Final Report

By Project Prime

March 26, 2019

1 Introduction

In this project, we were tasked with building a multi-host file synchroniser consisting of three components: a server, a mobile client and desktop client. The server is the *hub* while the clients are the *spokes*. The server was created using Node.js, where it adopted the role of an interface between the clients and MongoDB database. In addition, the server is stored in the Heroku cloud platform to provide both clients with universal file access and processing. The Node.js server can also process HTTP requests to allow both clients to upload, delete and download files. The mobile client was developed using Java and XML on Android Studio, whereas the desktop client was developed using Javascript, HTML and CSS on Electron. The output of the project is a fully-functioning file synchroniser that interacts with both clients via the Node.js server. This project addresses the problem of long durations to complete tasks; thus, the solution will take the form of two clients that minimise the duration to complete tasks as low as possible.

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.

In (Jim, Pierce and Vouillon, 2002), it is mentioned that "most of Unison's functionality is concentrated on the client side". At the start of a process, two roots are specified (to a create a replica) and then the client goes through the following steps:

- **Update Detection:** The metadata from older version of the files are compared to their current states and each host is notified whether the contents of the files are changed or not.
- **Reconciliation:** The list of changed contents are merged together into a "task list" and used to handle conflicts. A conflict is detected if a path is updated in the replica, whether its descendants have also been updated in the other replica, or the contents in the two replicas are not equal. The task list is then displayed to the user through the UI.
- **Propagation:** The changes in the task list are propagated through both the replicas and the server is then notified about the new contents of the file paths that were updated in the process.

The advantage of having a heavy Client-Side functionality is robustness. Having less burden on their server application increases their crash resistance as indicated in (Jim, Pierce, Vouillon, 2002).

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 application's functions should be accessible from the main page	Should Have
7	All files 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

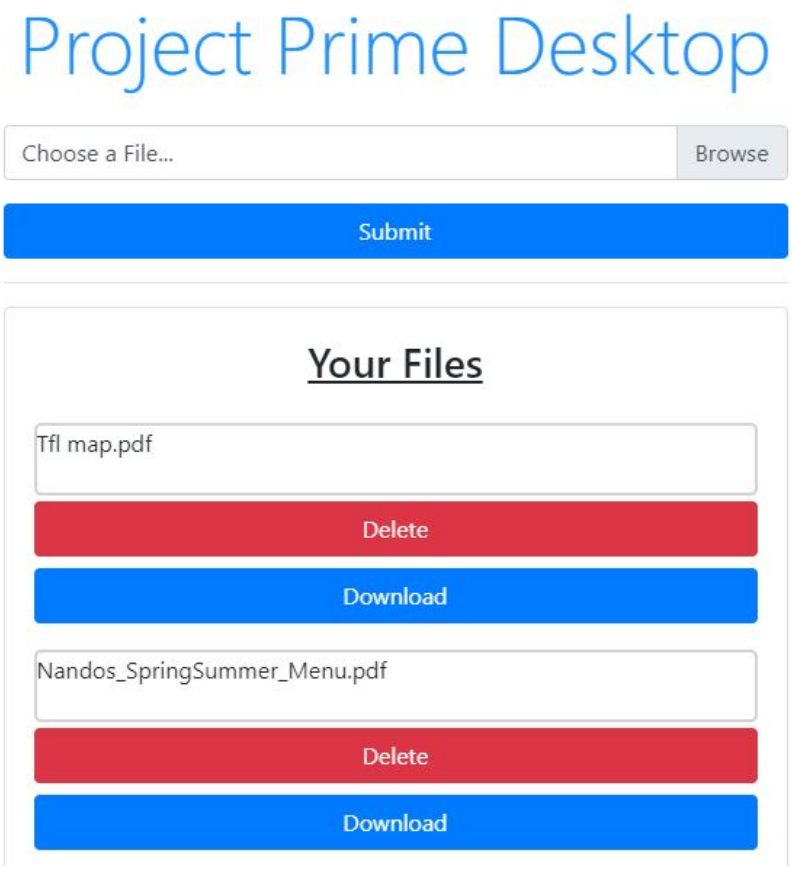
4 Implementation

4.1 Server Application

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

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 HTML, Javascript and CSS on Electron. Electron provides pre-written HTML code (Electron, n.d.) that creates a basic desktop app; the pre-written code was used as the foundation for the development of the desktop client. Afterwards, the client's front-end was modified using JS, HTML and CSS to reflect the project designs. On to

