

# BACKUP SOFTWARE APPLICATION DISTRIBUTED WITH THE LINUX OPERATING SYSTEM

Martin K. Sova

Final Project Report

**Abstract.** This project involves building a backup software application for intermittent backup of file systems. The final application will be distributed with the Linux operating system and will employ many of the backup schemes explored throughout the research phase of the project. To adhere to its rudimentary purpose, backup software is designed to enable restoration in cases of unfortunate loss of data. However, as will be discussed throughout this paper, the most advanced of today's backup applications often go above and beyond the most basic features to satisfy onerous software specific requirements and to stay relevant in today's cut-throat tech industry. Modern backup systems can effectively simplify or entirely eliminate the user's need to manually ensure that certain data is routinely backed up. The literature research phase of the project covers reputable backup systems, contemporary backup approaches and existing algorithms, which will enable me to accumulate knowledge in preparation for developing the software. The applicable backup schemes discovered during the research stages will be utilized for the realization of the project objectives. Throughout the project, an evaluation will ensue regarding the technical architecture of the software, alongside developing numerous sets of file system modifications with the bash command language to test the performance of the backup utility. The successful completion of the project yields a project report, a backup software and a presentation during which a demonstration of the application features is achieved.

*I certify that all material in this dissertation which is not my own work has been identified.*

A thesis presented for the degree of  
Computer Science BSc  
College of Engineering, Mathematics and Physical Sciences  
University of Exeter  
May 2<sup>nd</sup> 2018

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	The Challenge . . . . .	4
1.2	Project Motivation . . . . .	4
1.3	Project Aim . . . . .	5
1.4	Project Objectives . . . . .	5
1.4.1	Core Objectives . . . . .	6
1.4.2	Extension Objectives . . . . .	7
1.4.3	Future Enhancements . . . . .	7
1.5	Deliverables . . . . .	8
1.6	Literature Research . . . . .	8
1.7	Report Structure . . . . .	9
<b>2</b>	<b>Project Management</b>	<b>9</b>
2.1	Schedule . . . . .	9
2.2	Methodology . . . . .	9
2.2.1	Stage 1:Research and Preparation . . . . .	9
2.2.2	Stage 2:Design and Implementation . . . . .	10
2.2.3	Stage 3:Evaluation and Conclusion . . . . .	10
2.3	Evaluation plans . . . . .	10
2.3.1	Testing methodology . . . . .	10
<b>3</b>	<b>Background Research and Overview</b>	<b>11</b>
3.1	An introduction to the background literature research . . . . .	11
3.2	Existing backup systems . . . . .	11
3.2.1	AMANDA Network Backup Manager . . . . .	11
3.3	Design for backup systems . . . . .	12
3.3.1	Preference of strategy for this project . . . . .	12
3.3.2	File-based vs. device-based file system backup . . . . .	12
3.4	Full and incremental backup algorithms . . . . .	13
3.5	Summary . . . . .	13
<b>4</b>	<b>Development</b>	<b>14</b>
4.1	Introduction . . . . .	14
4.2	Back end . . . . .	14
4.3	Front end . . . . .	15
4.4	Configuration files for system communication . . . . .	16
4.4.1	configuration.json . . . . .	17
4.4.2	filemod.json . . . . .	17
4.4.3	status.json . . . . .	17
4.4.4	Technical details regarding configuration files . . . . .	17

<b>5</b>	<b>Technical Design and Architecture</b>	<b>18</b>
5.1	System architecture . . . . .	18
5.1.1	Design decisions and flaws for system build-up of fundamental componenets . . . .	18
5.2	Unified system architecture . . . . .	23
5.3	The front end restore utility . . . . .	24
5.4	Design deviations . . . . .	24
5.4.1	Deviations to hard link implementation . . . . .	24
5.4.2	Deviations to management of the system service . . . . .	25
<b>6</b>	<b>Technical Implementation</b>	<b>26</b>
6.1	Languages . . . . .	26
6.1.1	The C++ programming language . . . . .	26
6.1.2	C++ for the back end component . . . . .	26
6.1.3	C++ for the front end component . . . . .	26
6.2	Qt . . . . .	27
6.3	Libraries . . . . .	27
6.3.1	The Boost C++ Filesystem Library . . . . .	27
6.3.2	libusb . . . . .	27
6.4	git . . . . .	27
<b>7</b>	<b>Evaluation and conclusion</b>	<b>28</b>
7.1	Initialization . . . . .	29
7.1.1	Inotify . . . . .	29
7.1.2	Full backup after process initiation . . . . .	29
7.2	Incremental results . . . . .	30
7.2.1	Hard links and copies . . . . .	30
7.3	File deletion . . . . .	31
7.4	Full vs. Incremental backup . . . . .	32
7.5	Hard links in source file system . . . . .	32
7.6	Directories . . . . .	32
7.6.1	Directory creation . . . . .	33
7.6.2	Directory deletion . . . . .	33
7.6.3	File Omission . . . . .	33
7.6.4	Pruning . . . . .	33
7.7	Evaluation with comparison to literature and future work . . . . .	34
7.8	Conclusion . . . . .	34
	<b>Appendices</b>	<b>37</b>
<b>A</b>	<b>Additional Background Research</b>	<b>38</b>
A.1	Design considerations in file-based backup systems . . . . .	38
A.2	Device-based strategy . . . . .	39
A.3	Design issues of device-based dump backup strategy . . . . .	39
<b>B</b>	<b>Development</b>	<b>41</b>
B.1	Technical details of the configuration.json file . . . . .	41
B.2	Technical details of the filemod.json file . . . . .	41
B.3	Technical details of the status.json file . . . . .	42

<b>C</b>	<b>Unified system architecture</b>	<b>43</b>
C.1	Hard links in context of intermittent backup . . . . .	44
C.2	Supplementary description of the unified system algorithm . . . . .	45
C.2.1	Initialization of process . . . . .	45
C.2.2	Backup utility . . . . .	45
C.3	Inotify implementation algorithm . . . . .	48
C.4	Behavior of unified system architecture . . . . .	48
C.4.1	Backup Cycle 1 . . . . .	49
C.4.2	Backup Cycle 2 . . . . .	49
C.4.3	Backup Cycle 3 . . . . .	50
C.4.4	Backup Cycle 4 . . . . .	50
C.4.5	Backup Cycle 5 . . . . .	51
C.5	Discussion of unified system behavior . . . . .	51
<b>D</b>	<b>Evaluation</b>	<b>54</b>
D.1	Shell scripts . . . . .	55
D.1.1	Cycle 1 . . . . .	55
D.1.2	Cycle 2 . . . . .	55
D.1.3	Cycle 3 . . . . .	55
D.1.4	Cycle 4 . . . . .	55
D.1.5	Cycle 5 . . . . .	55
D.1.6	Cycle 6 . . . . .	56
D.1.7	Cycle 7 . . . . .	56
D.1.8	Cycle 8 . . . . .	56
D.2	Source file system . . . . .	57
D.3	File system modification results . . . . .	58
D.4	Pruning results . . . . .	61

# Chapter 1

## Introduction

In Chapter 1 of this report an introduction is achieved via: a description of the challenge in Section 1.1, the project motivation in Section 1.2, the project aim in Section 1.3, objectives set out for the project in Section 1.4, a list of the project deliverables in Section 1.5, a prelude to the literature research in Section 1.6 and, finally, a description of the report structure in Section 1.7.

### 1.1 The Challenge

In computing, a filesystem is a method for controlling how data is stored within a given storage medium to provide intuitive access for an end-user. As a central component of an operating system, file systems have received a great deal of attention from both the industrial and academic communities. As file systems expand, it becomes more and more difficult to ensure that all data is safely stored. As a result, backup systems have seen a similar increase in attention received from the industrial community. The cause for concern stems from the realization that access times and data rates of disk drives will increase at reduced rates - that is, at approximately 20% per annum[3]. On the other hand, as the capacity of new disk drives continues to increase with a higher rate of over 50% per annum[3], networked computing environments will expand to contain terabytes of disk storage. The disaccord in progression of these two fundamental components of data transfer indicates that the time to read contents of a disk drive and write them to a backup device will take increasingly longer. Over time, this trend suggests that traditional backup schemes will be too slow, so protecting rapidly expanding data repositories from software errors that may corrupt a filesystem, disk or other hardware failures, natural disasters, or loss of data by user error continues to be a growing challenge.

### 1.2 Project Motivation

The use of backup systems remains important not only for data recovery, but also for improving system resource utilization and the overall data maintenance process[21], especially in an environment where backups are performed on a regular basis[10][20]; this encourages a continuous search for improved backup products. Besides acknowledging the undeniable need for backup and restoration utilities in case of unfortunate data loss events, the decision to develop a backup software for the Linux file system was primarily born out of my interest in the diversity of procurable learning outcomes. Succinctly, I was interested in developing a Linux application entirely by myself, which would enable me to accumulate new skills throughout the course of the implementation phase of the system architecture, which is discussed in detail in Chapter 5.

My knowledge of this field was casual at first, knowing generally about software development from previous internships and academic projects, having basic understanding of the C++ programming language, and lacking proficiency with tools such as the Qt GUI application development framework, which

is integral to the implementation of the presentation layer. My former experiences are precisely what has guided my curiosity, which could be entertained with a project that gives the opportunity to resolve any of my unanswered questions regarding software engineering in general; as software engineering is a field that I am deeply interested in pursuing a career in, this project both appeases a genuine desire of enlightening my knowledge on this subject and undoubtedly eases the shift from academia to the early stages of my professional career.

The learning curve embarked on has been noticeably steeper than of any other individual projects that I have completed in the past. The combination of back end and front end components constitutes my first ever software development project culminating in a final product of this scale. Thus, it seemed a natural choice of project which I was sure would be both entertaining and profitable, considering I will most definitely be able to employ this experience in upcoming career interviews. This has been reflected in that even though my knowledge of the C++ programming language did not cater for the scale of work and my experience with Linux development was at a basic level at first, throughout the project I was able to acquire all knowledge necessary to implement a Linux desktop application that constitutes of both a data access layer and a presentation layer.

### 1.3 Project Aim

This project aims to build a user-friendly backup software application distributed with the Linux operating system. From the theory aspect, I have done much research into the principles of the Linux storage subsystem, which is split into two parts: the (1) file system layer and (2) storage devices support layer, which work together to determine how the kernel stores and retrieves data from secondary memory devices such as USB flash drives. Forming a complete understanding of the Linux file system is crucial to the project aim, as the project is structured around being able to easily communicate instructions to the Linux kernel in order to copy file system information from a Linux machine to a target storage device. To ensure a strong foundation of knowledge, other important areas must be researched, such as the management of Linux daemons, the use of hard links, handling file system events with the Inotify module of the Linux kernel subsystem, and establishing generic access to USB devices. Having extensive knowledge of related areas of research will provide grounds for a credible evaluation of the final product, as demonstrated in Chapter 7.

After accumulating relevant research, a backup software application is developed that consists of both a backup utility, which performs incremental backups of a file system, and a restore utility, which enables restoration of data made by the backup utility at a later time. It should be noted that this section constitutes a general aim of the project rather than comprehensive project specifications. See Sections 4.2 and 4.3 for a thorough outline of all specifications for the back end and front end components of the software, respectively.

### 1.4 Project Objectives

To ensure that the project aim, as detailed in Section 1.3, is met, a number of objectives must be delivered throughout the project. The project's objectives were classified into three sub-categories, which were based on the presumed time investment required to achieve them. Section 1.4.1 outlines the core objectives that are essential to complete. Section 1.4.2 lists the extension objectives identified as potentially within scope. Lastly, Section 1.4.3 discusses future objectives to provide a sense of how the project could develop further with additional resources, although they were exempt from the development cycle.

### 1.4.1 Core Objectives

The core objectives that are considered as fundamental to the project are listed below:

- *Identify, understand and review a range of industry-based design methods for performing file system backup.*

Information gathered in context of file layouts other than the Linux file system, such as the file-based systems and device-based dump strategies of the Write Anywhere File Layout, will be described with its relevance to this study.

- *Describe and gain an understanding of relevant advanced algorithms applicable to achieving efficient backup cycles to storage devices.*

Using the gathered information as groundwork, I will build up my own ideas on ways to integrate available algorithms to achieve the desired software behavior while retaining emphasis on performance, such as the combination of full and incremental backup algorithms to reduce required storage space per backup cycle.

- *Research, understand and review reliable industry-based software for data backup, and investigate the methods implemented for the back end component of the software as well as deriving inspiration from the presentation layer.*

Emphasis will be on the backup systems used widely, and how their techniques transfer to intermittent backup of file systems; not just in general terms, but the specific methods used to achieve overall efficiency and suitability, such as hard linking, program daemonization, or registering event listeners to listen for file system changes using a kernel subsystem.

- *Research applicable libraries, such as libusb and the Boost C++ Filesystem Library, and understand Inotify API programming.*

Since the software will involve direct API programming, it is familiarize myself with the facilities and restrictions that I will be working with.

- *Learn about the Qt application framework.*

To be learned specifically for the front end component of the software.

- *Design and build a backup utility capable of writing Linux file system data to a New Technology File System (NTFS) format storage device.*

This is the back end component of the software, which the technical details of are discussed thoroughly in Chapter 5. The NTFS file system format will be used as the only accepted format for storage devices registered with the software since it supports hard linking.

- *Design and implement a graphical user-interface with Qt that caters to the backup utility of the software.*

This will provide a front end to backup utility of the software, built to satisfy the specifications outlined in Section 4.3. Primarily, the user interface provides restoration capabilities, settings for configuration modifications and a search method through the stored versions of the source file system.

- *Evaluation of the results.*

This objective constitutes of a manifold project evaluation; a breakdown of what is to be achieved is contained in Chapter 7.

### 1.4.2 Extension Objectives

The possible objectives to extend the core requirements, if time is permitting, are:

- *Design backup cycles in such way that the timestamps of files copied to the target storage device are preserved. Achieved*

The challenge with copying files to the target device is that, even though most contents and metadata are sustained, the timestamp of a file is inevitably updated to the current machine time. Preserving the timestamp of backed up files by reassigning the modification time of the source file to the target file after the copy operation has been performed is not necessary for the performance of the software as a whole, but accounts for a better user experience since the backed up file system versions would exactly resemble the source file system.

- *Implement an additional software feature that enables the user to backup to a remote machine using the Secure Shell (SSH) cryptographic network protocol.*

A shell script would be written with a set of commands to perform equivalent logic as for locally connected storage devices, but file system data would be copied to a remote machine instead.

- *Change system service design such that sleep patterns of the daemonized program are dynamically adjusted to cater to backup cycle times and wake up at the exact time of impending backup cycle rather than at fixed intervals of 5 minutes.*

The potential of this objective is discussed more thoroughly in Section 5.1.1.3. Since this is not a difficult feature to implement, it is one to be explored rather than to be certainly integrated. Dynamically adjusting the sleep schedule of the daemonized program to wake up for the next due backup cycle would reduce the time spent between the scheduled and the actual times of a backup cycle. However, this is not a pressing issue, since this extension resolves discrepancies of less than 5 minutes in scope of relatively long backup intervals. Further, we must factor in that a user can adjust backup frequencies in the configuration.json file while the program is asleep, which means the daemonized program must be notified of this change and wake up to appropriately adjust the sleep patterns to ensure a complete and accurate implementation of this feature. Therefore, without further research this a potential extension rather than certain one at this time.

### 1.4.3 Future Enhancements

The following serve as advancements to the primary work planned set out to go above and beyond the core objectives, subject to any future plans regarding the project:

- *Build a Web-based application.*

In addition to a Linux desktop user interface, and web-based user interface would expand the control over the backup utility of the software to a web-based application. From there on, a new array of possibilities is born to expand the functionality of the software, such as providing an on-line help system.

- *Linux machine recovery capability.*

Include capability of entirely recovering Linux machine from a file system snapshot stored on a registered storage device. It should be noted that this is a difficult process. Even Apple's Time Machine backup is not immediately bootable through the user interface; although it does have a bootable hidden copy of the recovery HD installed, the user has to manually go through the process of partitioning and formatting. This feature differs from simply restoring the files from the file system snapshot, as the file system snapshot has to be bootable.



- *A storage device formatting enhancement to the UI settings.*

If a user attempts to register a USB device that is not of the required NTFS file system format, a formatting option will be offered as part of the user interface rather than relying on the user to reformat the device through Linux's Disk application.

- *Preserving hard links across file systems.*

The hard links created in the source file system could be preserved, such that a file is hard linked on the storage device accordingly instead of copying a new file, which would result in saving more storage space on the target device. This can be done by mapping inodes from source file system to those on the target file system; hence, when a hard link is created to an inode on the source file system, the corresponding inode is found for the target device and a file is hard linked to the correct path.

- *Utilizing the `IN_MOVED_FROM`, `IN_MOVED_TO`, and `IN_MOVE_SELF` inotify events.*

The listed inotify events could be utilized to monitor movement of directories and files within the watched source file system tree. As a result, methods could be developed to interpret movements of files to avoid unnecessary copies of files; for example, instead of copying a new file to the target storage device when a file is moved (as this is recognized as a deletion and creation action), appropriate actions could be performed. However, this is more complicated than might seem at first, since file system snapshots retain all information of the file system at that time, so the files would be hard linked as a result in the target device rather than moved; appropriate research would have to be conducted to achieve an accurate implementation of this objective.

## 1.5 Deliverables

The list below is a breakdown of what is to be achieved in a successful completion of this project:

- The project report, which incorporates literature research conducted prior to any software development, an outline of the project specifications and design, a discussion regarding the development process and, at last, a thorough evaluation and testing of the final product.
- A complete file system backup software distributed with the Linux operating system that, in addition to any enhancements, thoroughly satisfies the project objectives.
- A project presentation during which a system demonstration and an outline of the main features of the final product take place.

## 1.6 Literature Research

The literature research yields a discussion prior to the design and implementation of the software. The intent of background research is to acquire necessary knowledge and understanding of applicable material, related backup systems, and a precise understanding on how to achieve solutions to the development objectives outlined in Chapter 4. The survey of current trends and backup techniques serves as a prelude to establishing the fundamental features of backup systems, which is directly transferable to achieving the specifications set out for this project.

