Distributed File System Report

Neil Barry-Murphy
13327106

The goal of this assignment was to design and implement a distributed file system. The system uses python, flask, mongodb & disk cache, and is constructed using a RESTful service.

Distributed Transparent File Access

The system that I have designed makes use of a virtual file navigation, access and storage structure. The are two major components to this structure; directories, and the actual files themselves. An authenticated user (this will be discussed in more detail in the security section of this report), is capable of uploading, downloading and deleting files from the system. A RESTful request is sent by the user (user.py) to any of the currently operational directory servers (instances of directoryserver.py) in order to initiate a transaction, provided that the user has been successfully authenticated.

There are four types of transactions; upload, download, delete and rollback. For the purposes of this system, a rollback transaction is simply treated as a system-wide delete transaction, in order to maintain a simplistic design. The file “test.txt” will be the test file used in most testing scenarios. The database used to keep a record of all currently existing files, directories, servers, transactions and authorised users is mongodb.

A user is created and mimicked using the user.py script, and all mechanics of the file system are shielded from this script through a security service implementation. Simply run this script to simulate the behaviour of the user with regards the provided features of the file system.

Security Service

All network security pertaining to the user, and all generated files and directories is managed by the authentication server (authenticationserver.py). When a user is added to the database, they must be authenticated in order to be able to interact with the file system. A user will have an id, a password, and a public_key. It is easier to think of the user_id as a username. Separate RESTful POST requests are used to both create and authenticate the users. The user will register with the system with the above three parameters, which will be included with the POST request as headers. The authentication server will then add that user’s information the database, ensuring to hash the password using ECB base64 levels of encryption with the user’s public_key. A new field will also be added. This will be the

session_id, and will be used to encrypt the files and directories that the user wishes to upload/delete and manage respectively. Without this session key, a user cannot access the file system, ensuring that unauthorised users cannot gain access to the files and directories. Hashing the user's password ensures that if an external user manages to get access to the users collection (within the mongo database), they cannot directly see the user's chosen password, meaning that they still cannot gain access to the system. This solves two major security issues.

Once the user has been added to the db, they must now be authenticated. The user is required to add their username (user_id) and password to gain access. This password will be compared with the password in the system. Of course, the password in the database is encrypted at that point, so we simply decrypt the password in the database again using the public_key of the user. (We can find the user at this point in the db using the client_id). This is then compared with the password (pwd) provided in the headers of the POST request. If a match, the session_d is updated, so that somebody using an old session key cannot re-use a stale session. This current user is then set as a variable. Next, the session key must be hashed, and must be set to the correct length so that it can be hashed with the global AUTH_KEY. This is now the user's encrypted session_key. This is required by the system to access and modify the contents of the file system. If the authorisation phase is successful at this point, a json response containing the ticket that contains the session_id, server_host, server_port and access_key is returned to the user by the authentication server. The returned data is called a ticket, and is then hashed using the public_key of the user to provide an additional layer of security as it passed over the network. The server_host and server_port are required by the user so that the master_server is known. This allows for replication, and the completion of different types of transactions, which will be covered in greater detail at a later stage.

The data that is returned to the user by the authentication server is then assigned to variables. It is decrypted using the public_key of the user before any assignation can be determined. At this point the user has been successfully authenticated, provided that the response returned by the authentication server is not None. From here, the encryption of the files and directories, and the interaction of the user with any of the instantiated directory servers will be discussed.

The specified location of the file, (the directory) is set here by the user. For example, the user wishes to create a home directory called "fileserver" on the system. Within this home directory, a subdirectory called "location" will be created. It is within the subdirectory "location", that the file "test.txt" will be uploaded by the user in our test scenario. A virtual_structure_hash is created using the decrypted session_id of the user (provided by the authentication server). This has already been discussed above. This will be used to hash both the directory (file location) and file name, once these have been padded correctly, as they are initially not the correct size for encryption using the AES method of encryption adopted by the system here. The encrypted file name and directory name will be included as headers, along with the access_key, to the POST request: "/file/upload" when a file upload operation is simulated using the user script. The access_key is crucial to the success of this operation, as it is the used to derive the session_id when decrypted using the AUTH_KEY

(which is also provided in directoryserver.py for simplicity). The session_key can then be used to decrypt the actual file and directory names that were initially set by the user. This is the final layer of security that is provided by the system, and ensures that file and directory names cannot be modified by external sources.

At this point, all operations that are completed over the network are now secure. This ensures that the user's session remains uninterrupted by external sources, and validates the authenticity of the distributed file system.

Directory Service

The location of files is managed by directories and subdirectories. An example has already been provided above whereby the user creates a subdirectory "location" within the home directory "fileserver" in order that the file "test.txt" located in "test-files/test.txt" can be uploaded successfully, and that it is easy to keep track of this file. All of these files and directories are managed by the directory servers, of which there are three instantiated for the purposes of this demonstration and report.

The directory servers can handle three types of requests: upload, download and delete. Depending on the RESTful operation that is required, different actions will be undertaken in order to complete the goal of the user for a particular request.

