Final Report

By Project Prime

March 24, 2019

1 Introduction

In this project, we were tasked with building a multi-host file synchroniser consisting of three components: a server, and mobile and desktop client. The server is the "hub" whilst the clients are "spokes". The server was created using Node.js, where it acted as an interface between both clients and the database (which was created using MongoDB). In addition, this server processed HTTP GET and POST requests to deliver/updates files on the client's behalf. The mobile client was made using Java and XML on Android Studio, whereas the desktop client was created using Javascript, HTML and CSS on Electron. The result of this project is a fully-functioning file synchroniser that interacts with and manages multiple, heterogeneous clients via a Node.js server. Furthermore, this synchroniser possesses the necessary algorithms (e.g. rsync) to resolve file conflicts caused by clients.

2 Review

2.1 Server Application

2.2 Desktop Client

The common file hosting service, Dropbox, uses a method which they call "Sync" to keep its files updated across mobile and desktop clients. Its split into two stages: upload and download. Once a file is uploaded through the server, it is added to their database before other clients are notified of its existence. According to (Koorapati, 2014), the host client does not use a traditional file system directory and allocates a namespace for each user. Their metadata database also known as the Server File Journal stores this information and connects directly to one of Dropbox's server types: Metadata Server. The actual contents of the file are stored in block data servers. The advantage of having a separate database for the metadata here is that the main contents of the files are less vulnerable

to threats because they won't need to be referenced as often as the metadata for reviewing updates about the files. Furthermore, a good file synchronisation software is defined by how robust its security is and their block data servers are all encrypted using "256-Bit Advanced Encryption Standard" as mentioned in (Dropbox, n.d). Therefore, the protocol described in (Koorapati, 2014) for Dropbox's desktop client is as follows:

- The upload client commits to the metadata server first to check if there exists a namespace for the file stored in the block server already.
- For new files, the metadata server would return "need blocks" and the client then directly talks with the block data server to store the new file.
- The process is repeated and any updates to the file can be checked by only talking to the metadata server.
- To download a file, the downloading client pings the metadata server to retrieve a list of updates. The Server File journal notifies the client that a new file was added.
- The download client now contacts the block data server directly to access the new file.

This protocol shows that both the desktop client and server application in Dropbox are equally involved in the synchronization process. However, this is not always the case in other software, for example Unison, an open-source file sync.

2.3 Mobile Client

3 Requirements and Design

Requirements have been derived for each component. The requirements have been analysed using the MoSCoW prioritisation technique (Madsen, 2017); as a result, each requirement has been placed into one of the following categories: *Must Have* (critical requirements with the highest priority), *Should Have* (Important but unnecessary requirements for the final product), *Could Have* (Lowest-priority requirements that would be implemented if time permits) and *Won't Have* (Least critical requirements that are unrequired for project success). In addition, the subsections will show the designs that reflect the corresponding component.

3.1 Desktop Client

MoSCoW requirements for The Desktop Client		
Requirement No.	**Requirement**	**Priority**
1	Must be able to communicate with the Server Application	Must Have
2	Has to be able to upload files	Must Have
3	Has to be able to download files from database via server	Must Have
4	Make use of a file management system to upload any file from the user's system	Must Have
5	Check for updates regularly so that the desktop client is in sync with the Server and Mobile Client	Must Have
6	Display the current list of files from the database in the centre of the page	Should Have
7	Include a delete file functionality	Should Have
8	Provide a way to track and check recent changes to files	Could Have
9	Include a search bar to quickly find files if the list of files is too large	Could Have
10	Have User profiles for personalised access	Won't Have

3.2 Mobile Client

MoSCoW requirements for The Mobile Client		
Requirement No.	**Requirement**	**Priority**
1	Must be able to communicate with the Server Application	Must Have
2	Must be able to upload files into the database through the server	Must Have
3	Must be able to delete files	Must Have
4	All changes must be reflected in the files stored in the server	Must Have
5	The mobile UI must be simple and easy to navigate through	Should Have
6	All the applications functions should accessible from the Main page	Should Have
7	All files should be listed with both filename and time of last modification clearly visible to the user	Should Have
8	Include a search bar to quickly search through the list of files	Could Have