## 1.7 Report Structure

The structure of this report will go as follows:

The *second chapter* discusses the project management that is adhered to in order to accurately deliver the project objectives outlined in Section 1.4.1. The *third chapter* constitutes literature research conducted prior to any software development, which sets the background of the project by analyzing all off the relevant technologies that are used to build the final product. The *fourth chapter* provides a detailed outline of specifications to be met during the development of the software. The *fifth chapter* provides technical details regarding the system architecture of the final product. This includes the structure of the application, design principles, problems encountered and details of how they were overcome. Any deviations from the initial objectives are also discussed. A demonstration of the software in operation ensues to provide specific details on the system behavior and and to establish relevant context for discussion of the unified system architecture. The *sixth chapter* discusses components that are essential to the implementation of the system specifications described in Chapter 4. The *seventh chapter* contains the testing performed throughout the software development process, alongside an extensive evaluation of the results. The evaluation is achieved with consideration of the software specifications defined in Chapter 4, and in context of existing backup techniques and reputable backup systems discussed throughout the research phase of the project, as seen in Chapter 3. Finally, we will conclude on the final product and discuss challenges that have been encountered along the way. This includes a self-appraisal of my success in writing the software, attaining the project objectives and overall management of the project. At last, we will consolidate the results and point out areas of interest and improvements that could be made in future work on file system backup software.

## Chapter 2

# Project Management

### 2.1 Schedule

### 2.2 Methodology

With the aim to achieve the project objectives, the project is organized into three preeminent stages, which are the foundation to the structure of the project and allow for a methodical approach to achieving the project objectives set out in Section 1.4.1. The three main stages are outlined in chronological order.

#### 2.2.1 Stage 1: Research and Preparation

The first stage of the project constitutes of conducting a thorough amount of background research. Performing background research establishes an understanding of the state-of-the-art backup techniques applicable to intermittent backup of file systems. In essence, the aim is to become well versed in current trends in regards to both modern backup algorithms and commercial software, which are both fundamental areas of research for this project. It should be noted that the research and preparation stage spans past

research done solely on contemporary backup systems. The research stage at times delves beyond the scope of the initial background research shown in Chapter 3, as further research during implementation stages is required to understand relevant technologies after any initial project preparation. That is, to understand frameworks, shared libraries, IDEs and so on, in order to enable an intuitive realization of the project objectives; further research on these topics is discussed in Chapter 6.

The areas of research are combined and looked at for possible areas of overlap with the project objectives, and any applicable knowledge is leveraged for the software development part of the project. Essentially, ideas are derived regarding how intermittent backup can be performed in relevance to a file system backup software distributed with the Linux operating system, which forms the basis for this project. Once various possible solutions to file system backup have been approached during the research stage, an informed decision can be made regarding which backup techniques are most suitable to achieve the best realization of the software requirements. Thereafter, planning will begin for the rest of the project.

### **2.2.2 Stage 2: Design and Implementation**

During the second stage I will leverage the strong foundation of knowledge gathered in the research and preparation stages, which will allow me to derive potential designs and deployments of the system architecture when building the file system backup software. In addition to establishing the design for the technical architecture, which adheres to the outlined specifications in Chapter 4, this stage will yield both the implementation for the project and the bench mark for the evaluation stage. The design and implementation of the application most closely follows an agile sprint methodology, where the requirements and solutions evolve to conform to the needs of the project.

### **2.2.3 Stage 3: Evaluation and Conclusion**

In the third and final stage of the project, an evaluation will be achieved regarding the design and implementation details and of the final results. The details of methods used to achieve a thorough and accurate evaluation are outlined separately in Section 2.3. At last, a conclusion will provide closing thoughts on the results of the project, a self-appraisal and considerations to be made in future research.

## **2.3 Evaluation plans**

The main focus of the testing phase is on the performance of the backup utility of the software; while the front end provides a user-friendly control over the backup software, it has little to do with the core objectives of the software. Rather, the aim is to evaluate the performance of intermittent backup of the source file system, which is handled by the daemonized back end utility of the software. Further, the user interface mostly consists of rudimentary read and write functions to the configuration files and simple restore functions, so much more relevant and interesting is the evaluation of the back end of the software that handles the backup utility of the software. In short, the testing phase consists of forcing an array of changes in the source file system and observing how the backup utility handles these events with regards to satisfying the project specifications.

### **2.3.1 Testing methodology**

A combination of modifications to directories and regular files within the source file system are tested; the shell scripts to achieve this are included in Appendix D.1. In turn, the software's capability to manage hard links, file copy operations, pruning and performing full or incremental backups in response to file system changes is evaluated with respect to the desired system behavior defined in Chapter 4. By comparing the results a thorough evaluation is accomplished.

It should be noted that multiple sets of shell scripts containing file system modification have been tested overall, but a set of shell scripts equating to 8 different backup cycles is used in Chapter 7 to ensure a clear, yet detailed evaluation. Other than discussing project management, completed enhancements and comparisons to other reputable backup systems discussed in the literature phase of Chapter 3, the evaluation comes down to answering one important question: *has the project achieved what it set out to do?*

## Chapter 3

# Background Research and Overview

### 3.1 An introduction to the background literature research

In order to achieve software requirements, it is important to first compile relevant details on the problem at hand. This chapter discusses the background research relevant to the intermittent backup of file systems, which will provide the grounds for an informed evaluation of the final product as concerns the areas discussed throughout this chapter. Most especially, a rigorous research prior to the design and implementation stages of the solutions ensures an accurate realization of the preordained project specifications.

Apropos this project, the purpose of the background research can be best classified into three main pillars: to gain knowledge of current trends and relevant technologies used in the industry today, to review any applicable material (e.g. existing backup algorithms), and to gain a comprehensive understanding of how a solution to the problem could be implemented. In the following Section 3.2, we discuss commercially available systems that manifest similar objectives to that of this backup software, which are most carefully outlined in Chapter 4. This will serve as the foundation for the background research, which will enable further exploration in the areas discussed later in the chapter.

### 3.2 Existing backup systems

#### 3.2.1 AMANDA Network Backup Manager

The Amanda Network Backup Manager was founded at the University of Maryland at College Park to enable network backup of multiple UNIX machines in parallel[13]. Amanda takes advantage of UNIX backup programs such as tar and dump[3]. To achieve best possible performance, Amanda was designed to perform many backups in parallel to a temporary staging disk, which later writes data to a disk drive at high bandwidth[1][13].

Amandas backup cycles are determined by a central backup server host, which signals when to perform backups (most commonly in off-peak hours)[3], and also indicates the backup level to be performed for each distinct filesystem for a given cycle[25]. The Amanda backup manager provides optional file compression and full backup history for each filesystem[25].

Amanda takes error-checking measures prior to and after backups are performed[25]. Prior to backups, AMANDA verifies that the correct disk drives are used and that the staging disk has enough temporary storage[3][25]. After backups, the system produces a report that details any encountered problems; these may include backup program issues such software failure or permission errors, disk errors, or reporting

any client hosts that were down during the backup[3]. Although AMANDA can be considered as a reliable backup manager, there remain many limitations to existing backup schemes with regard to the increasingly demanding product requirements. These difficulties should be resolved by future work, as will be discussed in Chapter 7.

## **3.3 Design for backup systems**

### **3.3.1 Preference of strategy for this project**

Recent years have seen a discussion about the merits of file-based versus device-based backup schemes. However, there are very few systems in which the two schemes have been implemented with comparable attention to detail. Section 3.3.2.1 gives an introduction to the file-based backup strategy and Section 3.3.2.2 establishes why the file-based strategy is more suitable for this project over the device-based approach (Appendix A.3 provides more information on the design issues of the device-based strategy). Additionally, please see Appendix A.1, which discusses some of the performance and design flaws of the file-based approach (as embodied in WAFLs modified BSD dump) that I had to take into consideration when leveraging this technique for my project.

### **3.3.2 File-based vs. device-based file system backup**

#### **3.3.2.1 File-based strategy**

A file-based (or logical) system interprets filesystem metadata and determines which files and directories need to be copied to the backup device (details on use of metadata are explored in Section 2.2)[23]. Since the file-based strategy understands filesystem structure, the system is able to discover the files that need to be duplicated and write them to a backup device in canonical representation, which is highly desired for end-users who may not know much about the filesystem structure[18]. A filebased system reads the physical blocks of each file by traversing the pointers stored in each inode[18]. Thereafter, the backup software can store each file contiguously, which enables rapid single file recovery[3]. Unfortunately, writing each file contiguously to a backup medium means that more disk seek operations are required, which results in a decreased disk throughput and increased disk overhead[3][18]. These extra costly seek operations result in slower backup[3][18]. Another limitation of file-based backup scheme is that even minor a changes to a file demands that the entire file be backed up again[23]. Yet still, for reasons discussed in Section 3.3.2.2, file-based strategy is still more suitable for this project over the device-base approach; for that reason, the we will not delve into the specifics of the device-based strategy in this section, but can be found in Appendix A.2.

#### **3.3.2.2 Why a file-based strategy is the more suitable solution for this project**

We consider a full backup made based on a snapshot A. Snapshot B is created from which we desire to perform an incremental image dump. Figure 3 demonstrates the state of blocks as defined by the bits found in the snapshot bit planes. The objective is to perform an incremental dump for every block that is used in bit plane B but is not used in bit plane A, but not for any other blocks. We can also interpret the bitmaps to define sets of blocks, in which case we would perform incremental dump for blocks in the set B A.

An image dump should not be too reliant on the internal filesystem, otherwise the advantage of running at device speed is lost[18]. Hence, a device-based dump exploits the filesystem solely to gain access to the block map information, but bypasses the filesystem constructs and performs reads and writes directly by using the internal software RAID subsystem, which enhances performance[18].

Finally, the device-based dump provides several features that are not available with the file-based dump. While the file-based dump stores only the active filesystem, a device-based device can backup

all frozen snapshots of a live filesystem, which allows for a restoration of system that is identical to the one that was dumped[3]. Unfortunately, the two limitations discussed above - single file recovery and portability - continue to be inherent limitations of the device-based backup scheme.

### 3.4 Full and incremental backup algorithms

For this project, the full and incremental backup algorithms are combined to satisfy system requirements and optimise backup at a lower cost[8]. In the past, this has been seen for database systems. When a full backup is performed frequently, all copies of a database can be stored but at high cost[8]. Therefore, a backup technique was developed to ensure that data is safely stored at reduced backup time: an incremental backup, which only copies newly modified files since the last full backup, is executed at fixed intervals between the operation of full backups, which are performed at longer, prescheduled intervals[1][13]. This policy allows for performing the full backup, which has large overhead, fewer times - either at dispersed, prescheduled times or when the number of modified files in filesystem exceeds a predefined threshold value - while performing the incremental backup with small overhead at shorter intervals[8].

Seeking the optimal interval for a full backup will help minimise the expected cost of the database backup. In the context of database system, it seems straightforward that an incremental backup with small overhead should be employed to lessen the overhead of backups, and indeed, this is the case for most massive database systems[8]. However, the overhead of an incremental backup increases proportionally to the number of modified files in the system[8]. Therefore, it is highly beneficial to devise a suitable interval for performing full backups of a system. Although the overall performance of backup programs receives a great deal of attention from the industry community, end-user experience is also highly relevant, especially for commercial backup software. Some backup software demands that a filesystem stays quiescent while a backup is performed, which can harm user-experience and can be bypassed using an on-line backup approach[3][24]; such systems enable continuous access to the filesystem even during backup.

### 3.5 Summary

This chapter has evaluated literature research relevant to the realization of this project. We have demonstrated that, although challenging, it is possible for a backup system to find a balance in the performance of the backup and restore utilities. A file-based backup strategy enables recovery of single files and allows for a portable archival format. On the other hand, the device-based backup and restore strategy allows high throughput and supports multiple data replication strategies. Although both approaches have a lot to offer, the features of the file-based strategy are more suitable for the specifications set out for this project.

We classified the full and incremental backup algorithms, and shown that we can exploit the advantages of both schemes to improve backup performance for large systems by creating infrequent full backups with more frequent incremental backups. This discussion was prefaced by the description of the popular archiving tool AMANDA Network Backup Manager. As file systems grown size and continue to require more complicated backup maintenance, we can only expect to observe an array of new backup features and technologies making appearances in the industry.

# Chapter 4

## Development

### 4.1 Introduction

My project is a backup software application distributed with the Linux operating system. The software will make incremental backups of a filesystem, which will allow restoration at a later date of either individual files or an entire filesystem. The end-user will be able to perform a restore operation from the user interface. Each backup cycle will capture the latest state of the source filesystem. As the snapshots of the filesystem age, the more recent snapshots become prioritised over the older ones on the destination volume to save storage space; this is referred to as "pruning".

The software may be used with multiple external volumes. The default settings are such that 30 minute backups of a filesystem will be stored for the past 24 hours, daily backups will be stored for the past month, and weekly backups will be kept for snapshots older than a month. However, all backups older than a day will be kept at a daily frequency until half of the space on target storage device is used. The oldest weekly backups are deleted when the destination volume runs out of space.

The back end of the application will run as a Linux daemon, and will be responsible for the backup utility of the software. Since the back end is a system service void of direct end-user interaction, any necessary configurations will be read from a set configuration files, which contain parameters for the initial software setup and any details regarding the intermittent backup cycles to be performed on all of the registered devices.

The aim of the front end component of the application is to allow users to have control over the behavior of the back end component, mainly by enabling configuration file modifications through the graphical user interface (GUI), such as backup device registration and removal or omission of files from backup. However, limits to the user's flexibility will be placed to maintain appropriate software performance, such as the upper and lower limits of the backup frequency. The second and more notable responsibility of the front end is to provide access to the restore operations, allowing the user to easily recover data from the registered backup devices.

As suggested by the detail given to both components in Chapters 5 and 7, the front end is not the current main focus of this project, but rather the back end, which is responsible for the backup utilities. The combination of the components covers a rather large scope, so while the backup utility is the more intriguing part of the software, a frontend is actualized for a user-friendly management of the backup utility. To provide further comprehensive specifications, the following subsections below detail an overview of all specifications of the two of the primary application components: the back end and the front end.

### 4.2 Back end

- Runs as a system-level daemon (rather than user-level process), to allow processes to run even if

a user is logged off the machine; it is a system service that reads its configuration from a configuration file that can be modified from the user interface.

➤ The software works with locally connected storage devices. A folder on designated volume (disk image on a local drive\*) is created, onto which the software clones the directory tree of every locally connected disk drive, with the exception of files and directories that the end-user has chosen to omit (front end option) and the backup volume itself.

*\* A disk image can be thought of as a single file that stores the structure and contents of a disk volume.*

➤ Periodically, every X\* user-defined minutes (upper and lower limits are 5 and 1,440 minutes, respectively) after the creation of the original folder, the software makes another subordinate folder and backs up only those files that have been modified since the previous backup those files that files that have not been modified are not copied again, as the software will make hard links to defer to the existing files to save space.

*\* Upper and lower limits of X are 5 minutes and 24 hours, respectively.*

➤ A user will be able to search through the directory hierarchy of the backed up filesystem as if searching through a primary disk. An end-user should be able to manually search through the backup volume without the user interface; this is made possible by the use of hard links, which display each backup full disk copy.

➤ The software will be available to the Linux operating system it is limited to this operating system because the use of the inotify (inode notify) subsystem of the Linux kernel is central to its functionality, which is a module that is available only to the Linux operating system of the Unix family. The software operates on modification date of a file, which accounts for most intuitive design. This simply means that the system is concerned only with the last modified time metadata of a file, rather than executing MD5 checksums for a file, which can be inefficient for larger files e.g. large movie files. Therefore, the system operates on the assumptions that a file with new modification date but no changes to its contents is a rare occasion, as forcing such an occurrence does not pose any benefits to a user.

➤ The system has event listeners for modification of a file on the users filesystem, which is achieved by utilising the inotify subsystem\* of the Linux kernel that signals an event when a file is modified. This allows for being able to track directories that have been modified on the hard drive since the last backup. The software then needs to scan only those directories for modified files to copy in the next backup, since the unmodified files will be hard-linked, rather than having to check every file on the filesystem for its modification date every time a backup cycle occurs.

*\* Inotify operates to extend filesystems to notice changes to the filesystem, and report those changes to applications.*

## 4.3 Front end

➤ The GUI will be launched if a registered storage device is connected to the machine. In any other situation, a user can manually launch the user interface.

➤ The front end part of the software enables browsing of snapshots if a file system and the retrieval of directories and files within a particular directory via an animated user interface. This feature of the front end allows for a user to be able to recover a single file\*, but also to be able to entirely recover contents from the storage device to a Linux machine. This is so that a hard drive can be recovered from the backup in case that the users original disk loses all data (this can be achieved using the dd low level byte-for-byte copy utility).



- When restoring a subset\* or all contents from the backup device, the target disk needs to have enough storage space to accommodate this request. Otherwise, the front end the alert the user.

\* A user can recover a single file from the backup either by replacing the current file in the directory by the backed up file with the original modification date (in which case it is the users responsibility to be sure of this option, and chose Replace in the alert box), or to add the backed file to the directory of the current copy (Continue choice of the alert box). Alternatively, the user can Cancel the request.

- A user will be able to register and de-register backup volumes in the user interface.
- A user can register multiple volumes simultaneously, so an automatic back up of two (or more) separate destinations can occur in rotation. (Note: the software achieves this by rotating among the desired volumes each backup cycle in a manner that each backup is independent from the others.)

## 4.4 Configuration files for system communication

Configuration files are essential to the communication between the front end and back end components, but also to the performance of the backup utility alone. In lines 2 and 8 of Algorithm 3, which describes the behavior of the unified system architecture, we observe several referenced conditions that check for the existence of 3 configuration files, which must return true once when program first starts running and thereafter every time the process wake up. If the condition does not return true, the program creates the missing configuration file with the appropriate parameters, as the program cannot operate accurately without the three configuration files existing in the project directory: configuration.json, filemod.json and status.json.

