CS4532 - Distributed File System

1. General Information

A setup file has been provided which downloads the necessary libraries as well as the content of this github. The steps involved are:

1) If using docker, run `sudo docker run -i -t ubuntu /bin/bash` to start a docker container. All following steps should be run inside the docker container, or in a terminal window if not using docker.
2) Next run:
    - `$ apt-get update &&  apt-get install -y wget`
    - `$ wget https://raw.githubusercontent.com/vanhoutk/DistributedFileSystem/master/setup.sh`
    - This will download the setup script.
3) Run the script by using
    - `$ chmod +x setup.sh`
    - `$ source setup.sh`

Once the libraries have been installed, the distributed file system can be built using the build.sh file, and then run with the run.sh file.

A number of issues were encountered during testing, two of which can be resolved as follows:

1) Some versions of packages might result in build errors:
    - To solve remove the .stack-work folder (`rm -rf .stack-work`) for the affected application and rebuild
2) The client build might throw up a gtk2hsSetup error message. To solve run:
    - `$ stack build --resolver lts-7.16`
    - `$ stack install alex happy gtk2hs-buildtools --resolver lts-7.16`

Two common files were shared between all of the service. These were an APIs file, which contains:

- Logging Variables
- Port Variables
- Data Declarations
- API Declarations
- Security Variables and Functions

And a MongoDB functions file which contains a number of helper functions for using MongoDB with Haskell. The MongoDB functions were taken from the use-haskell project located at: https://bitbucket.org/esjmb/use-haskell

Log messages are implemented in each service, but can be turned off by changing the corresponding variables in the APIs file.

2. File Service

The file service which was implemented was a flat file system, i.e. each file server provides a single directory. It was decided to implement a flat file system as it made file:server mappings within the directory service simpler for this proof of concept system. A layered directory structure could be implemented for the file service, although the directory service logic would need to be changed quite a bit.

The file service stores files locally to disk in a directory named with the port which the fileserver is running on, for example “files8081”. When a file is deleted or uploaded, the fileserver sends a message to the directory server updating its list of files for this fileserver.

It should be noted that there is no check for validity of authentication token when a modification time request is received. It was decided that for simplicity the modification time would be returned regardless, as adding functionality to the cache to deal with the error would have added an extra level of complexity.

When a transaction occurs, a temporary copy of a file is uploaded to the file server, this temporary copy is denoted with a ~ after its name. If the transaction is aborted these files are deleted. If the transaction is committed these files are copied into the local files and then deleted.

The file service has a number of different functions, each of which is listed below.

1. File Upload
    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
        - File name encrypted with the Session Key
        - File contents encrypted with the Session Key
    - Return Values:

        - Response encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Decrypt the file name
        - Check if the token is still valid
            - If not, return a failed response
        - Decrypt the file contents
        - Send an update request to the directory server
        - Store the file
        - Return a success response

2. Delete a file

    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
        - File name encrypted with the Session Key
    - Return Values:
        - Response encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Decrypt the file name
        - Check if the token is still valid
            - If not, return a failed response
        - Send a delete request to the directory server
        - Remove the file
        - Return a success response

3. List the files

    - Return Values:
        - List of file names
    - Logic:
        - List the contents of the directory
        - Sort the list
        - Return the list

4. File Download
    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
        - File name encrypted with the Session Key
    - Return Values:
        - File name encrypted with the Session Key
        - File contents encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Decrypt the file name
        - Check if the token is still valid
            - If not, return a failed response
        - Read the contents of the file
        - Encrypt the contents
        - Return the encrypted file name and contents

5. Get Modify Time
    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
        - File name encrypted with the Session Key
    - Return Values:
        - Modification time encrypted with the Session Key
    - Logic:
        - Decrypt the session key
        - Decrypt the file name
        - Get the modification time of the file
        - Encrypt the modification time
        - Return the encrypted modification time

6. Commit Files
    - Inputs:
        - File name
        - Temporary file name

- Return Values:
  - Response message
- Logic:
  - Read the contents of the temporary file
  - Write these contents to the permanent file
  - Delete the temporary file
  - Return a response message

3. Directory Service

The directory service uses a MongoDB database, called FILE_SERVER_MAPPINGS, to store the information about the file:server mappings. The information which is stored in the database is: the name of the file, the name of the server on which it is stored, and the port number on which the fileserver is running.

