NTNU – Trondheim
Norwegian University of
Science and Technology

Intelligent Scheduled Backup Using Duplicity

Håkon Nymo Matland

Master of Science in Communication Technology
Submission date: June 2015
Supervisor: Danilo Gligoroski, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title: Intelligent Scheduled Backup Using Duplicity
Student: Håkon Nymo Matland

Problem description:

Duplicity is a full featured backup tool, that provides encrypted bandwidth-efficient backups using the rsync algorithm. It supports different backup server types, and even a chroot'd SFTP server. It works with Linux, FreeBSD, and OpenBSD (out of the box). It does not require root access on the backup servers.

Duplicity provides backup of directories by producing encrypted tar-format volumes and uploading them to a remote or local file server. Because duplicity uses librsync, a software library that implements the rsync remote-delta algorithm, the incremental archives are space efficient and only record the parts of files that have changed since the last backup. Duplicity uses GnuPG to encrypt and/or sign the archives. The encrypted archives will be safe from spying and/or modification by the server or during transmission.

In addition to full fledged server hosts, Duplicity comes with built in support for different major personal cloud providers such as Amazon S3, Google Drive, Rackspace and Mega. This allows easy and convenient cloud backup without having to set up your own server to act as backup destination.

The goals of the project are:

- Install and test Duplicity in different use case scenarios and report the advantages and disadvantages of the software.
- Identify potential pitfalls and propose guidelines on how to use duplicity in a secure and safe manner.
- Propose an intelligent scheduling system that distributes the data over several storage providers to increase availability and decrease probability of restoration failure or data corruption. The system should decrease the overhead of having several copies of the same data by distributing parts of the data in a way that enables restoration without all storage providers available.

Assignment given: 12 January, 2015
Responsible professor: Danilo Gligoroski, ITEM

Abstract

Digital information has rapidly become an important part of everyday human life. Consequently, backup solutions are important to ensure digital property is safely stored and protected. This thesis will do an in-depth study of Duplicity, a backup solution providing encrypted, bandwidth-efficient backup to both local and remote storage.

The first part of the thesis investigate Duplicity in different use case scenarios, reporting on the advantages and disadvantages of the software. Research is done to explore how various options affect backup and restoration time. Particularly, the impact of encryption, compression, and incremental backup chains are of interest. Tests are also conducted with four cloud storage providers to investigate if the choice of cloud provider has a large impact on Duplicity's performance.

Encryption's impact on backup execution time is concluded to be minimal. Users should rather perform analysis of data content to identify if execution time may be decreased through compression level settings. Investigation of incremental backup properties clearly shows some of the issues that arise with the use of incremental backups. While incremental backup techniques save bandwidth and storage cost when performing backups, the resources spent while restoring is greatly increased.

Finally, an original system for intelligent distributed backup to be used together with Duplicity is introduced. The system utilize erasure codes as the cornerstone of a minimalistic client application that distributes partial data to different storage hosts. The main objective of the system is to increase the availability and reliability of backups. System requirements and vital components are identified through analysing the systems main objectives. The ideas and architecture lead to a proof of concept prototype. Open source libraries and self-written source code show how the key components solve the objectives; increased availability and reliability.

Statistical analysis and calculations are utilized to show the availability properties of the system. Consequently, it is concluded that a backup solution using Duplicity and erasure codes is able to provide reliable distributed backup through encoding of the original data.

Sammendrag

Digital informasjon har raskt blitt en viktig del av menneskets hverdag. Følgelig er løsninger som utførerer sikkerhetskopiering viktig for å sikre at digital eiendom er sikkert lagret og beskyttet. Dette prosjektet gjennomfører et grundig studie av *Duplicity*, en backupløsning som tilbyr krypterte sikkerhetskopier med effektiv bruk av båndbredde til både lokale og eksterne lagringsmedium.

Første delen av prosjektet undersøker Duplicity i forskjellige bruksscenarier, og presenterer fordeler og ulemper med programvaren. Gjennom forsøk og eksperiment testes det hvordan forskjellige valgmuligheter i Duplicity påvirker tiden det tar å gjennomføre sikkerhetskopiering og gjenoppretting av data. Testene undersøker spesielt hvordan kryptering, komprimering og inkrementell sikkerhetskopiering påvirker programmet. Duplicity testes også sammen med forskjellige skylagringstjenester for å undersøke om lagringstjenesten har stor påvirkning på ytelsen.

Innvirkningen av kryptering på tiden det tar å gjennomføre sikkerhetskopi blir konkludert med å være minimal. Brukere burde heller analysere datainnhold for å undersøke om tid kan spares gjennom endring av komprimeringsnivå. Undersøkelsene viser tydelig noen av problemene som oppstår ved bruk av inkrementell sikkerhetskopiering. Både båndbredde og lagringskapasitet kan spares ved bruk av inkrementell backup, men ressursbruken under gjenoppretting øker kraftig når den inkrementelle lenken er lang. De fire skylagringstjenestene gir veldig forskjellige resultater når de brukes med Duplicity.

Andre del av prosjektet foreslår et system for intelligent distribuering av sikkerhetskopier for bruk sammen med Duplicity. Systemet benytter "erasure codes" som grunnsten i en minimalistisk klientapplikasjon som distribuerer deler av den opprinnelige dataen til forskjellige lagringsplasser. Hovedmålet med systemet er å øke tilgjengeligheten og påliteligheten av sikkerhetskopier. Systemkrav og vitale komponenter identifiseres gjennom analyse av systemets målsetninger. Idéene og arkitekturen brukes som mal for en prototype for å vise at konseptet fungerer. Prototypen viser hvordan de ulike nøkkelkomponentene løser utfordringene til systemet.

Statistisk analyse benyttes for å kalkulere påliteligheten til systemet. Det konkluderes med at det foreslåtte systemet kan brukes til å tilby pålitelig distribuert backup gjennom koding av de opprinnelige dataene.

Preface

This report is submitted to the Norwegian University of Science and Technology (NTNU) as the master thesis of my Master of Science in Communication Technology at the Department of Telematics (ITEM).

I would like to thank Professor Danilo Gligoroski (Department of Telematics) for being my responsible professor and supervisor. The quality of my project greatly increased through his valuable feedback and support throughout the project.

Contents

List of Figures

List of Tables

List of Source Code Examples

List of Abbreviations

API	Application Programming Interface.
AWS	Amazon Web Services.
CPU	Central Processing Unit.
FEC	Forward Error Correction.
GnuPG	GNU Privacy Guard.
MDS	Maximum Distance Separable.
NSA	National Security Agency.
NTNU	Norwegian University of Science and Technology.
SCP	Secure Copy.
SDK	Software Development Kit.
SFTP	SSH File Transfer Protocol.
SSD	Solid State Drive.
SSH	Secure Shell.
URL	Uniform Resource Locator.
XOR	exclusive-or.

Chapter 1 Introduction

1.1 Motivation

Security of digital information becomes increasingly important as more and more content become digital. Digital information includes personal documents, media and professional business related data. Business plans, databases, and digital correspondence may be inestimably precious if lost. Without a proper backup of digital information, users risk losing valuable, and sometimes irreplaceable, property.

Traditionally backup was often done to local storage media such as optical disks and tape drives. The last decade has seen several cloud storage providers create services that facilitates easy storage of files on the internet. The services are easy to use, even for users without a lot of computer experience. Services like Dropbox offer automatic synchronization of files through a desktop application using easy to understand concepts.[1] Smartphone applications offer features such as automatic upload of photographs taken. Users may view, upload or download files through their browser, or the client application of the platform.

Locally stored backups have limited risk of unintended disclosure of private or sensitive information. However, with the rise of cloud computing, providing users with easy-to-use and cheap cloud storage, new security concerns regarding user information and secure storage of backups is introduced [1]. In addition to the risk of the data being network connected, cloud storage techniques such as deduplication introduce additional attack vectors that needs to be considered [2]. Can data be trusted to large corporations abroad? Is it enough with secure transmission of data, if it is stored in plain text in their data halls? Users of cloud storage providers should be concerned with the privacy of their files.

Several backup software solutions roam the market, easing the process for users

[1] Dropbox: http://www.dropbox.com

with the wish of secured copies of their data. This thesis will focus on Duplicity[2]. Duplicity offers encrypted bandwidth-efficient backup and comes preinstalled on several Unix distributions. Duplicity and its features will be explained in detail in chapter 3.

1.2 Scope and Objectives

The scope of this project is to perform an in-depth analysis of the features of Duplicity. Three objectives have been identified to be the main focus of the thesis. The objectives of the project are:

- Install and test Duplicity in different use case scenarios and report the advantages and disadvantages of the software.
- Identify potential pitfalls and propose guidelines on how to use Duplicity in a secure and safe manner.
- Propose an intelligent scheduling system that distributes the data over several storage providers to increase availability and decrease the probability of restoration failure or data corruption. The system should decrease the overhead of having several copies of the same data by distributing parts of the data in a way that enables restoration without all storage providers available.

The use case scenario objective should inform about configuration and installation steps required to enable backup with Duplicity. The project will identify software dependencies required for different use cases, and showcase the wide array of storage possibilities Duplicity offers. The project will investigate and report on how Duplicity offers incremental and encrypted backup. The use case scenarios will be focused on the usage of different storage hosts, and if there is any significant performance difference between the tested options.

Guidelines on how to use Duplicity in a secure and safe manner will be given after Duplicity has been tested in different use cases. It will provide some basic, but useful steps on how to maximize utilization of Duplicity's capabilities.

The last, and most advanced topic of this project, will investigate how Duplicity can be utilized as part of a system to distribute backups to different storage hosts, increasing the availability and reliability of the backed up data. The project will design a possible system, implement a simple prototype, and provide an analysis of the systems availability and storage overhead properties.

[2] duplicity: http://duplicity.nongnu.org/

1.3 Method

As this project consist of several different, quite independent objectives, the method use varies depending on the objective in question.

The first part of the project consist of a literature study, identifying important background knowledge about incremental backup, encryption with GNU Privacy Guard and possible techniques to obtain increased availability in data communication and storage.

Prior to testing of use case scenarios, an initial test plan was created to identify interesting features to test and measure. Each step to solve each use case is logged, to enable rerun of the scenario if needed. The goal is that each test discussed should contain enough information to allow other parties to run similar tests, perhaps with different hardware and storage hosts. The different scenarios will be benchmarked by appropriate properties to allow analysis and comparison between different options tested. Further discussion and explanation of how appropriate properties are measures is found in chapter 4. Additional items are added to the test plan if interesting, or unexpected, new cases appear throughout testing. The iterative approach is selected due to the experimental nature of the project. The author has not previously used the software tested, and it is useful to consider new possible test scenarios and options as they are discovered.

The intelligent distributed backup system is designed through different phases to identify, develop and analyse different properties of the system. The first phase is to identify the properties the system should have, and discuss how the properties may be achieved. The second phase is to design a possible software architecture of the system. The purpose is to identify possible modularization of the system, to allow efficient implementation of a prototype. The last phase is to analyse the properties of the system. Calculations are done to answer if the objectives of the system are achieved, and to provide insight into the benefits and drawbacks of a distributed backup solution. Additionally, the analysis is important to identify potential improvements or issues that should be addressed in further work.

1.4 Outline

The thesis contains 6 chapters and 4 appendices. A short description of each chapter follows:

Background information on some important key concept, technologies and techniques used in the project is presented in chapter 2. The chapter provides information on the rsync algorithm, GNU Privacy Guard, erasure codes and data compression.

Duplicity is introduced in chapter 3. Information about how Duplicity achieves important features, such as incremental backup and encryption is investigated and discussed. The chapter also describes the initial setup required to use Duplicity, including additional configuration needed for cloud storage utilization.

Chapter 4 describe Duplicity in different use case scenarios. Some of Duplicity's different options are investigated, e.g. encryption, compression and asynchronous upload. The chapter test Duplicity with Dropbox, Amazon S3, Google Drive and OneDrive, reporting on execution time differences when using the different cloud storage providers. Generally, the results are analysed and discussed after they are presented, with further comparisons made as other appropriate tests are conducted. Concluding remarks, and identified possible future work rounds up the chapter.

An intelligent distributed backup system is presented in chapter 5. The system utilize Duplicity and erasure codes to enable distribution of partial data files in a way allowing data restoration even if some of the data is unavailable. It discusses and analyse the effect of using such a system, such as increased availability and reliability while keeping the storage and bandwidth cost lower than with simple redundancy through replication. Concluding remarks, and identified possible future work rounds up the chapter.

Finally, chapter 6 provides a conclusion on the project. Accordingly, it provides a summary of the most important results of the different studies conducted.

The report also contain several appendices. The appendices contain information and material for readers that wish to gain insight out of what is directly presented in the main report, i.e. source code listings, simulation scripts and tutorials.

Two partially independent parts are presented in this report. Consequently, sections on related work corresponding to the particular chapters are found in both chapter 4 and chapter 5.

Chapter 2
Background

This chapter provides background information on some of the technologies used by Duplicity and in this project. To properly understand the content of later chapters, it is vital to know about the key concepts happening behind the scene. Background information is provided on the rsync algorithm, GnuPG, erasure codes and data compression.

2.1 rsync

The rsync algorithm is an algorithm for efficient remote update of data developed by Andrew Tridgell as part of his PhD studies[3, 4]. It led to the widely used rsync software [5].

The rsync algorithm was developed to speed up remote update of data. The algorithm computes which parts of the source data that matches the remote data, with the intention to decrease bandwidth resources required to update the remote file to the new version[3].

The algorithm is explained by Tridgell in “The rsync algorithm” [3]:

> “Suppose we have two general purpose computers α and β. Computer α has access to a file A and β has access to file B, where A and B are "similar". There is a slow communications link between α and β. The rsync algorithm consists of the following steps:
>
> 1. β splits the file B into a series of non-overlapping fixed-sized blocks of size S bytes. The last block may be shorter than S bytes.
> 2. For each of these blocks β calculates two checksums: a weak "rolling" 32-bit checksum (described below) and a strong 128-bit MD4 checksum.
> 3. β sends these checksums to α.

> 4. α searches through A to find all blocks of length S bytes (at any offset, not just multiples of S) that have the same weak and strong checksum as one of the blocks of B. This can be done in a single pass very quickly using a special property of the rolling checksum described below.
> 5. α sends β a sequence of instructions for constructing a copy of A. Each instruction is either a reference to a block of B, or literal data. Literal data is sent only for those sections of A which did not match any of the blocks of B.
>
> The end result is that β gets a copy of A, but only the pieces of A that are not found in B (plus a small amount of data for checksums and block indexes) are sent over the link. The algorithm also only requires one round trip, which minimises the impact of the link latency."