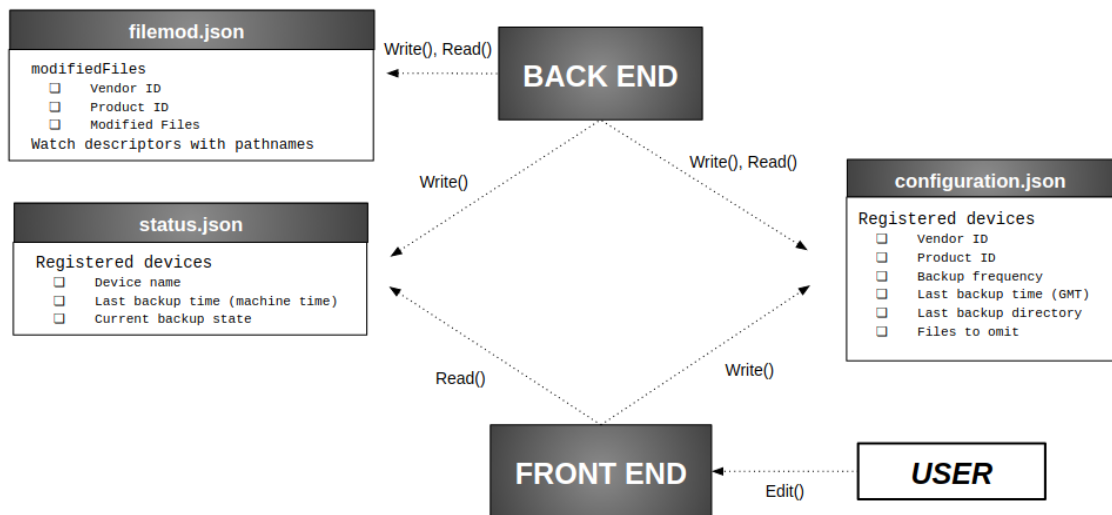


Figure 4.1: System communication

Figure 4.1 demonstrates the desired communication among the system components. Of course, it is possible that a user will directly edit a JSON files; that is, not via the user interface. This is not discussed in detail as it is not the intended use of the software. Rather, a user interface is provided that is, favorably, used by the user exclusively to adjust the software settings. Of course, it is necessary that cases are handled appropriately by error handlers if user is to directly edit a JSON file to produce JSON data not recognized by the backup utility.

Before resuming the discussion on technical system design and architecture in Chapter 5, we must first discuss the details of the configuration files, since much of the system architecture is built around externally storing and retrieving important information from JSON files.

#### 4.4.1 configuration.json

The “configuration” JSON file is responsible for any information related to the configuration of a backup cycle for all registered devices. Since each JSON object in the configuration.json file contains information related specifically to a single registered device, the program will create a JSON file with an empty “registeredDevices” object if the file does not exist in the project directory, as any previously stored information has been lost. Whenever an end-user registers a new device (this will most commonly be done through the presentation layer but can also be done manually), a JSON object will contain the device’s unique combination of vendor ID and product ID, the last backup time to the device in GMT, total time between backup cycles, the directory name of the last backup made to the registered device, and any files to be omitted from the device’s backup cycle.

```
{
  "registeredDevices": [
    {
      "filesToOmit": [
      ],
      "lastBackupDirName": null,
      "lastBackupTime": null,
      "productId": 25479,
      "vendorId": 1423
    },
    {
      "filesToOmit": [
        "/path/to/file",
        "/path/to/filep"
      ],
      "lastBackupDirName": "2018410121111",
      "lastBackupTime": 2018410131111,
      "productId": 1212421,
      "vendorId": 2408124
    }
  ]
}
```

Figure 4.2: configuration.json objects

#### 4.4.2 filemod.json

```
{
  "modFiles": [
    {
      "productId": 25479,
      "vendorId": 1423,
      "modifiedFiles": [
        "/home/martin/Desktop/TESTING/testmod.txt",
        "/home/martin/Desktop/TESTING/testmod2.txt"
      ]
    },
    {
      "productId": 1212421,
      "vendorId": 2408124,
      "modifiedFiles": null
    }
  ],
  "watchDesc": [
    {
      "path": "/home/martin/Desktop/TESTING",
      "wdInt": 1
    },
    {
      "path": "/home/martin/Desktop/.ipynb_checkpoints",
      "wdInt": 2
    }
  ]
}
```

Figure 4.3: filemod.json objects

The “filemod” JSON file is responsible for any information related to modified files returned by the inotify instance. The information contained within the filemod.json file is kept separately to the other configuration files because it contains only information generated by the inotify instance. This information is split into two JSON objects: modified files to be backed up for each registered device, and pairs of watch descriptors and its respective paths to enable retrieval of paths to modified files.

#### 4.4.3 status.json

The status.json exists to provide any relevant information to the front end of the application. The JSON data constitutes of

all registered devices and the relevant information to each of the devices: device name, the last successful backup time in machine time and backup cycle state (whether backup is occurring at the moment or not). Essentially, the status.json file contains all information that will be displayed in the user interface with regards to the backup utility part of the software. This information is first read when the user interface starts running so it can be initiated with the appropriate information, and an event listener is placed on changes done to the status.json to update the presentation layer accordingly with any changes occurring in the backup utility.

#### 4.4.4 Technical details regarding configuration files

Section 4.4 establishes the configuration files used for system communication and why certain data has been allocated to its respective JSON file. Appendices B.1, B.2 and B.3 explore additional information regarding the specific technical details of the configuration files;

```
{
  "registeredDevices": [
    {
      "deviceName": "DISK_IMG",
      "lastBackupCycle": "2018410131111",
      "backupState": "InProgress"
    },
    {
      "deviceName": "MARTIN",
      "lastBackupCycle": "2018410133451",
      "backupState": "InSleep"
    }
  ]
}
```

Figure 4.4: status.json objects

these are strictly supplementary details, as they do not relate to the system architecture of the backup utility directly but may be of interest to the reader.

## Chapter 5

# Technical Design and Architecture

### 5.1 System architecture

The developing of the system architecture is the third phase of the project, preceded by the imperative stages of background research and delineation of the project specifications. All relevant elements categorized under the technical structure requirements, the system buildup and the behavior of the file system backup software will be thoroughly discussed. This chapter mostly discusses the technical architecture of the backup utility, but the restoration implementation in the front end is briefly discussed as it is the main functional feature of the component.

#### 5.1.1 Design decisions and flaws for system build-up of fundamental components

With respect to the system requirements, the following subsections detail the fundamental components of the system architecture. In general, the system is tightly coupled with the Inotify API, makes extensive use of hard links, and exists as a daemonized system process, which is enabled by the Linux multitasking operating system. As displayed in Figure 4.1, the back end communicates any necessary information via the configuration files and the end-user interacts via the presentation layer of the file system backup application. The system architecture itself is therefore intended to be independent of any affecting environments, and is build-up in a modular and component-based way. The advantage of such a structure is having the possibility of independent parts interacting through defined interfaces, which makes the structure flexible and maintainable as will be discussed in the following subsections.

##### 5.1.1.1 Inotify module

Inotify is a Linux kernel subsystem that extends file systems in order to monitor any changes, and reports modifications to those applications interested in the event. The most important use of inotify is to have event listeners for file system events as opposed to repeatedly iterating through the entire filesystem to determine changes.

The Inotify module is inode-based, which is emulated in that it's an abbreviation for 'inode notify'. As an inode-based mechanism, the module enables a signal triggering for an event on any link to a monitored file; in other words, an event is triggered for any number of linked files across different directories. However, it should be noted that the Inotify API specifies that the file itself must be monitored and not merely the directory containing the file. This plays an important role in the implementation of the Inotify API when monitoring file system changes, as demonstrated by the pseudocode of Appendix C.3.

Let's first thoroughly examine the pseudocode of Appendix C.3, which provides the appropriate context for any further discussion. The core function of Inotify is to act as file change notification system—a kernel subsystem that allows applications to listen for a list of events on a set of files. By recognizing over

20 file system events (the events relevant to this software are listed in Figure 5.1), Inotify proves to be a fairly flexible and fast solution to the file change notification problem; this goes a long way for large-scale application that often watch thousands of files for changes.

Appendix C.3 provides an informal high-level description of the operating principles of my Inotify implementation. As observed, my implementation does not exist for the program to return modified files, seen in the use of the ‘procedure’ pseudocode structural convention over ‘function’. Rather, the Inotify feature operates in a modular and component-based way to allow integration with other fundamental features of the software. With the respect to the unified system architecture, this is to allow the software to perform recurring inquiries about modified files, perform specific function on the result set, and thereafter continue running as a background process. Before discussing the integrated system in latter sections of this chapter, let’s first discuss the design of the independent Inotify feature as a necessary prelude to assessing the unified system architecture.

### Initialization

Assuming the employed C library supports inotify and the appropriate files have been imported, the instantiation of an inotify instance is rather simple. We initialize via the `inotify_init()` system call, which creates an inotify instance inside the kernel and returns an integer file descriptor. A file descriptor is imperative to enable access to the file description in the kernel associated to the instance. An instantiation of an inotify instance is a first step to utilizing the Inotify API, as observed on line 2 of Appendix C.3.

### Adding and removing watches

The heart of the Inotify module is the ability to ‘watch’ files, which constitutes of a pathname designating what file to watch and an event mask specifying what events to watch for on that given file. This utility is largely responsible for why it is so enjoyable to work with the Inotify API; watches are available for an array of different events - reads, writes, opens, closes, moves, creates, deletes, unmounts, and metadata changes - truly providing flexibility as listeners can register for distinct events to manage specific behavior of the system. Moreover, a single inotify instance has the capacity to add a watch to thousands of files and each distinct watch can be added with a different list of a combination of events.

Watches can be added with the `inotify_add_watch()` system call. A call to `inotify_add_watch()` adds a watch for the one or more events given by the bitmask mask on the file path to the inotify instance associated with the file descriptor. On success, the call returns a watch descriptor, which is used to identify this particular watch uniquely.

Event	Description
IN_CREATE	File is created in watched directory.
IN_MODIFY	File was written to.
IN_ATTRIB	File’s metadata (inode or xattr) was changed.
IN_DELETE	File was deleted from watched directory.

Figure 5.1: Used inotify event masks

Figure 5.1 shows the set of monitored events used when creating new watch items within the file system watch list. On line 3 of Appendix C.3 we observe that all watches are recursively added to the root directory and all of its sub-directories. In context of a backup software, the designated root directory can be any path parameter, but most desirably will be the root directory of the file system. Adding a watch to a directory with the correct events resembles the following:

```

int wd;
wd = inotify_add_watch(fd, "/path/to/file", IN_CREATE | IN_MODIFY
    | IN_ATTRIB | IN_DELETE );
if (wd < 0) perror ("inotify_add_watch");

```

Here, the ‘fd’ variable refers to the file descriptor of the inotify instance.

The ‘wd’ variable is the watch descriptor referring to the newly created watch, and is a unique integer returned by `inotify_add_watch()`. Since the Inotify API uniquely identifies events via watch descriptors, they are imperative for retrieving the path to a file that has triggered an event. It was disclosed by the Linux Programmer’s Manual on Inotify that “it is the application’s responsibility to cache a mapping (if one is needed) between watch descriptors and pathnames”[15]. Therefore, as observed in lines 6 and 22 of Appendix C.3, each time a new watch is added, the unique watch descriptor and its respective path must be stored externally by the application. In context of my application, this is done by storing a JSON object with two members: a watch descriptor and a pathname. The member values can be retrieved at a later time to restore the path to the file that caused the event based on comparing the unique watch descriptors, as observed on lines 15 to 18 Appendix C.3.

### Receiving Events

As demonstrated by line 18 to 20 of Appendix C.3, a watch must be added to extend the initial watch list every time a new directory is created to sustain a correct monitoring of the target directory tree. This can be achieved by listening for the `IN_CREATE` event, and adding a watch if the file that triggered the event is a directory. While `IN_MODIFY` also covers a creation of a file, we utilize the `IN_CREATE` method uniquely for adding watches to any new subdirectories of a watched directory, which is why it must be included as part of event mask.

Including `IN_DELETE` as part of the event mask is necessary because the `IN_MODIFY` event mask does not notify the system when a file is deleted. Hence, we must listen for the `IN_DELETE` event to determine when a file is removed and omit it from the proceeding backup cycle. Although we listen for the `IN_CREATE` to perform an `inotify_add_watch()` function on any new directories, the implementation of the `inotify_rm_watch()` is not in fact necessary when a directory is deleted since a watch is automatically removed from the watch list when a file is deleted or the system is unmounted. The `inotify_rm_watch()` method will, however, become more relevant when the end-user of the software chooses to omit certain directories from backup. Nonetheless, as observed on lines 22 to 23 of Appendix C.3, it remains important to inquire for an `IN_DELETE` event caused by a deletion of a directory so that the watch descriptor and pathname values of the affected file can be removed from the externally stored JSON file. This way, only relevant information of currently watched directories is maintained in the configuration files, and thus efficiency is retained when iterating through the list of pairs later in time; the JSON file containing the watch descriptors and pathnames of watched directories should not be cluttered with information on previously deleted directories, which is resolved by including the `IN_DELETE` event mask when adding watches to directories and removing a deleted directory’s data from the configuration file when the `IN_DELETE` event mask returns true.

After having inotify instantiated and watches added, the application is prepared to receive events. All information associated with an event is contained within a distinct `inotify_event` structure, which is displayed in Figure 5.2.

```

struct inotify_event {
    __s32 wd; /* watch descriptor */
    __u32 mask; /* watch mask */
    __u32 cookie; /* cookie to synchronize two events */
    __u32 len; /* length (including nulls) of name */
    char name[0]; /* stub for possible name */
};

```

Figure 5.2: Inotify event structure [15]

### 5.1.1.2 Hard links

In regards to Linux or other Unix-like operating systems, a hard link allows to assign an additional file name to an already existing file. An inspiration for an intuitive implementation of hard links was derived from Apple’s Time Machine backup software distributed as part of the macOS operating systems. It could be argued that purpose of using hard links boils down to the simple intent of saving storage space. For software of exceptionally repetitive nature like incremental backup, leveraging hard links can result in considerable amounts of saved storage space even between only two consecutive snapshots of a file system, specifically as a result of avoiding unnecessary allocation of memory for unchanged files.

The original file name and any hard links made later in time will all point to the same inode, which is the crux of why hard links are so useful for this application; the user will not be able to tell a difference between two files in separate backup directories pointing to the same inode, as well as avoiding unnecessary memory allocation for unchanged files in consecutive backup cycles. It should be noted that not all file system formats support hard links. Therefore, the software ensures that the designated NTFS format is used for all registered storage devices. If the user attempts to register a device of different file system format a notification is generated, informing the user to either chose a different device or to reformat it.

While creating hard links is not a complicated matter in and of itself, the problem stems from integrating the use of hard links to accurately achieve the desired system behavior. Let’s consider Algorithm 1, which details the project’s technical implementation of hard links as part of the complete system architecture.

---

**Algorithm 1:** Hard links as part of the system architecture

---

**Result:** Successful hard linking of regular files in consecutive backup cycles  
**Input :** (*srcPath*): The source path of the file to be linked  
**Input :** (*dstPath*): The destination path of the linked file

```

1 procedure Hardlink (srcPath, dstPath);
2 if srcPath file is a directory then
3   create directory in dstPath;
4   for sub-directory in srcPath do
5     newSRC ← path to sub-directory;
6     Hardlink(newSRC, dstPath);
7   end
8 else
9   create hard link between srcPath and dstPath;
10 end

```

---

When called with the parameters of latest backup directory path and the new backup directory path, Algorithm 1 performs the preparation measures taken prior to a new backup cycle. The algorithm is responsible first for copying any directories from the preceding backup (the snapshot of a file system directly preceding the current cycle), and thereafter hard linking any regular files, effectively recreating the tree structure of the preceding snapshot with minimal use of storage space. This is all done in preparation of

a new backup cycle and strictly occurs in the confines of the target storage device; the only files that the software is concerned with at this stage are those that exist on the storage device. Thereafter, any consequent actions may be performed, such as copying any new directories or modified files from the source file system to the target storage device.

The creation of hard links is astoundingly simple as it can often be attributed to a single line of code, such as the `create_hard_link()` method of the Boost C++ Filesystem library. What is rather of interest to us is what we can achieve with hard linking in context of an entire system. Let’s consider Appendix C.1.

Figure of Appendix C.1 can be understood through two strategies that were extensively discussed in the background research section of this paper: full and incremental backups. Initially, we observe that a full copy of all contents under the source directory are copied to the target storage device. Thereafter, only the changes to the source file system are applied in incremental backups by appropriately copying or deleting modified files retrieved from the `filemod.json` file. What hard linking enables is to entirely eliminate the need to assign new inodes for all files every time a new backup cycle is due. Instead, a snapshot is created that resembles the current state of the source file system by hard linking all unchanged files from the last backup and only copying or deleting modified files. Therefore, if no files are modified from time of last backup, the next snapshot is simply an exact copy of the directory tree with hard links made for every regular file to the same inodes of the previous backup, which is precisely discussed by the

supplementary description of system behavior in Appendix C.5.

This section discussed the use of hard links for recreating the former directory in a backup cycle, which is an identical process for all backup cycles regardless of file system changes. The steps performed after the procedure described by Algorithm 1 to recreate an accurate snapshot of the current file system, such as copying or deleting any files modified in the watched source file system, are outlined by the unified Algorithm 3.

### 5.1.1.3 System service

The rudimentary purpose for the application to exist as a system service, or also known as a daemon in the Unix world, is rather straightforward: the program should run as a background process, rather than relying on the direct control of an interactive user. In Algorithm 2, we observe that the initial parameters for the daemonized program determine when the process is sleeping and when it is running. This means that the sleeping patterns of the process that are set programmatically have the underlying control of when the system inquiries for modified files and when backups occur.

Let's consider Algorithm 2, which details the process of daemonizing a program and how the daemon is managed to accommodate the behavior of the system.

---

**Algorithm 2:** Daemonization of the application

---

**Result:** Successful daemonization of the application  
**Input :** The root directory of the project *rootDir*

```

1 procedure Daemon (rootDir);
2 fork off the parent process;
3 if fork off was successful then
4   | let the parent terminate;
5   | child process becomes session leader by making it in-dependent;
6 end
7 implement a working signal handler;
8 fork off for the second time;
9 if second fork off was successful then
10  | let the parent terminate;
11 end
12 change the working directory to rootDir ;
13 close all open file descriptor;
14 open log file;
15 while process is running do
16   | perform all backup utilities (detailed in Algorithm 1);
17   | process sleeps for 5 minutes;
18 end

```

---

In lines 2 to 14 of Algorithm 2 we observe pseudocode that describes the steps taken to daemonize the program that is generally applicable to any C++ application. This procedure is fairly universal as there are predetermined steps that must be completed to accurately daemonize a program, so we will not discuss in great detail how a program is daemonized, as a general idea is given by the lines 2 to 14 of Algorithm 2. Rather, we will discuss the part of the system that is responsible for appropriately managing and accommodating the fact that the program is a daemon so that the software behaves as desired.

