Distributed File System

Gary Gunn - 12306421

February 16, 2017

1 Setup

In order to install this implementation of a distributed file system on a Ubuntu instance the following steps need to be taken:

- Download the 'setup.sh' shell script from the following link and place it in the parent directory of your choosing - https://github.com/Garygunn94/DFS/blob/master/setup.sh
- Execute the setup.sh script via './setup.sh'. The setup script will install stack, clone the repository and install mongo-db. Root access may be required.
- After the setup script has been executed, change directory to the 'DFS' directory (If not already inside) and execute the build.sh shell script via './build.sh'. This script will build each component of the project.
- Side Note: In order for this script to work I had to first run it without root access, and then with root access. If the script causes any issues email - ggunn@tcd.ie
- Executing './run.sh' will execute each of the components on a local machine. The user should then locate the terminal that executed the 'Client-Proxy' and follow the on screen instructions.
- Should distributed testing be required, the host IP's and ports must be changed in the CommonResources.hs file in the CommonResources package to reflect these changes.
- Should any issues occur with installation, contact ggunn@tcd.ie

2 Common Resources

In order to create a central hub of information that would be shared amongst all servers in the distributed file system, a local package named 'Common Resources' was created. Each of the other components of the distributed file system import this package which contains information they need. There are two files in this 'Common Resources' package:

- CommonResources.hs
- MongodbHelpers.hs

2.1 CommonResources.hs

CommonResources.hs contains the following data and variables:

- Data Declarations - Custom data types used by the distributed file system.
- Server Contact Information - IP address and port for all other servers.
- API Definitions - Definitions of the Api's used by the servers, allowing any server to become a client of any other Api.
- Encryption Information - Variables and functions needed by all servers to encrypt/decrypt information.

This file ensures that there are no conflicts between servers in terms of the information stored in this file. This allowed for faster and more robust development of the distributed system as trivial issues such as conflicting data types could not happen when this file is being imported by all servers.

2.2 MongodbHelpers.hs

MongodbHelpers.hs contains methods for accessing and storing information in a Mongodb instance. Mongodb allows for storage of data types in a format similar to JSON, allowing the servers to easily store and retrieve the information they need. Not all of the servers in the distributed file system require the use of data storage and retrieval. The ones who do are as follows:

- Directory Server - Filemappings, FileServers.
- Authentication Server - Storing user login details.
- Locking Service - Storing file lock status.
- Transaction Server - Storing user transaction data and which files are involved in said transaction.

The mongodb functions used in this file were sourced from the following bitbucket project https://bitbucket.org/esjmb/use-haskell. Having the mongodb function in a separate file ensures that the methods remain consistent among all the servers using these functions, making development of these servers easier.

3 File Server

The File Server implemented in this distributed file system is implemented as a flat file system. This means that for each file server connected to the service, each of them will contain one directory in which they can store files. Having the file servers implemented in this way provides an easy way to implement file mappings for the directory server, as files can be represented uniquely via a 'filename:fileserver' key, making ti easy to keep track of files. Another reason for implementing the file server in this way is that it makes the directory servers job easier as it does not need to worry about directory trees in each of the file servers it services.

Files are stored locally on disk in a directory created by the file server which goes by the naming convention of 'fileserver[IPADDRESS]:[PORT]'. The directories were named in this way to create a unique directory name for each of the file servers.

When a file server is execute, the first thing it will do will be to attempt to 'join' the directory server's list of online file servers. It does this by using the 'join' function of the directory server which is detailed in the 'Directory Server' section of this report. When attempting to join the service, each file server will send the directory server it's IP address and port number so that the directory server can store these details in a FileServers mongoDB record. This process assumes that the directory server has to be online before any file server joins the distributed file system service. Assuming a positive response from the directory server, the file server has now joined the service and its files and services are now accessible to clients.