In Tridgell's PhD thesis, the delta calculation is explained through the use of two signatures [4]. The first signature, the rolling checksum, needs to be very cheap to compute for all byte offsets [3, 4]. A stronger, more expensive checksum is calculated on all byte offsets where the cheap signature at A matches the one at B [3, 4]. Together, the two signatures find the balance between efficiency and probability of error.

2.1.1 librsync

librsync is a software library for remote file synchronization using rolling checksum influenced by the rsync algorithm [6]. It is licensed under GNU LGPL v2.1 and used in many tools, such as Duplicity and Dropbox [6, 7, 8].

The librsync library provides four basic operations to provide the possibility to use delta files that enables patching of a file to update it to the newest version [9]. The librsync programmer's manual list four basic operations [9]:

- gensig: Signature generation describing a file.
- loadsig: Loading a signature into memory.
- delta: Calculate a delta from an in-memory signature, and write result to a file.
- patch: Read a delta from a file and apply it to a basis file.

The concept and basic operations are similar to that of rdiff, a command-line scriptable interface to librsync [9, 10].

Algorithms such as rsync are vital to an incremental backup system, solving the hard task of identifying the information in need backup.

2.2 GNU Privacy Guard

GnuPG is an implementation of the OpenPGP standard defined by RFC4880 [11, 12]. It is licensed under the terms of GNU General Public License, allowing it to be used as part of other software [11, 13].

GnuPG was created to serve as a tool for other applications, and provides a cryptographic engine accessible from command prompts, shell scripts and other programs [11].

GnuPG is able to provide confidentiality and integrity to messages and data files [12]. The OpenPGP standard describes how to provide confidentiality through the use of a combination of symmetric-key encryption and public-key encryption. RFC4880 describes the process through a series of steps [12]:

1. The sender generates a random number to be used as a session key.
2. The session key is encrypted using the recipient's public key.
3. The sender encrypts the message using the session key.
4. The recipient decrypts the session key using the recipient's private key.
5. The recipient decrypts the message using the session key.

Encryption may also be provided without the use of public-keys. The symmetric key is then either derived directly from a passphrase through what the OpenPGP standard refers to as String-to-Key Specifiers, or through a two-stage process similar to the one listed above with session keys that are encrypted with a passphrase [12].

GnuPG supports a wide array of encryption algorithms. To view the supported algorithms in an installed GnuPG version, the following command can be run in terminal:

```
$ gpg --version
```

Well known algorithms such as 3DES, CAST5 and AES are among the supported cipher algorithms on the computer used in this project. Hashes can be computed using MD5, SHA1, SHA2 or RIPEMD. Version 1.4.18 is used on the project computer.

2.3 Erasure Codes

Erasure codes are often used to increase reliability of electronic communication. Erasure codes are closely linked to Forward Error Correction (FEC) techniques, which provide increased reliability through the use of error detection and correction codes [14]. An erasure code adds redundancy to a system of data to tolerate failures [15].

James S. Plank describes the simplest form of erasure code as replication. An example of this is RAID-1, where all bytes are stored on two disks, allowing restoration as long as one of the disks has a surviving copy of the bytes [15]. Replication is however quite expensive, in the example of RAID-1 doubling the required storage capacity, and thereby doubling the storage costs. Other erasure codes, such as Reed-Solomon codes are more complex to encode, but require less storage and may tolerate a wide range of failure scenarios. Reed-Solomon codes will be further explained in section 2.3.1.

The key idea behind erasure codes is explained by Luigi Rizzo in [14] as using an encoder to produce n blocks of encoded data from k blocks of source data. The data is encoded in a way allowing any subset of k encoded blocks is sufficient to reconstruct the source data [14]. Such an encoding allows data recovery from up to $n - k$ block losses. Erasure codes that allow reconstruction from $n - k$ block losses are called optimal erasure codes, or MDS codes. They are called optimal erasure codes due to featuring optimal correction capability [16]. Reed-Solomon codes is an example of MDS codes, and will be described further below.

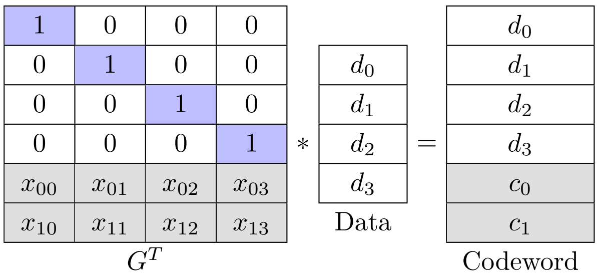
Erasure codes that do not offer optimal space-efficiency may be used in systems that prefer reduced computation cost [17]. One example of nonoptimal erasure codes is flat XOR-codes. XOR-codes compute redundancy elements by performing XOR operations on a subset of data elements, and are appealing because of the offered computational efficiency [17, 18].

Reconstruction of encoded blocks may be simplified through the use of *systematic codes*. When a systematic code scheme is used, the encoded blocks contain an exact copy of the original, i.e. the input data is reproduced among the output data. Systematic codes allow simple reconstruction when all original data blocks are available, as it only has to put the original data back together, without the use of calculations and arithmetic operations [14, 19, 20].

2.3.1 Reed-Solomon codes

Reed-Solomon codes were introduced by I. S. Reed and G. Solomon in “Polynomial codes over certain finite fields” in 1960 [21]. By using finite field arithmetic, computer precision problems are avoided. Ordinary arithmetic often require extra bits to be used to ensure representation without loss of precision [14]. The finite number of elements in finite fields allow total precision as long as the field size is large enough to contain enough different elements to solve the problem [14]. Reed-Solomon codes are created using finite field arithmetic in $GF(2^w)$ [22]. Most implementations use $w = 8$ due its performance, while leaving it possible to encode into 256 different blocks [22, 23].

Encoding with Reed-Solomon codes is done through linear algebra. Codewords are created using a generator matrix created from a Vandermonde matrix. The generator matrix is multiplied with data words to create encoded codewords consisting of the original data together with parity data. The encoding process is visually shown in Figure 2.1, figure reproduced from [22]. Reed-Solomon offers optimal efficiency, meaning that any available parity element can be substituted for any missing data element, a property that provides a deterministic behaviour [23].



G^T
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
x_{00} x_{01} x_{02} x_{03}
x_{10} x_{11} x_{12} x_{13}
$*$
Data
d_0 d_1 d_2 d_3
$=$
Codeword
d_0 d_1 d_2 d_3 c_0 c_1

Figure 2.1: Reed-Solomon coding for $k = 4$ and $r = 2$, where k is number of blocks containing data, and r is number of blocks that contain parity data. [22]

Reed-Solomon codes are quite expensive compared to other Erasure code techniques [22, 24]. In $GF(2^w)$ addition is cheap, due to it being carried out by performing exclusive-or (XOR) of elements. Multiplication is however much more expensive, and is often performed through the use pre-computed look-up tables [25]. The look-up tables size is fairly low in $GF(2^8)$, which is often used in computing due to its byte based nature [25]. The look-up tables do however grow very large for higher values of w, making it infeasible to make tables that match word size in 32-bit or 64-bit systems, namely $GF(2^{32})$ and $GF(2^{64})$ [25].

2.4 Data Compression

The discipline of data compression tries to decrease the size of digital data. If the original representation of digital data contain redundancies, the data is compressed by reducing or removing the redundant data [26]. To understand compression, it is important to understand the difference between information and data. Information is represented by data, and the goal of compression is to decrease the data size, without loosing information. If it is possible to shorten the data representation without removing any information, the data is compressible. David Salomon states that: "Any nonrandom data is nonrandom because it has structure in the form of regular patterns, and it is this structure that introduces redundancies into the data." [26], i.e. nonrandom data is compressible. The process of decompression reverses the compression, i.e. it reconstructs the original data.

Just as nonrandom data is compressible, random data is often hard to compress, as it has no structure. Data that has no redundancies can not be compressed. Similarly, data that is already compressed is impossible to compress further, as the redundant data is previously removed [26].

Data compression techniques are often used in backup solutions to decrease the storage and bandwidth cost of backups.

Chapter 3 Introduction to Duplicity

Duplicity offers encrypted bandwidth-efficient backup and runs on POSIX-like operating systems such as various Linux distributions and Apple OS X [8]. The main objective of this chapter is to investigate and report on how to use the basic features of Duplicity. The chapter will contain general information about Duplicity, and describe some of the most interesting options/features the software offers. The chapter should enable readers to perform simple Duplicity usage of their own.

Duplicity is a command line tool, and commands are executed through the computers shell terminal. Different front end solutions exist that use Duplicity as the back end. This project tests regular Duplicity, without any form of front end extension.

This chapter will contain general information about Duplicity, describe some of the most interesting options/features and how to set up a computer to enable backups with Duplicity. A secondary objective of the chapter is to prepare for the use case scenario tests described and discussed in chapter 4. Understanding the basic usage of Duplicity should also improve the design and development of the distributed backup system in chapter 5.

The first section of this chapter explains how to install and set up Duplicity. The next section discusses one of Duplicity's main features, namely incremental backup. Section 3.3 provides information about Duplicity's use of GnuPG, explaining the different ways Duplicity are able to encrypt backup archives, and explores useful options users should know about.

After the key features and concepts of Duplicity is introduced, a quick tutorial on how to perform backups and restorations with Duplicity is given in section 3.4.

Duplicity is able to perform backup over many different file transfer protocols. Section 3.5 and section 3.6 explain how to set up a computer to use Duplicity with popular file transfer protocols and cloud providers.

3.1 Installation of Duplicity

Duplicity is pre-installed on many Linux distributions, so before trying to install it is a good idea to check if Duplicity is already available on the system. One way to check if Duplicity is installed is by executing `duplicity --version` in the computers command line shell. If Duplicity is installed, it will show something similar to "`duplicity 0.7.01`", depending on the version installed.

A quite extensive list of dependencies are required to properly install and run Duplicity. The easiest way around this obstacle is through the use of major package managers such as apt-get (Ubuntu etc.) and homebrew (Apple OS X). It is however important to note that package managers might not be updated with he latest version of the software, and users should check if the version offered is the same, or similar, to the version offered on the Duplicity home page [8]. Another positive advantage derived from the use of package managers is easy updating of the software as new versions are added.

Before Duplicity is manually built/installed, the following requirements have to be met [27]:

- Python v2.4 or later
- librsync v.0.9.6 or later
- GnuPG v1.x for encryption
- python-lockfile for concurrency locking.

When the prerequisites are met, the following steps are required to install Duplicity:

1. Download the tarball containing the desired version of Duplicity from the Duplicity homepage [8]
2. Extract the archive
3. Build and install

Extraction and installation is done by running the following commands in terminal:

```
$ tar -zxvf duplicity-versionNumber.tar.gz
$ cd duplicity-versionNumber
$ python setup.py install
```

After completion of the installation steps, Duplicity should be runnable. Users may verify that the Duplicity command is available by executing the `duplicity --version` command.

Prerequisites mentioned in this section are the ones required for the minimum setup of Duplicity. Additional python libraries are required for running with different file transfer protocols. Some of these prerequisites, and how to install them is explained in section 3.5 and section 3.6

3.2 Incremental Backup

Incremental backup schemes are used to achieve faster and smaller backups on systems with frequent backups [28]. An incremental backup scheme strives to minimize the bandwidth and storage usage through only transmitting and storing the changes since the last backup. By updating existing backups with altered or added data, the use may decrease the storage capacity required greatly compared to a series of full backups [28].

The major advantage of incremental backup schemes is easily explained through an example:

- A user does an initial backup of a directory of 100 MB.
- One week later, the only change to the directory is a new file. The file is 4 MB.
- The incremental backup software identifies the new file as the only new/altered information, and uploads the new file.
- If the user did a new full backup instead, a new backup of 104 MB would be uploaded. Resulting in one backup of 100 MB and one of 104 MB.

The difficult part of incremental backup schemes is how to decide what information to upload. The problem is solved by algorithms such as rsync, previously described in section 2.1. The use of cryptographic hash function enables identification of changed or new information with minimal data transportation required between the backup client and storage host.

Duplicity stores backups as tar-format archives [8]. When Duplicity backups are executed, it will investigate if previous backups of the same source directory exist at the same target path. If no corresponding previous backup is found, it will create a full backup of the directory. The next time Duplicity is executed with the same configuration, it will discover that previous backups already exist. Metadata will be retrieved to calculate, and create, incremental delta archives storing the required changes to be able to restore the new version of the file from the chain of backup archives [29]. A chain of backup archives is a collection of corresponding backup archives from the previous full backup to the desired incremental backup [28].

One disadvantage with incremental backup schemes is the increased restoration time. To restore the newest version of a backup, the software requires every incremental backup since the last full backup [28]. The chain of archives is used to compute the desired version of the files. A long chain of incremental archives increases the computation required successfully restore the desired version of the files. The added computation also increases the probability of failed restoration, due to every link in the chain being dependent on every previous link. If one of the archives in the chain become corrupted, it will have an impact on every other archive later in the chain.

The disadvantage of incremental backup schemes can be mitigated by periodically doing full backups to limit the archive chain. This mitigation technique comes with the price of increased storage capacity required together with increased bandwidth usage. The tricky part is to find out with what frequency full backups should be executed.

3.3 Encryption

One of Duplicity's key properties is the ability to create encrypted backups. Duplicity encrypts the backup archives by default using GnuPG. All cryptographic operations are done client-side, enabling safe transmission and storage of backed up information [30]. The fact that encryption is done client-side allows users to store backups even at storage providers they don't fully trust.

As described in section 2.2, GnuPG offers both symmetric and asymmetric encryption. By default, Duplicity utilizes symmetric encryption by prompting the user for a passphrase, or looking for the `PASSPHRASE` environment variable. The use of environment variables is essential to allow scripts to execute backups without any user interaction.

Duplicity is also able to use public key encryption through passing the desired public key ID to the `--encrypt-key` option. If the user wish to use public key encryption, the client computer need to be set up with keys in the GnuPG keyring. If the user does not have a set of keys from previous use, he/she needs to create keys using `gpg --gen-key`.

Parts of the key generation sequence with GnuPG is shown in Figure 3.1. The user selects the desired key type, key length and key expiry date. In addition to the questions shown in the figure, the fields Real name, Email address and comment needs to be filled out. GnuPG will ask for a passphrase to protect the secret key, and

```
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
Requested keysize is 2048 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 1y
Key expires at Fri May 27 11:17:34 2016 CEST
Is this correct? (y/N) y
```

Figure 3.1: Key generation sequence with GnuPG. User is prompted to select desired key type, key length and expiry date

finally the keys will be created. The final output of the key generation sequence is shown in Figure 3.2. The public key ID is the ID that should be passed to Duplicity for public key encryption. It is marked by yellow in the figure. More information about generating keys may be found in the GnuPG manual [31].

```
public and secret key created and signed.
gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   2  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 2u
gpg: next trustdb check due at 2016-05-27
pub   2048R/ 088830C6  2015-05-28 [expires: 2016-05-27]
      Key fingerprint = 8604 2DC9 54C7 2FCC 9719  CDC3 B2B9 D58F 0888 30C6
gpg: conversion from ‘utf-8’ to ‘US-ASCII’ failed: Illegal byte sequence
uid                  Haakon Nymo Matland (Master thesis test key)
 ↪  <haakon@nymomatland.com>
sub   2048R/8E9A4827 2015-05-28 [expires: 2016-05-27]
```

Figure 3.2: Output from a successful key generation sequence with GnuPG.