Lines 15 to 17 describe the block responsible for how the daemonized program is

managed while the process is running. In Chapter 7, we observe that frequencies at which modified files are inquired are fairly short. However, that was a time parameter chosen merely for discussion purposes. As observed in Algorithm 2, the minimum set time between when the process is sleeping and when it is running is by default a 5 minute interval. This means that the system queries for modified files every 5 minutes, but also that the minimum possible time between two backups is 5 minutes.

The predetermined time variable of 5 minutes was decided based on a specific technical reason. In context of the entire system, the program will inquiry for any modified files every 5 minutes and write them to an external JSON file. Each time this occurs, the time difference between the current time and last backup time of each registered device will be compared to the desired backup frequency of the respective device (both data retrieved from the configuration.json file), which will allow the program to determine which devices are due for backup after querying for modified files.

Of course, the procedure described by lines 15 to 17 of Algorithm 2 has its flaws; it may be that a backup will be made some time after it was due to occur. However, since the lowest common denominator of time at which backups to devices can be made is 5 minutes, this is not a cause for concern. On the other hand, daemonizing a program has its clear benefits, since the program is not wasting unnecessary CPU resources and can perform independently without any user interaction.

## 5.2 Unified system architecture

---

**Algorithm 1:** Unified system algorithm

---

**Result:** A backup utility that satisfied the software requirements outlined in Chapter 4

**Input :**

```
1 System ();
2 if if any of the 3 configuration files are missing then
3   | create missing configuration files and instantiate appropriate JSON objects;
4 end
5 daemonize the program (precisely as detailed in lines 2 to 14 of Algorithm 2);
6 implement inotify (add watches precisely as detailed in lines 3 to 6 of Appendix C.3);
7 while process is running do
8   if if any of the 3 configuration files are missing then
9     | create missing configuration files and instantiate appropriate JSON objects;
10  end
11  modified_files  $\leftarrow$  any modified files since last iteration (as detailed in Appendix C.3);
12  if modified_files is not empty then
13    | for every registered backup device do
14      | insert modified files to filemod.json file;
15    | end
16  end
17  for devices due for backup do
18    | if device is connected then
19      | if target backup directory or last backup directory name and time for device does not exist then
20        | recursively recreate full copy of source file system to target storage device;
21      | else
22        | recreate copy of last backup with hard links on regular files (observed in Algorithm 1);
23        | modified_files  $\leftarrow$  any modified files since last iteration;
24        | sort modified_files from shortest to longest path;
25        | for file in modified_files do
26          | if if file not to be omitted from backup then
27            | get destination path to storage device from path of modified file;
28            | if source file does not exist and destination file exists then
29              | remove file at destination path;
30              | continue
31            | else if source file path does exist and destination path does not exist then
32              | copy source file to destination path;
33              | update file timestamps to that of source file;
34              | continue
35            | else if source file and destination file exist then
36              | if source file is a directory then
37                | update destination directory metadata to that of source directory;
38              | else
39                | copy source file to destination path;
40                | update file timestamps to that of source file;
41              | end
42            | end
43          | end
44        | end
45      | end
46 end
```

---



The file system backup application operates in a rather repetitive fashion, and its general behavior can be attributed to a fairly straight forward set of steps: the process is initialized and while the process is running the system intermittently inquiries for any modified files that are then backed up to a number of registered devices. However, some of the technical decisions behind achieving this desired behavior are complex. While we have not necessarily discussed the low-level details of the system architecture, such as the code responsible for a specific function, we thoroughly highlighted the fundamental design decisions imperative to the system architecture.

Algorithm 3 relates to the algorithms discussed earlier on, as seen from the references made on lines 5, 6, 11 or 22. As established earlier, some of the fundamental components of the software such as notify instantiation, hard linking and daemonization of the program are implemented in a modular way, which why the system architecture could be unified in Algorithm 3. It could be thought about in such way that subsections of Section 5.1.1 act as a prelude to the unified Algorithm 3, since incorporating the details of each component into one algorithm would be too exhaustive for a single pseudocode description. Nonetheless, each of the components discussed in the subsections of Section 5.1.1 constitute a highly relevant and imperative facet of Algorithm 3, and are appropriately cited in the algorithm that describes the unified system.

While Algorithm 3 was born out of the discussion of other fundamental components, a supplementary set-by-step explanation of the algorithm is included in Appendix C.2.

### 5.3 The front end restore utility

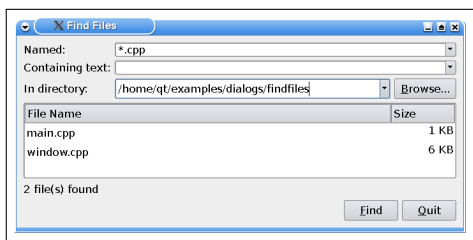


Figure 5.3: A Qt implementation of restore utility[27]

The restoration utility provided by the presentation layer is perhaps the only considerably complex operation that exists as part of the front end component; that is, of course, a good sign, as the front end logic should be minimal and should not be responsible for any of the backup utilities.

The restoration utility is born out of an integration of several Qt functions: `findFiles()`, `showFiles()`, `createFilesTable()` and `createComboBox()`. Of course, more methods are used in scope of the whole utility, which are implemented as part of a Qt Window class, which

inherits `QWidget`. Some of the logic requires using the `QDirIterator` to iterate through the resulting files that matched a file name, which is then used to create a `QStringList` of paths. Without getting into the low-level specifics, the resulting utility is embodied in a search window, which can be used to search for files in specific directories or files that contain certain text. The results are displayed in a table; similarly to the backend utility, the user specifies a destination path and a `copy()` method is used to copy the file.

### 5.4 Design deviations

The core project objectives had not changed since the research and preparation stages; that is, most broadly speaking, that the software performs intermittent backup of the Linux file system. However, several design deviations regarding the fundamental components of the software should be noted, as discussed in the following subsections.

#### 5.4.1 Deviations to hard link implementation

Although hard links were the certain tool to use from early on in the project planning, there was a brief consideration of symbolic links; although ruled out almost immediately, it is worth discussing, if not only to outline the clear advantages of hard links over other utilities.

Although symbolic links can span over file systems (opening up discussion about the possibility of links spanning from the source file system to the target storage device), this utility is highly unsustainable with regards to the objectives of this project. For example, all symbolic links that point to a file that has been deleted are also deleted, and not when all files to the inode are deleted like in case of hard links. In the case of deletion, the linked file becomes an "orphaned" or "dangling" file. This would complicate pruning; if a full backup is erased during pruning, all symbolic links made to that directory would be orphaned, while hard links would continue to point to their underlying inodes.

Further, when a hard link is created by the software, there is no obvious indication that it is any different from any other file. That is, hard links appear to be files of the same type as their target files (i.e., the files to which they are linked) when they are viewed with commands such as `ls` (i.e., list) and `file` (which is used to determine the type of any specified files). Likewise, when viewed in a GUI, the icons for hard links are identical to those for their target files. This is precisely desired in this file system backup software, as the use of hard links in consecutive backups should not be apparent to the end-user.

### 5.4.2 Deviations to management of the system service

Initially, the intended management of the daemon was going to involve updating the sleep patterns of the process dynamically rather than keep a fixed interval of 5 minutes, so that the process wakes up immediately prior to the next backup cycle; that is, eliminate as much time between the scheduled and actual times of an impending backup. This could be done by checking the time until the most recent backup of any registered device and adjust the sleep time of the process. However, due to the time taken to perform backup operations and the fact that this is a design consideration regarding less than 5 minutes (which is the maximum time of waiting before an impending backup cycle), it is more desirable to keep the sleep pattern at a fixed interval to avoid any unnecessary software complications. Also, it is most likely that the user will want to backup to devices at longer intervals, in which case a maximum time difference of 5 or less minutes to when the backup cycle was supposed to happen is not of huge concern. Perhaps most notably, a user can change the backup frequency of a device in the `configuration.json` file while the daemonized program is asleep, which would mean that for this design to be implemented thoroughly, the program would have to be signaled of this change and appropriately update the sleep pattern, otherwise it is not a complete implementation as the schedule backup time and actual time of backup would still not be synchronized.

During initial planning, the daemonized program was going to be designed to handle signals from other software components such as the user interface; however, this did not satisfy an intuitive design for numerous reasons. The presentation layer, for example, does not facilitate communication with the daemon through signals to the relevant process identifier (pid), but rather simply through writes to the configuration files, which the program reads when it wakes up. Since the program must be notified of any newly registered devices, changed backup cycle frequency, or newly modified files of the target file system, which in addition to other information is all contained in JSON files, it is most architecturally intuitive to simply have the process read from configuration files upon waking up, instead of signaling to inform the process of any of the listed changes.

## Chapter 6

# Technical Implementation

### 6.1 Languages

#### 6.1.1 The C++ programming language

Both the front end and back end components of the software were written exclusively in the C++ general-purpose programming language. In Sections 6.1.2 and 6.1.3 we discuss in more detail why C++ proved to be an appropriate choice for a programming language for both the back end and front end components of this desktop application.

#### 6.1.2 C++ for the back end component

As mentioned throughout the paper, the file system backup application is distributed exclusively with the Linux operating system. Although not mandatory, C++ is a popular choice for Linux development, since the Linux kernel was originally written in C (in 1991 when the Linux kernel was officially released C++ was not yet popular)[15].

Although C was invented for system development rather than application development, which is the focus of this project, there are still as many arguments to be made for the use C++ in application development as against it. Thus, the answer to why I have chosen C++ is a rather philosophical one. A significant reason why I chose to implement my application in C++ is because I will be applying for software engineering positions after academia, and C++ is yet another language to add to my skillset.

#### 6.1.3 C++ for the front end component

The decision to use the C++ programming language for the front end was partly a result of deciding to use Qt, which is a cross-platform application framework used for creating application software. In turn, the decision to use Qt stems from the fact that the framework provides a set of tools to enable front end development, which can normally be burdensome due to differing sets of development tools.

Qt Creator is a cross-platform IDE for C++ and QML and is used to entirely develop the presentation layer of the software. Due to the use of the Qt Creator IDE, C++ was the obvious choice of programming language for the front end, which happens to be complimentary to the use of C++ for the back end component of the software. Since the decision to implement the front end in C++ was a result of using Qt Creator rather than other factors specific to the language, we will thoroughly discuss the decision to use the Qt framework to develop the front end in Section 6.2. It should be noted that the choice to use C++ was arbitrary or ill-informed, since other languages such as Python may be used with Qt as well, although less commonly[27]. Writing in C++ accounts for great control, the opportunity to implement excellent libraries such as Boost, and any written code will compile to native binaries that will execute at full speed without the need for a virtual machine[27].

## 6.2 Qt

The Qt application framework constitutes one of the main components of the front end implementation. Upon some time conducting research, the consensus seems to be that the Qt toolkit is arguably one of the best free graphical user interface solutions for C++ currently available on the market.

Qt supports the GCC C++ compiler and uses C++ with extensions. One of these extensions are signals and slots[27], which is a language construct introduced in Qt for communication between objects; this construct simplifies event handling and enables easy implementation of the observer pattern without boilerplate code[27]. The essence of it is that GUI widgets are able to send signals containing certain event information, which is in turn received by other controls with the use of special functions known as slots. This is extremely helpful when developing a GUI that receives its own set of events and must process the information accordingly. For that reason, the signal and slot feature is highly relevant to the front end of this software, since it must listen for the event of a USB device connection to the machine and update the presentation layer block accordingly to display currently connected USB devices.

## 6.3 Libraries

In this section we discuss the external libraries imperative to the implementation of the software. Both the Boost Filesystem Library and libusb have been used extensively to accomplish the desired system behavior outlined in Chapter 4.

### 6.3.1 The Boost C++ Filesystem Library

The Boost Filesystem Library offers the ability to not only easily manage files and directories, but the paths that contain them. For this reason, this library is hugely advantageous in the implementation of the filesystem backup application.

In practice, the Boost Filesystem Library has provided many immensely useful methods to achieve an intuitive realization of the project objectives. Some of these methods include `recursive_directory_iterator()`, `is_directory()`, `is_regular_file()`, `copy()`, and `remove()` [12]. These methods can be used to achieve any of the imperative system behaviors discussed throughout in Chapter 5, such as recursively adding watches to directories of the source file system or copying the directories and performing hard links on regular files at the beginning of a new incremental backup cycle.

### 6.3.2 libusb

The library libusb provides a generic access to USB devices. The rudimentary reason why libusb is utilized throughout this project is to facilitate communication with USB hardware, and to retrieve information on any USB devices; this can be anything from determining whether a USB device is suitable as a target backup device to establishing whether a device due for backup is indeed connected to the machine

## 6.4 git

GitHub, a web-based repository hosting service, was the obvious choice for version control. Apart from being one of the world's leading software development platforms, I have a previously acquired personal experience using GitHub during many university projects and, more intensively, during my last year's software engineering internship. Since I was already familiar with GitHub's distributed version control prior to the project, it proved to be an obvious service of choice for me to use for this project.

Project link: <https://github.com/MartinSova/ECM3401>.

# Chapter 7

## Evaluation and conclusion

Throughout the chapter, the project is evaluated against the software specifications outlined in Sections 4.2 and 4.3. The findings are delivered through comprehensive testing of the final file system backup software, which will reveal that project objectives were indeed satisfied.

The states of backups captured in between backup cycles to provide a very clear illustration of the system behavior, such as increasing link count and new allocation of inodes, have been demonstrated in the supplementary Appendix C.5. Although not as extensive as the testing that will be shown in this chapter, Appendix C.5 provides the user with a first insight in that the software performs correctly under given circumstances of small-scale. We have also delved into the theory of the system architecture components in great detail, such as the use of the inotify module and hard links, highlighting the importance, implementation details as well as limitations thoroughly throughout the paper. The evaluation section will, in turn, focus on the resulting states of 8 consecutive backups after the source file system has undergone an array of extensive changes and how the results satisfy the software requirements; that is, staying away from the theory and pseudocode, which has been established thoroughly in Chapter 5. This will provide basis for an accurate evaluation of the resulting performance.

First, we must define the all parameters set out for the testing phase, upon which an accurate evaluation of the results relies. The configuration of the backup system will be different in real world use, but was altered for the testing period to enable clearer results, and so certain aspects, such as the pruning of older file system versions on the target storage device, can be tested in a reasonable time frame.

The list below defines all relevant parameters necessary for understanding, interpreting and evaluating the results discussed throughout the chapter:

- Software configuration: The **sleeping state** of the daemon is set for **12 seconds**, which i deliberately shorter than the backup frequency to ensure that modified files are inquired in time. This is more reasonable than waiting 5 minutes for file system changes to be recorded during the testing period, yet allows for enough time to perform any desired operations in one sleep cycle before the changes are recorded by the software; such as creating, modifying or deleting a file in the watched directory tree. The sleep pattern frequency is also not be too short, such as 4 seconds, as that is unnecessarily short and would result on constant writes to the configuration files.
- Target storage device configuration: The **backup frequency** of the **target storage device** is set to **15 seconds**. This allows for enough time to ensure that all necessary modifications to the source file system can be made via shell scripts for at least two sleep cycles, given the additional time of performing the operations. The configuration.json contains only the ID of the target device and the target device itself is empty as to imitate a newly registered storage device for a suitable testing environment.
- Source file system: A **directory tree** is constructed in light of demonstrating the system behavior, which will be the root watched directory of the software. The directory tree is constructed as an accurate imitation of a file system to establish a thorough and all-inclusive testing environment;

that is, providing a combination of files, such as subdirectories, regular files or hard links. The **initial contents** of the directory tree of the source file system are displayed in Appendix D.2.

➤ *Pruning changes:* The pruning for the testing phase is as follows: all backups are kept for 6 minutes, 2 backups are kept for 10 minutes, and 1 backup is kept past 10 minutes. Again, these parameters are very different the pruning frequencies established in Section 4.3, since the listed frequencies were chosen specifically to allow for manageable testing and evaluation phases. The backup frequency is set to 60 seconds to enable clearer pruning performance.

Before continuing in the discussion of results, we should make note of the shell scripts used for testing, displayed in Appendix D.1, which contain the modifications performed for each consecutive backup cycle to thoroughly test the performance of the backup utility.

## 7.1 Initialization

### 7.1.1 Inotify

First, it is interesting to observe that inotify watches are actually added when the process begins. We cannot observe whether watches have been added for the directories in the source file system directly via Linux commands, although concrete evidence of this can be found in the project code itself, the overall system behavior (in that the software must have added watches as it registers the file system changes to successfully complete very specific operations throughout backup cycles outlined in Appendix D.3), and relevant system log information, but this information is not entirely convincing in and of itself. Fortunately, we can prove that an inotify file descriptor that belongs to the process has been created.

```
martin@martin-MacBookPro:~$ pgrep dissertation
7579
martin@martin-MacBookPro:~$ sudo ls -l /proc/*/fd/* | grep notify
lr-x----- 1 martin martin 64 dub 27 17:15 /proc/7579/fd/1 -> anon_inode:inotify
```

The two Linux commands in displayed above were called after running the process via the Linux terminal. The process was initiated at 17:15 through the Linux terminal with a process identifier (PID) of 7579. As observed above, an inotify file descriptor has been created for a PID 7579 at the same exact time, which is equivalent to that of the project's process. Therefore, we establish that the PID of the process has indeed been registered with an inotify file descriptor. The accurateness of the directory watch registrations for the source file system “/home/martin/Desktop/Tests/ROOT” is emulated in the correct overall system behavior and the accurate handling of file modifications, as is thoroughly discussed in the following sections.

### 7.1.2 Full backup after process initiation

As shown in Algorithm 3 of the unified system, once the event watches have been added to the source file system, the system is ready to perform the main function: intermittently inquiry for any modified files in the source file system and perform backup cycles due for registered devices. As detailed in Chapter 5, there are several conditions that determine which of the two backup types are to be performed: a full backup, which copies all data from the source file system, or incremental backup, which copies only the changed files since the last backup to the source file system.

During the first backup cycle observed in Appendix D.3 a full backup is performed as the following condition is satisfied: when the process starts up for the first time (or even if backups have been made to a device in the past), any source file system changes have not been registered by the inotify instance since the last backup, which means that an incremental backup would result in an inaccurate file system

version and, thus, a full backup must be performed. This is the behavior observed in the first backup cycle in Appendix D.3.

