DISTRIBUTED SYSTEMS PROJECT REPORT
Sean Cowan
13325878
scowan@tcd.ie

My distributed file system is written is a simple working file system based on a RESTful architecture with simple API calls. You can upload and download a file from a file server that backs up copies of files to other servers. A user locks and unlocks a file when the download and update the file, so as to avoid race conditions, meaning one user can't overwrite to work of another user.

The design is simply a Directory Service that the user queries for files. When the directory receives a query is returns the ip address of the server with the primary copy of the file and the client downloads the file from there. The file is locked. When the user reuploads a file the file is unlocked and stored back on the correct server.

Distributed Transparent File Access:
The system supports an upload, download file access style. A file is uploaded by a user and when it is downloaded it is locked until it is reuploaded again. The user is shown a list of the files within the system but not told about the server outside of the haskell client package supplied.

Security Service
There is a mechanism for the registration of users using a username and password in the "Auth-Server" haskell stack package. It has a simple endpoint layout for an easy to use API.

Directory Service
The Directory Service Keeps track of all the files and the server on which they are located. It also keeps track of the "groups", which are the groups of servers composed of a single primary server holding the primary copies and the backup servers holding the backup copies. A user interfaces with the directory service api with calls to file/get and file/add so they can easily find out which server they are supposed to store files on. When the Directory Service receives these API calls it correspondingly locks and unlocks the files as needed.

File Servers also interact with the Directory Service. They use calls to creategroup, addmetogroup and makemeprimary to do this. The creategroup call orders the Directory service to create a new group and make the FileServer the primary of that group. The group will have no backups in this situation. The addmetogroup call orders the directory service to add the file server to some group. The Directory Service adds the File Server to the group with the least amount of backups. This helps achieve load balancing. The makemeprimary call is to allow the backup File Servers to detect that the primary has gone down for some reason and ask to be made the primary.

Replication
There are facilities for replication and back up servers for replication are included in the File system and directory service packages but they are not used.

Caching
There is no caching.

Transactions
Due to the files being stored in a database using persistent and sqlite all calls the the database are atomic which should stop errors related to half complete transactions from occurring. However certain things like if a user downloads a file and locks it and then never re uploads it that file is locked forever. This would need to be fixed with some kind of pinging to the user or just having a simple timer that the user is made aware of so they know they need to return the file in 3 hours.

Lock Service
The Directory Service controls locking. It keeps track of when users upload and download files and locks and unlocks the accordingly.

Building the System:

The main difficulty I encountered when building the system was learning how to use haskell, stack and servant. As I had not taken the functional programming class in third year I found the concept of monad and monad transformers as seen in the Handler type in servant to be confusing. I would make progress but then get stuck on something trivial like using a random integer in a function because that random integer is actually an IO Int and can't be used like a normal integer. I would eventually overcome these difficulties through reading a book on haskell and I also found the paper "Monad transformers step-by-step to be helpful". Learning stack wasn't too difficult, it just required carefully selecting the correct versions of servant and other libraries which caused me the occasional difficulty but I managed to get by. It took me about three weeks to become really proficient with Servant. The advanced type system being used with strings being part of types, and data kinds etc would often mean I had to do more research and I would get "caught in the weeds" and start researching mostly irrelevant details about haskell's type system that I probably could have safely ignored.

Once I became proficient with haskell, servant and stack it became relatively trivial to add functionality. With servant I didn't need much boiler plate and could essentially define and API, write the function for the api and it was done. With haskell's advanced type system I rarely made errors that weren't picked up by a "stack build". I believe if I write this in a language like python of java the code would be larger by a factor of 3 or 4. If I was writing it again I would have had less separate stack projects, and had the APIs in one shared repository. This would have allowed me to repeat myself less or not at all. I also would have planned out the system completely, in even greater detail than I did. I think this would have let me get work done faster because I would know exactly what I needed to build. Overall I think this project went well despite the hiccups involved in learning new technologies.