When the directory server handles an upload, the necessary security checks are performed. From here, the directories mongo database is checked to discover if a directory with this name already exists. If it does, continue. If not, create the new directory. It works by separating the directories by virtual structures. For example: "/fileserver" and "/fileserver/location" would be treated as two separate directory entities so that the overall structure design remains simple. Next, a check is performed to discover if the file that is being uploaded already exists in the system. If it does, it is simply removed, and then re-added in order to cater for updates to the file contents. If not, simply add the file. Upload the file to disk, so a "physical" virtual copy now exists.

When the directory server handles a download, the necessary security checks are performed. From here, a check is performed on the mongodb directories collection to see if the directory to which this file belongs exists. If is does, continue. Another check is then performed on the mongodb files collection to ensure that the actual file itself exists. If it does, check the cache for the file (more on caching later), and download. If no record exists in cache, download the file from disk.

When the directory server handles a delete, the necessary security checks are performed. From here, a check is performed on the mongodb directories collection to see if the directory to which this file belongs exists. If is does, continue. Another check is then performed on the mongodb files collection to ensure that the actual file itself exists. If it does, delete the file

from disk, from the cache (if exists in cache) and from the mongodb files collection. This is handled by the delete transaction, and will be discussed in more detail at a later stage.

Replication

The type of replication that is adopted by this system is passive replication. This means that the user is assigned the identifier of the master_server (from the authentication server) in the form of the master_server's server_host and server_port. This means that the replication need only occur at the level of the master_server, which will ensure that replication occurs down through all other servers that are not tagged with the master_server status, but are in_use. The form of replication prevents constant polling, and reduces overall system load, and the check is only performed once per operation before the completion of the required transaction.

For example, let us say that a user uploads a file. The directory server is aware of this, and calls on the transaction class to handle the upload. This is done on a per thread basis, and can only occur if the user has been assigned the host and port of the master_server, which must be set by the authentication server. If this does not occur, the transaction cannot be completed, and the file will not be replicated correctly across all currently active (in_use) directory servers. It is also important to be aware that all forms of transactions are handled asynchronously by this file system.

A check is also performed within the transactions class to ensure that a minimum number of servers are aware of a file uploaded (for example) to an individual server. This minimum number is defined by the status of a transaction (SUCCESS, FAILURE and UNKNOWN), but these will be covered in greater detail in the transactions section of the report.

Caching

Caching is an important aspect of this file system. It reduces lookup times for files that already exist within given directories significantly. The caching system used for this file system is python disk cache. When a user uploads a file, it is immediately placed in the cache. When a user wishes to download a file, a check is performed to determine whether or not this file exists in the cache (which is also dependant on the disk cache eviction policy, and length of time already spent in cache, etc.). If the file exists in cache, it is simply fetched and returned. If not, it must be retrieved from disk, which is a much slower operation. The effectiveness of the cache scales exponentially with the number of users of the system. The cache reference is simply a hash of the concatenation of the file['identifier'], the directory['identifier'] and the server_instance()['identifier'], where the server_instance() returns the current server.

The cache can also be used when completing the asynchronous upload transaction. The hash is generated and the cache is checked. If the file exists in cache, it is simply retrieved from the cache and replicated across all currently in_use servers. If not, the file must be read from disk. Again, the effectiveness of the cache for this instance scales exponentially with the number of currently in_use directory servers.

Transactions

There are two main types of transactions that are handled by this file system; upload and deletion transactions. Rollback transactions are just an extension of the delete transaction, and are handled in a similar fashion.

It is very important to note at this point how a delete transaction (or rollback transaction) would work. The criteria for a successful transaction is based on the theory of a quorum. For example, when a file is uploaded by a user to the system, the current directory server (master_server) will acknowledge the change and update the “ledger” (status) column in the transactions table. If the change is recognised, this would be marked as a “SUCCESS”. The other types of status are “FAILURE” and “UNKNOWN”, where a fail is an unrecognised change (replication has not occurred for the server with this status), and an unknown status deals with issues like a server that is not operational. The quorum is determined by calculating the total number of each types of status. The total number of fails and unknowns are added together. If the total number of successes is greater than or equal to the total number of failures plus the total number of unknowns, then the upload transaction would be considered a success, and the transaction would be acknowledged by the system to definitely say that the change has been recognised by the system.

In the case that this quorum scenario fails (total number of successes is less than the total number of failures plus the total number of unknowns), the transaction will not be acknowledged by the system, meaning the system would not recognise the change. In this case, the transaction would then be rolled back (or deleted, in the case of the implementation of this file server).

Lock Service

This is a simple matter of managing thread locks at the time of managing transactions. For example, we cannot allow multiple threads (generated from an asynchronous transaction method call from the directory server) to delete a file continuously. It should only be deleted once, and a single thread should handle this operation.

This check prevents multiple unnecessary, and potentially dangerous, transaction modifications, as well as file and directory changes by another user or users at the time of a transaction pertaining to a particular file or directory.