As seen in Appendix D.3, a full backup is a straightforward operation; a full copy of the source file system tree is made to the target backup directory. We observe that the contents of the source ROOT directory displayed in Appendix D.2 are equivalent to the first backup snapshot. Since the files are copied, we observe different inode numbers for the files in the source and target directories, but modified file times are sustained (retrieved from original file in the source file system and assigned to the copy on the storage device), which is desired for user experience, so that the target file system appears to be an exact copy of the source file system.

In this section, a full backup due to first initiation of the process is discussed. This is different from performing a full backup over an incremental one due to other reasons rather than simply creating a first copy ever of the source file system, which is why it is discussed separately as part of the initialization section. The other occasions of when a full copy is appropriate are discussed in Section 7.4.

## 7.2 Incremental results

Please refer to consecutive backup cycles displayed in Appendix D.3, which are the basis for the evaluation, as they provide concrete evidence for exploring the range of possible changes to the source file system. Each respective components of the system behavior will be discussed in relevance to the results contained in Appendix D.3.

### 7.2.1 Hard links and copies

One of the fundamental software features discussed extensively throughout the paper is the use of hard links. The linking of files is a rather straightforward Linux operation and its importance, in context of this project, comes down to saving storage space by avoiding unnecessary replication of unchanged files from previous backups.

The successful use of hard links is evident for any cycles prior to which files have been modified (attributes or otherwise) in the source file system; that is, modified files are copied from the source file system rather than hard linked to the last backup. This is clear from the inode numbers of those files. We will discuss a few of these cases for a clear analysis.

Firstly, for advantageous design reasons discussed in Chapter 5, it is part of the the system architecture to first recreate the directory tree and hard link all of the regular files of the new backup to those in the previous backup, regardless of any modified files between the backup cycles. Therefore, for backup cycle 8 of Appendix D.3, we observe an exact copy of the previous backup where all regular non-modified files share an inode (of course, for reasons explained earlier, directories cannot be hard linked in Linux, so they are simply copied). In fact, linking a file sustains all file information (even metadata a such as modification time), which, as mentioned throughout the paper, is highly desired for user experience so that consecutive file system versions look like identical copies; that is, files on the target storage device should maintain identical attributes and contents to that of the related files in the source file system.

#### 7.2.1.1 Modifications for existing files with the Linux ‘touch’ command

```
touch "/home/martin/Desktop/Tests/ROOT/RESOURCES/Future_Work"  
touch "/home/martin/Desktop/Tests/ROOT/Essay_Final_Submission"
```

Prior backup cycles 1 and 3, we observe (in Appendix D.3) that the files `Future_Work`, `Essay_Final_Submission` and `Software_Updates` were modified with the ‘touch’ command (these are just a few example). For existing files, the Linux touch command can be used to update the access date and modification times of a file, but can also be used for the creation of new files, which is discussed in Section 7.2.1.2. In this Section, we discuss the use of the ‘touch’ as a command for testing that the software indeed recognizes and accurately handles file modifications.

In backup cycle 2 we observe the touch command performed for the existing file `Software_Updates`. This can also be seen in other backup cycles, such as cycle 5 with file `Successful_Test_Results`. It is evident that the files were successfully copied from the source file system due to the changed file attributes. The inode of the touched files is different to those of previous backups, so hard linked file has been removed and replaced with new copy. This is observed for all backup cycles where existing files were modified using the Linux touch command in the source file system.

We also observe that the software properly handles the issue of incorrect timestamps for modified files in the backup directories due to the copy function, which is required to copy the modified file to the target storage device. It is inevitable that a file modified in the source file system will have its timestamps changed when copied to the storage device, which is undesirable since we want to preserve the file's metadata. This is resolved by reassigning the file's timestamps programmatically after it has been copied during a backup cycle.

This success of this operation is evident from cycles 3 or 5, since we can compare the directory name (exact time of backup cycle in machine time) to a touched file's metadata of the same cycle, such as file `file1`. We can clearly see that the metadata is different to the backup time, as the file metadata suggests that it has been modified at time 11:26:50 and the backup cycle occurred at time 11:27:19. This is due to the correct reassignment of the modified time from the source file system in the source file system to its copy on the storage device, which would otherwise have the modified time equivalent to the time of the backup cycle.

#### **7.2.1.2 File creation**

The second case that requires a copy to be made of a regular file to the target storage device occurs when a new file is created in the source file system. This occurrence is demonstrated in cycles 3 and 5. As observed, the files `file1`, `file2` and `Successful_Test_Results` were accurately copied to the destination directory during the respective cycles. File creation is handled very similarly to file modification but does not require that a hard link file is removed from the target directory. Instead, the created file is simply copied to the appropriate directory. Again, the timestamps of the new file must be reassigned to match those in the source file system, which is demonstrated by time difference of the files (matching its source file's modification and access times) and the modification time of the backup directory to which they belong.

#### **7.2.1.3 Summary on hard links and copies**

The systems ability to hard link consecutive backups and accurately apply any file system changes by copying relevant files has proven to work flawlessly. The directory tree is correctly recreated in new backups, and any files that have not been modified in the source file system since the last backup are hard linked in the new backup directory, which, as a result, saves a considerable amount of storage space on the target device. In addition, any files that have been modified or created either replace the hard linked file on the target device with a new copy or are simple copied to the correct directory on the target file system, respectively.

### **7.3 File deletion**

File deletion is correctly handled by the system, which is proven by the desired behavior displayed in backup cycles 2 of Appendix D.3. This operation is rather straightforward; when the file is deleted with the shell script commands shown in D.1, the inotify instance registers the event and stored the path, which, in turn, is used by the system to determine that the file exists in the target file system but has been removed from the source file system. This suggests that file must be deleted from the newest backup,



and the correct operation ensues by deleting the file from the appropriate backup directory. This similar to directory deletion, but that is discussed in Section 7.6.2

## 7.4 Full vs. Incremental backup

Once a first full backup of the source file system has been performed, as discussed in Section 7.1.2, any following incremental backups can be performed since links can be made to the first file system version. However, there remain cases where a full backup is desired over an incremental one, even after incremental backups have already been made to the device. There are other conditions that are highly imperative in deciding which type of backup, full or incremental, to perform other than just whether the process has newly started. These conditions are checked for repeatedly every time the process wake up, and include: missing last backup time and directory name fields in the configuration.json for the given registered device.

If a last backup directory name is missing from the configuration.json file, it is the correct decisions to perform a full backup since it is not certain which is the latest snapshot of the source file system. Of course, the system could use the names of the directories to determine which was the last backup time. However, this poses ambiguity (especially since the directory names are in machine time which can be easily altered through the system settings), and using the correct directory for incremental backup is imperative as otherwise the whole backup becomes wildly inaccurate.

The reason for why to perform full backup when the last backup time (in GMT) is missing from the configuration.json file is similar to when a directory name is missing. Although a backup could be made immediately in the next backup cycle, there remains an ambiguity in how this information has gone missing and it might be possible that the backups are inaccurate. Instead, a new full backup is made to the target storage device and the backup frequency is set to the default frequency that is used every time a new device is registered. This decision design eliminates any possibility of inaccurate backup, which is the absolute priority.

This is tested in cycle 7, which yielded the directory 201814112819. In Appendix D.1 we observe that a bash command is used to remove the most recent backup directory from the target storage device. As observed, the system correctly performs a full backup since the last backup directory does not exist on the target storage device; the new backup directory contains no hard links to the last possible backup, but instead creates a new copy of all files in the source file system, seen as all inodes are different. In the following backup cycle of directory 20181411284, we observe that the system continues in incremental backups by recreating the last full backup directory with hard links on regular files.

## 7.5 Hard links in source file system

```
ln "/home/martin/Desktop/Tests/ROOT/RESOURCES/Software_Specifications" "*"
↪ home/martin/Desktop/Tests/ROOT/RESOURCES/
↪ Software_Specifications_Updated"
ln "/home/martin/Desktop/Tests/ROOT/Essay_Final_Submission" "/home/martin/
↪ Desktop/Tests/ROOT/Essay_Final_Revised"
```

This section explores the occurrence of new links made in the source file system. The desired system behavior is such that new hard links in the source file system are treated

equivalently to file creation, in that they are copied to the new target backup directory and timestamps are updated. In Appendix D.1 observe that links are made to files `Software_Specifications` and `Essay_Final_Submission` in cycle 3. The resulting behavior was such that the files were treated simply as new files, and correctly copied as thoroughly discussed in Section 7.2.1.2. The files have been copied to the correct directories and modification times have been updated to match those of the files in the source file system.

## 7.6 Directories

In this section we discuss the management of directories separately to regular files, in order to illustrate cases where directory files are handled differently.

### 7.6.1 Directory creation

The management of directory creation in the source file system is perhaps the only operation that resembles file creation almost entirely. As discussed in Chapter 5, newly created directories must be copied before any regular files, as files cannot be copied to a non-existent directory (in case directories are created that also contain new files in a single backup cycle). However, one important difference to regular file creation is the addition of new inotify watches for any new directories when created in the source file system. This is successful, which is emulated in that when the directory `Family_Photos` is created in the source file system in backup cycle 4, and a newly created file `Vacation.jpg` is copied in backup cycle 5, the system successfully listens for this event and copies the file accordingly, which is the desired behavior when creation of new directories occurs during incremental backups.

```
[ 16014 11:27:34] Family_Photos
└─ [ 16024 11:27:20] Vacation.jpg
```

### 7.6.2 Directory deletion

It may occur that a directory is deleted in the source file system. As observed in backup cycle 5 of Appendix D.3, this requires the deletion of the directory and any of the contents (files or sub-directories) that it may contain. The main difference between the deletion of regular files and directories is observed at code level; instead of removing a single file, a method must be used to recursively delete all of the directory's contents (in this case, the Boost Filesystem Library method `remove_all()` method is utilized).

### 7.6.3 File Omission

Other than handling file system changes to create accurate and most up-to-date snapshots of the source file system, one of the software's defining features is the ability to omit certain files from backup. This feature is significant, since there might be files that the user does not need backed up under any circumstances. In case of large files like high definition movie files, omission from backup is crucial to avoid unnecessary copies of large amounts of data. Moreover, in cases where a user has a storage device with very limited space but still wants to leverage the application to backup a small subset of files, an omission feature will allow this. Although a rather simple feature in context of the entire system architecture, the implications are considerable. The shell script for backup cycle 2, we observe a clear omission of the `File_to_omit` file from the backup cycle by the system.

```
"lastBackupDirName": "201814112634",
"registeredDevices": [
  {
    "filesToOmit": [
      "/home/martin/Desktop/Tests/ROOT/File_to_omit"
    ],
    "lastBackupDirName": "",
    "lastBackupTime": 2,
    "productId": 5675,
    "vendorId": 2385
  }
]
```

Figure 7.1: Files to omit for registered device

```
touch "/home/martin/Desktop/Tests/ROOT/File_to_omit"
```

### 7.6.4 Pruning

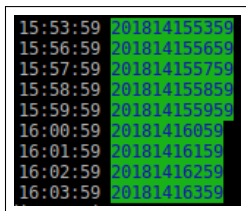


Figure 7.2: Pruning results

Pruning, which is the removal of older backups to provide more storage space for newer backups, has resulted in successful behavior. The testing for this is rather simple, as file system modifications have no overlap with this feature; rather, we simply observe that backups are removed as desired according to the testing parameters stated earlier. Since pruning does not crossover with modification handling, we can use longer backup frequencies for clearer pruning demonstration over slightly longer periods of time.

As we observed in Appendix D.4, the older versions of the source file system are correctly removed based on the established parameters of: all backups for past 6 minutes, 2 backups for 10 minutes, and 1 backup past. As desired, the pruning removed the oldest file system versions during the pruning phase.

## 7.7 Evaluation with comparison to literature and future work

The results have shown that the final product achieves the objectives set out for the project. However, there are other backup algorithms and methods that could be considered, which exploit filesystem constructs to resolve backup and restore challenges more efficiently. The rsync algorithm, for example, is able to efficiently identify the contents of a source file that are equivalent to the contents of a destination file, and only transfer the data that does not match. Although rsync requires that each file on the source file system is read to determine which contents have been modified, it would be interesting to see if there are potential features applicable for improving this project in the future.

This project has successfully applied many of the areas explored throughout the literature research; for example, Time Machine, which partly inspired this project, resolved efficiency problems by tracking changes made to a file system rather than checking each individual file for changes. This is accomplished by utilizing the inode notify subsystem of kernel.

It is no surprise that as file system grow in size, new backup strategies are created to protect them. Currently, as suggested by the background research, the most promising strategy for handling large-scale file system backup is the incremental-only backup. I have applied the knowledge accumulated during the research phase, such as the implementation of incremental back ups, to successfully realize the objectives set out for this project in an intuitive manner.

## 7.8 Conclusion

The aim of this project was to build a backup software application distributed with the Linux operating system. To achieve this, specifications were set out in Chapter 4. The evaluation stage of the project yielded results that have shown a successful completion of the preordained objectives.

As a result of the considerable amount of research conducted prior to delving into the software development phase of the project, I was pleased to not have faced any major problems regarding understanding the theory; it was evident that the background research has prepared me thoroughly for the scope of the project. For most part, the time-consuming challenges related to other issues than ones concerning the development stages of the system architecture itself; this would include partitioning my laptop to support the Ubuntu 16.04 operating system, or understanding how to utilize libraries that I had never used before, such as libusb. However, it was certainly challenging for me to acquire so many new skills to deliver a single project; this is the first time that I have worked with daemonized programs or the Inotify module. While the use and implementation of the fundamental components of the software is straightforward to me now, the first steps of the implementation phase were rather intimidating. While challenging at first, I am delighted with my ability to grasp so many new concepts to realize the objectives set out for the project.

Although the project was overall a success, there remain possible future enhancements that could be accomplished in addition to the core objectives, as outlined in Sections 1.4.3 and 7.7. From the potential of future work listed in Section 1.4.3, I would mostly like to dedicate time to exploring how this software could exist as a Web-based application. Regarding the presentation layer, I would also like to give more attention to visual aspects of the front end in any future work, as the focus of this project was mainly on the performance of the backup utility.

I would like to conclude this project with a statement regarding how delighted I am that I chose to complete a software of this nature. Although proven to be difficult at times, the outcome is rather tremendous and the learning outcomes attained along the way are directly transferable to my future career plans.

# Bibliography

- [1] AMANDA <http://www.amanda.org/>. 2017.
- [2] Andrew Tridgell and Paul Mackerras The rsync algorithm. The Australian National University, 1996.
- [3] Ann Chervenak, Vivekanand Vellanki, Zachary Kurmas. Protecting File Systems: A Survey of Backup Techniques. Joint NASA and IEEE Mass Storage Conference, 1998.
- [4] Apple. AirPort Time Capsule. <https://www.apple.com/uk/airport-time-capsule/>.
- [5] Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Andrea C. File System Implementation. Arpaci-Dusseau Books, 2014.
- [6] Francisco Javier Thayer Fbrega, Francisco Javier, and Joshua D. Guttman Copy on Write. 1995.
- [7] C. Mee and Eric Daniel. Magnetic Storage Handbook, 2nd Edition. McGraw-Hill Professional, 1996.
- [8] Cunhua Qian, Syouji Nakamura, and Toshio Nakagawa Optimal Backup Policies for a Database System with Incremental Backup, Volume 85, Issue 4. Electronics and Communications in Japan (Part III: Fundamental Electronic Science), 2002.
- [9] Dave Hitz, James Lau, and Michael Malcolm File system design for an NFS file server appliance. Network Appliance Corporation, 1994.
- [10] David Cane and David Hirschman Handbook of Network and System Administration. Connected Corporation, 1998.
- [11] David Salomon Data Compression: The Complete Reference. Springer Science & Business Media, 2013.
- [12] Filesystem Boost, [www.boost.org/doc/libs/1\\_67\\_0/libs/filesystem/doc/index.htm](http://www.boost.org/doc/libs/1_67_0/libs/filesystem/doc/index.htm)
- [13] Guan-qun Sy, Hong-cai Tao Design and implementation of remote data backup and recovery system based on Linux. School of Information Science and Technology, Southwest Jiaotong University, 2012.
- [14]
- [15] Inotify(7) - Linux Manual Page, [man7.org/linux/man-pages/man7/inotify.7.html](http://man7.org/linux/man-pages/man7/inotify.7.html). James da Silva and Olafur Guthmundsson The Amanda Network Backup Manager. Seventh System Administration Conference (LISA 93). Monterey, California, November, 1993.
- [16] Jin-Sun Suk and Jae-Chun No File System Snapshot, Volume 47, Issue 4, pp.88-95. Journal of the Institute of Electronics Engineers of Korea CI, The Institute of Electronics Engineers of Korea, 2010.
- [17] Michael Kerrisk Filesystem notification, part 2: A deeper investigation of inotify. LWN.net, 2014.
- [18] Mostafa Khalil Storage Design and Implementation in vSphere 6: A Technology Deep Dive, 2nd Edition. vmware Press, 2017.

- [19] Neeta Garimella Understanding and exploiting snapshot technology for data protection, Part 1: Snapshot technology overview. IBM, 2006.
- [20] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley Logical vs. Physical File System Backup. Proceedings of the 3rd Symposium on Operating Systems Design and Implementation. University of British Columbia, Network Appliance, Inc. New Orleans, Louisiana, February, 1999.
- [21] Paolo Di Francesco Design and implementation of a MLFQ scheduler for the Bacula backup software. Universit degli Studi dell'Aquila, Italy, 2012.
- [22] Peter Macko, Margo Seltzer, and Keith A. Smith Tracking back references in a write-anywhere file system. USENIX Association FAST 10: 8th USENIX Conference on File and Storage Technologies, 2010.
- [23] Pratap P. Nayadkar and Prof B.L Parne Logical vs. Physical File System Backup. (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 5 (6) , 2014, 8236-8238. Dept. of Computer Technology, Yeshwantrao Chavan College of Engineering, Nagpur (MS), India, 2003.
- [24] Robert Love Linux Kernel Development. Third Edition, Pearson Education, Inc., 2010.
- [25] Russell J. Green, Alasdair C. Baird and J. Christopher Davies Designing a Fast, On-line Backup System for a Log-structured File System. Digital Technical Journal Vol. 8 No. 2, 1996.
- [26] Steve Shumway Issues in On-line Backup. SunSoft, Inc., 1991.
- [27] Qt Creator Manual, [doc.qt.io/qtcreator/](http://doc.qt.io/qtcreator/).
- [28] W. Curtis Preston Backup & Recovery. O'Reilly, 2007.
- [29] Windows Dev Center BY HANDLE FILE INFORMATION structure. [https://msdn.microsoft.com/en-us/library/windows/desktop/aa363788\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363788(v=vs.85).aspx), Windows.
- [30] Zachary Kurmas and Ann L. Chervenak Evaluating Backup Algorithms. College of Computing, Georgia Tech USC Information Sciences Institute, 2000.
- [31] Zachary Kurmas and Ann L. Chervenak Evaluating Backup Algorithms. College of Computing, Georgia Tech USC Information Sciences Institute, 2000.

