library, to abstract the low-level details associated with its usage, such as expiration periods for set data, and retrieving cached data. This provided a programmatic interface to key operations, such as cache writes, reads, and invalidation/expiration.

When a client uploads a file to the master server, or the master server replicates a file to a non-master copy, the file data is cached. A separate thread of execution writes the file data to disk on the server, using the data obtained from the cache. The file is replicated asynchronously, to the non-master copies, using the data stored in the cache of the master copy, as opposed to on-disk data. This assumes that the cached entry has not been invalidated by the point at which replication begins. Otherwise, the master copy seeks the data from disk. As reading from the cache is more performant than reading from disk, this helps to improve overall system performance.

When a client seeks to retrieve an existing file on the distributed server system, the system first checks whether a non-master copy has a copy of the file, first in the system cache and upon absence from the system cache, the system checks server's disk. If the requested file is neither present in the cache nor the disk, we seek a copy of the file from the master server. If this fails, then an error response is returned to the client. Otherwise, we return the requested file.

A key concern considered while devising and implementing a caching design was cache invalidation in the face of multiple concurrent clients reading and writing from the same file. If client E reads file Y, and client F writes to file Y, then client E's cache copy is synchronously invalidated on the master copy server and asynchronously on the replica servers. A notification of a file change by another client is not pushed to a client who had read the file contents prior to a change. It is the client's responsibility to re-request a copy of the file's data, at which point, they can identify that the file has been update since their previous read.

For simplicity, I cache all of a file's contents. I did explore the possibility of caching part of a file. However, implementing this proved to be too complex.

If a file is updated or deleted, the cached entry is invalidated on all host machines; i.e, all replicas and the master copy. Additionally, the file is also deleted on disk. This is performed synchronously on the master server; it is performed asynchronously on the replicas, by forwarding the initial client's request onto the replicas.

Transactions
When a client uploads or modifies a file on a server, this is updated in a thread outside of the main thread of execution.

Upon successfully receiving a file on the master server and saving it to both cache and disk, a transaction thread is spawned and executed asynchronously. The role of the transaction thread is to replicate the client's request to the replica copy servers.

The transaction thread writes to the replica servers using the standard Directory Server API.
I implemented two levels of transactions: Single-server transactions, which employ the shadowing technique, and secondly, transactions pertaining to many files, located on different servers. To achieve this, I did not implement a separate transaction service; instead, I incorporated it into the existing directory server, operated by the master copy. I used MongoDB to record the state of transactions. The master copy server is responsible for managing execution of the transaction. Transactional changes are those that involve the creation or modification of files; hence, this does not pertain to the reading of files.

When a client sends a request to create or update a file, the master copy server records "ready to commit" transaction entry against a file reference in the MongoDB service. The master copy server then sends a "ready to commit" request to each of the replica copy servers. Upon receipt of success responses from a requisite number of replica servers (a quorum of 50% of all servers within the distributed system must be satisfied), the master copy obtains a commit token from each of responsive replica servers. The master copy records the commit tokens against the transaction record in MongoDB. The master server then proceeds to re-transmit the file changes to the replica servers, using the tokens received from the "ready to commit" request. When a replica copy server accepts a "ready to commit" request from the master copy server, a server-specific record is created in the replica server, persisting the transaction token returned, along with the filename and directory path to which the transaction pertains. When a replica server receives the file data along with a transaction token, it uses the transaction record to access the file's meta information (such as filename, directory, and the identifier of the client who is creating or modifying the file). The transaction record is then removed from the replica server copy. Transactions records are deleted on both the master and replica servers upon successful completion of the transaction, of failure to satisfy the quorum necessary. If any of the replica servers that returned a transaction token fail to complete the transaction during the final stage of the transaction (i.e., post-"ready to commit" stage), then the master server's request to complete the transaction is retried until such time as the replica server in question goes offline (i.e., is no longer part of the distributed system) or the request is accepted. Once the transactions have been successfully completed on all replica servers, the transaction record stored on the master copy is also deleted.

Lock Service

In the context of my project, locking is an integral part of the transaction server. A transaction may only begin once the directory server has secured a semaphore lock on a file. The client is not exposed to this detail; their request is is queued until such time as another client's transaction has resulted in the lock on a file being relinquished. Lock management occurs at the directory server level. The directory server maintains a MongoDB 'lock' record type, which is used to record those files that are currently locked as part of a transaction. The MongoDB lock record type include a unique lock identifier, and a reference to the file in question. If a client makes a request to update a file which another client already has a lock on, by virtue of a transaction, the second client's transaction request is queued until such time as the lock has been relinquished. A client may read from a file for which another client has a lock on. However, they cannot update a locked file, until the lock has been released.

It is important to note that files may made available to clients other than the original uploader of a file. This is where locks play a central role; coordinating shared access and updated among different clients concurrently.

Conclusion

In conclusion, I successfully implemented all of the required components of a distributed file system.