3.4 Tutorial on Basic Duplicity Usage

This section will explain the basics of how to perform backups and restoration with Duplicity. It will show how it is done with local storage as backup host. The most basic command for performing a backup is very simple:

```
$ duplicity <path to directory that will be backed up> file://<path
↪  to where backed should be stored>
```

Figure 3.3 show the output from executing a full backup with Duplicity. The first part of the output state that no previous signatures are found, and that it will execute a full backup. Information about the directory backup up is shown in the backup statistics starting at line 7. It contains information such as the size of the backup and number of files backed up. Fields such as `DeletedFiles` and `ChangedFiles` are used for incremental backup information, and contain no interesting information in this example output.

```
$  duplicity master/ file://master_backup
Local and Remote metadata are synchronized, no sync needed.
Last full backup date: none
GnuPG passphrase:
Retype passphrase to confirm:
No signatures found, switching to full backup.
--------------[ Backup Statistics ]--------------
StartTime 1431519841.15 (Wed May 13 14:24:01 2015)
EndTime 1431519841.26 (Wed May 13 14:24:01 2015)
ElapsedTime 0.11 (0.11 seconds)
SourceFiles 15
SourceFileSize 1614383 (1.54 MB)
NewFiles 15
NewFileSize 1614383 (1.54 MB)
DeletedFiles 0
ChangedFiles 0
ChangedFileSize 0 (0 bytes)
ChangedDeltaSize 0 (0 bytes)
DeltaEntries 15
RawDeltaSize 1613703 (1.54 MB)
TotalDestinationSizeChange 1296842 (1.24 MB)
Errors 0
-------------------------------------------------
```

Figure 3.3: Example output from a full Duplicity backup

Duplicity will automatically discover that a previous backup exist if the previous command is run again. If a previous backup exists, Duplicity will default to incremental mode and compute any source directory changes. The output of an incremental backup is shown in Figure 3.4. As seen in the output, Duplicity found 2 new and 1 modified file, increasing the backup size by 321 bytes.

In some cases users may want to force a full backup even if a previous backup exists. This is done by adding the full flag after duplicity:

```
$ duplicity full <path to directory that will be backed up>
↪  file://<path to where backed should be stored>
```

```
$  duplicity master/ file://master_backup
Local and Remote metadata are synchronized, no sync needed.
Last full backup date: Wed May 13 14:23:56 2015
GnuPG passphrase:
Retype passphrase to confirm:
--------------[ Backup Statistics ]--------------
StartTime 1431521260.52 (Wed May 13 14:47:40 2015)
EndTime 1431521260.54 (Wed May 13 14:47:40 2015)
ElapsedTime 0.02 (0.02 seconds)
SourceFiles 16
SourceFileSize 1614470 (1.54 MB)
NewFiles 2
NewFileSize 463 (463 bytes)
DeletedFiles 0
ChangedFiles 1
ChangedFileSize 54 (54 bytes)
ChangedDeltaSize 0 (0 bytes)
DeltaEntries 3
RawDeltaSize 82 (82 bytes)
TotalDestinationSizeChange 321 (321 bytes)
Errors 0
-------------------------------------------------
```

Figure 3.4: Example output from an incremental Duplicity backup

The restoration command is very similar to the backup command. The two paths simply change position in the command:

```
$ duplicity file://<path to where backed is stored> <path to store
↪  the restored files>
```

Duplicity will automatically discover that a protocol path is given as the first argument, and enable the restoration mode of operations. Manual restoration is also possible, but quite complex without the use of Duplicity. The procedure includes manual decryption with GnuPG, extraction of files from archives and potentially concatenation of parts split on different volumes. If trying to restore an incremental backup, rdiff must be used to fully restore the latest version.

Restoration of a backup as it was at a specific time is enabled through the `--time` or `-t` option. The command below would restore the backup as it was 1 week ago:

```
$ duplicity -t 1W file://<path to where backed is stored> <path to
↪  store the restored files>
```

Other syntax variations are found in the Duplicity manual, such as specifying a specific date [29].

Some of Duplicity's options are listed below. For a full overview of actions and options refer to the Duplicity manual page [29]

`--full-if-older-than time`: Forces a full backup if the latest full backup is older than provided time.

`--encrypt-key` keyid: Encrypt with public key corresponding to keyid.

`--gpg-options` options: Pass GnuPG options for Duplicity to use with GnuPG.

`--exclude` shell_pattern: Exclude any files that match the provided shell pattern.

`--exclude-filelist` filename: Exclude the files listed in the provided file

`--log-file` filepath: Write Duplicity output to file.

`--full-if-older-than time`: Perform a full backup if the latest full backup is older than the given time.

3.5 File Transfer Protocols

Duplicity comes with built in support for a large number of different file transfer protocols. Some of the protocols work out of the box, while others require specific back-ends to be installed in addition to Duplicity [29]. The wide support of file transfer protocols is useful in situations where the user has limited permissions to storage host computer. Instead of being forced to install a specific program to handle communication with the client computer, the user may identify which protocols the host computer already support, and set up Duplicity accordingly. Duplicity may also be used without any form of network transmission at all. Duplicity is able to store the backup at the local file system, making it possible to store backups to external hard drives etc.

Security of the different file transfer protocols should be considered when using Duplicity for backup. Even if the backup archives are encrypted, information will leak if the communication between the client and storage host is captured and readable. Some information that is gained by just being able to read the name of the files:

- The files are created by duplicity.
- Date and time of the backup.

- The files are encrypted through the use of GnuPG.
- If the backup was full or incremental.

Generally, it is preferred to give away as little information about personal data as possible, and thus secured transfer protocols should be chosen, even if the transferred data is encrypted.

Many of the protocol require additional back-ends to be installed on the client computer. Most of the protocol or storage provider specific back-ends may be installed with pip, a package manager for python libraries/modules [32]. Pip provides easy installation and updating of libraries and modules for python and is recommended by PyPA [33].

SSH File Transfer Protocol

SSH File Transfer Protocol (SFTP) is a protocol layered on top of Secure Shell (SSH) to provide secure transfer of files over network connections [34]. SFTP derives its security by communicating over an SSH connection, and is thus as safe as the SSH connection [34]. SFTP is a protocol packaged with SSH to provide secure transfer of files over network connections. SFTP is one of the possible protocols to use with Duplicity, and is the protocol used in this thesis when using a Ubuntu computer as storage hosts.

To enable Duplicity backups with SFTP as file transfer protocol, the remote storage host must have SSH capabilities installed and running. One way to solve this requirement is through the use of OpenSSH [35]. OpenSSH is a free SSH connectivity tool created by the OpenBSD Project that provides SSH, Secure Copy (SCP) and SFTP capabilities.

In addition to SSH capabilities at the storage host, the client computer requires one of the two SSH back-ends supported by Duplicity, namely the paramiko back-end or the pexpect back-end. In this project, the paramiko back-end is used. Paramiko is a python module that implements the SSH2 protocol, written entirely in python [36]. Paramiko may be installed by following the directions on the project GitHub repository [36], or through pip, the package manager previously mentioned:

```
$ pip install paramiko
```

After every dependency required to use SFTP is met, the following command may be used to back up with Duplicity:

```
$ duplicity <path to source directory>
 ↪  sftp://user[:password]@other.host[:port]//<absolute path to
 ↪  backup destination>
```

For additional security, OpenSSH may be set up to use public key authentication instead of password authentication [34]. Another positive effect of using public key authentication is to avoid the password prompt when executing a duplicity backup, easing the process of using duplicity together with for example Cron daemon.

3.6 Cloud Provider Support

Duplicity supports the use of several cloud storage providers as storage hosts. Cloud storage is an alternative for users that either lacks a computer/server to store backups on or prefers to have the backups accessible on high availability cloud servers. Cloud providers often have high redundancy on the stored data, providing users with a way to store data safely [37].

While cloud storage provides many positive properties, some users may dislike the idea of storing their data on corporate servers. How can a user be sure the company does not read, or use, their private data? Leaks from whistleblowers have shown how the National Security Agency (NSA) targets inter-datacenter communications from some of the largest cloud providers [38]. Duplicity defaults to encrypted backups, decreasing the risk of unintended disclosure of private or sensitive information.

3.6.1 Amazon S3

Amazon offers cloud storage as part of their Amazon Web Services (AWS) portfolio. [1] The price of Amazon S3 depends on storage used, in addition to data transfer out. Amazon offers a free trial if the user prefers to test the solution before spending money on it.

Amazon S3 store files in what it calls a bucket. To set up duplicity to back up to an Amazon S3 bucket, a Python package named *Boto* must be available on the client computer [29, 39]. The Boto Python package provides a Python interface to many Amazon Web Services, and among them, Amazon S3.

The Boto package can be installed from the shell terminal via pip [32]:

```
$ pip install boto
```

[1] Amazon Web Services (AWS) - Cloud Computing Services: http://aws.amazon.com/

As soon as Boto is installed, Duplicity should be ready to handle backups to Amazon S3 buckets.

The user is required to create Access Keys to use Amazon S3 together with Duplicity. The Access Keys are used by Boto to authenticate with Amazon's servers. Access Keys are created at the AWS console. Begin by pressing the button in the top right corner called "Security Credentials". The next step is to go to "Access Keys (Access Key ID and Secret Access Key)", press create New Access Key, and download the credential comma separated file. The credentials should be stored somewhere safe, as it is not possible to download the Secret Access Key again at a later stage. If the secret key is lost, a new set of keys must be created.

The region of the Amazon S3 bucket should be chosen carefully, as it has an impact on the options that needs to be passed to Duplicity. For the setup used in this thesis, a bucket in Ireland was used, requiring the options `--s3-use-new-style` and `--s3-european-buckets` to be passed to Duplicity on backup and restoration execution. The region selected also impacts the host Uniform Resource Locator (URL) to use with Duplicity. A list of the different region URL's is found in the AWS documentation [40]. The S3 endpoint for Ireland is s3-eu-west-1.amazonaws.com.

The command below show how to execute Duplicity with S3 as storage host.

```
$ duplicity [region options] <path to source directory>
↪  s3://<endpoint url>/<bucketname>/<backup directory name>
```

3.6.2 Google Drive

Google's cloud storage solution is named Google Drive.[2] Google offers 15 GB of free storage, which may be upgraded for a monthly sum if 15 GB is not enough. The storage is shared across Google Drive, Gmail and Google+ Photos.

Duplicity's Google Drive back-end requires a python library named *GData Python Client* [29, 41], it is easily installed with pip:

```
$ pip install gdata
```

Google Drive authenticates the user through the `FTP_PASSWORD` environment variable or the password directly in the Duplicity command. If the password is not provided in the command or by the `FTP_PASSWORD` environment variable, Duplicity will prompt the user through the terminal at command execution. The command below shows how to execute Duplicity with Google Drive as storage host:

[2] Google Drive - Cloud Storage & File Backup: https://www.google.com/drive/

```
$ duplicity <path to source directory>
 ↪  gdocs://username[:password]@hostname/<backup directory name>
```

`hostname` will normally be gmail.com for most Google accounts, but may also be other domain names that are used with Google Apps.

3.6.3 OneDrive

Microsoft's consumer cloud storage service is named OneDrive.[3] Similar to Google, Microsoft provide 15 GB of storage for free, with additional storage available with a monthly paid subscription.

Duplicity's OneDrive back-end require two Python libraries on the client computer. The first library, Requests, is a python library for better HTTP capabilities [42]. The second library, Requests-OAuthlib, providers OAuth capabilities for Requests [43]. The two libraries are easily installed through pip:

```
$ pip install requests
$ pip install requests_oauthlib
```

```
$ duplicity master onedrive:///master_backup

In order to authorize duplicity to access your OneDrive, please open
 ↪  https://login.live.com/<redacted identifiers> in a browser and copy the URL of
 ↪  the blank page the dialog leads to.

URL of the blank page: https://login.live.com/oauth20_desktop.srf?code=<redacted
 ↪  identifiers>
Local and Remote metadata are synchronized, no sync needed.
Last full backup date: none
GnuPG passphrase:
Retype passphrase to confirm:
No signatures found, switching to full backup.
```

Figure 3.5: Shell output from OneDrive authentication sequence.

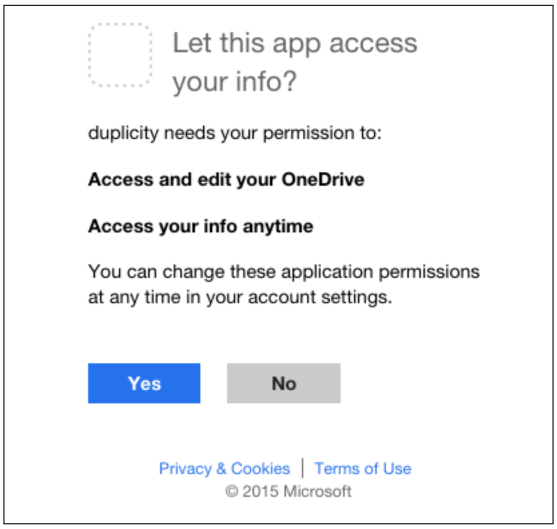
The first time Duplicity is executed with the OneDrive back-end, the user will be prompted with a URL in the shell window. The URL is used to authorize Duplicity to use the OneDrive Application Programming Interface (API) to access the users account. The shell output from an OAuth prompt is shown in Figure 3.5. Users will be given a URL to open in a web browser, where OneDrive will ask the user if Duplicity should be granted authorization to alter OneDrive content.

[3] Microsoft OneDrive: https://onedrive.live.com

The browser permission message is shown in Figure 3.6. When permissions are granted, a blank page will be shown in the browser. The URL of the blank page is pasted into the terminal window, and Duplicity stores the OAuth tokens in the file "`~/.duplicity_onedrive_oauthtoken.json`".

Because the OneDrive credentials are stored in a file, the Duplicity command with OneDrive as storage host is very minimal:

```
$ duplicity <path to source directory> onedrive:///<target
↪  directory>
```


Let this app access your info?

duplicity needs your permission to:

Access and edit your OneDrive

Access your info anytime

You can change these application permissions at any time in your account settings.