# Appendices

# Appendix A

## Additional Background Research

### A.1 Design considerations in file-based backup systems

WAFLs file-based backup utility has been around for almost twenty years[18]. In that time, the BSD dump has been ported to platforms like Solaris and Linux[18]. One of the prominent advantages of the BSD dump format has been its exceptional ability to cross-restore data from one environment to another[18]. The BSD dump format operates on inodes, which is the distinguishing factor from archiver utilities such as cpio or tar [16] [18][24]. In the backup, every directory must precede its files, and both files and directories are stored in ascending inode order[18][22]. Directories are stored in a common format of the file name followed by the inode number[22]. Every backed up file and directory is prefixed with 1KB of header meta-data, which includes: file type, a map of the 1KB holes in the file, permissions, group, size, and the owner[18]. The backup itself includes two bitmaps that describe the system at the time of backup[18], which allows for a format that facilitates what is a relatively simple dump utility[9]. The first map identifies the inodes being used in the dumped subtree during the time of the dump, and helps in determining which files have been deleted between incremental dumps[18]. The second map identifies the inodes written to the backup storage device, and help verify the degree of correctness of a restore[18].

The main merits of the file-based backup utility can be accredited to its ability to use the file system structure to its advantage[23]. In a file-based back approach, an end-user can chose to backup only a subset of data in a file system, which saves significant backup time and backup device space[23]. In that sense, a file-based backup is able to exploit to use of filters, such as omitting given files from backup. Once again, this saves device space and reduces backup time. Furthermore, file-based backups allows for relatively easy user error recoveries; if the system administrator deletes a file by accident, a file-based restore operation can locate the given file on disk and restore that single file[3]. A file-based approach is tremendously resilient to even small corruption of the disk, or errors caused by the backup utility of the software - that is, since each individual file is self-contained, minimal disk corruption will most of the time corrupt only that file. An error on the backup side of the software may cause a number of files being excluded from backup, but a majority of the file system copy will still be successfully stored on the disk[18] .

Since the data of each distinct file is grouped together, a file-based backup stream will have to tendency to assume minimal or no knowledge of the underlying file system[18]. Therefore, it is no surprise that the primary disadvantage of a file-based backup and restore utility can be accredit to the use of the source file system[18]. For instance, the dump utilities must rely on the source file system for basic operation, which accounts for additional overhead[18]. It may happen that a file systems caching policies and read-ahead are not optimized for backup. Consequently, the backup softwares use of a certain file system may have considerable impact on user experience. This means that performance is often a significant issue. Similar problems concern restore operations. For example, data and meta data must be written for each stored file[18]. However, many file systems are not optimized for the data access patterns of a restore operation[18]. In such cases, we observe a performance trade off for functionality. Although the

improved functionality often attempts to alleviate the performance degradation, it is often an unsuccessful attempt[18].

## A.2 Device-based strategy

In contrast to the file-based approach, device-based (or physical) backup systems copy an entire filesystem onto the backup medium with minimal or no interpretation of the target filesystem structure[23]. Device-based backup has initially been developed to migrate data from one device to another[3], but has since evolved and been used as a fully functional backup strategy by Digital[18]. One of the advantages of duplicating an entire physical medium is that even filesystem attributes that may not be representable in the standard archival format are copied[3]; examples of such attributes are filesystem configurations, hidden files, or snapshots, which are further discussed in section 2.2.1[18]. Neglecting the file structure when copying disk blocks onto a backup medium allows for fewer necessary seek operations, which improves overall backup software performance[3]. However, since a device-based approach may not store files contiguously on backup medium, it has a slower restore than the file-based approach[23]. To enable file recovery, device-based backups must keep information on how files and directories are stored on disks to link blocks on the backup medium with their respective files[18]. As a result, device-based programs often vary in implementation for different filesystems and are, therefore, seldom portable[3][18]. On the other hand, the file-based software utilities like tar contain contiguous files, which means they are more portable because the concept of files is considerably universal[3]. Another disadvantage of the device-based approach is that it may cause data inconsistencies[3][18]. Prior to writing to disk, it is possible that an operating system kernel will buffer write data. Typically, devicebased backup schemes neglect this file cache data when traversing disk blocks and back up older versions of files, which will undeniably introduce data inconsistencies[3]. By contrast, a file-based backup scheme will first check the file cache and back up the most up to date versions of the files[3].

## A.3 Design issues of device-based dump backup strategy

A device-based backup is the complete migration of data from one device to another[23]; with regard to filesystem backup, the source media are disks and the destination media may include solid state storage media such as flash memory, optical storage media such as CDs, or remote backup services[18]. It is a simple augmentation to the traditional device-based backup scheme to be able to understand filesystem meta-data enough to identify which disk blocks are in use and should therefore be backed up. All filesystems must include a method for identifying what blocks are free[18], which a backup procedure can use to back up only those blocks that are in use; of course, this requires that the block address of every block stored on a backup storage medium is recorded so that the restore operation can recover data to the correct address[18][23].

The primary advantage of using the device-based backup schemes stems from its simplicity[3]; all data from the source device is migrated to the destination device, so the backup procedure can neglect the format of the data[18]. Another advantage is speed; the device-based backup scheme can place the accesses to the source device in the order it deems as most efficient[18]. There are several limitations to device-based backups, however.

First, because data is not interpreted when stored, it is not portable[3]. Only if a filesystem layout of the source disk has not been modified since the last backup can the backup data be used to recreate a filesystem[18]. In fact, it may even be required to recover a filesystem to a disk of same configuration and size as the original depending on the underlying filesystem organisation[18].

Second, to recover a small subset of the filesystem, such as an individual file that was deleted due to user error, is highly impractical[23]. The complete filesystem must then be recovered in order to identify the individual disk blocks that constitute the requested file[18].



Third, the disk blocks stored on disk will most likely be internally inconsistent if a filesystem is modified while performing a backup[18]. It could be said that the indelicate nature of the device-based backup approach is a cause of its own problems; since minimal to no interpretation of filesystem information occurs during a backup procedure, both backing up subsets of a filesystem and incremental backups are unavailable features of a device-based approach. A garden hose provides an appropriate analogy for raw device-based backup. Although data flows from the source device to the target storage medium at high speed and with operational simplicity, the control of this flow is limited, since all a person can do is to hold the hose shut - any finer grained control of the data flow is very difficult.

# Appendix B

## Development

### B.1 Technical details of the configuration.json file

The pair of a device's vendor ID (VID) and product ID (PID) is used to uniquely identify any registered devices. Device information such as the serial number is not appropriate, since not all devices possess a serial number and, although still negligible, there is a higher chance of duplicate serial numbers in a list of devices. VIDs and PIDs are 16-bit numbers commonly used to identify USB devices in the industry because each VID is assigned by the USB Implementers Forum to a specific company, which in turn assigns a unique PID to its individual products. They are also convenient for software development since both IDs are embedded in the product and are communicated to the operating system when a device is plugged into a USB port.

The last backup time to device is stored in GMT, since machine time is simply too unreliable to accomplish a consistent backup cycle; machine time can be easily altered by the user through system settings, or automatically change by the system in case of change of time zones, which would disturb a consistent backup schedule. Using GMT to record backup cycle times and finding the difference to the current GMT time when inquiring for any impending backup cycles is a much more reliable method, since it is independent of any system changes. The variable is of the format: YYYY/MM/DD/HH/MM/SS.

The last backup directory field holds the value for the name of the last made snapshot of the file system. Directories created to contain a file system backup are formatted according to its creation time: YYYY/MM/DD/HH/MM/SS. The backup directory names are stored in machine time to provide a user-friendly format relevant to the user's current machine time, while, as discussed above, the last backup time stored field in the configuration.json that is used to manage backup cycles is stored in GMT to account for unreliability of system time changes.

### B.2 Technical details of the filemod.json file

It is essential to the technical design to save a list of modified files for each registered device separately. If the application enforced a single registered device limit or same backup cycle frequency across all registered devices, all modified files could be stored within a single JSON object. However, this would make for a poor application. Instead, the application supports any number of registered devices with their own backup intervals (adjusted through user interface or manually in configuration.json), but this comes with the necessity of storing modified files due for backup for each device separately. This reason for this is that device backups are not expected to happen concurrently, but rather at different intervals. The list of modified files of the target device must be cleared each time a cycle is completed as a cycle of newly modified files must begin. Since it may occur that two or more devices begin their backup cycles at different times, it should be accounted for that while modified files were backed up to one device (and deleted), those same files were not backed up to other devices yet. While the modified file lists will be

cleared at different times, any newly modified files will be written immediately to the JSON objects of all registered devices.

Even if backups of all devices happened synchronously, it would still be appropriate design decision to manage the modified files for each device separately, as it may occur that even though a backup cycle succeeded to one device, the cycle was interrupted for the second device, and in that case a backup cycle would be repeated for the first device since the modified files lists has not yet been cleared. This would be wasteful, and the better design, even in a system where backups occur at same, is to keep separate the modified files lists for each device and clear the list once a backup to a device succeeds. Therefore, all modified files will be stored in its separate JSON object together with the VID and PID of the device that the file must be backed up to.

The filemod.json file also stores the pairs of watch descriptors and the corresponding pathnames. The management of this information is imperative in establishing the paths to any modified files. When a signal occurs for a file for any of the registered events, the system can compare its watch descriptor to those retrieved from the filemod.json file, and thereafter store the newly modified file name concatenated to the correct path in filemod.json to the modified files of each registered device.

### **B.3 Technical details of the status.json file**

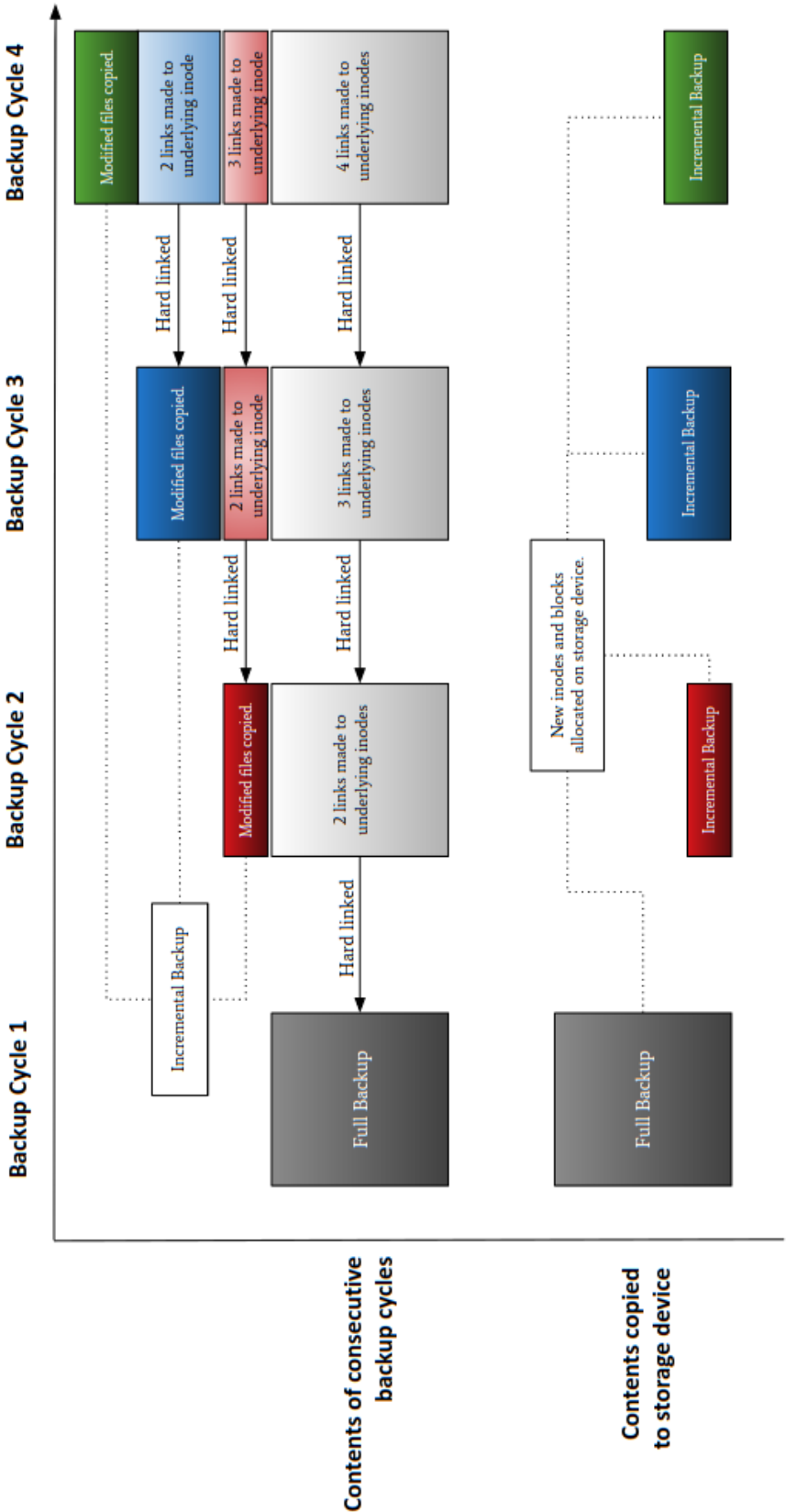
The reason why the last backup time to a device is written in machine time in the status.json rather than in GMT is because this information is used strictly for the presentation layer. The front end component of the application then retrieves this information to provide appropriate information in the presentation layer. A user would not be interested in seeing the time of last backup in GMT, but rather the current time displayed on their computer.

The reason why backup state is included is to specifically state in the presentation layer whether backup is occurring at the very moment; apart from good user experience, this informs the user that it would be a bad idea to remove the device at the moment. This is contained as a separate variable because the backup state cannot be stated for certain from the last backup time.

## Appendix C

# Unified system architecture

C.1 Hard links in context of intermittent backup



## C.2 Supplementary description of the unified system algorithm

### C.2.1 Initialization of process

Prior to entering the main loop of the daemon, a few measures must be taken to ensure desired behavior.

Firstly, the we must check that the 3 configuration files, which were discussed in detail in Section 4.4, exist in the project directory; this condition will also be checked for routinely inside the while loop on lines 8 to 9. If a configuration file happens to be deleted while it is being written to, such as when writing a list of modified files to the `filemod.json` file, the operation will inevitably fail; the is a reality of software development that, although not expected to happen, can occur under unlikely circumstances, such as the user or another process deleting a configuration file. At the beginning of every iteration of the main while loop on line 7, all three configuration files will be checked for and recreated if the files are missing. Thereafter, the program can resume in populating the JSON objects in proceeding operations.

Secondly, the program must be daemonized, as observed on line 5. In practice, this will be a call to another method within the main class, which will perform all the steps necessary to daemonize a program, as discussed in detail in Algorithm 2.

Thirdly, `inotify` must be implemented exactly as described in Algorithm 2. Succinctly, this constitutes of `inotify` instantiation, adding of watches to directories of target file system, and writing all watch descriptors with pathnames to the `filemod.json` file.

### C.2.2 Backup utility

#### C.2.2.1 Configuration files

Upon entering the main while loop of the program on line 7, most of the operations are designated to accommodate the backup utility of the software (other than, for example, managing the `status.json` file that communicates information to the front end logic). As stated earlier, we must perform intermittent checks for any missing configuration files; even if a configuration file is deleted, the software will continue performing since the missing file is recreated.

#### C.2.2.2 Querying for file modifications

After performing any checks for missing configuration files in the project directory, the modified files read from the `inotify` instance are written to the `filemod.json`. The architectural decision to store modified files externally stems from the fact that many `inotify` reads for modified files may occur before a backup cycle is due, so the modified files read in the 5 minute intervals of the main while loop need to be stored without loss of information. Each time new modified files are read, the JSON objects in `filemod.json` must be updated for all of the registered devices.

Before writing the list of modified files to the `filemod.json` file, all duplicates are first removed from the list as file names returned by the `read()` function on the `inotify` instance can be repeated if a file is modified multiple times, for example. The post conditions seen on lines 28 to 40 of a Algorithm 3 will perform any necessary condition checking when a backup cycle is due with just one copy of path to a file that triggered any of the 4 events in the source file system. Further, after eliminating any duplicates, it may be that some of the modified file paths that are already stored `filemod.json` are duplicates of the new modified files being written after the 5 minute while loop interval. Therefore, the new modified files are sorted alphabetically which allows for an efficient insertion to the already alphabetically sorted modified files list in the `filemod.json` file.

#### C.2.2.3 Full or incremental backup cycle

After inquiring for any modified files, the system queries whether any devices are due for a backup by comparing the difference of the current GMT time and the last backup time with the device's backup

frequency. If any devices are due for a backup, we begin a backup cycle which is demonstrated by lines 18 to 45 of Algorithm 3. First, we must determine whether a device due for backup is indeed connected to the machine. The essential logic of this condition relies on the use of the `libusb.h` library. In short, the `libusb.h` method `libusb_get_device_list()` returns a list of USB devices currently connected to the machine, from which, using the `libusb_device_descriptor()` method, we can retrieve the VID and PID pairs of the connected devices. If any of the devices due for backup are in this list, a backup cycle will resume. It is not an issue if a device is ejected mid-backup cycle, since the last backup directory name and last backup time would have not yet been altered (since this is overwritten after a backup has succeeded), so backup is simply made next time the device is connected.

During a new backup cycle, four conditions are tested for to determine which of the two type of backups will take place: (1) a full copy of the target file system onto the storage device or (2) hard linking the previous backup on the storage device and thereafter applying file system changes.

Firstly, the directory containing all backups must exist on the target storage device, which is shown as “PROJECT” in Appendix C.4. If this directory is missing from the storage device, the device is either newly registered or the directory has been deleted, and in both cases the project directory has to be created and a first full copy of the source file system has to be made inside this directory to resume regular behavior of proceeding hard linked backup cycles.