The file server Api can carry out the following operations:

1. getFiles
    - Inputs:
        - No inputs required for this service.
    - Outputs:
        - String array of all the file names in the unique local directory
    - Process:
        - Retrieves all file names inside the unique file server directory and returns them in an array.
2. downloadFile
    - Inputs:
        - ticket - encrypted version of sessionKey with shared secret key.
        - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.
        - encryptedFN - Encrypted file name of the file which is to be downloaded. Decrypted with sessionKey

- Outputs:
  - Encrypted FileName
  - Encrypted File Contents
- Process:
  - Decrypt ticket with shared secret key to retrieve sessionKey
  - Decrypt timeout with sessionKey
  - Check if client session is valid with timeout - If timeout return error
  - Decrypt file name with sessionKey
  - Read in file
  - Encrypt file contents
  - Return encrypted file name and encrypted file contents

3. uploadFile

- Inputs:
  - ticket - encrypted version of sessionKey with shared secret key.
  - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.
  - encryptedFN - Encrypted file name of the file which is to be uploaded. Decrypted with sessionKey
  - encryptedFC - Encrypted file content of the file which is to be uploaded. Decrypted with sessionKey
- Outputs:
  - Encrypted Response
- Process:
  - Decrypt ticket with shared secret key to retrieve sessionKey
  - Decrypt timeout with sessionKey
  - Check if client session is valid with timeout - If timeout return error
  - Decrypt file name with sessionKey
  - Decrypt file content with sessionKey
  - Write to file
  - If all goes well, encrypt successful response
  - Return encrypted response

4. deleteFile

- Inputs:
  - ticket - encrypted version of sessionKey with shared secret key.
  - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.

- encryptedFN - Encrypted file name of the file which is to be deleted. Decrypted with sessionKey

- Outputs:
  - Encrypted Response
- Process:
  - Decrypt ticket with shared secret key to retrieve sessionKey
  - Decrypt timeout with sessionKey
  - Check if client session is valid with timeout - If timeout return error
  - Decrypt file name with sessionKey
  - delete file from local storage
  - If all goes well, encrypt successful response
  - Return encrypted response

4 Directory Server

The directory server acts as both a hub for user requests and an interface for the file servers. All requests to do with file IO (download, upload, file list, ect) are sent to the directory server which then delegates the request tot he appropriate file server.

The directory server keeps track of which file servers have joined to service and which files are stored on which file server through the use of a mongodb instance. The file servers and file mappings are stored in separate records.

For example if a client wants to download a file, the directory server receives this request and searches the filemappings in order to determine which file server the file is located on. The directory server will then download the file from that file server and return it to the client who initially requested it.

In the case of an upload, the directory server will randomly choose one of the file servers it has connected to to upload the file to. The file will be sent to that file server and a file mapping will be inserted into the mongodb instance in order to remember where that file is stored.

The directory server provides the following services:

1. fsJoin
   - Inputs:
     - FileServer IP address
     - FileServer Port
   - Outputs:
     - String Response
   - Process:

- Stores fileserver IP address and port in mongodb record called 'FILESERVER_RECORD'
- IF no errors, return successful Response string

2. openFile
    - Inputs:
        - ticket - encrypted version of sessionKey with shared secret key.
        - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.
        - encryptedFN - Encrypted file name of the file which is to be downloaded. Decrypted with sessionKey
    - Outputs:
        - Encrypted FileName
        - Encrypted File Contents
    - Process:
        - Decrypt ticket with shared secret key to retrieve sessionKey
        - Decrypt timeout with sessionKey
        - Check if client session is valid with timeout - If timeout return error
        - Update "FILEMAPPING_RECORD" by retrieving list of files from each file server via getFiles method from file server Api.
        - Decrypt file name with sessionKey
        - Search "FILEMAPPING_RECORD" for file name - Return error if this fails
        - Send download request to file server (retrieved from filemapping) via file server api function 'downloadFile'
        - Retrieve encrypted file from file server
        - Return encrypted file name and contents to client
3. closeFile
    - Inputs:
        - ticket - encrypted version of sessionKey with shared secret key.
        - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.
        - encryptedFN - Encrypted file name of the file which is to be uploaded. Decrypted with sessionKey
        - encryptedFC - Encrypted file content of the file which is to be uploaded. Decrypted with sessionKey
    - Outputs:
        - Encrypted Response