Figure 3.6: OneDrive prompts the user to allow Duplicity permission to access and edit OneDrive content.

3.6.4 Dropbox

Dropbox is another popular cloud storage service.[4] Dropbox offer 2 GB of free space, which may be increased for a monthly sum.

The Dropbox duplicity back-end require the Dropbox Software Development Kit (SDK) [29]. The Dropbox Python SDK may be downloaded and installed by following the instructions given by Dropbox themself [44]. It is also easily installed with pip:

```
$ pip install dropbox
```

[4] Dropbox: https://www.dropbox.com

Similar to the OneDrive authentication mechanism, users will be prompted with a URL the first time they execute Duplicity with the OneDrive back-end. The OAuth token obtained is saved in the file " `~/.dropbox.token_store.txt`" to enable future Duplicity executions without browser interaction [29].

Duplicity backups with Dropbox as storage host require the backups to be stored in the "*/Apps/duplicity*" directory of the Dropbox account. Duplicity is also unable to create new directories, forcing the user to manually create an empty directory through the Dropbox web interface, a disadvantage not found with any of the other storage host back-ends tested.

If the user has a Dropbox client installed and logged into the account he/she intend to use with duplicity, it can be a good idea to turn off synchronization with the duplicity backup directory. It is not very useful to keep a local version in addition to the version stored at Dropbox storage servers. The copy stored locally by the Dropbox client will not be used by the Duplicity Dropbox back-end, i.e Duplicity will upload the backup to Dropbox leading to the client downloading it, leading to unnecessary high bandwidth costs.

Because the Dropbox credentials are stored in a file, and the user is forced to manually create the directory to host the Duplicity backup, the Duplicity command with Dropbox as storage host is very minimal:

```
$ duplicity <path to source directory> dpbx:///<directory created
↪  in /Apps/duplicity>
```

3.7 Concluding Remarks

This chapter has investigated basic usage of Duplicity. It has shown how to perform simple backup and restoration procedures to both local file systems and remote storage hosts, and set up the project client computer for further tests and investigation. Duplicity comes with a wide range of different features and options, we have introduced a few in this chapter, but to utilize the full potential of Duplicity a user should read the documentation to know every possibility.

Duplicity enables storage provider independent secured backups through the use of GnuPG. The backed up data is secured through cryptographic operations at the client, making sure the data is safeguarded prior to being uploaded to the storage host.

Duplicity struggles with fragmented software requirements. Full utilization of different features requires additional software, giving the software a somewhat difficult learning curve. The Duplicity manual page provides limited information about how to install different libraries, and usually direct the user to other web pages for information and installation guidelines.

As shown throughout the chapter, the setup process of Duplicity is simplified through the use of package managers. Package managers automatically make sure dependencies are met, and allow easy installation, update and installation of software packages.

Chapter 4 Testing of Duplicity in Different Scenarios

This chapter will investigate and test some of the different possible ways of using Duplicity. Duplicity offers incremental backup with client-side encryption of data, enabling efficient backup to remote storage providers while denying unauthorized disclosure of information.

The first part of this chapter, section 4.2 and 4.3), describes the setup used for testing. It informs what hardware the tests are run at, what the test data contains, and how the test data was created.

Section 4.4 and 4.5 investigate how different encryption options and compression levels impact the execution time of backing up and restoring with Duplicity using local storage. The local tests are followed by similar tests with another computer as storage host in section 4.6. Running the same tests with both local and remote storage provide greater insight on how the different phases of the program execution affect the total runtime.

The execution time of backup and restoration when utilizing incremental backup is investigated in section 4.8. The tests focus on verifying the hypothesis that restoration time is increased when the backup is part of an incremental chain.

Duplicity with different cloud storage providers is investigated in section 4.9. The goal is to identify differences between the different providers performance together with Duplicity. The following cloud storage providers will be tested:

- Amazon S3
- Google Drive
- Dropbox
- OneDrive

Each of the different cloud storage provider require different back-end libraries for Duplicity support. This chapter will investigate if the choice of cloud storage provider has any significant impact on backup and restoration execution time. The performance analysis of using Duplicity with different storage hosts are helpful for users that are unsure about which storage provider they should use with Duplicity, but also provides useful information for chapter 5; Intelligent Scheduled Backup.

The performance of the different scenarios is in general measured by the execution time of the scenario. In this chapter, execution time is defined as the real time provided by the time command in the clients shell. The real time provided by the time command is the elapsed time between invocation and termination of the command it runs[1].

4.1 Related Work

Santos and Bernardino conducted interesting experiments involving Duplicity in "Open Source Tools for Remote Incremental Backups on Linux: An Experimental Evaluation" [45]. The study tests five different remote incremental backup tools in the categories:

- Experiments without compression/encryption.
- Experiments with compression.
- Experiments with compression and encryption.

Duplicity is tested in the two last categories, and provided good results compared to the other solutions tested. Santos and Bernardino concluded that rsync was the most efficient backup tool for simple data replication, while Duplicity provided the overall best results for tools capable of compression and encryption.

Santos' and Bernardino's study provide an interesting comparison between the different incremental backup tools, but does not serve as an in-depth study of Duplicity. The main focus of Santos and Bernardino is to find the best performing tool, while this study tries to test the various options and settings of Duplicity against each other. Santos and Bernardino do for example not state if the restoration times provided are for the incremented backup set, or for the full backup set. Additionally it is not noted if the encryption tests are conducted with symmetric or asymmetric encryption.

[1] Time command manual page: http://linux.die.net/man/1/time

The study presented in this chapter contains experiments investigating how the restoration time of Duplicity changes with the length of incremental backup chains. It also performs tests to see if there is any performance difference between using Duplicity with GnuPG's symmetric encryption scheme and Duplicity's with GnuPG's asymmetric encryption scheme.

This chapter also does several tests not directly related to the experiments done by Santos and Bernardino. Especially the experiments measuring Duplicity's performance with different cloud storage provider should be interesting for users interested in utilizing Duplicity to perform incremental and encrypted backups to the cloud.

4.2 System Setup

The use cases in this chapter have been tested in Apple OS X Yosemite. The directions given on how to set up duplicity is applicable also for other Unix operating systems. Every benchmark/measurement is done on an Apple Macbook Pro with specifications listed in Table 4.1. All tests are done at NTNU, granting stable high-end internet performance.

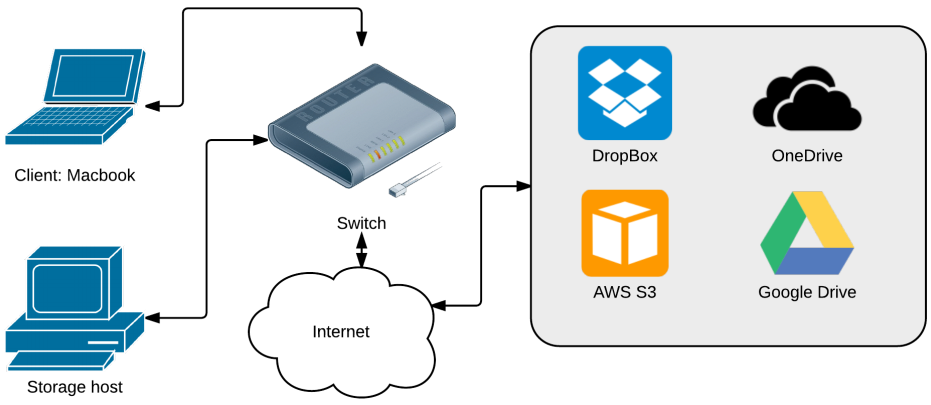


Figure 4.1: The client computer is set up to enable backup to several different storage options.

In addition to local file system storage, remote storage will be tested with a local Ubuntu computer, and the cloud providers listed in the chapter's introduction. The setup used in the tests is shown in Figure 4.1.

Duplicity is tested using the development release 0.7.01. The last stable version is 0.6.25, but lacks back-end support for OneDrive which is one of the storage providers

Specification of Client Machine	
CPU	2.4 GHz Intel Core i5
Memory	8 GB 1600 MHz DDR3
Harddrive	256 GB Solid State Drive
Operating system	Apple OS X Yosemite
Network adapter	Apple USB Ethernet Adapter

Table 4.1: Specifications of Apple Macbook Pro running the use case scenarios

tested in this project. The Duplicity web page also states “The 0.6 series is in the process of being deprecated” [8].

4.3 Setup of Test Data

This section will describe the files used for testing, and how they were created.

The tests are done using the same data to avoid any inconsistencies in the results due to file differences. The directories, with it’s content, were created using a bash script shown in Appendix A. The test directory contains:

- 10 directories.
- Each directory contain another 10 directories.
- Each subdirectory contain:
    - 200 files of size 1KB.
    - 45 files of size 100KB.
    - 5 files of size 1MB.

This creates a total of 110 folders, 25000 files and a total directory size of 1,07 GB. The content of the files is described further in section 4.3.1.

A copy of the directory described above are run through another script to prepare it for incremental backup testing. The following modifications are done:

- In each directory containing files, the following files are deleted:
    - 40 files of size 1KB are deleted.
    - 9 files of size 100KB are deleted.
    - 1 file of size 1MB is deleted.

- In each directory containing files, the following files are created:
    - 40 files of size 1KB are generated.
    - 9 files of size 100KB are generated.
    - 1 file of size 1MB is generated.
- In each directory containing files, the following files are modified:
    - 40 files of size 1KB have 512B random data removed, and 512B random data appended.
    - 9 files of size 100KB have 50KB random data removed, and 50KB random data appended.
    - 1 file of size 1MB has 512KB random data removed, and 512 KB random data appended.

The modified directory will have the same size as the previously created directory, but a significant portion of the original files have been deleted or heavily modified, and several new files have been created.

4.3.1 Generation of test files

The content of the files are pseudo-randomly generated bytes mixed with zero byte data. The zeroes provide the files with redundant data, and are important to gain insight into the compression capabilities of Duplicity and the underlying technologies. The zero-data complements the random data that gives almost no compression possibilities. The files consist of 75% pseudo-random bytes and 25% zeroes.

To decide how to generate the pseudo-random bytes, a quick test were done with the OpenSSL rand tool [46] and dd with **/dev/urandom** as input [47]. The test were done using the time command for timing statistics about the program runs. This test showed a significant difference in the execution time of file generation. The result is showed in Table 4.2, and the script used to measure is found in Appendix B.

The results show that dd with infile **/dev/urandom** is quicker on very small files, while the OpenSSL approach is a lot quicker on bigger files. For consistency, all further file generation in this project is generated with the OpenSSL approach.

The zeroes part of the files are generated through the use of dd with **/dev/zero** as input [47].

Filesize	**OpenSSL Rand**	**dd with if=/dev/urandom**
1KB	0.008s	0.006s
10KB	0.008s	0.005s
512KB	0.024s	0.043s
1MB	0.040s	0.089s
100MB	3.512s	8.683s
1GB	34.831s	85.849s

Table 4.2: Timing statistics of creating random files with OpenSSL and dd with infile /dev/urandom

4.4 Encryption's Impact on Execution Time

As discussed in chapter 3, duplicity has built in encryption capabilities. This section will report the result of tests done to investigate the impact encryption has on the backup and restoration process.

The goal of this test is to investigate to what degree it is preferable to disable encryption when storing the backup on a trusted storage provider. Both backup and restoration is tested with and without encryption. The tests are done storing the backup on the local file system, to avoid any external bottlenecks to impact the result.

The same folder was backed up using GnuPG symmetric encryption, GnuPG asymmetric encryption and with encryption disabled. The tests are done with fresh full backups. The effect encryption has on incremental backups is not tested in this section. Each full backup was run 5 times, to mitigate inconsistent result due to other processes on the client machine.

The following commands were ran to execute the tests in this section:

```
#Symmetric encryption enabled (default settings):
duplicity testDirectory/ file://<target path>
#Asymmetric encryption enabled:
duplicity --encrypt-key <gpg pub key id> testDirectory/
↪  file://<target path>
#Disabled encryption:
duplicity --no-encryption testDirectory/ file://<target path>
```

The backup execution time result, shown in Table 4.3 yield very similar execution time in all three cases. The small impact encryption has on the backup execution

Encryption	Backup time	Restoration time	Directory size
Symmetric	57.73s	47.23s	761.7 MB
Asymmetric	56.27s	49.88s	760.6 MB
Disabled	56.10s	24.38s	786.9 MB

Table 4.3: Average execution time of duplicity with symmetric, asymmetric and disabled encryption

time led to an investigation to find out if there might be other differences that make this result as it is. One reason for this might be that when encryption is enabled, duplicity uses gpg to compress the files. In the case with encryption disabled, the compression job is instead given to the GzipWriteFile function in gpg.py [48], which in turn use the gzip library of python. A possible reason for the lower than expected decrease in execution time may be the use of different compression modules. It is also a possible explanation of why the backup directory size with encryption disabled is higher than with encryption.

The difference in restoration time with the different encryption options is quite large. The restoration time with encryption disabled is around half of the restoration time of the encrypted backups. To verify that this result is not due to different compression libraries and algorithms, an extra test was done with GnuPG told to disable compression by passing the command `--gpg-options "--compress-algo=none"` to Duplicity. The restoration execution time on the backup with GnuPG told to not compress was 49.66s, a slightly higher value than the 47.23 with default settings. With these settings, some extra time is used to decrypt the extra data from not compressing, some extra time is from the additional input and output operations, while some time is removed due to removing the decompression stage.

The impact of compression on Duplicity execution time is further investigated in section 4.5.

4.5 Compression's Impact on Execution Time

As described in section 2.4, the discipline of data compression tries to remove redundant data without loosing any information. By default, Duplicity compresses the backup archives. This section will run a similar test to the one in section 4.4, but with compression level set to zero. This proved to be harder than expected, as the `--no-compression` option listed at the Duplicity manual page [29] does not do what it should according to tests done as part of this project, and according to the "Duplicity-talk" mail-list [49]. At the GzipWriteFile function in gpg.py of the duplicity source code [48] the compression level is hardcoded to 6, disabling the possibility to set the compression level with encryption disabled from command line tools.

The compression level done by GnuPG with encryption enabled is possible to set through the use of `--gpg-options` when running duplicity. To make a comparison possible, the local version of duplicity was patched to enable a new `--compress-level` option from the command line. The patch, and how it was implemented, is described in Appendix C.

The following commands were ran to execute the tests in this section:

```
#Symmetric encryption. Compress level 0:
duplicity --compress-level 0 testDirectory/ file://<target path>
#Disabled encryption. Compress level 0:
duplicity --compress-level 0 --no-encryption testDirectory/
↪  file://<target path>
```

Encryption	**Backup time**	**Restoration time**	**Directory size**
Symmetric	38.78s	58.79s	1049.67 MB
Disabled	34.66s	35.77s	1047.22 MB

Table 4.4: Average execution time of duplicity with compression level 0.