When the directory server starts up, it requests a file list from each of the fileservers that it is connected to. As such, the fileservers need to be run in advance of this directory server. (This is implemented in the run.sh provided). The list of files on each server is then stored in the database.

In the case where a new file is being uploaded, the fileserver to upload to is randomly selected. This was done for simplicity, although a more complicated model based on the number of files or the size of the files on each server could easily be implemented instead.

Replication is implemented through the upload function of the directory server. As the transaction server also needs to maintain consistency between files, the directory server has a function which returns all of the fileservers that a file is on, which the transaction server uses for its own replication.

The directory service has a number of different functions, each of which is listed below.

1. Get a fileserver for a file

   - Inputs:
     - Session Key encrypted with the Shared Server Secret
     - Timeout on the Authentication Token, encrypted with the Shared Server Secret
     - File name encrypted with the Session Key
   - Return Values:

        - Fileserver Port number encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Decrypt the file name
        - Check if the token is still valid
            - If not:
                - Encrypt the port number 0 (which is used to signify error)
                - Return the encrypted error port
        - Search for the file name in the file mapping database
            - If the file mapping exists
                - Get the first file mapping (if there's more than one)
                - Get the port from the file mapping
                - Encrypt the port
                - Return the encrypted port
            - Otherwise:
                - Encrypt the port number 0 (which is used to signify error)
                - Return the encrypted error port

2. Get all fileservers for a file
    - Inputs:
        - File name
    - Return Values:
        - List of port numbers
    - Logic:
        - Search for the file name in the file mapping database
            - If the file mapping does not exist, return an empty list
            - Otherwise:
                - Get the port number for each file mapping and store in a list
                - Return the list of port numbers

3. Upload a file
    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret

        - File name encrypted with the Session Key
    - Return Values:
        - Response message encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Decrypt the file name
        - Check if the token is still valid
            - If not, return a failed response
        - Search for the file name in the file mapping database
            - If a file mapping exists
                - Upload the file to each of the fileservers
                - Note: There was no success check on upload here although this could easily be implemented by storing the return values of the mapM function.
                - Return a success response
            - Otherwise:
                - Randomly select a fileserver
                - Upload the file to this fileserver
                    - If this upload succeeds, return a success response.
                    - Otherwise return a failed response.

4. List the files on all servers

    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
    - Return Values:
        - List of file names encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Check if the token is still valid
            - If not, return an encrypted list containing error information
        - Get all of the file mapping which are currently stored
        - Make a list of the file names in each file mapping
        - Sort the list and remove any duplicate file names
        - Encrypt the list

        - Return the encrypted list

5. Update lists
    - Inputs:
        - Type of update
        - Port of the server sending the request
        - Filename
    - Return Values:
        - Response Message
    - Logic:
        - If the type is a “delete” update
            - Calculate the server name from the port number
            - Delete the file mapping for this filename and fileserver
            - Return a success response
        - If the type is an “update” update
            - Calculate the server name from the port number
            - Update the file mapping for this filename and fileserver
            - Return a success response
        - If the type is something else
            - Return a failed response

4. Authentication Service

The authentication service provides a level of security to the distributed file system. The authentication scheme used within this project is the three key security model which was provided in the project description. A shared server secret was used to simplify the scheme, although storing individual server keys could be done without too much difficulty. It would just increase the number of requests the authentication server receives from each client.

Encryption and decryption are done through the use of a simple XOR function. This encryption mechanism was chosen because it provided simple and easy to implement symmetric key encryption. A more secure encryption scheme could be implemented, but was not deemed necessary for the scope of this project.

It should be noted that there are currently no checks in place in any of the services to see if the session key has been corrupted/tampered with. Logic for this could easily be added but it was decided that this was outside the scope for a proof of concept project.