- Process:
  - Decrypt ticket with shared secret key to retrieve sessionKey
  - Decrypt timeout with sessionKey
  - Check if client session is valid with timeout - If timeout return error
  - Decrypt file name with sessionKey
  - Search "FILEMAPPING_RECORD" to see if file already exists
  - If file already exists we overwrite, else new file written to random file server.
  - Send file write request to chosen file server via file server Api's uploadFile method
  - Retrieve encrypted response from file server
  - Return encrypted response

4. allFiles
   - Inputs:
     - ticket - encrypted version of sessionKey with shared secret key.
     - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.
   - Outputs:
     - Encrypted list of file names
   - Process:
     - Decrypt ticket with shared secret key to retrieve sessionKey
     - Decrypt timeout with sessionKey
     - Check if client session is valid with timeout - If timeout return error
     - Update file mappings by reading file list from all file servers located in "FILESERVER_RECORD"
     - Retrieve all instances in "FILEMAPPING_RECORD"
     - Extract only file names from the filemapping instances into a list
     - Encrypt list of files
     - Return encrypted file list
5. removeFileName
   - Inputs:
     - ticket - encrypted version of sessionKey with shared secret key.
     - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.
     - encryptedFN - Encrypted file name of the file which is to be downloaded. Decrypted with sessionKey

- Outputs:
  - Encrypted Response String
- Process:
  - Decrypt ticket with shared secret key to retrieve sessionKey
  - Decrypt timeout with sessionKey
  - Check if client session is valid with timeout - If timeout return error
  - Decrypt encryptedFN using sessionKey
  - Update file mappings by reading file list from all file servers located in "FILESERVER_RECORD"
  - Search "FILEMAPPING_RECORD" for requested file
  - Send delete request to file server listed in retrieved file mapping using file server Api's deleteFile method
  - Receive encrypted response from file server
  - Return encrypted response to client

5 Authentication Server

The purpose of the authentication server in the distributed file system is to provide both a means for handling user registration and login, as well as acting as a hub for security through the implementation of the three key security model.

The authentication server is connected to a mongodb instance and uses the 'USER_RECORD' to store user login info. Usernames and passwords are stored in this record so that they can be retrieved at login. When a user wishes to login, they will send their username and their username encrypted with their password. Upon receiving this information, the authentication server will check to see if that username is in the 'USER_RECORD' which will return the username and password of the user. The encrypted username is then decrypted using the password retrieved from the user record. If the username and decrypted username match, then we know the user's password is correct and login can continue.

Once the user's username and password have been verified, the security measures are implemented. A sessionKey is randomly generated. This key will serve as the encryption key for all messages the client sends to the servers. The sessionKey is then encrypted with a shared secret key, a key which all of the servers have access to via the CommonResources package. The encrypted sessionKey is stored in a variable named ticket. A session timeout is then, giving the client half an hour to carry out whatever tasks they need to. This session timeout is then also encrypted using the shared secret key. Finally, the sessionKey is then also encrypted with the users password so that only the user can decrypt it. The login process will then return the ticket, the encrypted sessionKey and the encrypted session timeout to the user.

A shared secret server key was used instead of an individual one for each server as it was felt that using a shared key was simpler in order to complete development within the time frame. Individual keys could be implemented easily enough but it would be a time consuming process.

The encryption method used was a simple XOR function, although not the most secure, it provides a simple method for symmetrical key encryption and it was felt that a more secure method was not needed based on the scope of the project. However with more time, a more secure method could be implemented.

The authentication server Api is described as follows:

1. login
    - Inputs:
        - uName - Username of the client.
        - encryptedMsg - The client's username encrypted with their password.
    - Outputs:
        - encrypted sessionKey - (sessionKey encrypted with user password)
        - ticket - (sessionKey encrypted with shared secret)
        - encryptedTimeout - (session timeout encrypted with shared secret key)
    - Process:
        - Search 'USER_RECORD' for username provided by client
        - If not found return error
        - If found, user password returned from record search to decrypt message sent from client.
        - If decrypted message is the same as the username, client has been verified.
        - Generate random sessionKey
        - Encrypt sessionKey with user password
        - Create ticket by encrypting sessionKey with shared secret key
        - Get current time and add 30 mins
        - Encrypt new time with shared secret
        - Return encrypted sessionKey, ticket and encrypted timeout to the user.
2. register
    - Inputs:
        - uName - Client username
        - password - Client password
    - Outputs:

- String Response

- Process:
  - Insert the user's username and password into the 'USER_RECORD' record so that it can be retrieved on login.
  - Return response string if action is complete

6 Locking Service

The locking service provides a method for ensuring exclusive access to files within the distributed file system. Ensuring exclusivity is important as it prevents to users from editing the same file at the same time which would cause conflicts. Any time a client requests to either read or write to a file, a lock request is first sent to the locking service which maintains a mongodb instance 'LOCK_RECORD'. The data stored in this record is the filename and lock status of the file (locked being True and unlocked being False). As of writing, if a client goes offline while a file is locked, that file can become permanently locked, although this may be fixed at a later stage.

Because a flat file system is being used in this distributed file system, it is not necessary for the locking service to lock entire directories because it was felt that locking the files themselves was sufficient for the project scope. Adding support for directory locking would increase the complexity of the logic of the locking server and while possible to implement, it seemed unnecessary and required the locking service to interface with other servers whereas the implemented method allowed the locking service to be a standalone, modular service.

The locking server provides both a locking and unlocking method and are implemented as follows:

1. lockFile

   - Inputs:
     - ticket - encrypted version of sessionKey with shared secret key.
     - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.
     - encryptedFN - Encrypted file name of the file which is to be downloaded. Decrypted with sessionKey
   - Outputs:
     - Encrypted Response String
   - Process:
     - Decrypt ticket with shared secret key to retrieve sessionKey
     - Decrypt timeout with sessionKey
     - Check if client session is valid with timeout - If timeout return error
     - Decrypt file name with sessionKey

- Search 'LOCK_RECORD' for file locking instance
- If no instance found, create a new instance with lock status set to True, and return encrypted positive response string to user
- If instance found, but lock status is already set to True, file locked by another user, return encrypted negative response
- If instance found and file status is Unlocked, file is available to lock, set locking status to True and return positive encrypted response.

2. unlockFile

   - Inputs:
     - ticket - encrypted version of sessionKey with shared secret key.
     - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.
     - encryptedFN - Encrypted file name of the file which is to be downloaded. Decrypted with sessionKey
   - Outputs:
     - Encrypted Response String
   - Process:
     - Decrypt ticket with shared secret key to retrieve sessionKey
     - Decrypt timeout with sessionKey
     - Check if client session is valid with timeout - If timeout return error
     - Decrypt file name with sessionKey
     - Search 'LOCK_RECORD' for file locking instance
     - If no instance found, return error as this should never happen.
     - If instance found, but lock status is already set to False, file unlocked by another user, return encrypted negative response
     - If instance found and file status is locked, file is available to unlock, set locking status to False and return positive encrypted response.

7 Transaction Server

The purpose of the transaction server is to allow users to make changes to multiple files at once. It does this in such a way that temporary copies of the files they wish to edit (denoted by TMP FILENAME) are created on the file servers. The non-temporary versions of the files are only overwritten when the user is finished with all files involved in the transaction and is ready to commit the changes they have made. When the user commits these changes, the non-temporary files are overwritten and the temporary files are deleted.

The transaction server keeps track of individual transactions through the implementation of two mongodb instances: 'TRANSACTION_ID_RECORD' which keeps track of which users are currently engaged in a transaction by storing their sessionKey (unique to each user), and the 'TRANSACTION_FILE_RECORD' which stores information about each file involved in a users transaction by mapping the sessionKey to the filename.

Currently, the transaction server only allows to client to edit already existing files and does not allow for the uploading of new files. The ability to upload a new file in a transaction could potentially be implemented but it's implemented was decided against in order to complete the project within the time frame.