The results with compression level set to 0 is shown in Table 4.4. The result provides an interesting insight into how the compression process of Duplicity affects the total execution time. Disabling compression has the potential to significantly decrease the backup execution time, and may be something to consider in situations where backup execution time is of more importance than the size of the backup. At the same time, the restoration execution time of an uncompressed backup is slower than the restoration execution time of the compressed backup. This is likely due to the added input/output operations as a result of the increased backup size.

Another interesting finding is that the difference between the two restoration times is almost the same difference as found in section 4.4. This clearly indicates that the time difference indeed stems from the decrypt stage of restoration.

The results in this test are achieved with local storage only, giving a limited view on how compression affects Duplicity used with remote storage. With remote storage providers, the network throughput adds another factor to Duplicity's performance. It is also important to note that a large part of the test data is random, without much redundancy. The next section will investigate how the additional data in need of transportation influence the backup and restoration execution time. The additional data in need of transportation will decrease, perhaps even remove, any execution time decrease achieved by disabling compression.

4.6 Backup and Restoration with Ubuntu as Storage Host

This section will investigate the backup and restoration execution time of Duplicity when backing up to a Ubuntu computer on the same local network. While the previous tests have been done with the same computer as Duplicity client and storage host, these tests introduce a new bottleneck: the network bandwidth. In section 4.5, Duplicity with different compression levels were tested. Similarly, the tests in this section will be done with the same combinations of encryption and compression settings to investigate how the execution time changes with the new storage host. The investigation will uncover if the decrease in backup time with compression disabled is removed when network bandwidth limits the performance of the backup.

The following commands were run to execute the tests in this section:

```
#Symmetric encryption enabled, default compression level:
duplicity testDirectory/ sftp://username@hostname/<target path>
#Disabled encryption, default compression level:
duplicity --no-encryption testDirectory/
↪  sftp://username@hostname/<target path>
#Symmetric encryption. Compress level 0:
duplicity --compress-level 0 testDirectory/
↪  sftp://username@hostname/<target path>
#Disabled encryption. Compress level 0:
duplicity --no-encryption --compress-level 0 testDirectory/
↪  sftp://username@hostname/<target path>
```

Encryption	**Compression**	**Backup**	**Restoration**	**Directory size**
Symmetric	default	122.04s	122.44s	761.68 MB
Disabled	default	116.55s	103.20s	758.76 MB
Symmetric	zero	131.56s	146.59s	1049.6 MB
Disabled	zero	118.78s	121.56s	1047.11 MB

Table 4.5: Average execution time of full duplicity backup to local Ubuntu desktop computer

The results of running a full Duplicity backup transmitted to an Ubuntu desktop computer using SFTP is shown in Table 4.5. This is the first tests done to a remote storage location, where network traffic impacts the execution time of Duplicity.

As expected, the execution time is increased with all the tested options compared to the previous tests where the client and host machine was the same. The difference in execution time with symmetric encryption and disabled encryption is very similar to the differences found in section 4.4: Encryption's Impact on Execution Time.

In section 4.5 the impact of disabling compression level was investigated. However, in that test, a fast Solid State Drive (SSD) acted as both the backup source disk and storage destination disk. In this test, the backup and restorations are done between two computers on the same local network. The decreased execution time of disabling compression is lost, and both backup and restoration execution times are higher with compression disabled than with compression level at the default value.

The Central Processing Unit (CPU) times in the tests with compression are still higher than in the tests with compression level set to zero, clearly indicating that the increased execution time comes from additional network traffic. This information may be used to a user's advantage depending on the system properties he/she prefers. On clients with a low network bandwidth, compression may save a lot of time, at the cost of CPU resources spent. If the user is in a situation where he prefers that Duplicity spends as little CPU resources as possible, and execution time or network bandwidth is not an issue, decreasing the compression level may be the preferred solution. The CPU time divided by total execution time in the different backup scenarios is shown in Table 4.6

Encryption	**Compression**	**CPU time**	**CPU time as %**
Symmetric	default	97.20s	0.79
Disabled	default	86.56s	0.74
Symmetric	zero	91.10s	0.69
Disabled	zero	68.47s	0.58

Table 4.6: Average CPU time of full duplicity backup to local Ubuntu desktop computer

The users preference of compression level should also take into consideration the files that will be backed up. As previously described in section 2.4, Gzip/GnuPG should be unable to do much more compression on previously compressed data. If the files the user intends to use is already in a highly compressed format, a lot of system resources are spent without actually saving any storage capacity or bandwidth.

These claims are verified by a test done with the Macbook as the client machine, and the local Ubuntu desktop computer as the storage host. The test was done with an encoded MPEG-4 movie file. The results of the test, shown in Table 4.7, shows that with this kind of encoded/compressed data, it is a lot quicker to back up with the compression level set to zero. The decreased execution time was over one minute, and Duplicity used less CPU resources measured in CPU time spent. The compression done by Duplicity saved less than 0.5 % of the storage capacity required by the uncompressed version. This test clearly shows how users may benefit of thinking about what kind of compression level they should choose when backing up with Duplicity.

Compression level	**Backup time**	**CPU time/total time**
Default (6)	295.23s	0.72
Disabled (0)	221.74s	0.64

Table 4.7: Backup execution time of H.264 encoded MPEG-4 movie file with default and disabled compression level

4.7 Backup with Asynchronous Upload

The Duplicity manual mention an experimental option that enables asynchronous upload of backup archives [29]. The CPU time in the tests done with local storage was significantly higher than in the Ubuntu tests, giving a strong indication that the increased backup execution time is a result of network limitations. In this test, full encrypted backup will be tested with the Ubuntu computer as storage host. The only difference between the two scenarios is that the `--asynchronous-upload` option is enabled.

The `--asynchronous-upload` option make Duplicity perform file uploads asynchronously in the background. By default, Duplicity finished upload of a volume before it begins to process the next volume for upload [29]. By enabling asynchronous upload, Duplicity allows two volumes to be prepared and uploaded simultaneously, with the goal to utilize both bandwidth and CPU more consistently. Use of the option should increase backup execution time, at the cost of increased required temporary storage.

This section will investigate if the experimental option provides better performance than the default setting. The hypothesis is that the backup execution time is decreased, while the CPU time divided by total time is increased. The asynchronous option has no effect on restoration, and only backup time is covered in this section.

The following commands was run in the test done in this section:

```
#Async upload enabled. Default encryption and compression:
duplicity --asynchronous-upload testDirectory/
↪  sftp://username@hostname/<target path>
```

Encryption	Compression	Backup time	CPU time/total time
Symmetric	default	83.98s	1.30

Table 4.8: Average execution time of full duplicity backup to local Ubuntu desktop computer with asynchronous upload enabled

The result of the test is shown in Table 4.8. As predicted, the backup execution time is significantly decreased compared to the similar test in section 4.6. The backup execution time is decreased by 31%, while the CPU time to total time ratio is heavily decreased. The CPU time is actually higher than the total time, a result only possible with multiprocessing, such as performing asynchronous preparation of volumes. The asynchronous upload option clearly shows very promising results as a mechanism to decrease backup execution time.

4.8 Incremental Backup's Impact on Execution Time

This section will test incremental backup with the previously used local Ubuntu desktop computer as storage host. It will perform a backup update with the data changes described in section 4.3. The test will investigate how backup and restoration time changes with incremental backups. The main hypothesis is that restoration will be slower when consulting a chain of backup archives [28].

Encryption	Compression	Time	Directory size	Dir Increase
Symmetric	default	47.40s	1016.84 MB	255.16 MB
Disabled	default	47.76s	1012.96 MB	254.2 MB
Symmetric	zero	43.22s	1377.87 MB	328.27 MB
Disabled	zero	43.09s	1374.80 MB	327.69 MB

Table 4.9: Average execution time of incremental duplicity backup to local Ubuntu desktop computer

The average execution time of incremental backup with different combinations of encryption options and compression level is shown in Table 4.9. The low backup execution time with compression level set to zero was at first a bit surprising, as it does not follow the pattern of the full backup done with the same setup. However, the low values are likely due to how the source directory has been altered. The files that have been modified, have only altered the randomly generated content. The randomly generated content has very poor compression capabilities, and thus, the tests with default compression level spent a lot of CPU resources on trying to compress incompressible content. The main advantage of incremental backup schemes is seen in the test results. While a new full backup would require 761 MB of the storage capacity, the directory increase with incremental backup was only 254 MB.

Encryption	Compression	Restoration time
Symmetric	default	160.07s
Disabled	default	144.20s
Symmetric	zero	198.23s
Disabled	zero	168.29s

Table 4.10: Average restoration time of incremental Duplicity backup from local Ubuntu desktop computer

The average restoration time of the incremental backups done is shown in Table 4.10. The restored directory is the same size as the previous full tests done, making the results comparable to the restoration times in section 4.6. In other words, the results in section 4.6 should mimic the restoration time if the incremental data set was backed up with duplicity in full backup mode. The results show a significant increase in restoration time when the backup is part of an incremental backup chain. The amount of data that needs to be downloaded from the storage host is increased, and additional computation is required to reconstruct the original data from the incremental chain.

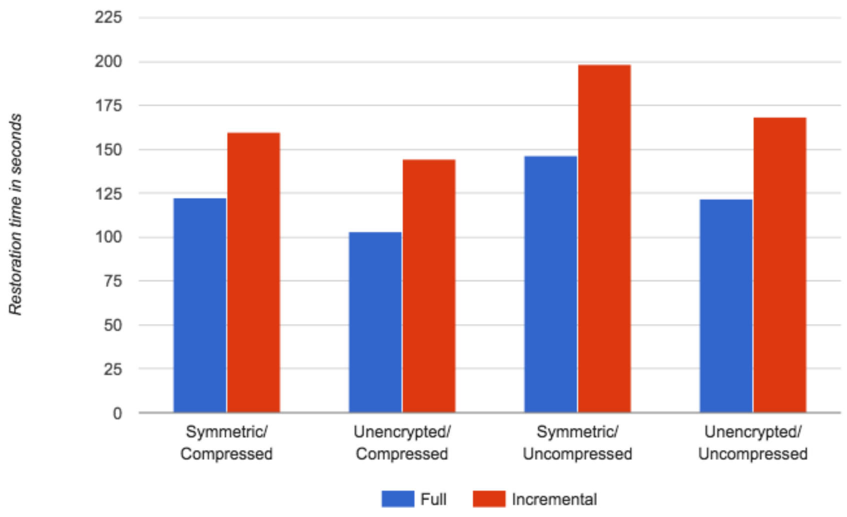


Figure 4.2: Comparison of the restoration time of full and incremental backups

The restoration times from the incremental tests in this section is compared with the results from section 4.6 in Figure 4.2. As shown, the restoration time is significantly increased for every combination of encryption and compression option.

Some additional verifications test were done to investigate if the restoration time further increased with a longer backup chain. The incremental data set was changed another time through the changes described in section 4.3, backed up as an incremental backup to the previously incremental backup completed. The test was done with default compression and symmetric encryption. The backup execution time was similar to the backup time in Table 4.9. As predicted, the restoration time is further increased due to a longer incremental backup chain. After two incremental updates, the average restoration time was 209.57 seconds, an increase of almost 50 seconds compared to the restoration time with one incremental update. The increase compared to the restoration of a full backup is 87.53s. The results clearly show how the restoration time increases with longer incremental backup chains.

4.9 Backup and Restoration with Cloud Storage Providers

Chapter 3 gave instructions on how to prepare the Duplicity client machine for use with cloud storage providers as backup hosts. This section will investigate the backup and restoration speed of four different cloud storage providers. Amazon S3, Google Drive, Dropbox and OneDrive will be tested with the test-data described in section 4.3: Setup of Test Data. Each test is run 5 times to give a broader spectrum of results to analyse.

The results from the test with a Ubuntu computer as storage host showed that disabling compression made the execution of Duplicity backup and restoration slower with the test data of this chapter. In the tests towards cloud storage providers, the throughput of data over the network is lower than between two computers on the same local network. Because of this, only encrypted backup with compression level at default is tested in this section.

Setup of tests with Amazon S3

As mentioned in section 3.6.1, duplicity requires some extra setup to be used with Amazon S3 buckets. Before execution of Duplicity the two AWS authentication keys were added as the two environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.

The following commands were used in the Amazon S3 tests:

```
Backup:
```

```
$ duplicity --s3-european-buckets --s3-use-new-style testDirectory/
 ↪  s3://s3-eu-west-1.amazonaws.com/bucketname/backup
Restoration:
$ duplicity --s3-european-buckets --s3-use-new-style
 ↪  s3://s3-eu-west-1.amazonaws.com/bucketname/backup <local path>/
```

Setup of Tests with Google Drive

Before execution of Duplicity the authors Google account password was set as the `FTP_PASSWORD` environment variable. The following command was run to backup and restoration tests with Google Drive:

```
Backup:
$ duplicity testDirectory/ gdocs://haakon@nymomatland.com/backup
Restoration:
$ duplicity gdocs://haakon@nymomatland.com/backup <local path>/
```

Setup of Tests with Dropbox

Before execution of the tests with Dropbox as storage provider, the client computer authenticated through the browser as described in section 3.6.4. It also seems that Duplicity is unable to create new directories on Dropbox, and empty test folders were created through the Dropbox web interface prior to the tests.

Because the authentication is done through URL prompting, saving the credentials in a file on the client machine, the Duplicity commands become quite simple:

```
Backup:       $ duplicity testDirectory/ dpbx:///backup
Restoration:  $ duplicity dpbx:///backup <local_path>
```

Setup of Tests with OneDrive

Similarly as when setting up Dropbox, the client computer authenticated through the browser as described in section 3.6.3. After giving Duplicity authorization to change OneDrive content, the commands to run Duplicity with OneDrive as storage host is:

```
Backup:       $ duplicity testDirectory/ onedrive:///backup
Restoration:  $ duplicity duplicity onedrive:///backup <local_path>
```

4.9.1 Backup execution time

The mean execution time of the initial full backup to the four different cloud storage providers is shown in Table 4.11. As expected, the execution time of backing up to cloud storage is increased with all four providers compared to the tests using a computer on the same local network. The standard deviation of the tests is also provided to show how inconsistent the execution time is with the different providers.

Cloud provider	**Backup time**	**Standard deviation**
Amazon S3	184.07s	35.95s
Google Drive	241.89s	8.02s
Dropbox	445.11s	101.59s
OneDrive	448.59s	65.00s

Table 4.11: Average execution time and standard deviation of duplicity backup with symmetric encryption to cloud storage providers

Amazon S3 has the lowest mean execution time, almost a minute faster than the nearest competitor. Only one of the runs with Amazon S3 was slower than any of the other test runs in this section. Even if Amazon S3 outperform the other cloud providers, it gave some inconsistent results, with a difference between the fastest and slowest execution of 103.06s.