Secondly, the last backup directory name retrieved from the `configuration.json` has to exist in the JSON object. If this variable is missing, the software simply does not know which is the latest file system snapshot, and a new full copy has to be made to retain accurate versions of the of the source file system.

Thirdly, if a valid last backup directory name is retrieved from the configuration file, the path to the directory has to actually exist on the storage device.

Lastly, the last backup time (in GMT) must exist in the `configuration.json` for the device being backed up. Of course, we could use the last backup directory name, if it exists, and simply carry out a backup cycle immediately if the last backup time is missing. However, this raises suspicious behavior as it is not certain that the last backup directory name is indeed the most updated version of the file system due to other missing information, so it is safer to create a new full copy of the source file system. Maintaining an accurate backup copy should always be prioritized over saving some storage space and increasing efficiency; once the wrong file system version is used as the target directory to hard link for any proceeding backup cycles, the result can become dangerously skewed if newest directories are missing from the device file system version.

If any of the four conditions listed above return false, a full copy of the source file system is made to the device due for backup. Else, the other type of backup cycle resumes, which begins with recursively recreating the directory tree structure from the previous backup directory into a new directory and thereafter hard linking any regular files. It is imperative that the directory tree is recreated first, as any hard links into a nonexistent directory will not be made. This is achieved by using a `directory_iterator()` method of the Boost C++ Filesystem library to first recreate the directory tree structure and then hard linking any regular files. Directories must be copied first when copying any modified files to the new directory from the source file system as well, since you cannot copy a file to a nonexistent directory; any new directories to be copied must be copied before the files they contain are copied. To ensure this, the list of all modified files to be copied to the new target directory is sorted from shortest to longest, which means that any parent directory of a file will be copied to the new backup directory before its children files.

#### C.2.2.4 Applying file system changes to hard linked backup directory

In lines 26 to 41 we observe the main block of logic responsible for the operations to be carried out on modified files. Since modified files can be stored over periods as long as an hour, for example, before ever being copied to the storage device, it would be poor design to store modified file paths multiple times with its events, and then determining which is the appropriate action to be performed on a single file based

on the event patterns. Instead, only one instance of a modified file path needs to be stored between each cycle, and appropriate actions are to be carried out based on four conditions regarding files' existence at backup time to avoid any issues. Such issue may include backing up a file that was deleted right before a backup cycle but was modified some time ago and therefore is listed in the `filemod.json` file, which would result in a failed copy action and an inaccurate filesystem snapshot.

Firstly, we determine whether the file should be omitted from the device's backup based on the list of omission files retrieved from the `configuration.json` file. It is necessary check this condition at backup time for similar of reasons as to why additional conditions are made for modified files at backup time: discrepancies can occur if a file is immediately omitted from the list of files to be backed when written to `filemod.json`, since there is no guarantee that the omitted list will not change before the files are due for backup. In any case, it would be best practice to check for this condition at backup time when relevant information is most up to date to avoid any unnecessary complications. If the file should indeed be backed up, we must first determine the relative path to the storage device based on the modified file in the source file system. This can be done by concatenating the path to the source file to the path pointing to the new directory on the storage device.

Secondly, we check whether the modified file no longer exists in the source file system at the time of backup. The condition that a modified file does not exist on the source file system but does exist in the destination path of the target storage device suggests that the file has been deleted from the source file system. Therefore, the file hard linked in the newest directory on the storage device must be deleted to maintain an up to date file system snapshot. If this conditions succeeds, there will be a "continue" call to proceed with rest of modified files since the appropriate function on this modified file has been competed.

Thirdly, if the source file path does exist at time of backup and the equivalent path of the file to the storage path does not exist, we must copy the file from the source file system to the storage device to maintain a newest accurate version of the file system. This conditions quite clearly suggests that this is a newly created file on the source file system since the file was not present in the former backup cycle. Again, we perform a "continue" call to resume with operations on the next modified file.

Finally, if the second and third conditions have not succeeded, this suggests that the file exists both in the source file system and in the new hard linked backup directory; in most cases, this suggests that modifications to the file occurred. In this case, we remove the hard linked file and replace it with the new modified file from the source file system. However, in the case that the file is a directory, solely metadata is updated as we should not remove a directory that might contain othe files.

If the file is returned as part of the modified files list but exists neither on the source file system nor on the storage device, no operation must be performed since the file seems to have been created, its path saved to the `filemod.json` in one of the 5 minute while loop intervals, and then deleted before a backup cycle even occurred to the given storage device. This case is unfortunate, since there is no stored copy of the file prior to its deletion from the source file system. Unfortunately, this is the reality, but the user can avoid this by setting a shorter backup frequency for a device, which makes it more likely that files with a short lifespan will be copied at least once to the backup device before they are deleted from the source file system.

### C.2.2.5 Post backup cycle

After an accurate snapshot of the current version of the file system has been made to the newest directory on the backup storage device, the program must proceed to update the JSON configuration files: clear all files from the device's modified files list in `filemod.json` since changes have just been applied in the latest backup cycle, update the latest directory name made to a device in the `configuration.json`, update the latest backup time (in GMT) of the device in `configuration.json`, and update the latest backup time (in machine time) of the device in `status.json`. Thereafter, the process proceeds to sleep for 5 minutes before repeating the the steps outlined in lines 7 to 46.

Now that we understand the architecture of the individual components as well as the unified system,



let's consider Appendix C.4, which demonstrates the software in operation; specifically Appendix C.5 offers a clear and extensive description of the system behavior based on results found in Appendix C.4, provided for the reader to refer to for any additional information.

### C.3 Inotify implementation algorithm

---

#### Algorithm 2: Inotify implementation

---

**Result:** Successful implementation of the Inotify API to monitor a source file system

**Input :** The root watched directory *rootDir*

```

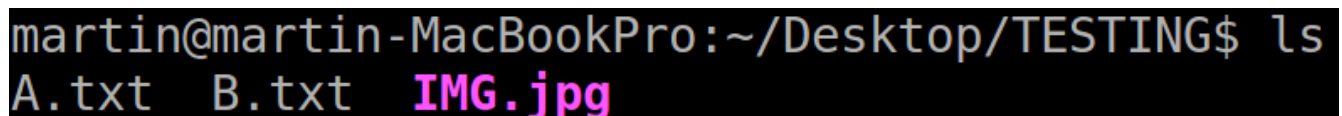
1 procedure Inotify (rootDir);
2   initialize an inotify instance using inotify_init();
3   for each subdirectory in root directory rootDir do
4     add watch to directory with inotify_add_watch();
5     externally store pair of watch descriptor and the respective full path of watched directory;
6   end
7   while process is running do
8     modified_files ← empty vector;
9     poll inotify instance with associated pollfd structure;
10    if poll succeeded and some events occurred then
11      len ← count bytes read from file descriptor into the buffer starting at buf;
12      i ← 0;
13      while i < len do
14        ie ← inotify event structure from event buffer at index i;
15        for every pair of watch descriptors and their respective paths of all watched directories rootDir
16          do
17            if watch descriptor of ie matches that of current pair then
18              concatenate path from pair of matched watch descriptor with filename of ie;
19              if file is a directory then
20                if event is the creation of file then
21                  add watch to new directory;
22                  externally store the pair of watch descriptor and full path of of newly watched
23                  directory;
24                else if event is the deletion of file then
25                  remove stored pair of watch descriptor and full path of deleted watched directory;
26                  append full path of modified file to modified_files;
27                end
28              break;
29            end
30          end
31        i ← i + size of inotify event structure + length of event;
32      end
33      perform appropriate backup utilities with stored modified_files (explored further in Algorithm 3);
34    end
35  end
36 end

```

---

### C.4 Behavior of unified system architecture

Contents of watched TESTING directory



A terminal window showing the command 'ls' being executed in the directory ~/Desktop/TESTING. The output shows three files: 'A.txt', 'B.txt', and 'IMG.jpg'. The file 'IMG.jpg' is highlighted in pink in the original image.

```

martin@martin-MacBookPro:~/Desktop/TESTING$ ls
A.txt  B.txt  IMG.jpg

```

### C.4.1 Backup Cycle 1

PROJECT directory contents after first backup cycle.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT$ ls
2018410131111
```

Contents of the 2018410131111, which is the directory created in the latest back up cycle and, thus, constitutes the latest file system version.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131111$ ls
A.txt B.txt IMG.jpg
```

File status of A.txt is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131111$ stat -L A.txt
File: 'A.txt'
Size: 0                Blocks: 0                IO Block: 4096   regular empty file
Device: 821h/2081d    Inode: 10179             Links: 1
```

Files status of B.txt is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131111$ stat -L B.txt
File: 'B.txt'
Size: 0                Blocks: 0                IO Block: 4096   regular empty file
Device: 821h/2081d    Inode: 10162             Links: 1
```

Files status of IMG.jpg is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131111$ stat -L IMG.jpg
File: 'IMG.jpg'
Size: 94963            Blocks: 192             IO Block: 4096   regular file
Device: 821h/2081d    Inode: 10163             Links: 1
```

### C.4.2 Backup Cycle 2

PROJECT directory contents after second backup cycle.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT$ ls
2018410131111 2018410131141
```

Contents of the latest 2018410131141 directory.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131141$ ls
A.txt B.txt IMG.jpg
```

File status of A.txt is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131141$ stat -L A.txt
File: 'A.txt'
Size: 0                Blocks: 0                IO Block: 4096   regular empty file
Device: 821h/2081d    Inode: 10179             Links: 2
```

File status of B.txt is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131141$ stat -L B.txt
File: 'B.txt'
Size: 0                Blocks: 0                IO Block: 4096   regular empty file
Device: 821h/2081d    Inode: 10162             Links: 2
```

File status of IMG.jpg is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131141$ stat -L IMG.jpg
File: 'IMG.jpg'
Size: 94963            Blocks: 192             IO Block: 4096   regular file
Device: 821h/2081d    Inode: 10163             Links: 2
```

### C.4.3 Backup Cycle 3

PROJECT directory contents after third backup cycle.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131211$ ls
A.txt B.txt IMG.jpg
```

Contents of the latest 2018410131211 directory.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT$ ls
2018410131111 2018410131141 2018410131211
```

File status of A.txt is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131211$ stat -L A.txt
  File: 'A.txt'
  Size: 11          Blocks: 1          IO Block: 4096   regular file
Device: 821h/2081d Inode: 10186       Links: 1
```

File status of B.txt is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131211$ stat -L B.txt
  File: 'B.txt'
  Size: 0           Blocks: 0          IO Block: 4096   regular empty file
Device: 821h/2081d Inode: 10162       Links: 3
```

File status of IMG.jpg is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131211$ stat -L IMG.jpg
  File: 'IMG.jpg'
  Size: 94963       Blocks: 192       IO Block: 4096   regular file
Device: 821h/2081d Inode: 10163       Links: 3
```

### C.4.4 Backup Cycle 4

PROJECT directory contents after fourth backup cycle.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT$ ls
2018410131111 2018410131141 2018410131211 2018410131241
```

Contents of the latest 2018410131241 directory.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131241$ ls
A.txt B.txt C.txt IMG.jpg
```

File status of A.txt is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131241$ stat -L A.txt
  File: 'A.txt'
  Size: 11          Blocks: 1          IO Block: 4096   regular file
Device: 821h/2081d Inode: 10186       Links: 2
```

File status of B.txt is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131241$ stat -L B.txt
  File: 'B.txt'
  Size: 0           Blocks: 0          IO Block: 4096   regular empty file
Device: 821h/2081d Inode: 10162       Links: 4
```

File status of IMG.jpg is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131241$ stat -L IMG.jpg
  File: 'IMG.jpg'
  Size: 94963       Blocks: 192       IO Block: 4096   regular file
Device: 821h/2081d Inode: 10163       Links: 4
```

File status of C.txt is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131241$ stat -L C.txt
  File: 'C.txt'
  Size: 0           Blocks: 0          IO Block: 4096   regular empty file
Device: 821h/2081d Inode: 10187       Links: 1
```

## C.4.5 Backup Cycle 5

PROJECT directory contents after fifth backup cycle.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT$ ls
2018410131111  2018410131141  2018410131211  2018410131241  2018410131311
```

Contents of the latest 2018410131311 directory.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131311$ ls
A.txt  C.txt  IMG.jpg
```

File status of A.txt is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131311$ stat -L A.txt
  File: 'A.txt'
  Size: 11          Blocks: 1          IO Block: 4096   regular file
Device: 821h/2081d Inode: 10186       Links: 3
```

File status of C.txt is listed below.

```
martin@martin-MacBookPro:/media/martin/DISK_IMG/PROJECT/2018410131311$ stat -L C.txt
  File: 'C.txt'
  Size: 0           Blocks: 0          IO Block: 4096   regular empty file
Device: 821h/2081d Inode: 10187       Links: 2
```

## C.5 Discussion of unified system behavior

In Appendix C.4, we observe the details of files captured during the course of five consecutive backups to help facilitate our understanding of the file system backup software in operation. First, it is important to understand the `stat()` function, which was used rather than, for example, the `ls -l` terminal command to retrieve information on files during the backup cycles. It is more desirable to use the `stat()` function as it returns more detailed information necessary for us to understand the operation of the software. Specifically, we use `stat()` function to observe inode, links, and size of files between consecutive backups, which allow for an informed discussion about the software behavior between consecutive backups.

The data displayed in Appendix C.4 was retrieved from a Linux Terminal window after running my software with certain parameters. As displayed at the beginning of Appendix C.4, the source directory “/home/martin/Desktop/TESTING” is used as the root directory that is watched for file modifications, which contains an empty A.txt file, another empty B.txt file and the JPEG file IMG.jpg. It should be noted that a simple folder containing 3 regular files has been used as a source directory instead of the root of a file system to provide an explicit demonstration of the software’s functionality at a smaller scale. However, in case of the finalized product, the watched directory will in most cases be the root directory of the source file system. It should also be noted that the set backup cycle frequency is 30 seconds as suggested by the time difference derived from the consecutive directory names, which were the machine time at the time of the backup. The backup frequency parameter has been set to 30 seconds specifically for discussion purposes, but in practice backup cycles of any registered devices can be expected to occur at much longer intervals.

At the beginning of Appendix C.4, we first list the files of the watched TESTING directory to provide appropriate information regarding the files involved in the first full copy made by the software. In Appendix C.4.1, we observe the first backup cycle made to DISK\_IMG, which constitutes of copying all files from the TESTING directory to a directory contained within the PROJECT directory on the registered DISK\_IMG storage device. Based on conditions discussed earlier, the software understands that first a full copy has to be made based on the following conditions: the root directory PROJECT containing all backups must exist on the storage device, and the last backup time and directory name retrieved from the configuration.json file must be valid. During the backup cycle that resulted in the new target directory displayed in Appendix C.4.1, none of the conditions were satisfied, hence a full copy of the files contained in TESTING directory was made to the target directory 2018410131111; as discussed

earlier, the backup directory names are derived from the machine time at the time of the backup cycle. The first backup cycle to this device therefore occurred on 10th of April, 2018, at 13:11:11. We can observe that the files were indeed copied based on the information of each file displayed by the `stat()` function. The files copied from the TESTING directory to the PROJECT directory have different inode numbers and have only 1 link to the underlying inode number, which tell us that each files were allocated new memory when copied to the storage device.

In Appendix C.4.2, we observe the second backup cycle; in practice, this is the second iteration through the main while loop of Algorithm 3. Since the last backup time and directory name were written to the `configuration.json` during the last backup cycle, the software no longer performs a full backup of the source directory. Instead, the software recreates the directory tree and hard links any regular files of directory from the last backup cycle. I made sure not to modify any files between backup cycles 1 and 2. Hence, we observe that all files contained in the new directory 2018410131141 share the same inode numbers with the files of the directory that resulted from backup cycle 1, and the link count of the files increased to 2 since a new file is linked to each of the underlying inodes that were first allocated during backup cycle 1. Between backup cycles 2 and 3, I decided to modify the A.txt file in the source TESTING directory. The inotify instance listened for the `IN_MODIFY` event and therefore the system wrote the path of the modified file to the `filemod.json` file. When the 2018410131211 directory for the third backup cycle was created, as displayed in Appedix C.4.3, all regular files were again hard linked. However, since listed in the modified files JSON object in the `filemod.json` file, the hard linked A.txt file was removed and replaced by the newly modified A.txt, as desired. Hence, we observe that while the B.txt file and IMG.jpg file still share the same inode number to the previous backups and now have 3 links, the A.txt file has a different inode number and only 1 link. We also observe that A.txt is no longer a regular empty file as it was copied from the watched TESTING directory after it was modified to contain some text.

Between backup cycles 3 and 4, I decided to create a new text file C.txt in the TESTING directory. The software picked up on the `IN_CREATE` event, and stored the path to the C.txt file in the `filemod.json` file. After creating a new directory 2018410131241 and copying the directory tree and hard linking all regular files of the directory created in the third backup cycle, the software observed that the C.txt existed in the TESTING directory but not in the target 2018410131241 directory, so the file was copied. Now, we observe that B.txt and IMG.jpg still have the same inode number from the first backup cycle and share an inode number with 3 other files (4 links in total) as the files have not yet been modified since the first full copy was made. The A.txt file shares the same inode only with the A.txt copied from the TESTING directory in backup cycle 3, and therefore has 2 links in total. The C.txt file has only 1 link and unique inode number since it was newly copied from the TESTING directory during the most recent backup cycle. – for last backup cycle add that I have edited the metadata of A.txt

Prior to the last backup cycle displayed in Appendix C.4.5, I had decided to delete the B.txt file and modify the metadata of the IMG.jpg file. The watch added to the TESTING directory included the `IN_DELETE` and `IN_ATTRIB` event masks and, therefore, methods were invoked in the proceeding while loop of when the events were triggered to write the paths of the B.txt and IMG.jpg files to the `filemod.json` file. Again, the file status' of the files in the resulting 2018410131311 are as one would expect. That is, any non-modified files (A.txt and C.txt) are hard linked from the previous backup, so we observe equivalent inode numbers and an increase of 1 of the link count relative to backup cycle 4; that is, 3 links for A.txt and 2 links for C.txt. As demonstrated by condition on lines 26 to 41 of Algorithm 3, it is required that the B.txt is deleted from the latest backup since the file no longer exists in the TESTING directory; hence, to maintain an accurate snapshot of the file system, the newest versions of the file system on the storage device must resemble that of the watched root directory. Since IMG.jpg was returned as part of the modified files list from the `filemod.json` file and still exists in the TESTING directory, the IMG.jpg file in the new backup directory is replaced with latest copy of the IMG.jpg in the source TESTING directory. Hence, we observe a link count of 1 and a different inode number to that of the IMG.jpg file in directories created by former backup cycles. It should be noted that the conditions displayed in this section are not exhaustive as it would be too lengthy to demonstrate even a small subset

of all possible file system change combinations. Rather, a few scenarios are chosen to demonstrate the general behavior of the system in operation. Extensive testing of how the system architecture handles a range of scenarios of file system changes is explored in Chapter 7.

