Faculty of Natural & Mathematical Sciences

7CCSMGPR – Group Project

Assignment 1 of 1 – Final Report (Default Team)

Authors
Alain Doebeli
Christakis Vasiliou
Ignacio Dominguez Garcia-Guijas
James Kayongo
Konstantinos Oikonomou
Nikolaos Rizos

Module Tutor
Dr. Laurence Tratt

March 25, 2019

Contents

1 Introduction

The advent of distributed file systems and the increasing computing power of portable computing devices in forms of tablets, smart phones and laptop has since changed the work place environment from the traditional desktop computers stationed in office building or the home office. In todays world a lot more work is done on the move away from office building through the use of interconnected portable computing devices. The portable devices mostly have ad-hoc or on demand connectivity to the internet and the applications used on these devices are data-centric. This created a number of unique challenges to this new computing paradigm of distributed file systems these include; the files synchronization problem, file storage the fact that portable devices have limited storage capacity and can therefore can only store limited files depending on their storage capacities as compared to their desktop counterparts, getting file access and maintaining files across the numerous devices platforms and how to deal with the file updating, conflicts and versioning.

This project implements a file synchronizer application designed to address the aforementioned challenges. The application architecture is designed to provide the most efficient methods of providing the required functionality and support different computing platforms. The architecture design decision points included; cross platform development framework, file synchronization method, file sizes and types and endpoint device capabilities. The implemented application provides file sharing of different file sizes and types, file access across different devices such as desktop computers and mobile computing devices, file updating, versioning and conflict handling, file storage both remotely (in the cloud) and locally on device storage, and lastly multiuser functionality. The application implementation in the interest of delivering the solution on time focused on achieving the core functionalities first and a few other desired functionalities that enhance the application usability.

2 Review

The File synchronization is a critical component to the current distributed file systems computing and is therefore a widely researched field with numerous products both commercial and open source solutions with fully em-

bedded feature sets. A comparison study of file synchronization to determine the best file synchronization methods to solve a file transferring problem between student tablets devices and education institutional hubs provides a comparative comparison of the different file synchronizations tools Rsync, DropBox, HadoopRsync, Syncany and Unison [1]. According to this study HadoopRsync an open source framework is most suitable for meeting the aforementioned requirement. First because it can be scaled to a very large network of users and the HDFS framework used for storing data across numerous nodes provides resilience to data node failures. Secondly HadoopRsync uses Rsync [2] which provides the best file synchronization method because it handles file changes with minimal transfer overhead through tracking file and block lists both locally and remotely, transferring only the changes and reconstructing the file at the destinations. This technique was adopted and implemented in this projects file synchronizer. HadoopRsync uses peer-to-peer file synchronization technique, client-to-server technique is used for this project because its implemented over http to provide reliable delivery and flow control.

In todays world, the number of user managed devices continues to increase and the need for synchronizing multiple file hierarchies distributed over devices with ad hoc connectivity is becoming a significant important issue. [3] This paper reviews cloud file synchronizer architectures such as Google Drive, Dropbox, SkyDrive, iCloud, ReaiSync, Box identifies issues relating to availability, security, privacy and trust and proposes a product Cloud Enabled File Synchronizer that addresses the identified shortcomings in a single solution. This project design architecture is similar to cloud file synchronizer architectures reviewed in this work.

The file synchronization problem is a well-known challenge faced in maintaining huge chunks of files across distributed environments. Rsync [1][3] has been proposed as the suitable method and implemented widely in applications such as personal user files, business files, remote backups, mirroring of large webpages, ftp sites and content distributions. This paper [3] however, proposes improvements to Rsync in solving the synchronization problem when dealing with very large chunks of files synchronization over slow connections.

3 Requirements and Design

In the following section the requirements that came from our study of the problem will be discussed as well as the architecture and design we are going to use to satisfy them.

3.1 Use Cases

After considering the problem and different possible solutions we came up with the following use cases.

- User can create an account
- User can log in into his account
- User can log out from his account
- User can delete his account
- User can see his files that are both local and online
- User can upload a local file to the server
- User can download a file from the server
- User can update a file on the server
- User can delete a file both locally and on server
- User is notified if there is a conflict
- User can handle a conflict if it occurs

3.2 Architecture Design

We decided that the best way to deliver the above functionality to the user will be through a client server architecture. In particular the client will communicate with the server in the back to send and receive data. This allows for modularity because we can have different clients in different platforms talking to the same server. Specifically, as shown in the figure we are going to have a mobile app and desktop app clients that would provide users with a graphical user interface to interact with the server. The server which runs on a Linux operating system will store all the files and users

information into a database and the files themselves will be stored in the Linux file system.