Google Drive had this scenarios lowest standard deviation, with a 21.89s difference between its fastest and slowest execution. Even though it is slower on average than Amazon S3, it was able to outpace Amazon S3's slowest run once.

Backing up to Dropbox was very slow, spending over 7 minutes to finish on average. The execution time also varied a lot, with a 257.6s difference between the fastest and slowest run. Dropbox had the slowest run of every backup run of every run tested in this section, taking over 10 minutes to finish.

The slowest average backup execution time in this series of tests is with OneDrive as storage host. It's average performance is quite similar to that of the Dropbox runs. It has a lower deviation of results compared to Dropbox, but still performed quite a lot more inconsistent than Amazon S3 and Google Drive.

The result shows us a vast difference between the four providers. Some key points from the test results:

- Amazon S3 gave by far the quickest execution times. Only one of the runs were slower than any of the other cloud providers runs.

- Google Drive gave very consistent results. Only 20.89s differed between the fastest and slowest run.

- Both Dropbox and OneDrive gave very slow results. The mean backup execution time with these two were more than twice as high as the mean backup execution time with Amazon S3.

- The slowest backup with Dropbox was 464s, close to 8 minutes, slower than the fastest Amazon S3 backup.

The CPU time used when backing up to the different providers does not differ much. The large difference in backup time with the different cloud storage providers is likely due to different upload bandwidth granted. The difference may also partially stem from the different Duplicity back ends used, and how they implement the protocols and API used by the cloud providers for upload and download of files.

4.9.2 Restoration execution time

The mean execution time of restoration of the test directory from the four different cloud storage providers is shown in Table 4.12. As with the backup tests, the restoration time is increased compared to the scenario using a local Ubuntu desktop computer. However, the difference is lower in these tests than it was in the backup tests. The standard deviation is also provided to show the consistency of the restoration times with the different providers.

Cloud provider	Restoration time	Standard deviation
Amazon S3	148.05s	3.70s
Google Drive	162.81s	1.35s
Dropbox	207.09s	6.16s
OneDrive	327.71s	10.12s

Table 4.12: Average execution time and standard deviation of duplicity restoration symmetric encryption from cloud storage providers

Once again, Amazon S3 provides the fastest results. The results are also very consistent, with a difference of 8.00s between the fastest and slowest restoration. On average it takes 25,61 extra seconds to restore the test data from Amazon compared to restoring from the local Ubuntu desktop computer.

Google Drive provides the second best results. The standard deviation was also very low, with only a 3.17 second difference between the fastest and slowest restoration.

In general, the gap in performance between the different providers is greatly decreased when performing restoration compared to the performance difference in the backup tests. OneDrive gives the slowest results once again, spending twice the time of Amazon S3 to complete a restoration.

4.10 Guidelines for Backing up with Duplicity

Duplicity is an open source software, with limited documentation and recommendation for correct usage. This section will discuss some of the potential pitfalls users should avoid when using Duplicity for backup, and propose some simple guidelines based on the analysis of use case scenario results.

Encryption has limited impact on backup execution time

The performance gain from disabling encryption is very limited. As the drawback in execution time is minimal, users should look into other options to improve performance instead of potentially disclose information to unauthorized parties. The backup execution time with encryption disabled decreased the execution time by less than 5%. The best way to preserve the confidentiality of backups with Duplicity is through using asymmetric encryption with properly set up GnuPG keys. Keys with a large enough length make it infeasible for attackers to guess the key required to decrypt the backup archives.

Store private keys safely somewhere in addition to the backed up machine

If Duplicity is used with asymmetric encryption, users should ensure that their private key is stored safely. If the private key is only stored on the computer doing backups, problems will arise if the computer is lost or has a fatal crash leaving the key corrupted or unusable. Without the private key, the user is unable to decrypt and restore backups, leaving the backups worthless.

Choose a secure file transfer protocol

Even if the backup is encrypted, metadata about the backup is disclosed by transporting the backup in an insecure way. As discussed in section 3.5, an eavesdropper may gain a lot of information about the communication, even though the main content is encrypted.

Analyse the compression capabilities of the data set

The compression level of backups has the possibility to greatly decrease backup and restoration time if used correctly. The file type has a large impact on which compression level should be used. If the file are mainly compressed audio or video, performance may be gained without any additional storage or bandwidth cost. Compression level is also a possible mechanism to control which resource is used. On data with compression capabilities, CPU usage may be decreased at the cost of increased storage and bandwidth requirements. Similarly, if CPU usage is not an issue, an increased compression level may save bandwidth and storage costs. Users actively analysis the data set of a backup may use compression capabilities and properties to use more/less of the resources they value the most.

Limit incremental backup chains

Doing full backups every now and then is positive for both the flexibility and performance of Duplicity backups. While incremental backup is a great way to limit storage and bandwidth cost, it also increase the probability of data corruption and computation cost.

Limiting the incremental backup chain will increase the bandwidth cost due to additional data in need of transportation. A similar increase will be seen in storage cost, but may be mitigated by removing old snapshots no longer required. One alternative is to keep every full backup, but remove old incremental archives. This approach allows old snapshots to be restorable, but at a decreased time interval between snapshots.

4.11 Concluding Remarks and Future work

This chapter has investigated duplicity used with various encryption and compression settings. The use of different settings may tune Duplicity's performance to suit better for specific usage patterns.

In section 4.4 and 4.6 the impact of encryption on backup and restoration time was investigated. The tests showed a minor decrease in execution time when disabling encryption. The decrease was so low, that the author would consider it bad practice to disable encryption for performance gain.

Section 4.6 showed some of the potential of changing the compression level of Duplicity backups. Compression has the possibility to greatly impact the overall performance of Duplicity. If the data is in a highly compressed format prior to backup, decreasing the compression-level may significantly decrease the system resources spent. Analysing the content of the directory to be backed up has the potential to greatly decrease the system resources spent, at the cost of limited increase of network and storage usage.

Section 4.8 investigate the impact on restoration time when the backup consists of an incremental backup chain. The tests show a significant restoration time increase, showing one of the disadvantages of incremental backup schemes. Future work should benchmark the bandwidth spent with different incremental backup scenarios, to investigate how much extra bandwidth is spent when restoring incremental backups.

Duplicity offers a lot of different options, and this chapter only touches a few of them. Experimental options such as `--asynchronous-upload` is techniques that potentially greatly improves Duplicity's performance. The option makes Duplicity perform file uploads asynchronously in the background, giving a great potential for better utilization of resources to decrease total execution time. The tests done in section 4.7 showed a significant decrease in backup execution time with the option enabled. Future work should test how the asynchronous upload option affect the backup execution time with the different cloud storage providers.

The results from the tests with cloud storage providers, section 4.9, show the major execution time decrease that may be gained through using specific cloud providers. Amazon S3 gave significantly better results than the other tested providers. For users where execution time is of great importance, the correct choice of cloud storage will impact the total execution time a lot more than any gain through disabling encryption or compression.

The major difference in performance with different cloud storage providers is something that would benefit from further work. Additional tests could investigate where the difference stems from, if using professional accounts has any difference and how experimental options such as asynchronous upload affect cloud storage usage.

Chapter 5 Intelligent Scheduled Backup

This chapter describes a new original system for intelligent distributed backup to be used together with Duplicity. A proof of concept prototype is presented along with an in depth availability analysis of the proposed system.

The chapter begins with section 5.1, which discuss the motivation behind creating a distributed backup system, what the system should achieve, and the specifications of the system. Related work done on distributed storage systems utilizing erasure codes is presented, giving a quick overview on some similar projects and systems.

A system overview is offered in section 5.2. The overview includes a potential system architecture, clearly defining a set of components solving the different objectives and specifications identified in the system introduction.

Section 5.3 describes a proof of concept prototype implementation of the system. A potential implementation is discussed for every component of the system architecture.

An availability and reliability analysis of the proposed system is presented in section 5.4. Through a series of logical thinking and statistical calculations, the system's key properties are examined.

5.1 System Introduction

The goal of the intelligent distributed backup system is to increase availability and reliability, while preserving Duplicity's key features. Increased availability and reliability often come at the cost of increased resource overhead, such as computing power and storage capacity required. Consequently, the system should strive to minimize the overhead while accomplishing the desired availability increase. The required properties are summed up:

1. Increased availability compared to a backup at a single storage provider.

2. Decreased probability of restoration failure compared to a backup at a single storage provider.

3. Decreased probability of data corruption compared to a backup at a single storage provider.

4. Decrease the overhead of having several copies of the same data.

Utilizing several different storage providers to host the backup data will decrease the backups dependence on a specific storage provider being available to allow successful restoration of data. The same line of thought is used for the probability of restoration failure and data corruption. Allowing parts of the backup data to be missing or unusable without disabling restoration is the key objective of the solution. The challenge of this system is how to achieve the first three properties, while decreasing the overhead caused by redundant data.

This chapter will investigate if the availability and reliability of backups may be increased through the use of erasure code encoding and distribution of partial data blocks. As discussed in section 2.3, erasure codes are a possible mechanism to increase the reliability of electronic communication. The key property of erasure codes is the ability to reconstruct the original data even if some of the partial data blocks are missing. This property will be used as the cornerstone of the proposed system. Figure 5.1 shows the idea of encoding every file in a directory into partial data files, followed by distribution of corresponding partial files to different storage hosts, allowing restoration even though a subset of the storage providers are unavailable.

Readers are referred to chapter 2 and chapter 3 for background information on Duplicity and erasure codes.

5.1.1 Availability

The primary goal of the system is to have a higher availability than a regular duplicity backup to a single storage host. The obvious, and simplest, solution to this problem is to store the backup at several different storage providers, creating redundancy through several exact copies of the data. In that kind of solution, restoration is possible as long as one storage provider is available. The disadvantage of such a backup scheme is that the storage and network traffic required for backup is multiplied by the number of copies stored.

In the proposed system, the data is distributed in a way requiring more than one storage host available to successfully restore the backed up data, making the thought process about availability a bit different. Availability in this system can be viewed in

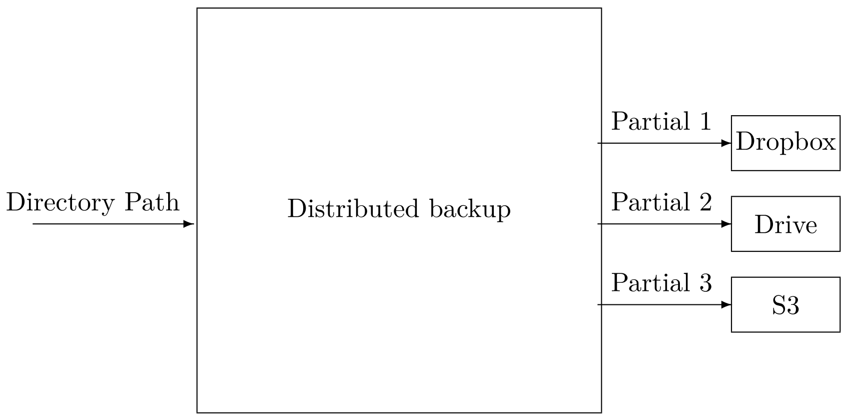


Figure 5.1: The system distribute erasure coded partial blocks to different storage host.

two different scenarios. The scenario where a user wishes to perform a backup, and the scenario where a user wished to restore backed up files.

The availability in the restoration scenario is calculated by finding the probability that enough storage providers are available to reconstruct the data back to its original state. In the proposed system, where it is not sufficient that one provider is available, the availability will be determined by how the data blocks are distributed, and the number of data blocks required to decode the data. It is also important to note that if a storage provider was unavailable at the time of the backup, it will affect the restoration availability greatly. Some scenarios will require every uploaded block to be available for download to enable successful restoration and reconstruction of the original data. Other scenarios will disable the possibility to reconstruct completely, as the number of uploaded partials may be lower than the number of partials required for reconstruction.

In the scenario of a user performing backup, things quickly become even more complicated. If the backup scheme is simply storing a full data copy to each storage provider, it is not an issue if one of many providers is unavailable. However, in a setup distributing the data between different storage providers, it may cause several problems if only parts of the distribution succeed. As already mention, it may temporarily leave the system in a vulnerable state, where each of the uploaded blocks holds greater importance than intended.

Availability of the proposed system is analysed in depth in section 5.4.

5.1.2 Restoration failure and data corruption

As with all electronic communication and storage, there is a chance that the data stored is corrupted, leaving it in a unrepairable state. As discussed in section 2.3, erasure codes enable reconstruction even if only a subset of the encoded blocks are required to decode back to the original state. Because of this property, the consequences of data corruption is reduced. If one block is corrupted, reconstruction may still be possible. At the same time, similar concerns as with availability arise: what happens to the likelihood of restoration failure due to data corruption if one or more storage providers are unavailable? If only the number of blocks required to decode the erasure code is available, and one of the blocks becomes corrupted, restoration is not possible. On the other hand, if all blocks are available, restoration is possible even if some of the data is corrupted.

To ensure reconstruction of the original data is not done with corrupted partial blocks, the system need to be able to discover if data has become corrupted. If reconstruction is done with incorrect partial files, the reconstructed file will contain errors, and the restoration has failed. The system is required to have mechanisms to detect if data reconstruction was successful.

5.1.3 Related work

The topic of distributed backup has seen a large increase in popularity as the general public has gained faster and better internet connection. Some distributed backup systems use a peer-to-peer model, where very user are storage hosts for other users [50, 51]. The system presented in this report is not a peer-to-peer system, but still share several properties and techniques with the peer-to-peer systems, such as erasure coding. In “iDIBS: an improved distributed backup system”, data is encoded with erasure codes and distributed to available peers. Through the use of erasure code, restoration is possible even if some of the peers are unavailable or offline, increasing the reliability of the backup solution [51].

Slamanig and Hanser provides an overview of different strategies to realize reliable distributed storage systems in “On Cloud Storage and the Cloud of Cloud Approach” [52]. The paper discuss ten different system, and their approaches to realize distributed cloud storage. Of the ten, RACS in particular is very related to the system proposed in this project [53]. RACS is an abbreviation for Redundant Array of Cloud Storage, and is a cloud storage proxy that stripes data across multiple cloud storage providers. RACS use erasure coding to create redundancy, and enabling toleration of unavailable storage providers without losing data.

The open source system Tahoe-LAFS provides distributed storage with confidentiality and integrity capabilities [54, 55]. Similar to the system presented in this report, Tahoe-LAFS uses a library named zfec for erasure coding [56]. Tahoe-LAFS uses a gateway to perform all cryptographic functionality such as encryption and integrity check [52].