The timeout for the authentication token was encrypted with the shared server secret to prevent the client from tampering with it and extending the validity of the token. This was not further encrypted with the user's password, although adding the extra encryption would be trivial.

The authentication service uses a MongoDB database, called USER_ACCOUNTS, to store the username and password tuples. A function has been created for use by "admins" to add new users to this database. As only admins should be able to add new users, there is no requests to this endpoint through the client application. New users can be added to the database using the following command.

```
curl http://localhost:8090/addNewUser/username/password
```

The authentication service has two functions within it, the logic of each is explained below.

1. Login User
    - Inputs:
        - Username
        - A message encrypted with the user's password
    - Return Values:
        - Authentication token which contains:
            - A session key, encrypted twice, first with the Shared Server Secret, follow by the user's password, which is the authentication ticket
            - A session key, encrypted with the user's password
            - A timeout for the token, encrypted with the Shared Server Secret
    - Logic:
        - Search for the user in the user database
            - If the user does not exist
                - Return an error Authentication Token
            - Otherwise:
                - Decrypt the message the user sent with the stored password
                - If the message does not match the username
                    - Return an error Authentication Token
                - Otherwise:
                    - Generate a random string for the session key
                    - Encrypt the session key with the user's password to create an encrypted session key

- Encrypt the encrypted session key with the shared server secret to create an authentication ticket
- Encrypt the ticket with the user's password to create an encrypted ticket
- Generate a timeout for the token
- Encrypt the timeout with the shared server secret so that users can't change the timeout.
- Return the Authentication Token

2. Add New User
    - Inputs:
        - Username
        - Password
    - Return Values:
        - Response message
    - Logic:
        - Add a new username and password tuple to the database if one doesn’t already exist for that username, or update the password if a tuple already exists.
        - Return a success response

5. Lock Service

The locking service allows for the exclusive access of files in the distributed file system. Any time a client requests a file with read/write permissions, either through singly or as part of a transaction, a lock is requested for the file, and the file is accessed if and only if the request for a lock is successful.

The locking service uses a MongoDB database, called FILE_LOCKS, to store the information about the locks. The information which is stored in the database is: the name of the file, the status of the lock, the time the lock is valid until, and the session key of the user who locked it. A timeout of ten minutes is used for each lock; this prevents locks from being permanently locked if a client goes offline without unlocking the file.

The session key of the user was used as a unique identifier for the user. While generating a unique identifier for the user would potentially be a better approach, it was

decided that the session key was suitable for the purposes of this project. The unique identifier prevents other users from unlocking a file which they didn't lock originally.

It was decided that locks would only be provided on files, and not on entire directories. There were a couple of reasons for this. The first was that as a flat file system was used, and only a small number of file servers were created, locking an entire file system could cause trouble for other users.

The second reason was that it simplified logic and reduced network accesses. In the case where a lock already exists, to lock a directory, it would require a request to the directory server to find out the file server name for a particular file, followed by a lock request for that server. Whereas a single request is required when locking a file.

The program is designed in such a way that adding functionality to lock an entire directory would be relatively trivial if it was required.

The locking service has three functions within it, the logic of each is explained below.

1. Locking a file
    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
        - File name encrypted with the Session Key
    - Return Values:
        - Response encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Decrypt the file name
        - Check if the token is still valid
            - If not, return a failed response
        - Search for the file name in the lock database
            - If the lock exists and is currently locked
                - Check if 10 minutes has passed since it was locked
                - If so, return a success response and the update the lock with this user's session key
                - Otherwise, return a failed response
            - Otherwise:
                - If a lock record exists but is unlocked
                - Change the status to locked, set a time out of ten minutes and return a success response

- If no lock record exists
  - Creates a new one with a timeout of ten minutes and return a success response