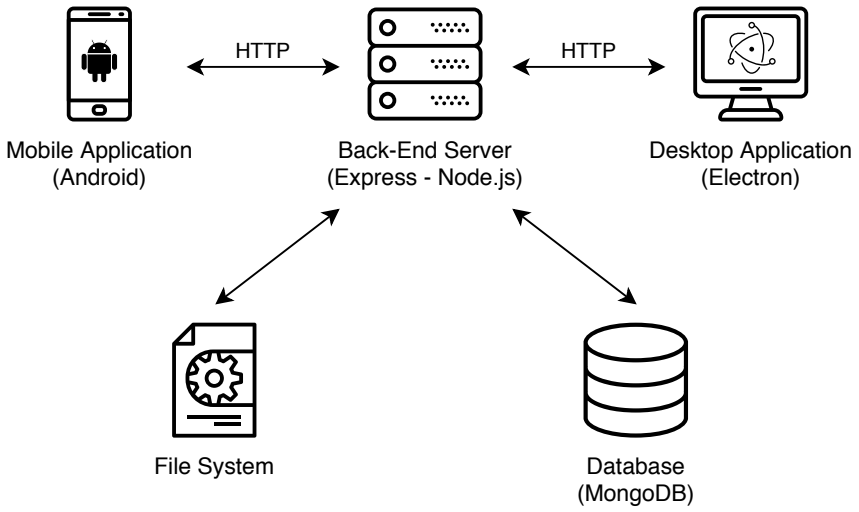

Figure 1: An illustration of the architecture design

3.3 Project Objectives

We set the following objectives for the implementation of the system. Specifically, what needs to be implemented for the clients and the server and the priority of each objective.

Objectives	
Server	**Desktop/Mobile**
Set up Node.js app	Set up Electron and Android apps
Receive & save a file through HTTP	Create the GUI
Check for conflicts of a file being uploaded	Show files in a local dir
Send file information through HTTP	Request file information from server
Send latest file version through HTTP	Push file changes to server
Deal with possible conflicts	Pull outdated files from server
Implement different users	GUI and functionality for multiple users
Implement permissions amongst users	Allow users to give permissions

Table 1: Colour-Coded Objectives

In the figure we can see each one of the objectives having a colour that

states its priority. Green objectives represent the bare minimum functionality of the corresponding component and should be completed first. Yellow objectives represent important functions of the subsystems and should be done immediately after. On the contrary, red objectives represent features that are not essential for the system to be operational but could potentially be added to expand the services it provides. Finally, blue objectives are quality of life improvements that will be implemented only if the quality of the system has been thoroughly tested and time allows for their development.

All the functionality above will be able to handle almost any kind of file with the exception of directories. We decided that it would be best not to support the synchronization of directories as it would be too complex for us to store the structure of directories and implementing it within our time constrains for the project. Also having directories could cause performance issues as a user may create infinite number of sub directories.

3.4 Conflicts Handling

We discussed thoroughly about what is the best way to go with handling the conflicts and how to detect them in the first place. Firstly, for the detection of a conflict we came out to the conclusion that the following occasion should qualify as a potential conflict. If a user tries to upload a file without already possessing the latest version of that file will qualify as a potential conflict because it means that someone else has updated the latest version before he had the chance to do so and therefore his version will be different.

Now to handle that conflict as soon as a user tries to upload a file that causes the conflict, we will give to him three options. To either upload the file anyway and replace the latest version or just get the latest version first or a third option that a diff file will be presented to him so he can handle the conflict on his own. The diff file will display to him the differences between the version he is trying to upload and the current latest version on the server.

3.5 UI Design

We have attempted to do the user interface as friendly as possible both for the mobile and desktop applications, implementing the latest guidelines in

material design. Also, we made the two client interfaces as consistent as possible to ensure the user can adapt between the two platforms.

4 Implementation

4.1 Desktop Application

Why we chose Electron

The desktop application was developed on the Electron JS platform. Electron uses Chromium and NodeJS, so the application was built using HTML, CSS, and JavaScript. Electron offers great cross platform compatibility and with a single code base for all the major platforms, the packaging and distribution to Mac, Windows, and Linux, is easy. User Interface (UI) and User Experience (UX) can be the same to all platforms and are easy to handle since it is mainly web design. It provides extended interactive features such as keyboard shortcuts and low-level accessibility to the hardware and operations system components. Electron separates clearly the components by making use of separate processes for each web page and a separate, main process that is used to interact with any backend/server systems.

When opening the application, the login form is shown to the user. After the successful login, the user is prompted to select the folder that will be checked and synchronized with the server application. With the folder selection, the process of reading the local files within the folder starts. First, we iterate through the local files through the folder chosen and we get some statistics like name, size and their hash and we save all the information in an array of json objects for easy access. Following this, a `GET` request - containing the users cookie as received from the login - to the server is sent to retrieve the logged in users files that are uploaded to the server. When the response is received, the received information is compared against the local files array in order to check their status. The server response contains a hash of the content and the file version that is uploaded on the server. Each file is represented as an object with attributes that include: flags showing if the file is synchronized and whether the content is different, and the version number. These flags make both the functionality and the visualization easier as based on them, the appropriate options and icons can be given to the user.

Subsequently, the files list is rendered on the screen. For all the files the

user has the option of deleting them either locally, or from the server or from both sides. For the files that exist only locally, the user has the option of uploading them to the server, and likewise, if the files exist only on the server, the user has the option to download them in the folder specified in the application start. In case the files exist both locally and on the server, there are multiple cases:

- The files have the same version and the same hash content: in this case the file is up to date and the user does not have any options available except for opening the file with the specified default program for opening the files of this filetype in the OS
- The files have the same version but different hash contents: in this case the file was changed by the user locally, so he gets the option to upload the file to the server
- The files have different versions: in this case, someone else has modified and uploaded the file on the server, and the user has a different version of the file. As a result, he gets the options of forcing uploading or downloading and overwriting the contents of either the server of the local file. In case of a simple text file, the option to see the file differences in a new window is given to the user

The option to logout is accessible from every screen. Upon logging out all local storage kept for the application is deleted. *This was decided mainly as the process of dealing with multiple users wanting to synchronize files from the same folder would require a different approach to the way we save files in the server and the way we store information about files status locally.* Finally, the user has the option to delete his account.