This project report presents a system enabling distributed storage of backup with a simple client application. It is storage independent, in the sense that Duplicity uses standard file transfer protocols to enable remote storage of backups. It does not require any additional server software, and strives to keep the application simple without any additional communication overhead in addition to what is provided by Duplicity.

5.2 System Architecture

This section will describe the planned architecture of the distributed backup system. The architecture should address, and solve, the properties introduced previously. This section will describe how different components of the architecture solve different key tasks. The different components are responsible for distinctive and independent tasks that together enable distributed backup with the use of erasure codes and Duplicity.

The key tasks of the system are identified as:

1. Identify all files in need of encoding/decoding
2. Encoding/decoding of identified files
3. Distribution/collection of encoded partial files

The system should, as with regular use of Duplicity, back up a directory and its content. The first key task is solved by identifying files and directory structure through directory traversal. In the backup scenario, it will identify files, and prepare them for encoding with erasure codes to create easily distributable partial files. In the restoration scenario partial files need to be identified, and matched with other partial files that stem from the same original file, to permit reconstruction of the original file.

The third key task is distribution/collection of encoded partial files. In the backup scenario, the encoded partial files are distributed to different storage hosts through several instances of Duplicity. In the restoration scenario, Duplicity is used to restore partial files from enough storage hosts to enable reconstruction of the original data.

The system will strive to keep a modular design. A modular design allows each software component to be easily defined, as well as providing the possibility to use the different components individually. For example, the component responsible for encoding should be usable without encrypting, compressing and distributing the partial files. A well designed modular system makes it easier to change certain parts of the software, without it having any impact on the other components. One example could be to provide different erasure code schemes, allowing the user to use the scheme of his/hers preference.

If the system architecture is broken down to small modules, each responsible for a special task, testing and debugging becomes easier and more well defined. The main idea is to decouple parts of the software that are likely to change, so that they can be changed independently. The idea was introduced as “information hiding” by Parnas in “On the Criteria To Be Used in Decomposing Systems into Modules” [57]. In “The structure and value of modularity in software design” the information hiding design is predicted by Sullivan et al. to be superior to modularization that are based on sequencing steps based on input to the output [58].

The system architecture of the proposed system is quite simple, and thus makes it hard to differentiate between the parts that are most likely to change, and the sequence of steps required by the software. The general plan is that it should be possible to exchange the encoder/decoder or the distributor/collector without changing other parts of the software.

The next subsections will describe the different core modules of the system, the tasks they solve, and how they solve them.

5.2.1 Erasure code modules

The erasure code modules are listed together, even though they are two separate modules. The two components have to correspond, as the decoder has to be able to decode the output of the encoder. The decoder should decode according to the encoder in use.

The erasure code modules show one of the positive effects of using a modular design, as it is possible to change coding technique without it making any impact on the rest of the system. If several different coding modules are implemented, the modular design allows the user to select the coding technique of his preference.

The encoder and decoder description below is if the system utilizes a MDS code such as Reed-Solomon. As previously discussed in chapter 2, non-optimal erasure codes may require more than k partial files to enable successful reconstruction.

The encoder and decoder is largely influenced by Luigi Rizzo's explanation of erasure codes in "Effective Erasure Codes for Reliable Computer Communication Protocols" [14]. In this system, partial files correspond to the blocks of encoded data, and the blocks of source data are all gathered from one file.

Erasure code encoder

The erasure code encoder component is responsible for encoding the files it receives as input. As shown in Figure 5.2, the encoder takes a file as input, and encodes it into $k + r = n$ partial files, where k partial files are required to enable restoration of the original file through the use of a decoder.

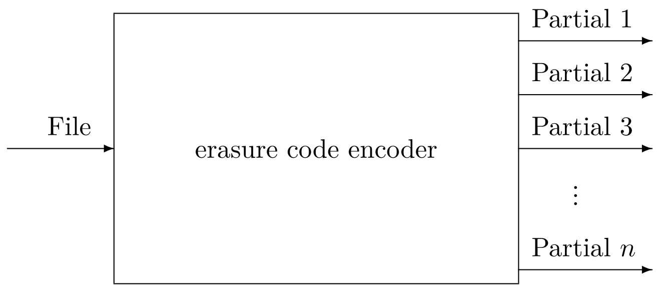


Figure 5.2: The erasure code encoder module takes a file as input, and outputs $k + r = n$ partial files

Erasure code decoder

The erasure code decoder component is responsible for decoding the partial file set it receives as input. As shown in Figure 5.3, The decoder takes at least k partial files as input, and outputs the decoded file. In other words, the decoder is able to reconstruct with up to r missing partial files.

5.2.2 Traversal component

Duplicity backs up entire directories, while the erasure code encoder/decoder takes files as input. The traversal component is needed to identify and feed the directory files to the encoder/decoder. The traversal component has slightly different tasks in the backup scenario and the restoration scenario:

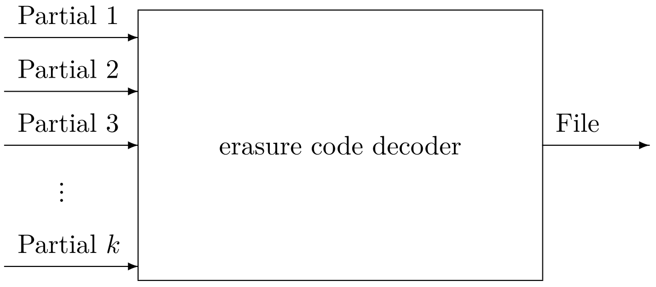


Figure 5.3: The erasure code decoder module takes k to n partial files as input, and outputs to original file

Encode phase

In the encode phase, the traversal component is responsible to find every file in the directory tree, and feed them to the encoder. When the traversal is complete, the component moves the partial files to a file structure that allows efficient use together with duplicity. The file structure contains n directories. Each directory has the same directory structure as the source directory structure, but includes one partial block of each file, instead of the actual file.

Decode phase

In the decode phase, the traversal component is responsible to find every partial file in the restored directory, including any sub directories. After the traversal is complete, the traversal component has mapped every partial file that belong together, enabling decoding through the decoder module.

5.2.3 Duplicity backup component

The Duplicity backup component is responsible for invoking one Duplicity command for each storage host. The command should be invoked with options according to the settings set in the configuration file, including any environment variable required to authenticate with the storage hosts, or to encrypt the backup archives.

5.2.4 Duplicity restoration component

The Duplicity restoration component is similarly to the backup component responsible for invoking Duplicity commands according to the configuration file. It should set up any environment variables required for authentication or decrypting of files.

The restoration component also has some extra responsibility that the backup component does not have. The component should only restore data parts from storage hosts that hold the same, most recent version of the original source data. If the host was unavailable when he last backup procedures were run, the data it holds is outdated, and should not be used for decoding as it will create incorrect files.

Ideally, it should only restore enough parts to enable reconstruction of the original data. Bandwidth may be saved by limiting the number of parts that are restored. If a systematic erasure code is used, i.e. the original data is reproduced among the encoded data, the parts that contain reproduced data should be prioritized to reduce reconstruction cost.

5.2.5 Argument parser and configuration file

Different user provider parameters are required to set up the system properly. Direct user input is received through the argument parser. In addition to the arguments, a configuration file is used to define how the different components work. As the setup of several different Duplicity storage host quickly become quite complex, the system uses a configuration file to hold most of the information.

5.2.6 Detection of corrupted backups

Data that is corrupted at the storage host, or during transfer to the restoration client may cause the decoder to fail at successfully reconstruct the original data. Thankfully, Duplicity will automatically detect if the content of the backup archive is changed or corrupted.

Duplicity utilizes SHA-1 to calculate a hash for every backup archive it creates. The hash is stored in the Duplicity manifest file, which is also uploaded to the storage host. When Duplicity restores a backup, it will compare the hash of the downloaded archives with the hash stored in the manifest file. If the two hashes mismatch, the restoration is aborted. The shell output from a restoration that was aborted because of a mismatch is shown in Figure 5.4

```
Invalid data - SHA1 hash mismatch for file:
duplicity-full.20150324T100427Z.vol1.difftar.gpg
Calculated hash: 85d1bb65be20e253f3997d03240554231361180f
Manifest hash: 0868122b23d613b5b089a7ca0623891f47b018ca
```

Figure 5.4: Shell output from Duplicity restoration with altered/corrupted data.

Because Duplicity aborts restorations with wrong checksums, any successfully restored partial file is assumed to be without any data corruption.

5.3 Implementation of Prototype

After the initial draft of system architecture, implementation of a prototype started. The prototype is written in the Python programming language. The prototype was implemented according the specification and system architecture described in section 5.2. The implementation was done with focus on modularization to allow different FEC/erasure code schemes to be utilized.

Git was used for revision control of the implementation. Git is an open source distributed version control system that gives the user a great revision history of the code changes between commits [59]. The source code of the prototype may be viewed or downloaded from the authors web-based Git repository at the hosting site GitHub [60].

The system architecture/overview previously described allowed each component to be developed and tested separately. In addition to the previous architecture planning, the flow of different key tasks was sketched to envision how the flow of the code should look like.

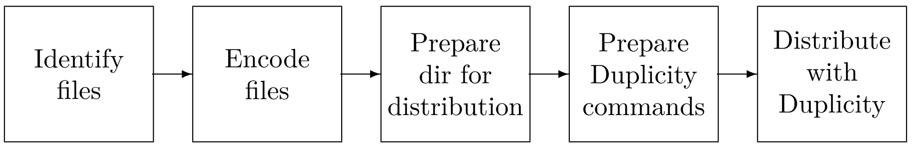

Identify files
Encode files
Prepare dir for distribution
Prepare Duplicity commands
Distribute with Duplicity

Figure 5.5: Flow of key tasks when performing distributed backup

Figure 5.5 show the flow of key tasks when performing a distributed backup. The first key task, further discussed in section 5.3.1, is to identify the files that should be encoded. The next step, described in section 5.3.2, is to encode the files with erasure codes to enable distribution of partial files. The last three steps consist of preparing the files for distribution, creating the duplicity commands required for distribution and finally execute the commands that uploads backup archives with duplicity.

As shown in Figure 5.6 flow of the restoration scenario looks a little bit different than the backup scenario. In this scenario, preparing and running duplicity are the first tasks to be run. The program need to set up duplicity commands correctly according the configuration file, and restore the distributed archives containing the

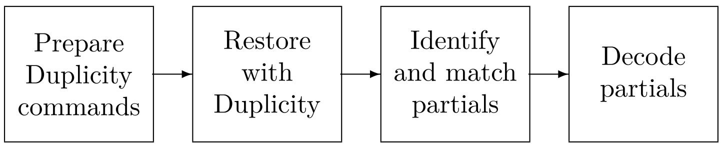


Figure 5.6: Flow of key tasks when performing restoration of distributed backup

partial files. After Duplicity has done its part of the job, partial files need to be found and matched to enable decoding to the original source files.

5.3.1 Directory traversal

A key part of the distribution system is to identify files, and prepare them for encoding through the erasure code modules. Directory traversal is solved through the use of Python's os.walk function. The Python code shown in Listing 5.1 does a top down directory walk starting at the path parameter. Every file path found in the walk is appended to the result list, and when the directory walk is complete, the list is returned to the caller of the function.

Listing 5.1 Top down directory walk that returns a list of the paths to every file encountered.

```python
def traverse_and_identify_files(path):
  file_paths = []
  for root, dirs, files in os.walk(path):
    for file_name in files:
      file_paths.append(os.path.join(root, file_name))
  return file_paths
```

Python's `yield` statement could also be used to return the paths. The function would then work as a generator, but the list version was preferred as it allows easier multiprocessing usage in other parts of the code. The use of a generator could save memory spent on the client computer, but the memory used is insignificant compared to the gain in execution speed gained by enabling multiprocessing of encoding/decoding later.

The result from the directory walk is further used together with a pool of workers from the multiprocessing library of Python. This allows several files to be encoded at the same time, using different processes/threads on the computer. Early tests showed a significant reduction in execution time of encoding directories through the use of multiprocessing.

Identification of partial files is done in a similar manner. However, some extra processing is required, as the partial files that belong together need to be mapped before being sent to the decoder. Partial file mapping is done through analysing the file paths. Additional file extensions are appended to the file name to allow easier recognition and matching.

5.3.2 Erasure codes

The modular design allows easy replacement of single modules. The prototype is implemented with two different erasure codes, one provided by a Reed-Solomon based library and one self-implemented XOR-based coding scheme provided by this project's supervisor Danilo Gligoroski.

Zfec library

The Reed-Solomon based encoding and decoding used in the distributed backup prototype is provided by the software library zfec [56]. It is licenced under GNU General Public License version 2 and largely based on Luigi Rizzo's FEC library [13, 56]. The software library can be used directly from the command line, or through APIs in Python, C or Haskell.

Zfec is based on Reed-Solomon coding with Vandermonde matrices when $w = 8$. The implementation of RS coding is heavily tuned, and utilizes pre-computed multiplication and division tables for $GF(2^8)$ [22]. Reed-Solomon coding has previously been described in section 2.3.1.

The python API solves the tasks of the erasure code modules in the system. In *filefec.py* the two functions `encode_to_files` and `decode_from_files` are defined, solving the tasks of the erasure code encoder and erasure code decoder described in section 5.2.1.

Zfec is able to encode with a large variation of user provided k and n values, where k is number of blocks number of blocks required to reconstruct the original data, and n is the total number of blocks. This allows easy encoding of partial files into the format required by the distribution components of the prototype.

XOR encoder and decoder

The system prototype was also tested with a systematic $(10, 6)$ XOR scheme provided by Danilo Gligoroski, this projects supervisor. The equations of the scheme is showed in Table 5.1. As seen from the set of calculations, the first 6 output parts contain the 6 input parts. The last four output parts contain redundant information created by performing XOR operation on the input data parts.

$$\begin{aligned}
y[1] &= x[1]\\
y[2] &= x[2]\\
y[3] &= x[3]\\
y[4] &= x[4]\\
y[5] &= x[5]\\
y[6] &= x[6]\\
y[7] &= x[3] \oplus x[5] \oplus x[6]\\
y[8] &= x[2] \oplus x[4] \oplus x[6]\\
y[9] &= x[1] \oplus x[4] \oplus x[5]\\
y[10] &= x[1] \oplus x[2] \oplus x[3]
\end{aligned}$$

Table 5.1: Encoding equations of a systematic $(10, 6)$ XOR scheme, i.e. with 6 data parts and 4 redundant parts.