the desktop client's front-end, Bootstrap CSS files and JS scripts (Bootstrap, n.d.) were also included to add and format certain elements (e.g. file upload bar and button), along with CSS files written by the desktop subgroup.

```
<!--Linking the html file to the css file-->
<link href="./index.css" type="text/css" rel="stylesheet">

<!--Bootstap CSS file-->
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">
```

```
<!--JS scripts from Bootstrap-->
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1" crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM" crossorigin="anonymous"></script>
```

```
/* Setting the format of the title */
.title {
    text-align: center;
    color: dodgerblue;
}

/*The format for the blocks that contain each file*/
.inner-block {
    border: 2px solid lightgray;
    width: 100%;
    height: 50px;
    border-radius: 5px;
    margin-top: 18px;
}

/* Positioning the title to the center of the page */
h3 {
    text-align: center;
}
```

After setting the foundation, the desktop subgroup concentrated on displaying files that resided in the database. To do so, the server would return a JSON object (which contains each file in the database) after a connection was formed between the desktop client and server. Using a loop, the object was traversed through and for every file encountered, that file would be presented on the client's UI. In addition, each file would be encapsulated in a file container (presented as a grey-bordered box).

```
<!-- Displaying files that are stored in the database -->
<div id="server-files" class="card card-body mb-3">
<h3><u>Your Files</u></h3>
    <script>
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = function() {
      if (this.readyState == 4 && this.status == 200) {
        //fileObj = A list of files residing in the server
        var fileObj = JSON.parse(this.responseText);
        var block, deleteForm, deleteBtn, dwnldForm, dwnldBtn;
        // Traverses through each file
        for (var i = 0; i < fileObj.length; i++) {
          // File-containing blocks
          block = document.createElement('div');
          block.className = "inner-block mb-1";
```

```
// Creates and displays the files, and delete and download buttons on the desktop client
document.getElementById('server-files').appendChild(block);
block.appendChild(document.createTextNode(fileObj[i].filename));
document.getElementById('server-files').appendChild(deleteForm);
document.getElementById('server-files').appendChild(dwnldForm);
```

```
xmlhttp.open("GET", "https://rocky-plateau-19773.herokuapp.com/files", true);
xmlhttp.send();
```

Next, the subgroup wrote code that enabled the client to upload files into the database via the server. After pressing the *Browse* button, the local host's file manager opens, which enables the user to choose a file and subsequently, the label of the file upload bar changes to the name of the selected file; therefore, indicating which file was selected by the user. Afterwards, the user would press the *Submit* button which transmits a http post request to the server, which causes the latter to store the selected file inside the database. In addition, the label of the file upload bar will revert to its initial form (i.e. "Choose a File...") after the *Submit* button is pressed.

```
<!--Uploads a file into the DB via the node.js server (in the Heroku cloud) after pressing Submit-->
<form action="https://rocky-plateau-19773.herokuapp.com/upload" method="POST" enctype="multipart/form-data">
  <div class="custom-file mb-3 ">
    <label id="fileLabel" for="file" class="custom-file-label">Choose a File...</label>
    <input id="file" name="file" type="file" class="custom-file-input" onclick="changeLabel()">
  </div>
```

```
<!--Shows which file was selected by the user-->
<script>
  var textFile = document.getElementById('file');

  function changeLabel() {
    var label = document.getElementById('fileLabel');
    // The filename and path of the selected file
    var textFileValue = textFile.value;
    // Finds the end of the filepath
    var fileNameStart = textFileValue.lastIndexOf('\\');
    // Isolating the filename
    textFileValue = textFileValue.substring(fileNameStart + 1);

    if (textFileValue !== '') {
      // Changes the label text into the filename
      label.innerText = textFileValue;
    }
  }
  // Runs 'changeLabel' method after input filename changes
  textFile.addEventListener('change', changeLabel, false);
</script>
```

```
<!--Inserts files into the database upon button press-->
<input type="submit" value="Submit" class="btn btn-primary btn-block" onclick="revert()">

<script>
  // Shows the initial label after pressing submit button
  function revert() {
    var label = document.getElementById('fileLabel');
    label.innerText = 'Choose a File...';
  }
</script>
```