3.3 Server Application

MoSCoW requirements for Server Application		
Requirement No.	**Requirement**	**Priority**
1	Has to be connected to a database in order to store and retrieve files	Must Have
2	Must be able to communicate with the Desktop and Mobile Clients simultaneously	Must Have
3	Use HTTP requests to send data to the clients	Must Have
4	Handle conflicts using Rsync	Should Have
5	Use security encryption to protect the data	Could Have

4 Implementation

4.1 Server Application

The server was developed using Node.js and the database used is a cloud service by MongoDB.

Test Cases for Server Application				
Test ID	**Test Name**	**Expected**	**Actual**	**Pass/Fail/Solved**
1	...	...	...	...

4.2 Desktop Client

The desktop client was developed using the Electron framework; thus, the implementation primarily involved HTML, Javascript and CSS coding. To start with the foundation, Electron provides JS and HTML code (Electron, n.d.) to form a basic desktop app and webpage, respectively; therefore, we used this pre-existing code as the starting point for the development of the desktop client. Afterwards, we modified the webpage by adding a title (i.e. "Project Prime Desktop"), a file upload bar and a display of the list of server-contained files using HTML and CSS. To upload a file into the database, the desktop client would transmit a POST request to the server after the user selects a file; as a result, the file gets stored in the database. Next, in order to display the database-contained files, the client sends a GET request to the server, which relays the files to the client. In addition, a delete button is underneath each file on the desktop client, which after being pressed, deletes the corresponding file in the server; thus, removing the file from the client UI. The rationale behind this layout is to grant simplicity to the desktop application, as well as efficiency in terms of usage.

Test Cases for Desktop Client				
Test ID	**Test Name**	**Expected**	**Actual**	**Pass/Fail/Solved**
1	Upload file	A file gets uploaded into the database through server after pressing *Submit* button.	...	...

4.3 Mobile Client

The mobile client was developed using Android Studio; thus, the implementation primarily involved Java and XML.

Test Cases for Mobile Client				
Test ID	**Test Name**	**Expected**	**Actual**	**Pass/Fail/Solved**
1	...	...	...	...

5 Teamwork

Project Prime consists of six members: Yusaf, Sandipan, Saloni, Shefali, Cameron and Manny. To balance the workload, the team was divided into three subgroups of two teammates and each subgroup was assigned to the development of one component. As a result, Yusaf and Sandipan were assigned to the desktop client, Saloni and Shefali were assigned to the server, and Manny and Cameron were assigned to the mobile client. Within these subgroups, the work was distributed between both teammates through discussion. Despite the team division, members from other subgroups were permitted to intervene in the development of a component that they weren't assigned to, in order to fix issues that the assigned subgroup couldn't correct, for example. Moving onto communication, the team remotely communicated using an instant messaging app (i.e. Whatsapp), which allowed arrangements of group sessions at a certain date-and-time, notifying teammates of open pull requests, etc. Furthermore, the sessions were conducted in booked study rooms at least once per week and these sessions involved group discussion and coding of all components. Next, the project involved Github, where the project implementation was contained in a public repository called "Project Prime Dev". In addition to Git, the team followed the feature branch workflow, where a component feature was developed inside a branch seperate to the master branch and subsequently, the former would be merged into the latter branch.