Libraries

The request library [4] was used to handle all the API requests to the server as it offers cookie handling, response streaming to any file type as well as supporting both promises and callback interface natively. In order to use the request library easier, the util module from NodeJS was also used in conjunction to it, in order to handle the requests as promises and not dealing with callbacks.

The `fs` module [5] of NodeJS was used to interact with the file system as we needed to handle files (open/close/delete/write) and read statistics about

them.

The `crypto` module [6] of NodeJS was used to create the local files sha256 hash in order to use it to compare it with what the server sends back so we can check quickly for differences.

The electron json storage library [7] was used to persist and read user settings for the application. Upon logging in, the selected folder was saved as a setting so as the user wouldnt have to select the folder again unless he logged out.

The `isbinaryfile` module [8] was used to check if a file is binary or simply a text file so that we check if we can show the file differences simply in the app or not.

`Diff2html` [9] was used to visualize the differences between the local and server files in case of simple text files.

Bootstrap [10] is the most popular HTML, CSS and JS framework for responsive web development. It has predefined design templates and classes that can be used where they fit and is fully customizable. Moreover it offers a great responsive, mobile grid system that is easy to use and can be adjusted for showing or hiding content based on screen size. Last but not least, Bootstrap is a framework that has uniform and consistent results in every platform and browsers (Internet Explorer, Chrome, Firefox).

4.2 Mobile Application

Mobile Application Implementation

For the implementation of the mobile app the android studio [11] is used. The reason we choose to use android studio its because is easier for the mobile app to access the mobile local files in an android phone. It is for same reason that also hybrid development framework could not be used as it would be difficult to handle local file on both operating systems using the same codebase. Also, some external libraries where used to help do things like sending requests to the server which we will discuss later on.

Implementation logic

Generally, the mobile app will use the internal storage of the device to store information needed for the version control of the local files. The actual local files of the user will be stored in an external storage directory called WorkingDirectory which is created the first time the user lunches the app. Also, during the first lunch the app will ask the user for the necessary permissions to access the local file system and use the internet to send requests to the server.

If the user is not logged in the usual logic screen will appear to him with also the button that will direct him to another form to register if he does not already have an account. From the point that the user logs in he will remain logged in until he chooses to log out and for every time he opens the app he will get the main screen of the app. The main screen of the app will contain a list of the following:

- All the file names
- Show which files are only stored locally on the phone
- Show which files are only available online on the server
- Show which files are both online and locally and are updated
- Show which files are both online and locally places and are not updated
- Next to each filename a button with the appropriate action if there is
- Next to each filename a delete button

To obtain the list above the following implementation logic is used:

- Save all the file names from the local storage in a list
- Save all the file names you get from the server in a list
- Compare lists to find out which files are only local which only online and which on both
- Put them in a single list with a code field with value depending on their status
- Initiate the adapter that will create the list of files
- For each file do the following

- Check the status of the file
- If file is local
    - Show local file symbol next to file name
    - Initiate an upload button that on click does the following
        - Uploads the file
        - Saves the file version as 1
        - Refreshes screen
    - Initiate a delete button that will trigger the delete file dialog
- If file is only online
    - Show online file symbol next to file name
    - Initiate a download button that on click does the following
        - Downloads the file from server
        - Saves the file version as the version that is stored on the server
        - Refreshes screen
    - Initiate a delete button that will trigger the delete file dialog
- If file is on both places
    - Initiate a delete button that will trigger the delete file dialog
    - If stored file version is different from the version stored on server
        - Show the not updated symbol next to the file name
        - Initiate a conflict button that will trigger the conflict dialog
    - If stored file version is the same as the version stored on server
        - If local file hash is different than the file hash stored on server
            - Show the not updated symbol next to the file name

- ◇ Initiate a synchronise button that will
  - ⋆ Upload the file
  - ⋆ Increment the saved file version by 1
- · If local file has the same hash as the hash stored on the server
  - ◇ Show the updated symbol next to the file name

As you can see from the logic, a local file is kept on private storage that will contain all the file versions of the local files which are compared with the file versions stored in the server to determine if a file is up to date or not and what action is required. Then the versions will be updated on the local file based on the action performed by the user. That way we can detect potential conflicts that will occur if a user tries to update a file that has a more recent version stored on the server. When a conflict is detected as soon as the user initiates the action a conflict dialog will appear that has three options. Either for the user to force upload the file, just download the file version or get a diff file that will be generated and saved on the local storage with the same name as the file the user tried to upload. The logic for those actions is similar to the logic discussed above. Also, for the delete dialog the user will always have the option to delete the file either from the server or from local storage or from both places with the version of the file also deleted from the local information file.

Libraries

Now lets discuss the different external libraries that were used

OkHttp

OkHttp [12] is a library available on GitHub that allows you to perform get and `POST` requests on a server. The reason we used this library is for its simplicity into the building of the requests and that it allows you to easily carve multipart `POST` requests that accept a binary file in the `POST` requests body that we will use to upload the files to the server.

PersistentCookieJar

PersistentCookieJar [13] is an extension to the OkHttp library that allows you to easily capture session cookies and reuse them in your OkHttp re-

quests. The main reason we use it for its simplicity that will prevent us from having to manually process the session cookies.