# Appendix D

## Evaluation

## D.1 Shell scripts

*Note: Full backup omitted, but included in Appendix D.3.*

### D.1.1 Cycle 1

```
#!/bin/bash
clear
touch "/home/martin/Desktop/Tests/ROOT/RESOURCES/Future_Work"
touch "/home/martin/Desktop/Tests/ROOT/Essay_Final_Submission"
```

### D.1.2 Cycle 2

```
#!/bin/bash
clear
rm "/home/martin/Desktop/Tests/ROOT/Journal"
touch "/home/martin/Desktop/Tests/ROOT/Software_Updates"
touch "/home/martin/Desktop/Tests/ROOT/File_to_omit"
```

### D.1.3 Cycle 3

```
#!/bin/bash
clear
ln "/home/martin/Desktop/Tests/ROOT/RESOURCES/Software_Specifications" "/"
↪ home/martin/Desktop/Tests/ROOT/RESOURCES/
↪ Software_Specifications_Updated"
ln "/home/martin/Desktop/Tests/ROOT/Essay_Final_Submission" "/home/martin/"
↪ Desktop/Tests/ROOT/Essay_Final_Revised"
rm "/home/martin/Desktop/Tests/ROOT/RESOURCES/CONSIDERATIONS/C++
↪ _libraries_to_use"
touch "/home/martin/Desktop/Tests/ROOT/RESOURCES/CONSIDERATIONS/file1"
touch "/home/martin/Desktop/Tests/ROOT/RESOURCES/CONSIDERATIONS/file2"
```

[w]

### D.1.4 Cycle 4

```
#!/bin/bash
clear

mkdir "/home/martin/Desktop/Tests/ROOT/Family_Photos"
```

### D.1.5 Cycle 5

```
#!/bin/bash
clear
rm -rf "/home/martin/Desktop/Tests/ROOT/RESOURCES/CONSIDERATIONS"
touch "/home/martin/Desktop/Tests/ROOT/Succesful_Test_Results"
```



```
touch "/home/martin/Desktop/Tests/ROOT/Family_Photos/Vacation.jpg"
```

### **D.1.6 Cycle 6**

NO CHANGES

### **D.1.7 Cycle 7**

```
#!/bin/bash
clear
rm -rf "$(stat -c "%Y:%n" * | sort -t: -n | tail -1 | cut -d: -f2-)"
```

### **D.1.8 Cycle 8**

NO CHANGES

## D.2 Source file system

/home/martin/Desktop/Tests

```
└─ [3285455 10:42:26] ROOT
   └─ [3285465 13:48:36] Business_Ideas
   └─ [3285464 00:52:29] Essay_Final_Submission
   └─ [3285467 08:12:53] Journal
   └─ [3285457 09:15:41] RESOURCES
      └─ [3285459 17:01:33] Background_Research
      └─ [3285462 17:08:33] CONSIDERATIONS
         └─ [3285463 17:04:39] C++_libraries_to_use
      └─ [3285458 17:01:25] Future_Work
      └─ [3285461 17:03:32] Software_Requirements
      └─ [3285460 17:03:06] Software_Specifications
   └─ [3285466 13:48:36] Software_Updates
```

3 directories, 9 files

### D.3 File syste modification results

/media/martin/DISK\_IMG/DISSERTATION

```
.
├── [ 15969 11:26:19] 201814112619
│   ├── [ 15973 11:26:19] ROOT
│   │   ├── [ 15987 11:26:19] Business_Ideas
│   │   ├── [ 15985 11:26:19] Essay_Final_Submission
│   │   ├── [ 15984 11:26:19] Journal
│   │   ├── [ 15974 11:26:19] RESOURCES
│   │   │   ├── [ 15976 11:26:19] Background_Research
│   │   │   ├── [ 15980 11:26:19] CONSIDERATIONS
│   │   │   │   └── [ 15981 11:26:19] C++_libraries_to_use
│   │   │   ├── [ 15975 11:26:19] Future_Work
│   │   │   ├── [ 15982 11:26:19] Software_Requirements
│   │   │   └── [ 15983 11:26:19] Software_Specifications
│   └── [ 15986 11:26:19] Software_Updates
├── [ 15988 11:26:34] 201814112634
│   ├── [ 15989 11:26:34] ROOT
│   │   ├── [ 15987 11:26:19] Business_Ideas
│   │   ├── [ 15993 11:26:22] Essay_Final_Submission
│   │   ├── [ 15984 11:26:19] Journal
│   │   ├── [ 15990 11:26:34] RESOURCES
│   │   │   ├── [ 15976 11:26:19] Background_Research
│   │   │   ├── [ 15991 11:26:34] CONSIDERATIONS
│   │   │   │   └── [ 15981 11:26:19] C++_libraries_to_use
│   │   │   ├── [ 15992 11:26:22] Future_Work
│   │   │   ├── [ 15982 11:26:19] Software_Requirements
│   │   │   └── [ 15983 11:26:19] Software_Specifications
│   └── [ 15986 11:26:19] Software_Updates
├── [ 15994 11:26:49] 201814112649
│   ├── [ 15995 11:26:49] ROOT
│   │   ├── [ 15987 11:26:19] Business_Ideas
│   │   ├── [ 15993 11:26:22] Essay_Final_Submission
│   │   ├── [ 15996 11:26:49] RESOURCES
│   │   │   ├── [ 15976 11:26:19] Background_Research
│   │   │   ├── [ 15997 11:26:49] CONSIDERATIONS
│   │   │   │   └── [ 15981 11:26:19] C++_libraries_to_use
│   │   │   ├── [ 15992 11:26:22] Future_Work
│   │   │   ├── [ 15982 11:26:19] Software_Requirements
│   │   │   └── [ 15983 11:26:19] Software_Specifications
│   └── [ 15998 11:26:35] Software_Updates
```

```

[ 16007 11:27:19] 201814112719
└─ [ 16008 11:27:19] ROOT
    └─ [ 15987 11:26:19] Business_Ideas
    └─ [ 16003 11:26:22] Essay_Final_Revised
    └─ [ 15993 11:26:22] Essay_Final_Submission
    └─ [ 16011 11:27:19] Family_Photos
    └─ [ 16009 11:27:19] RESOURCES
        └─ [ 15976 11:26:19] Background_Research
        └─ [ 16010 11:27:19] CONSIDERATIONS
            └─ [ 16004 11:26:50] file1
            └─ [ 16005 11:26:50] file2
        └─ [ 15992 11:26:22] Future_Work
        └─ [ 15982 11:26:19] Software_Requirements
        └─ [ 15983 11:26:19] Software_Specifications
        └─ [ 16006 17:03:06] Software_Specifications_Updated
    └─ [ 15998 11:26:35] Software_Updates
[ 16012 11:27:34] 201814112734
└─ [ 16013 11:27:34] ROOT
    └─ [ 15987 11:26:19] Business_Ideas
    └─ [ 16003 11:26:22] Essay_Final_Revised
    └─ [ 15993 11:26:22] Essay_Final_Submission
    └─ [ 16014 11:27:34] Family_Photos
    └─ [ 16024 11:27:20] Vacation.jpg
    └─ [ 16015 11:27:34] RESOURCES
        └─ [ 15976 11:26:19] Background_Research
        └─ [ 15992 11:26:22] Future_Work
        └─ [ 15982 11:26:19] Software_Requirements
        └─ [ 15983 11:26:19] Software_Specifications
        └─ [ 16006 17:03:06] Software_Specifications_Updated
    └─ [ 15998 11:26:35] Software_Updates
    └─ [ 16020 11:27:20] Succesful_Test_Results

```

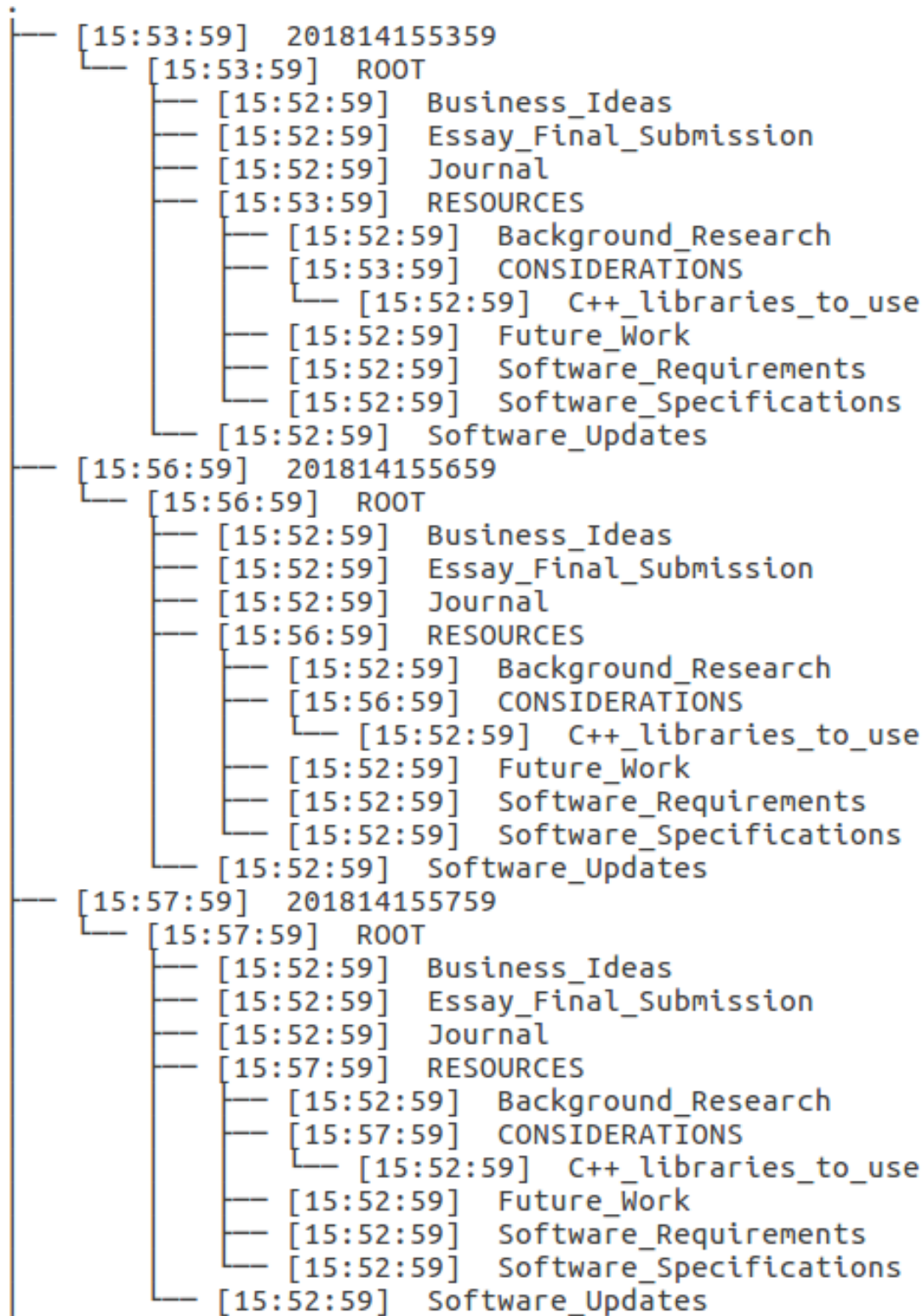
```

[ 15999 11:27:04] 20181411274
├── [ 16000 11:27:04] ROOT
│   ├── [ 15987 11:26:19] Business_Ideas
│   ├── [ 16003 11:26:22] Essay_Final_Revised
│   ├── [ 15993 11:26:22] Essay_Final_Submission
│   ├── [ 16001 11:27:04] RESOURCES
│   │   ├── [ 15976 11:26:19] Background_Research
│   │   ├── [ 16002 11:27:04] CONSIDERATIONS
│   │   │   ├── [ 16004 11:26:50] file1
│   │   │   └── [ 16005 11:26:50] file2
│   │   ├── [ 15992 11:26:22] Future_Work
│   │   ├── [ 15982 11:26:19] Software_Requirements
│   │   ├── [ 15983 11:26:19] Software_Specifications
│   │   └── [ 16006 17:03:06] Software_Specifications_Updated
│   └── [ 15998 11:26:35] Software_Updates
└── [ 16023 11:28:19] 201814112819
    ├── [ 16044 11:28:19] ROOT
    │   ├── [ 16042 11:28:04] Business_Ideas
    │   ├── [ 16043 11:28:04] Essay_Final_Revised
    │   ├── [ 16040 11:28:04] Essay_Final_Submission
    │   ├── [ 16045 11:28:19] Family_Photos
    │   └── [ 16039 11:28:04] Vacation.jpg
    │   ├── [ 16046 11:28:19] RESOURCES
    │   │   ├── [ 16035 11:28:04] Background_Research
    │   │   ├── [ 16033 11:28:04] Future_Work
    │   │   ├── [ 16036 11:28:04] Software_Requirements
    │   │   ├── [ 16037 11:28:04] Software_Specifications
    │   │   └── [ 16034 11:28:04] Software_Specifications_Updated
    │   ├── [ 16041 11:28:04] Software_Updates
    │   └── [ 16031 11:28:04] Successful_Test_Results
    └── [ 16029 11:28:04] 20181411284
        ├── [ 16030 11:28:04] ROOT
        │   ├── [ 16042 11:28:04] Business_Ideas
        │   ├── [ 16043 11:28:04] Essay_Final_Revised
        │   ├── [ 16040 11:28:04] Essay_Final_Submission
        │   ├── [ 16038 11:28:04] Family_Photos
        │   └── [ 16039 11:28:04] Vacation.jpg
        │   ├── [ 16032 11:28:04] RESOURCES
        │   │   ├── [ 16035 11:28:04] Background_Research
        │   │   ├── [ 16033 11:28:04] Future_Work
        │   │   ├── [ 16036 11:28:04] Software_Requirements
        │   │   ├── [ 16037 11:28:04] Software_Specifications
        │   │   └── [ 16034 11:28:04] Software_Specifications_Updated
        │   ├── [ 16041 11:28:04] Software_Updates
        │   └── [ 16031 11:28:04] Successful_Test_Results

```

## D.4 Pruning results

/media/martin/DISK\_IMG/DISSERTATION





```

[15:58:59] 201814155859
└── [15:58:59] ROOT
    ├── [15:52:59] Business_Ideas
    ├── [15:52:59] Essay_Final_Submission
    ├── [15:52:59] Journal
    └── [15:58:59] RESOURCES
        ├── [15:52:59] Background_Research
        ├── [15:58:59] CONSIDERATIONS
        │   └── [15:52:59] C++_libraries_to_use
        ├── [15:52:59] Future_Work
        ├── [15:52:59] Software_Requirements
        ├── [15:52:59] Software_Specifications
        └── [15:52:59] Software_Updates

[15:59:59] 201814155959
└── [15:59:59] ROOT
    ├── [15:52:59] Business_Ideas
    ├── [15:52:59] Essay_Final_Submission
    ├── [15:52:59] Journal
    └── [15:59:59] RESOURCES
        ├── [15:52:59] Background_Research
        ├── [15:59:59] CONSIDERATIONS
        │   └── [15:52:59] C++_libraries_to_use
        ├── [15:52:59] Future_Work
        ├── [15:52:59] Software_Requirements
        ├── [15:52:59] Software_Specifications
        └── [15:52:59] Software_Updates

[16:00:59] 20181416059
└── [16:00:59] ROOT
    ├── [15:52:59] Business_Ideas
    ├── [15:52:59] Essay_Final_Submission
    ├── [15:52:59] Journal
    └── [16:00:59] RESOURCES
        ├── [15:52:59] Background_Research
        ├── [16:00:59] CONSIDERATIONS
        │   └── [15:52:59] C++_libraries_to_use
        ├── [15:52:59] Future_Work
        ├── [15:52:59] Software_Requirements
        ├── [15:52:59] Software_Specifications
        └── [15:52:59] Software_Updates

```

```

[16:01:59] 20181416159
└─ [16:01:59] ROOT
    └─ [15:52:59] Business_Ideas
    └─ [15:52:59] Essay_Final_Submission
    └─ [15:52:59] Journal
    └─ [16:01:59] RESOURCES
        └─ [15:52:59] Background_Research
        └─ [16:01:59] CONSIDERATIONS
            └─ [15:52:59] C++_libraries_to_use
        └─ [15:52:59] Future_Work
        └─ [15:52:59] Software_Requirements
        └─ [15:52:59] Software_Specifications
    └─ [15:52:59] Software_Updates

[16:02:59] 20181416259
└─ [16:02:59] ROOT
    └─ [15:52:59] Business_Ideas
    └─ [15:52:59] Essay_Final_Submission
    └─ [15:52:59] Journal
    └─ [16:02:59] RESOURCES
        └─ [15:52:59] Background_Research
        └─ [16:02:59] CONSIDERATIONS
            └─ [15:52:59] C++_libraries_to_use
        └─ [15:52:59] Future_Work
        └─ [15:52:59] Software_Requirements
        └─ [15:52:59] Software_Specifications
    └─ [15:52:59] Software_Updates

[16:03:59] 20181416359
└─ [16:03:59] ROOT
    └─ [15:52:59] Business_Ideas
    └─ [15:52:59] Essay_Final_Submission
    └─ [15:52:59] Journal
    └─ [16:03:59] RESOURCES
        └─ [15:52:59] Background_Research
        └─ [16:03:59] CONSIDERATIONS
            └─ [15:52:59] C++_libraries_to_use
        └─ [15:52:59] Future_Work
        └─ [15:52:59] Software_Requirements
        └─ [15:52:59] Software_Specifications
    └─ [15:52:59] Software_Updates

```