6 Evaluation

To start with what went well, one positive aspect was the firm strength of team communication, as all members proactively shared their thoughts and concerns in physical meetings and in the WhatsApp group. In addition, communication was carried-out in a respectful manner at all times; thus, there were no verbal conflicts during the project. Another positive aspect was the fairness in workload distribution by creating subgroups that were assigned to the development of a specific component. The rationale behind the formation of subgroups was to avoid the possibility of one member feeling overwhelmed from having lots of work. Furthermore, by assembling subgroups, this influenced the members of the subgroup to cooperate together to build the component; thus, being in a subgroup generated a sense of teamwork. Finally, a notable positive aspect was our ability to adapt to changing circumstances. For instance, it was initially planned to store files in a SQL database. Unfortunately, we discovered SQL databases were unideal for file storage since it is limited by file size. In

response, we decided to migrate to MongoDB, which is a cloud service that allows the creation of document-oriented database systems that can contain files of any size.

Moving onto what didn't go well, our initial plan was weak as we didn't know how to approach the task, due to having no experience in developing file synchronisers. Although we met various project objectives, our commitment to the plan was low, as we rarely compared our progress to the initial plan during our weekly meetings. Another negative was the poor interactive feedback of certain features in particular components. For example, after pressing the *Browse* button in the desktop application, there is no feedback (e.g. temporary colour change, mouse cursor change) to indicate the button-pressed.

Relative to the initial plan, there were differences between the rough timetable and actual progress. For instance, instead of using an SQL database, we decided to create and use a MongoDB database since it's document-oriented and has the capability to store files of any size, whereas SQL storage is limited by data type and size, and it cannot process text files properly; thus, with MongoDB being more suited to our needs, we abandoned the plan to use an SQL database and shifted to MongoDB instead. Another difference is we didn't sketch each component because we wanted to concentrate our efforts on the actual implementation of the clients and server, as well as to save time.

On to how the team worked together, the team was divided into three subgroups as initially planned, where each subgroup developed one component of the file synchronising system; as a result, the workload was divided fairly amongst all members. Throughout the majority of the project's lifetime, the team attended group meetings once per week where group discussion and coding was conducted. Along with group meetings, communication was also conducted in a whatsapp group where we scheduled groups meetings in booked study rooms at a certain date-and-time, notified each other about opened pull requests, reminders about incomplete work, etc. The reason for choosing whatsapp as our main form of remote communication was due to everyone's familiarity with the app and its simplicity (in terms of usability); thus, there was no need to use Skype since the team was comfortable with using whatsapp, in contrary to the original plan. Despite developing the system outside the meetings, one major weakness was scheduling meetings once per week since it delayed the completion of work, which could have been completed faster if we met more than once per week by completing tasks together. Overall, the team worked well together, as there were no conflicts and each member contributed their unique skills to the group. For instance, since Yusaf possessed leadership experience, he was elected as team leader to command the group, ensure each member's involvement in the project's activities, etc.

In conclusion, this project caused the realisation that despite our computer science backgrounds, our technical prowess is still very basic and there is much for us to learn. In response to this discovery, we will commit to thoroughly relearning and practising how to use different technologies in our spare time,

including programming languages and Git (e.g. BitBucket). In retrospect, if this project was repeated, our different approach would be to partner stronger members with novice ones, in terms of technological skill; thus, providing novice members with the opportunity to enhance their skills by learning whilst working on the job.

7 Peer Assessment

Peer Assessment of Project Prime					
Yusaf	**Sandipan**	**Saloni**	**Shefali**	**Cameron**	**Manny**
17.5	16.5	16.5	16.5	16.5	16.5

8 References

- Madsen, S. (2017) *How to Prioritize with the MoSCoW Technique* [online] Available at: https://www.projectmanager.com/training/prioritize-moscow-technique [Accessed on 10 March 2019]
- Electron (n.d.) *Writing your First Electron App — Electron* [online] Available at: https://electronjs.org/docs/tutorial/first-app [Accessed on 12 March 2019]
- Nipuun Koorpati. (2014) *Streaming File Synchronization* [online]. Available at: https://blogs.dropbox.com/tech/2014/07/streaming-file-synchronization/ [viewed 17th March]
- Dropbox Business. (n.d) *Under the hood: Architecture Overview*[online]. Available at: https://www.dropbox.com/business/trust/security/architecture [viewed 17th March]