Differences between mobile application and desktop application

- Mobile Application saves the generated diff file with the same name as the file the user tried to upload but desktop application just shows the diff file to the user on screen
- Desktop Application shows information about each files size while the mobile application does not
- In the desktop app the user has the option to choose a local directory for his while in the mobile app a default directory is created that is always the same

4.3 Server Application

Why we chose NodeJS

The server application was built using NodeJS. One of the main advantages of NodeJS is the ability of building fast and scalable network applications. It is able to handle big amounts of simultaneous requests due to its non-blocking and asynchronous architecture. Each connection is handled in the same thread using non-blocking I/O calls which means that one request does not have to wait until the previous one is finished to be executed. This is crucial for the efficiency of the server application as writing files to disk or transferring them through HTTP can take long amounts of times. So performing various requests at a time will decrease the response time for each request.

Another great advantage of NodeJS is the Node Package Manager (NPM). This package manager allows to reuse components and helps with managing the versions and dependencies of the components. NPM is public and anyone can publish their modules there for other people to use.

Our team also considered that NodeJS uses JavaScript which will also be used in the desktop application making it easier for all the team members to adapt from one to the other.

Together with NodeJS it was decided to use MongoBD as the database to store the user and file information. This decision was made taking into consideration that MongoDB is a non-relational database that stores the documents as json objects which makes it very convenient to work with JavaScript in the backend and send the information through HTTP.

We wanted to have a common server and database to make developing and testing easier, the server is hosted in an Amazon Web Services EC2 server running Ubuntu. Nginx is being used as a reverse proxy that redirects traffic from port 80 to our NodeJS application. Although the database could be hosted in this server too we decided to host it in an mLab [14] server as the AWS server has very limited disk space.

API description

The server application provides end points for the client-side applications to send or receive information. There is a total of 8 end points which will be explained in this section although further explanation can be found in the `docs.md` file inside the server project folder. These 8 end points have been implemented in 2 different files `/src/routes/users.js` for all the user account related functionality and `/src/routes/files.js` for the file related functionality.

A new user account can be created by calling the sign-up function which has the end point `/user/signup`. This function requires a `POST` request with the parameters username and password. The function checks that the username has not already been taken by another user and saves the username and a salted hash of the password in the database.

User log in is done by calling the `/user/login` end point this function requires a `POST` request with the username and password the hash of the password is recalculated and compared to the one stored for that user. If the authentication is successful a session cookie will be sent in the response. The client is expected to use this cookie to identify himself later.

Users can logout of the system by calling the logout function at the end point `/user/logout`. This function receives a `GET` request from an authenticated user (the session cookie must be included in the request) and will invalidate that session and logout the user.

To delete a user account the user has to send a `GET` request will being logged

in to the `/user/delete` end point. This function will invalidate that session and delete his user entry in the database.

The rest of the API functions deal with the file synchronization part of the system. For all of them the user must be authenticated by sending the session cookie in the headers of each request. In order for the user to know all the information about the files that he has uploaded to the server the end point `/file/getInfo` can be called with a `GET` request that will return a json array with all the information about that user's files. This function accepts a `filename` parameter which acts as a filter to only get the file information for one file. The file information includes filename, server version and the hash of that file among other useful information.

The push function implements a lot of our system functionality. It can be called through a `POST` request and receives 5 parameters depending on what the user wants to do. The first parameter is `file` this is an actual file that the user wants to upload to the server. If `file` is included the user either wants to upload a new file or update an existing one. To distinguish between these two scenarios the database is checked to see if this user already has a file with that filename. If it is a new file, the information about the file will be stored in the database and the file stored in the file system. In the case that the file already existed the hash will be checked to detect if the file has been modified. If it has not been modified there is no need to save the file, so an error will be sent to the user. If the hash is different the server must ensure that the version of the file that the user modified was the latest in the server to avoid conflicts. In order to do this the user must include the `version` parameter with the version number of the file that he modified, and the server will check if it is the latest version. If it is not the latest version the server will respond with the error code 409 Conflict indicating that there is a conflict which he will have to resolve. If there is no conflict the server will save the new information for that file, increase the version number and update the file in the file system. The other main functionality of this function is to delete files from the server. To do this the `file` parameter will be ignored, the `delete` must be set to true and the `filename` parameter to the name of the file to be deleted. The server will check the version number provided in the request and if there are no conflicts delete the file from the file system and the database. The `force` parameter is used to ignore the version and therefore the conflicts. This is useful to resolve conflicts in the case where the user wants to overwrite the latest version with his version.