2. Unlock a file
   - Inputs:
     - Session Key encrypted with the Shared Server Secret
     - Timeout on the Authentication Token, encrypted with the Shared Server Secret
     - File name encrypted with the Session Key
   - Return Values:
     - Response encrypted with the Session Key
   - Logic:
     - Decrypt the timeout
     - Decrypt the session key
     - Decrypt the file name
     - Check if the token is still valid
       - If not, return a failed response
     - Search for the file name in the lock database
       - If the lock exists and is currently locked
         - If the user who sent this request is the one who locked the file
           - Unlock the file and send a success response
         - Otherwise:
           - Check if 10 minutes has passed since it was locked
             - If so, unlock the file and send a success response
             - Otherwise, return a failed response
       - Otherwise:
         - Return a failed response

3. Check the lock on a file
   - Inputs:
     - Session Key encrypted with the Shared Server Secret
     - Timeout on the Authentication Token, encrypted with the Shared Server Secret
     - File name encrypted with the Session Key
   - Return Values:

        - True/False whether or not the file is locked
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Decrypt the file name
        - Check if the token is still valid
            - If not, return True (stops the client from getting a lock)
        - Search for the file name in the lock database
            - If the lock exists and is currently locked
                - Check if 10 minutes has passed since it was locked
                    - If so, unlock the file and return False
                    - Otherwise, return True
            - Otherwise:
                - Return False

6. Replication Service

A simple replication model was used within this project. It was decided that a set of "core" files should be present on each file server. The system is designed in such a way that if a filename already exists, then the fileserver it's on is what will be passed back to the client. As such, for multiple copies of a file to exist (i.e. replication), the files need to originally be copied to each fileserver manually by an "admin".

The benefit of a replication model like this would be seen in the case where certain fileservers were closer to the client than others, and so accessing of these core files would be quicker as the closest fileserver could be returned.

As all of the fileservers in this project are running on a local host, the directory server selects the first one in the list to point the client to, although logic could easily be added which would allow the "closest" fileserver to be selected.

Replication is currently implemented through both the directory server and the transaction server. The directory server maintains a list of all of the fileservers that the file is on, and sends the updated copy to each of these fileservers. The transaction server gets the list from the directory server before doing the same. The logic in both of these cases is explained within the relative sections of this report.

7. Transaction Service

The transaction service allows the user to make group changes to files, by storing temporary copies of the files on fileservers, and only committing the changes once the client has finished and decided to commit them, or deleting those changes if the client decides to abort the transaction.

A basic transaction service was implemented for this project. When the client decides to start a transaction, all subsequent actions which the client performs are passed through the transaction service, until the client decides to commit or abort the transaction.

The transaction service uses two MongoDB databases, called TRANSACTIONS and TRANSACTION_MAPPINGS respectively. The first of these solely stores the unique transaction ID, which is the Session Key of the user who is asking for the transaction. The second database stores the information about each transaction, the transaction ID, the original filename, the temporary filename and the fileserver port.

The Session Key was used as the transaction ID because it was already present and should be unique. A new random transaction ID could be generated instead if desired, but was not required for this project.

It should be noted that functionality was not implemented to allow a client to upload new files through a transaction. Rather, transactions only allow the editing of currently existing files. Functionality for the uploading of new files could easily be added, but it was felt that this wasn't required for this project.

The transaction service has a number of different functions, each of which is listed below.

1. Start Transaction
    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
    - Return Values:
        - Response encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Check if the token is still valid
            - If not, return a failed response
        - Store the transaction ID (Session Key) in the transaction database
        - Return a success response

2. Implement a download transaction
    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
        - File name encrypted with the Session Key
    - Return Values:
        - File encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Decrypt the file name
        - Check if the token is still valid
            - If not
                - Delete the transaction in the transaction database
                - Abort the transaction (see details later)
                - Return a failed response
        - Request a list of all fileserver ports for this file from the directory server
            - If this fails, return an error file
        - Add new mappings for each of the ports, with the original file name and temporary file name both the same
        - Download the file from the first port
            - If the download fails, return an error file
            - Otherwise return the file.