Moving on to file deletion, the desktop subgroup wrote code that enabled the client to delete files; as a result, removing the corresponding document from the database. To add to this, deleting a file resulted in removing that file from the list of documents displayed on the client's UI. After pressing the *Delete* button, the client would send a delete request to the server and this request would include the file's unique ID; therefore, the use of unique identification ensures the prevention of multiple files of the same name being deleted, for example.

```
// Deletes the file by removing it from the DB via the server
deleteForm = document.createElement('form');
deleteForm.setAttribute('method', 'POST');
deleteForm.setAttribute('action','https://rocky-plateau-19773.herokuapp.com/files/'+ fileObj[i]._id + '?_method=DELETE');
deleteBtn = document.createElement('input');
deleteBtn.setAttribute('type', 'submit');
deleteBtn.setAttribute('class', 'btn btn-danger btn-block mb-2');
deleteBtn.setAttribute('value', 'Delete');
deleteForm.appendChild(deleteBtn);
```

Finally, the desktop subgroup wrote code that enabled the client to download files from the database; as a result, bringing the corresponding document into the storage of the local host. After pressing the *Download* button, the client sends a GET request to the server, which causes the server to return the file in the local host's file manager and the user stores the file in the local host's storage by pressing the *save* button of the file manager.

```
// Downloads the file from the DB via the server
dwnldForm = document.createElement('form');
dwnldForm.setAttribute('method', 'GET');
dwnldForm.setAttribute('action', 'https://rocky-plateau-19773.herokuapp.com/download/'+ fileObj[i].filename);
dwnldBtn = document.createElement('input');
dwnldBtn.setAttribute('type', 'submit');
dwnldBtn.setAttribute('class', 'btn btn-primary btn-block');
dwnldBtn.setAttribute('value', 'Download');
dwnldForm.appendChild(dwnldBtn);
```

Test Cases for Desktop Client				
Test ID	**Test Name**	**Expected**	**Actual**	**Pass/Fail/Solved**
1	Connection to server	A JSON object will be returned to indicate the connection between the server and client.	As expected	PASS
2	Display file(s)	Display files stored in the Mongo database.	As expected	PASS
3	Upload file	A file gets uploaded into the database via server after pressing the *Submit* button. Next, the file is displayed in the list of documents stored in the server.	File was uploaded but does not appear in the list unless the page is refreshed.	1/2
4	Delete file	A file gets removed from the database via server after pressing *Delete* button. Next, the list of documents is displayed without the deleted file.	File was deleted but isn't removed from the list unless the page is refreshed.	1/2
5	Download file	A file gets downloaded into the local host after pressing the *Download* button.	As expected	PASS

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.

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 Bibliography

Reference List
Bootstrap (n.d.) *Introduction* [online] Available at: https://getbootstrap.com/docs/4.3/getting-started/introduction/ [Accessed on 25th March]
Dropbox Business. (n.d) *Under the hood: Architecture Overview* [online] Available at: https://www.dropbox.com/business/trust/security/architecture [Accessed on 17th March]
Electron (n.d.) *Writing your First Electron App — Electron* [online] Available at: https://electronjs.org/docs/tutorial/first-app [Accessed on 12 March 2019]
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]
Nipuun Koorpati. (2014) *Streaming File Synchronization* [online] Available at: https://blogs.dropbox.com/tech/2014/07/streaming-file-synchronization/ [Accessed on 17th March]
Jim, T., Pierce, B., Vouillon, J. (2002) *How to build a File Synchronizer* [online] Available at: http://web.mit.edu/6.033/2005/wwwdocs/papers/unisonimpl.pdf [Accessed on 26th March]