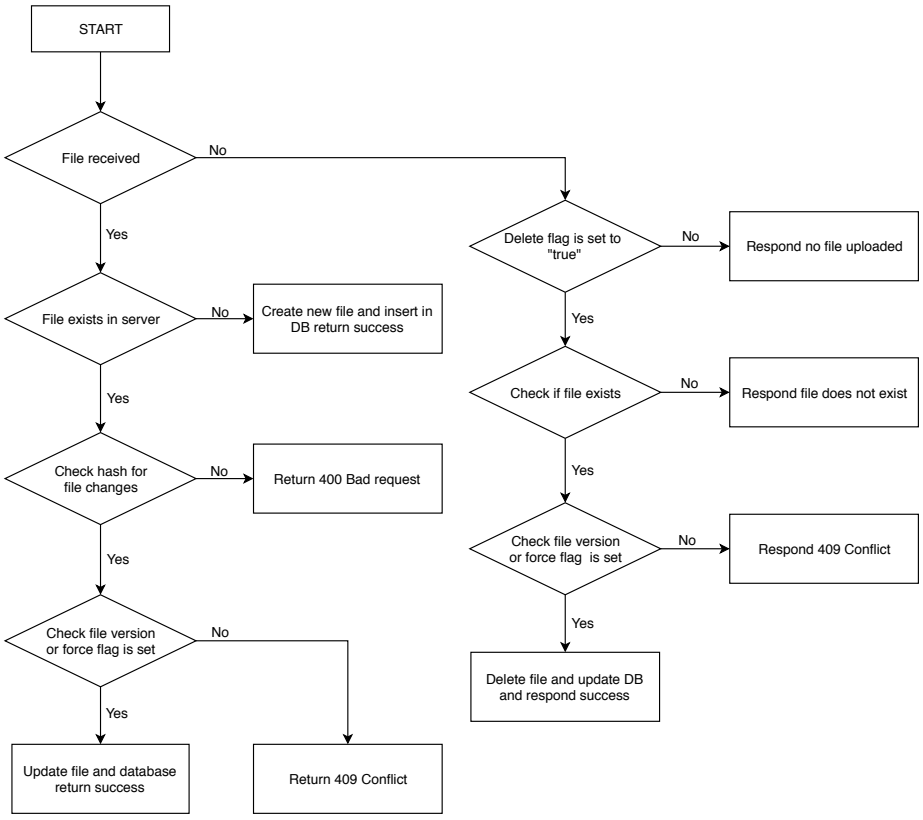

Figure 2: Flow chart of the server implementation logic

Once the user gets the information from the files he has stored in the server he might want to download them in order to do this he can use the download function located in the end point `/file/getFile`. This function receives a `GET` request with the parameter `filename` which indicates the file the user wants to download. If the user has previously uploaded a file with that filename the file will be sent back in the response.

In the case of updating a file and receiving a response stating that there is a conflict the user can decide to discard his changes and download the latest version, use the push function with the `force` flag set to true or use the get differences function located in the `/file/getDiff` end point. This function receives a `POST` request with the `version` and the `file` that has the conflict. The server checks that indeed there is a conflict with that file by checking the version numbers and checks if the file is a text or binary file.

If it is a text file it then generates a diff file similar to what `git diff` would generate. The content of this file is sent on the response and the client can easily resolve the conflicts in the file and then use the push function normally to update the file taking into consideration the latest version of the file.

Libraries

One of the greatest advantages of NodeJS is the huge number of modules available through NPM which speed up development and help make code more reusable. The main module that the application requires is ExpressJS [15], express provides functionality to set up a http server and handle the requests. MongooseJS [16] has been used as a object modeling tool for MongoDB, it allows for easy creation of models for the different objects stored in the database and provides an asynchronous way to make the database interactions. Morgan [17] has been used to log all the requests done to the server and all the possible error that can occur. HTTP-errors [18], helps creating meaningful HTTP errors for express. PassportJS [19], allows to authenticate users in different ways including google, facebook or twitter accounts. In our case we used it together with connect-mongo [20], express-sessions [21] and cookie-parser [22] to do the authentication of users with accounts in our own database. Express-fileupload [23], helps managing files uploaded through HTTP making it easier to save them and collect information about them. To do the SHA256 hashes of the files the application uses bcryptJS [24]. Finally, in order to generate the diff files when handling conflict `isbinaryfile` [8] is being used together with jsDiff [25].

5 Testing

It was decided to opt for a black box testing [26] approach. Black box testing, also known as functional testing, places a focus on discovering erroneous functions in an application — all without knowledge of the underlying code. Figure 3 below illustrates the black box testing approach.

Figure 3: An illustration of the black box testing approach

Test cases, derived from an analysis of the requirements specification, act as the input. An input can be clicking or typing something specific, as defined by a particular test case. The output is the result of such action, which is then verified against the expected outcome. A test case was marked as "PASS" when the outcome of an output upon a given input matched the expected outcome. Conversely, an unexpected outcome would have concluded a "FAIL", of which there were none.

A dedicated table for each of the three applications, containing a list of test cases associated and tailored to each particular application, can be found in Appendix A. The test cases were further organised into categories, based on the features they test. Both the desktop and mobile application test suites had their test cases categorised into: *packaging/compilation and launch*, *user sign-up*, *user log-in*, *file management*, *user log-out*, and *user delete*. The test suite for the server application, responsible for the communication between the desktop and mobile application, only has one category as there is nothing more to it: *connections and requests*. Its functionality is already implicitly tested through the desktop and mobile application.

All of the testing was conducted by every group member in order to ensure that there were no oversights or that potential bugs would not slip through unnoticed. Furthermore, as black box testing does not presume any programming or implementation knowledge, and for sake of avoiding developer bias, it was decided to also conduct hallway usability testing [27] with third party participants. In essence, ten random individuals with no particular skillset were selected to test the applications, following the test cases given in Appendix A. No errors or unexpected behaviour was uncovered in the applications.

6 Team Work

During our first team meeting we discussed the approach that we should choose for our project. We decided to follow an agile approach, as it is a natural fit for software development and it provides working iterations of the final product. More specifically, we chose the Scrum Methodology [28] with some necessary adjustments to better fit our project and timetable. We opted for 2-week Sprints and decided that there was no need for a Scrum Master. After the end of each Sprint we had a Sprint meeting in order to reflect on our performance and decide the goals of the next Sprint. Furthermore, we had an extra meeting after the first week of a Sprint to talk about specific problems we had and brainstorm solutions regarding them. An important goal for us was for every team member to be involved in every aspect of the project and the delegation of the work in each Sprint aided that goal.