The transaction server Api has the following functionality:

1. beginTrans
    - Inputs:
        - ticket - sessionKey encrypted with shared secret key
        - encryptedTimeout - The client's session timeout encrypted with shared secret key.
    - Outputs:
        - Encrypted Response String
    - Process:
        - Decrypt ticket with shared secret key to retrieve sessionKey
        - Decrypt timeout with sessionKey
        - Check if client session is valid with timeout - If timeout return error
        - Insert the client's sessionKey into the 'TRANSACTION_ID_RECORD' in order to register their transaction.
        - Assuming all goes well, return encrypted positive response string
2. downloadTrans
    - Inputs:
        - ticket - encrypted version of sessionKey with shared secret key.
        - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.
        - encryptedFN - Encrypted file name of the file which is to be downloaded. Decrypted with sessionKey
    - Outputs:
        - Encrypted FileName
        - Encrypted File Contents
    - Process:
        - Decrypt ticket with shared secret key to retrieve sessionKey
        - Decrypt timeout with sessionKey

- Check if client session is valid with timeout - If timeout return error
- Decrypt file name with sessionKey
- Retrieve the file from the file server using the directory server's 'openFile' Api method.
- Assuming successful retrieval of the file, store the user's sessionKey and the file name in the 'TRANSACTION_FILE_RECORD' record to register that the file is part of a the user's transaction
- Return the encrypted filename and the encrypted file content to the user

3. uploadTrans

- Inputs:
  - ticket - encrypted version of sessionKey with shared secret key.
  - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.
  - encryptedFN - Encrypted file name of the file which is to be uploaded. Decrypted with sessionKey
  - encryptedFC - Encrypted file content of the file which is to be uploaded. Decrypted with sessionKey
- Outputs:
  - Encrypted Response
- Process:
  - Decrypt ticket with shared secret key to retrieve sessionKey
  - Decrypt timeout with sessionKey
  - Check if client session is valid with timeout - If timeout return error
  - Decrypt file name with sessionKey
  - Create the temporary file name using the decrypted file name in the form of 'TMP FILENAME'
  - Upload a new file to the file servers using the directory server's 'closeFile' method using the encrypted temporary filename and the encrypted file content
  - Assuming all goes to plan, return encrypted positive string

4. CommitTrans

- Inputs:
  - ticket - encrypted version of sessionKey with shared secret key.
  - encryptedTimeout - Can be decrypted with sessionKey. Used to determine if user session has timed out.
- Outputs:

- Encrypted Response

- Process:
  - Decrypt ticket with shared secret key to retrieve sessionKey
  - Decrypt timeout with sessionKey
  - Check if client session is valid with timeout - If timeout return error
  - Using the sessionKey, select all instances from 'TRANSACTION_FILE_RECORD' relating to the user's sessionKey.
  - For each instance returned, use the returned filename to create the temporary file name for each file.
  - Use the temporary file names to send a download request to the directory server's 'openFile' method, returning the data stored in the temporary file.
  - Use the file content returned to overwrite the original file using the the directory server's 'closeFile' method
  - Delete the temporary file from the file server using the directory server's 'removeFile method
  - If all goes well for all files involved, encrypt successful commit response
  - Return encrypted response

8 Caching

A cache is implemented on the client side of the distributed file system in order to provide quick access to commonly used files. The cache implemented here uses key, pair values to match file names to file content and is stored in memory in a hashed queue. The cache is instantiated upon client login and is terminated upon closing the application, thus the cache is not persistent for multiple sessions.

The cache implemented in this distributed file system is a least recently used (LRU) cache, meaning if the cache is full and a new file needs to be inserted into the cache, the file in the cache which was least recently used is removed, and the new file is inserted into the cache.

The cache implements a queue in order to fulfill its duties as an LRU cache, when a file is inserted it is inserted to the bottom of the queue. If the cache is full, the file at the top of the queue is removed to allow space for the incoming file.

The cache contains three simple methods:

1. ioinput

   - Input:
     - Key - In this case a file name

- value - The file content
- cache - The cache itself (hashed queue)

- Output:
  - No output provided
- Process:
  - Insert the file name and file content as a key pair value in the cache

2. iolookup
   - Input:
     - Key - In this case a file name
     - cache - The cache itself (hashed queue)
   - Output:
     - Maybe value - The file content if found
   - Process:
     - A lookup function is used to search the cache for the filename requested via the key.
     - If the file is already in the cache, the file content is returned to the client
     - If it is not found, the lookup function will return Nothing
3. iogetContents
   - Input:
     - cache - The cache itself (hashed queue)
   - Output:
     - List of file names (keys)
   - Process:
     - Return all key values in the cache to the client