3. Update maps with a cached transaction
    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
        - File name encrypted with the Session Key
    - Return Values:
        - Response encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Decrypt the file name
        - Check if the token is still valid

                - If not
                    - Delete the transaction in the transaction database
                    - Abort the transaction (see details later)
                    - Return a failed response
            - Search the transaction mapping database to see if a mapping already exists
                - If so, return a success message
                - Otherwise
                    - Request a list of all fileserver ports for this file from the directory server
                        - If this fails, return a failed response
                        - Otherwise
                            - Add new mappings for each of the ports
                            - Return a success response

4. Implement an upload transaction
    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
        - File name encrypted with the Session Key
    - Return Values:
        - Response encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Decrypt the file name
        - Check if the token is still valid
            - If not
                - Delete the transaction in the transaction database
                - Abort the transaction (see details later)
                - Return a failed response
        - Search the transaction mapping database to see if a mapping already exists
            - If not, return a failed message
            - Otherwise
                - Create a temporary file name by adding ~ to the end of the file name
                - Upload the temporary file to each fileserver that has the original file

        - Update the transaction maps with the name of the temporary file
        - Return a success response

5. Commit Transaction

    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
    - Return Values:
        - Response encrypted with the Session Key
    - Logic:
        - Decrypt the timeout
        - Decrypt the session key
        - Check if the token is still valid
            - If not
                - Delete the transaction in the transaction database
                - Abort the transaction (see details later)
                - Return a failed response
        - For each transaction mapping
            - If the original file name and temporary file name are different, send a commit message to the file server
            - Unlock the lock for that file
            - Delete the Transaction Mapping
        - Delete the transaction in the transaction database
        - Return a success response

6. Abort Transaction

    - Inputs:
        - Session Key encrypted with the Shared Server Secret
        - Timeout on the Authentication Token, encrypted with the Shared Server Secret
    - Return Values:
        - Response encrypted with the Session Key
    - Logic:
        - Decrypt the session key
        - Abort the transaction:
            - For each transaction mapping

        - If the original file name and the temporary file name are different, send a delete message to the file server with the name of the temporary file.
        - Unlock the lock for that file
        - Delete the Transaction Mapping
    - Delete the transaction in the transaction database
    - Return a success response

8. Client Application

When the client first starts up, the user is requested to log in. A request is sent to the authentication service for an authentication token, and this token is passed to both the cache and the editor. The cache is described in section 9.

The Editor is the main interface which the client interacts with, although some console interaction is required as well. Through the Editor the client can choose to create a new file, read or read/write a file, see a list of all the existing files, upload a file or start a transaction. File names need to be input through the console when reading files from the file server, and the file list/log messages are displayed in the console.

The Editor disables the use of certain buttons when a transaction is not occurring and when a transaction is occurring. This prevents the client from circumventing the transaction. It was decided that for the proof of concept this was a suitable mechanism, although other options could be explored.

Given more time, an editor which does not require any console interaction would be created.

A client API handles all of the backend of the client, such that most of the operation of the client application is hidden from view, although it is visible through the console logging which is occurring. The client API handles the requests to all of the other services in the distributed file system, as well as interacting with the cache in the case of file interactions.

9. Cache Service

Caching of files was implemented on the client side, by storing a copy of the file on disk on the client. As an upload/download model was used, it was decided that storing a copy in memory on the server side wasn't useful, and that storing on disk on the client provided a satisfactory result. The cache is cleared when the client application is closed.

The cache service is closely linked to the client API which is handles the backend of the client application. The cache service checks for invalidation of files through polling every five minutes, although this time period could easily be extended or shortened. The cache has a limited size, and an LRU policy is used to remove files when the cache exceeds the allotted size.

The cache is passed the authentication token which the user receives on login so that its network accesses can all be encrypted.

The logic behind cache invalidation is shown below.

1. Check for Invalidation
    - Logic:
        - Wait 5 minutes
        - List all of the files currently in the cache ("/temp" folder)
        - For each file in the cache:
            - Get the local access time of the file
            - Request the port of the fileserver from the directory server
            - Get the modification time of the file on the file server
            - Check if the modification time on the file server is later than the access time of the local file
                - If so, download the remote file and updated the local copy
                - Otherwise, do nothing