In order to communicate more efficiently we created a group in a mobile messenger application. It proved a quicker and easier way to communicate than email and was very handy in organising the meetings. Still, the review of the code was done with the pull requests in GitHub and during the actual team meetings. Another tool that proved really useful was the Projects tab in GitHub. We created separate folders for the server, the mobile and the desktop application along with the final report. These folders where populated with objectives that had the proper status (to do, in progress, not yet started, etc) and in many cases also stated the team member that was currently working on them.

Finally, the most important aspect of our work as a team, and the one that helped us the most, was that every disagreement was discussed and every team member was willing to listen and adapt. It was very beneficial for our progress that when different opinions were present, they were discussed and people were willing to take a step back if they were convinced by the arguments of the opposite side.

7 Evaluation

7.1 Schedule and Objectives

At the beginning of our project and after discussion, we settled on the timetable that we will follow. As mentioned before and since we followed the Scrum methodology, the schedule consists of 2-week Sprints and some extra time at the end of it for Testing and creation of the final report.

Timetable	
Date	**Description**
17/1 - 23/1	Planning and Initial Task Delegation
24/1 - 6/2	Sprint1: Green + Yellow (as many as possible)
7/2 - 21/2	Sprint2: Remaining Yellow + Red (optional)
22/2 - 6/3	Sprint3: Remaining Yellow + Red (optional)
7/3 - 27/3	Testing, Final Report, Presentation, Contingency allowance

Table 2: Project Timetable

In general, we followed the schedule from Table 2 above and indeed finished all the green, yellow and red objectives by the indicated time (7/3). The only deviation from it was between Sprints 2 and 3, as some objectives were completed earlier than expected while one objective (conflict handling) was extended for another Sprint as it was not fully completed. The generous 20 days we allowed for the final timeframe proved very useful as we tested the applications to a greater extent and had time to correct the bugs that the testing uncovered, properly write the final report and prepare the presentation.

Finally, we managed to complete all the objectives from Table 1 except for the blue ones. That means all the objectives that were labelled as *initial* (green), *subsequent* (yellow) and *optional* (red) were implemented and tested in all of the server, desktop and mobile applications. The sum of these objectives represents a fully functional application with some extra functionality as well. Due to time constraints, we were unable to implement the *quality of life* (blue) objectives that revolved around giving users permissions to access specific files.

7.2 Teamwork and Changes

All the team members worked well together and there was no incident of someone not contributing enough. The atmosphere during the team meetings was great and everyone shared their ideas and solutions. There were different proposals but their advantages and disadvantages were always thoroughly discussed and the final decisions were unanimous. This great team chemistry helped us achieve our objectives in a timely fashion and moreover made the whole project a fun and learning experience.

During the course of our project there were times that we needed to adapt to the situation and changes were required. A good example is the initial plan regarding the mobile application. At first, we decided to build the app using Ionic [29] and started working on it utilizing the aforementioned framework. However, when trying to create an early prototype, we found out that trying to read the user files was an unnecessarily difficult procedure to implement in Ionic. Therefore, we discussed about alternatives and decided to opt for Android Studio instead. Thankfully, the change was decided relatively early in the lifecycle of our project so the schedule was not impacted in a significant way.

7.3 Weaknesses

Due to the relatively small amount of time afforded for the development of our project, the final result has some weaknesses that are worth mentioning.

Firstly, we use HTTP instead of HTTPS and this is suboptimal for security and encryption reasons. HTTPS is a much safer protocol to use and can protect the user of the application from various attacks of his private information. In order to test our applications, we tried a man in the middle attack [30] on the desktop application. The result was that the use of HTTP as expected left the system vulnerable and we were able to retrieve the username and the password of the user as he attempted to login.

Furthermore, both the mobile and the desktop applications cannot load a folder that is inside a folder. This is not a major problem for the mobile application, as the directory that is used is unlikely to have folders. On the contrary, a desktop folder is very likely to contain more folders and this is something that needs to be improved in future versions of the software.

Another issue is the file size which is currently capped at 50mb. This can be altered if needed but the overall size of the server led us to choose this particular cap. Finally, a last weakness of the project is that the mobile application was developed using Android Studio and therefore it is only compatible with Android.

7.4 Strengths

One of the strengths of our project is the thoroughness and the extensiveness of our testing. We wanted to produce a bug-free application, to the extent this is possible, and for this reason we dedicated enough time for proper testing. Testing was done from multiple team members in order to avoid programmer bias, as well as from random users with the hallway testing method and we are fairly confident about the consistency of the software we produced.

Furthermore, we managed to implement users for both the mobile and the desktop application. Our architecture can register new users, manage existing ones and only allow each user to access his own files. This feature was considered optional during the planning of the project and it adds extra functionality by expanding the usability of our software.

Another strength of our work is that we dont use a traditional SQL database and thus the software is not vulnerable to SQL-Injection attacks [31] that are very common in SQL databases. A relevant strength is that all inputs are sanitized so that no directory path traversal is possible. This increases the security level of the server and protects us from relevant attacks [32].

7.5 Future Work

The majority of the future work proposed revolves around tackling the aforementioned weaknesses. A first major improvement would be to switch to the HTTPS protocol and make the necessary adjustments for the connection of the server with the desktop and mobile applications. Another future project could be developing a similar mobile application for the iOS operating system. Furthermore, we could try in the future to implement the Quality of life(blue) objectives we established during the planning phase. These involve having different levels of permissions for each user and enabling the user to allow access to specific files from his repository to other users.