9 Client Proxy

The client proxy is a client side program which is executed by any user wishing to access the distributed file system. It provides a text-based interface to the distributed file server and provides instructions to the client in order for them to carry out their desired tasks

Upon executing the program, a client will be given the choice to login or register for the service. If the client already has an account, they interface with the authentication server's 'login' method to verify login and to retrieve the encryption keys the program will need. If the user does not have an account, they will be asked to enter a username and password combination which will

be sent to the authentications server's 'register' method via the authentication server's Api. Upon approval of the new account, the user will return to the login/register screen so that they can login with their new account.

After login, the cache is initiated and the client moves to the main program loop where they are provided with a number of options in terms of which tasks they'd like to carry out.

1. Files
2. Download
3. Update
4. Transaction
5. Close

At the time of writing, it is planned that a 'Delete' option will be available as the functionality is there however it is not implemented on client side.

9.1 Files

Selecting the 'FILES' will send the encrypted timeout and ticket to the directory server Api method 'allFiles'. This method will return a encrypted list of all file names currently in the distributed file system, which is the decrypted and printed for the user to read. The user will then be returned to the program's main loop.

9.2 Download

When a user wishes to download a file from the distributed file system, the cache first makes sure that all the files inside of it are up to date by using the cache's iogetContents method which will return all file names in the cache. This list of files are then downloaded using the directory server's 'openFile' method to download the latest file content for each file in the cache. The caches ioinsert method is then used to update each of the files with the updated file contents.

Next, the client is asked to enter the name of the file they wish to download. The client will then request a lock on the file via the locking service's method 'lock'. If successful, the cache is searched for the file the user wishes to download via the iolookup method. If it is found, we download the content from the cache and if it is not found, the directory server method 'openFile' is used to download the file.

Upon download from either the cache or directory server, the file is saved in a local folder named 'localstorage' and is immediately opened in a vim window, allowing the user to open the file and edit the contents. Upon exit of the vim window, the user is asked to confirm that they're finished editing the file, after which they are asked if they wish to upload the changes. If they select no, the file remains in localstorage. If they say yes, the file is uploaded to the directory

server via the 'closeFile' method and written to the cache via ioinsert.The file is then unlocked via the locking service, and the user will return to the main program loop once again.

9.3 Upload

If a client wishes to upload a file, they will be asked to enter the name of the file they wish to upload. This file name will then be locked via the locking service to ensure exclusive access. A vim window will then be opened using the filename given by the user. If a file of that name is already located in the local storage, that file will be opened and its content displayed in the window. This allows users to upload files created in a previous session. Upon closure of the vim window, the client will be asked to confirm that they are finished with the file After which the file content will be read into into a variable. Both the file name and file content are then encrypted using the session key and sent to the directory server via the 'closeFile' method. Assuming this is successful, the file is the inserted into the cache via the ioinsert function and the file is finally unlocked via the locking service, upon which the user will be returned to the main program loop.

9.4 Transaction

When a user wishes to carry out a transaction, a transaction initialization request is first sent to the transaction server so that the transaction server knows that the user is currently carrying out a transaction.

Once the transaction has been initialized successfully, a file list is then pulled from the directory server and the user is asked to enter the names of the files they wish to download in the transaction delimited by a comma. The file names are then read into a list and a lock is attempted on all of them. The transaction will only continue if all of the locks are successful.

If the locking is successful a new 'tmp' directory is created and one by one each file is downloaded via the transaction server's downloadTrans into this directory. A vim window will open for the first file downloaded, allowing the user to edit and save their changes. After saving the changes, the file is then uploaded via the transaction server's uploadTrans method which will store the changes in a temporary file. Once the user is finished with the first file, the vim window is closed, the file is stored in the cache and a new vim window will open for the next file. This process continues until all files have been edited and uploaded.

Once all file changes have been stored in temporary files on the file servers, the changes will be commited via the transaction server's 'commitTrans' method. It should be noted that at the time of writing, it is planned that a user will have the option of aborting the commit at this stage, however this is not yet implemented. After all of the files have been commited, they are unlocked via the locking service and the user will return to the main program loop.