8 Peer Assessment

It was decided to distribute the 100 available points evenly across all team members.

Team Member	**Points**
Alain Doebeli	16.66
Christakis Vasiliou	16.66
Ignacio Dominguez Garcia-Guijas	16.66
James Kayongo	16.66
Konstantinos Oikonomou	16.66
Nikolaos Rizos	16.66

Table 3: Distribution of points among team members

References

[1] G. Shiala, S. K. Majhib, and D. B. Phatakc. "A Comparison Study of File Synchronization". In: *International Conference on Intelligent Computing* (2015), pp. 153–164.

[2] T. Suel, P. Noel, and D. Trendafilov. "Improved file synchronization techniques for maintaining large replicated collections over slow networks". In: *Proceedings. 20th International Conference on Data Engineering* (2014). ISSN: 2319-7293.

[3] Aaysha Shaikh et al. "A paper Study on Cloud Enabled File Synchroniser". In: *Global Journal of Engineering, Design & Technology* (2004), pp. 153–164. ISSN: 1063-6382.

[4] *GitHub - request/request: Simplified HTTP request client.* `https://github.com/request/request`. [Online; accessed 17-March-2019].

[5] *fs - Node.js Manual & Documentation.* `https://nodejs.org/docs/v0.3.1/api/fs.html`. [Online; accessed 17-March-2019].

[6] *Crypto — Node.js v11.12.0 Documentation.* `https://nodejs.org/api/crypto.html`. [Online; accessed 17-March-2019].

[7] *electron-json-storage - npm.* `https://www.npmjs.com/package/electron-json-storage`. [Online; accessed 17-March-2019].

[8] *isBinaryFile - npm*. https://www.npmjs.com/package/isbinaryfile. [Online; accessed 17-March-2019].

[9] *diff2html*. https://diff2html.xyz/. [Online; accessed 17-March-2019].

[10] *Bootstrap*. https://getbootstrap.com/. [Online; accessed 25-March-2019].

[11] *Android Studio*. https://developer.android.com/studio. [Online; accessed 20-January-2019].

[12] *OkHttp library*. https://github.com/square/okhttp. [Online; accessed 8-February-2019].

[13] *PersistentCookieJar library*. https://github.com/franmontiel/PersistentCookieJar. [Online; accessed 25-February-2019].

[14] *mLab*. https://mlab.com/. [Online; accessed 20-January-2019].

[15] *ExpressJS*. https://expressjs.com. [Online; accessed 20-January-2019].

[16] *MongooseJS*. https://mongoosejs.com. [Online; accessed 20-January-2019].

[17] *Morgan*. https://github.com/expressjs/morgan#readme. [Online; accessed 20-January-2019].

[18] *HTTP-errors*. https://github.com/jshttp/http-errors#readme. [Online; accessed 20-January-2019].

[19] *PassportJS*. http://www.passportjs.org/. [Online; accessed 20-January-2019].

[20] *connect-mongo*. https://github.com/jdesboeufs/connect-mongo#readme. [Online; accessed 20-January-2019].

[21] *express-sessions*. https://github.com/expressjs/session#readme. [Online; accessed 20-January-2019].

[22] *cookie-parser*. https://github.com/expressjs/cookie-parser#readme. [Online; accessed 20-January-2019].

[23] *express-fileupload*. https://github.com/richardgirges/express-fileupload#readme. [Online; accessed 20-January-2019].

[24] *bcryptJS*. https://github.com/dcodeIO/bcrypt.js#readme. [Online; accessed 20-January-2019].

[25] *jsDiff*. https://github.com/kpdecker/jsdiff#readme. [Online; accessed 20-January-2019].

[26] *What is Black Box Testing: Advantages and Disadvantages*. https://www.invensis.net/blog/it/black-box-testing-advantages-disadvantages/. [Online; accessed 18-March-2019].

[27] *Hallway Usability Testing*. https://www.techopedia.com/definition/30678/hallway-usability-testing. [Online; accessed 18-March-2019].

[28] L. Rising and N. S. Janoff. "The Scrum software development process for small teams". In: *IEEE Software* 17.4 (2000), pp. 26–32. ISSN: 0740-7459. DOI: 10.1109/52.854065.

[29] *Ionic*. https://ionicframework.com/. [Online; accessed 20-January-2019].

[30] F. Callegati, W. Cerroni, and M. Ramilli. "Man-in-the-Middle Attack to the HTTPS Protocol". In: *IEEE Security Privacy* 7.1 (2009), pp. 78–81. ISSN: 1540-7993. DOI: 10.1109/MSP.2009.12.

[31] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. "A classification of SQL-injection attacks and countermeasures". In: 1 (2006), pp. 13–15.

[32] M. Danforth. "Towards a Classifying Artificial Immune System for Web Server Attacks". In: *2009 International Conference on Machine Learning and Applications* (2009), pp. 523–527. DOI: 10.1109/ICMLA.2009.38.

A Test Cases

A.1 Desktop Application

Test Case Number	Test Description	Result
	Packaging and Launch	
1	The code does not contain errors and is packaged successfully for Linux, macOS, and Windows using the provided build scripts.	PASS
2	Each OS specific version of the desktop app can be launched on Linux, macOS, and Windows.	PASS
	User Sign-Up	
3	Providing empty credentials (username and password), an already taken username, or not matching passwords, results in a failed sign-up.	PASS
4	Providing valid credentials (username and password), including matching passwords, results in a successful sign-up.	PASS
	User Log-In	
5	Providing empty or invalid log-in credentials (username and password) results in a failed log-in.	PASS
6	Providing valid log-in credentials (username and password) results in a successful log-in.	PASS
7	Upon successful log-in, the user is asked to specify the target directory containing the files for synchronisation (if not already set previously).	PASS
8	Upon successful log-in, and when the target directory containing the files for synchronisation is set, all the local files within that directory are listed.	PASS
9	Upon successful log-in, and when the target directory containing the files for synchronisation is set, all the remote files on the server associated with the current user are listed.	PASS
	File Management	
10	Clicking the refresh button will refresh the list of files in the local directory and all the remote files on the server associated with the current user.	PASS
11	If a specific file does not exist on the server, it can be uploaded to the server by double-clicking on the shown upload button.	PASS

12	If a specific file does not exist in the local directory, it can be downloaded to the local directory by double-clicking on the shown download button.	PASS
13	If a specific version of a file exists on both the server and in the local directory, no action can be taken.	PASS
14	If a version of a file exists on the server that is newer than the version of the same file in the local directory, a fix button is shown that can be double-clicked, opening a dialog, to act on the conflict.	PASS
15	Clicking the "Do nothing" button will close the dialog without further action.	PASS
16	Clicking the "Download from server and replace local file" button successfully downloads the new version of the file from the server, replacing the old version of the file in the local directory.	PASS
17	Clicking the "Upload local file and replace server file" button successfully uploads the old version of the file from the local directory, replacing the new version of the file on the server.	PASS
18	Clicking the "See file differences" button, only shown when the file is not a binary, will open a dialog highlighting the differences in content of the local and server file.	PASS
19	In the file list, double-clicking the trash button for a specific file will open a dialog allowing the user to delete that file from the local directory, the server, or both (where applicable).	PASS
20	Clicking the "Oops! Don't do it!" button will successfully close the dialog without further action.	PASS
21	Provided the file exists in the local directory, clicking the "Delete the local file" button will successfully delete the file from the local directory.	PASS
22	Provided the file exists on the server, clicking the "Delete the server file" button will successfully delete the file from the server.	PASS
23	Provided the file exists in the local directory and on the server, clicking the "Delete both" button will successfully delete the file from the local directory and the server.	PASS
	User Log-Out	
24	Clicking the log out button will destroy the current user's authenticated session and the initial log-in page will be displayed.	PASS

User Delete		
25	Clicking the "Delete account" entry in the app's menu will delete the currently authenticated user account and all of the files stored on the server associated with the current user.	PASS

A.2 Mobile Application

Test Case Number	Test Description	Result
Compilation and Launch		
1	The code does not contain errors and is compiled successfully for Android.	PASS
2	The mobile app can be launched on Android.	PASS
User Sign-Up		
3	Providing empty credentials (username and password), an already taken username, or not matching passwords, results in a failed sign-up.	PASS
4	Providing valid credentials (username and password), including matching passwords, results in a successful sign-up.	PASS
User Log-In		
5	Providing empty or invalid log-in credentials (username and password) results in a failed log-in.	PASS
6	Providing valid log-in credentials (username and password) results in a successful log-in.	PASS
7	Upon successful log-in, all the local files within the mobile app are listed.	PASS
8	Upon successful log-in, all the remote files on the server associated with the current user are listed.	PASS
File Management		
9	Clicking the refresh button will refresh the list of files in the mobile app and all the remote files on the server associated with the current user.	PASS
10	If a specific file does not exist on the server, it can be uploaded to the server by clicking on the shown upload button.	PASS
11	If a specific file does not exist in the mobile app, it can be downloaded to the mobile app by clicking on the shown download button.	PASS
12	If a specific version of a file exists on both the server and in the mobile app, no action can be taken.	PASS
13	If a version of a file exists on the server that is newer than the version of the same file in the mobile app, a fix button is shown that can be clicked, opening a dialog, to act on the conflict.	PASS

14	Clicking the "Get the latest version" button successfully downloads the new version of the file from the server, replacing the old version of the file in the mobile app.	PASS
15	Clicking the "Upload anyway and replace last version" button successfully uploads the old version of the file from the mobile app, replacing the new version of the file on the server.	PASS
16	Clicking the "Get diff file and solve manually" button, only shown when the file is not a binary, will open a dialog highlighting the differences in content of the local and server file.	PASS
17	In the file list, clicking the trash button for a specific file will open a dialog allowing the user to delete that file from the mobile app, the server, or both (where applicable).	PASS
18	Clicking the "Cancel" button will successfully close the dialog without further action.	PASS
19	Provided the file exists in the mobile app, choosing the "local file" option will successfully delete the file from the mobile app.	PASS
20	Provided the file exists on the server, choosing the "server file" option will successfully delete the file from the server.	PASS
	User Log-Out	
21	Clicking the log out button will destroy the current user's authenticated session and the initial log-in page will be displayed.	PASS
	User Delete	
22	Clicking the "Delete account" entry in the app's menu will delete the currently authenticated user account and all of the files stored on the server associated with the current user.	PASS

A.3 Server Application

Test Case Number	Test Description	Result
Connections and Requests		
1	The server can successfully accept HTTP connections.	PASS
2	The server can successfully accept and process GET and POST requests.	PASS