If any of the data parts are lost, they may be reconstructed through the use of the redundant data blocks. For example if y[3] is lost, it may be reconstructed through the redundancy data in $y[7]$ or $y[10]$. Because $y[7] = x[3] \oplus x[5] \oplus x[6]$, it is known that $x[3] = y[7] \oplus x[5] \oplus x[6]$. To allow easier implementation of the erasure code scheme, the different ways of calculating the 6 data blocks were identified. The different possible ways to calculate the data blocks is shown in Table 5.2.

$$\begin{aligned}
y[1] &= y[2] \oplus y[3] \oplus y[10] && = y[4] \oplus y[5] \oplus y[9]\\
y[2] &= y[1] \oplus y[3] \oplus y[10] && = y[4] \oplus y[6] \oplus y[8]\\
y[3] &= y[5] \oplus y[6] \oplus y[7] && = y[1] \oplus y[2] \oplus y[10]\\
y[4] &= y[2] \oplus y[6] \oplus y[8] && = y[1] \oplus y[5] \oplus y[9]\\
y[5] &= y[3] \oplus y[6] \oplus y[7] && = y[1] \oplus y[4] \oplus y[9]\\
y[6] &= y[3] \oplus y[5] \oplus y[7] && = y[2] \oplus y[4] \oplus y[8]\\
y[7] &= y[3] \oplus y[5] \oplus y[6]\\
y[8] &= y[2] \oplus y[4] \oplus y[6]\\
y[9] &= y[1] \oplus y[4] \oplus y[5]\\
y[10] &= y[1] \oplus y[2] \oplus y[3]
\end{aligned}$$

Table 5.2: Decoding equations of a systematic $(10, 6)$ XOR scheme

One of the primary goals of non-optimal XOR erasure code schemes is to provide fast encoding and decoding, as the only computations required are XOR operations [17].

The XOR erasure code scheme was implemented to allow encoding and reconstruction of files, to fit with the distributed backup system.

The implementation of the encoder is quite simple. The six data parts are created by splitting the original file into six equally sized parts. The six data parts are subsequently used to encode the redundancy blocks. The encode function, shown in Listing 5.2, takes a list containing data parts, and calculates the four redundant parts. The `strxor` function of Python's *crypto* library is used to perform the actual XOR operations. Several different XOR techniques for python was tested, but the function provided by the Crypto library outperformed every other tested approach.

Listing 5.2 The encode function of the XOR scheme implementation takes a list containing 6 elements, and returns a list with the 4 calculated redundant parts.

```python
def sxor(s1, s2):
  if len(s1) > len(s2):
    s2 = s2.ljust(len(s1),' ')
  elif len(s2) > len(s1):
    s1 = s1.ljust(len(s2),' ')
  return Crypto.Util.strxor.strxor(s1,s2)

def encode(data):
  y7 = sxor(sxor(data[2],data[4]),data[5])
  y8 = sxor(sxor(data[1],data[3]),data[5])
  y9 = sxor(sxor(data[0],data[3]),data[4])
  y10= sxor(sxor(data[0],data[1]),data[2])
  return [y7,y8,y9,y10]
```

Every detail of the XOR encoder implementation may be viewed in Listing D.2 in Appendix D, or in the `xor_encode.py` file at the projects GitHub repository [60].

Additional logic is required in the decoder implementation due to the need to identify missing parts, and compute how to reconstruct them. Because the coding scheme is systematic, the decoder needs to ensure that the six data parts are available to enable reconstruction of the original file. In this implementation, the data parts are the original file divided into six pieces, and simple concatenation of the data parts is enough to reconstruct as soon as the data parts are available.

An algorithm was created to identify and compute missing data parts. A simplified version of the algorithm is shown in Algorithm 5.1. The algorithm continues as long as a data part is missing, or it is unable to reconstruct any missing data or redundancy part. When the algorithm is complete, it is certain that either every data part is available or reconstruction of the original file is impossible.

Algorithm 5.1 Simplified algorithm that tries to reconstruct encoded parts until every data part is available.

```
function DECODE IF POSSIBLE
    while data part is missing do
        for every missing part do
            if reconstruction of missing part is possible then
                reconstruct missing part
            end if
        end for
        if no part was reconstructed in this iteration then
            return False
        end if
    end while
    return True
end function
```

The equations listed previously in Table 5.2 is used to identify if reconstruction of a partial file is possible. If every other partial file in the equation is available, it restores the partial file, which subsequently may be used to reconstruct other missing parts. The full python implementation of the XOR decoder is shown in Listing D.3 and Listing D.4 in Appendix D.

Further performance increase is possible through the use precomputed tables of recoverable combinations. The precomputed possible combinations are used as a lookup table, quickly determining if further resources should be spent on reconstruction. The lookup table only requires 95 entries to contain every combination of missing parts where reconstruction is impossible. Utilization of a lookup table may potentially significantly decrease the execution time when reconstruction is impossible. At the same time, it is for some unrecoverable situations possible to reconstruct some data. A partially constructed file may in some cases be valuable. Consequently, users should be able to choose if they want to abort decoding if insufficient data is available, or if the decoder should reconstruct what it can.

Pre-calculation of reconstruction sequence is another possible mechanism to achieve decreased computation cost. The 6 data parts are required to successfully reconstruct, and thus further improvements to the implementation are gained if the decoder already know the ideal sequence of equation solving. The scheme is deterministic, and it is possible to identify the best approach for every possible scenario.

The XOR erasure code implemented always outputs 10 partial files, and is only suited for use with the distributed backup prototype with 10 hosts. The erasure code scheme is not optimal, and does not promise reconstruction with $n - k = 6$ data parts available. An in-depth availability and performance analysis is performed in section 5.4.

Encoder and decoder class

The encoder and decoder class is implemented to allow both the zfec library, and the XOR-based erasure code implementation. The code examples in this section are shown with the functions from zfec's *filefec.py*. Only minor changes are required change the erasure code scheme to the self-written XOR implementation.

Listing 5.3 contain the wrapper function that prepares the variables requried for zfec's `encode_to_files` function. The code shows how the path the partial files should be stored at is calculated through the a series of `os.path` operations. The variables not calculated are self values set at initiation of the encoder class instance. The code also opens the source file for reading, as zfec's function takes an open file as parameter. The only change required to encode with the XOR scheme is to comment the filefec line (line 8), and uncomment the xor line (line 9). If the XOR encoder is used, the different k and n values are not used, as the encoder always encodes into 10 partial files.

Listing 5.3 Encode function that prepares the variables required for the file encoding functions

```
1 def encode_file(self,file_path):
2   prefix = os.path.basename(file_path)
3   relative_path = os.path.dirname(os.path.relpath(file_path,
     ↪  self.root_source_path))
4   output_path = os.path.join(self.target_root_path, relative_path)
5   utility.make_sure_directory_exists(output_path)
6   filesize = os.stat(file_path).st_size
7   with open(file_path, 'rb') as f:
8     filefec.encode_to_files(f, filesize, output_path, prefix, self.k,
       ↪  self.n, self.suffix, False, False)
9     #xor.encode_to_files(f, filesize, output_path)
```

The function parameters are set up as simplistic as possible to work well with the multiprocessing Pool class mentioned in the section about directory traversal (section 5.3.1). Listing 5.4 shows how the `encode_directory` function utilize the Pool class. The code gets all file paths through the use of the function showed in Listing 5.1, and proceeds by feeding that list into the Pool.map function. The Pool.map function takes two parameters, the function it should execute and a list of arguments for that function to run. In the prototype implementation, the `unwrap_self_f` function is called, which is a unwrap shell to allow class instance functions work with the Pool class.

Listing 5.4 Example of Pool class enabling encoding of several files in parallel

```python
from multiprocessing import Pool
def encode_directory(self):
  utility.make_sure_directory_exists(self.target_root_path)
  files = self.traverse_and_identify_files()
  p = Pool(5)
  p.map(unwrap_self_f, zip([self]*len(files), files))
```

The same general idea is used when decoding files. Some extra processing is required prior to using multiprocessing as the different partial files needs to be mapped to the other matching partial files. Because most modern computers has several cores in the CPU, this approach greatly decreases the execution time required to encode/decode a directory.

Discrepancy with file permissions between Duplicity and erasure code implementations

Throughout the system implementation it was discover an important lack of a specific feature in zfec. File permissions are not preserved when files are encoded and decoded. Consequently, the file permissions of a reconstructed file may not be the same as the original file had. Similarly, the same flaw is found int he XOR scheme implemented.

File permission preservation is one of Duplicity's key features. Consequently, Duplicity's file permission preservation feature is removed through encoding of files. Users on systems with multiple user accounts may run into problems if the restored files does not share the same permissions as the original files had.

In the current prototype, file permissions are not preserved.

5.3.3 Execution of duplicity instances

Initially, the idea was to use Duplicity through its Python code. However, the Duplicity source code is quite complex, with dozens of options/arguments that undergo parsing and verification. To enable easier implementation of this prototype, Duplicity is executed as a subprocess with the built in Python function `call`.

Python's `call` function allow the software to run Duplicity as it would in shell. While it is a very simple solution, the solution has drawbacks, such as poor error handling, that would have to be addressed with further development.

Storage host configuration is done in `duplicity_conf.py`. In this prototype, users should set up n hosts, i.e. equal number of hosts as encoded partial parts. To set up a host, the following variables need to be added:

`remote_host` Protocol+address, e.g. “sftp://matland@192.168.0.1”.

`remote_host_options` Duplicity options the user want to pass, e.g. “–s3-european-buckets”.

`remote_path` The path to where the data should be stored at storage host, e.g. “backup/”.

Users may also set environment variables such as `PASSPHRASE`(used as symmetric encryption key by GnuPG through Duplicity), `FTP_PASSPHRASE` (used to authenticate with for example Google Drive) and AWS authentication keys.

The logic for setting up hosts for backup and executing the commands is in the `Duplicity_distributer` class. The class is responsible for generating the duplicity commands based on the configuration file, setting up any environment variables end execute the duplicity commands. Similarly, the class `Duplicity_restorer` is responsible for setting up hosts for restoration and executing the commands. Some of the source code is found in Appendix D, but for the full source, readers are referred to the project’s GitHub repository [60], or in Appendix D.

```
Local and Remote metadata are synchronized, no sync needed.
Last full backup date: Tue Mar 24 14:31:32 2015
Collection Status
-----------------
Connecting with backend: BackendWrapper
Archive dir: /Users/Matland/.cache/duplicity/3c0b0a7c636e27ac4e2876e6267ef4b9

Found 0 secondary backup chains.

Found primary backup chain with matching signature chain:
-------------------------
Chain start time: Tue Mar 24 14:31:32 2015
Chain end time: Mon May 18 14:56:00 2015
Number of contained backup sets: 2
Total number of contained volumes: 53
 Type of backup set:         Time:                      Num volumes:
        Full                Tue Mar 24 14:31:32 2015   40
        Incremental         Mon May 18 14:56:00 2015   13
-------------------------
No orphaned or incomplete backup sets found.
```

Figure 5.7: Example output from the collection-status option of Duplicity

Duplicity's `collection-status` action is used to only allow restoration from hosts that has the latest version of the backup. The `Duplicity_restorer` class will invoke collection status commands, retrieving metadata about the backup. Example output from the collection-status action is shown in Figure 5.7. Particularly the "Chain end time" is of interest, and is used to verify that the different hosts have the same version of the original data stored. Comparisons of the hosts chain end time equip the class with means to verify that the time difference between the different backup archives is acceptable. Only the host with a timestamp close enough the most recent timestamp will be restored, and ready for decoding.

5.3.4 Limitations of the prototype

While the goal of the prototype is to prove some of the potential of the system, it has some clear limitations and problems that should be addressed in further development.

Additional data corruption detection is not included in the prototype. Necessarily, it is assumed that partial files successfully restored by Duplicity is correct. Further investigation and testing should be done to verify that this presumption is correct.

Additionally, the prototype is not able to detect erasure code errors. One idea could be to utilize a hash function on every file prior to encoding, and store it as metadata with the backup. This would allow the system to verify if the decoded file's hash match the hash that was computed before encoding.

As mentioned in section 5.3.2, the prototype does not preserve file permissions. Restored files may have different file permissions compared to the original files, causing potential problems. Future improvements should include metadata in some way storing details about the files permissions, enabling the system to not only restore the data of a file, but also the file's permissions.

Future potential improvements should include improved scheduling of restoration. Ideally, the system should only restore from the number of hosts required for reconstruction. If systematic erasure codes are used, the verbatim data partials should be prioritized, as it decreases reconstruction computation cost. If the XOR scheme is used, different recoverable combinations may have different bandwidth requirements. The system should have an option allowing users to minimize the bandwidth cost of restoration, even if it comes at the cost of increased computation required. In the current implementation, every available part is restored, regardless of how many parts are required for reconstruction.

5.4 Analysis of the Distributed Backup System

The first part of this section is an availability analysis of the system when the Reed-Solomon based zfec library is used. The second part performs an availability analysis of the system with the less configurable, non-optimal XOR scheme presented in section 5.3.2.

Two different erasure codes schemes are tested with the system. Because one of the key properties of sub-optimal erasure codes usually is decreased computation cost, a performance evaluation of the two codes is conducted. The performance evaluation tests zfec and the XOR scheme implementation on different files, and reports on the execution time of encoding and decoding.

The storage overhead of the system quickly rounds up the section, providing insight into how different erasure code parameters affect the total backup storage cost.

5.4.1 Availability with optimal erasure codes

The total number of data blocks is described with n, and the number of redundant data blocks is r. With optimal erasure codes, such as Reed-Solomon codes, $k = n - r$ data blocks are required for reconstruction. x is used as the symbol for number of hosts available at time of backup, and y is used as the symbol for number of hosts available at time of restoration.

Availability of restoration for a distributed backup is quite complex to calculate, as it depends on the availability of the different storage providers used, in addition to the chosen values k and n. In this prototype the number of storage hosts equal the chosen n value, making it easier to analyse the availability compared to cases where some hosts might store more than one partial file. This section will analyse the availability in one specific setup, in addition to showing how calculations may be done in a more mathematical way for any (n, k) MDS code.

The calculations and thought processes in this section are done with one important assumption: the availability of a host at the time of restoration is independent of the availability of the host at time of backup. The assumption is fair because it is presumed that a backup set is not restored very shortly after the backup was executed.

In a situation with $k = 2$, $n = 3$ and 3 hosts (H_1, H_2 and H_3), a minimum of two hosts are required to be available to enable decoding of the partial files. Each host has stored one partial file of every original file. Consequently, it exists 8 different possible states, or combinations, of available and unavailable hosts.

State	H_1	H_2	H_3	Probability
s_1	0	0	0	0.01^3
s_2	0	0	1	$0.01^2 * 0.99$
s_3	0	1	0	$0.01^2 * 0.99$
s_4	0	1	1	$0.01 * 0.99^2$
s_5	1	0	0	$0.01^2 * 0.99$
s_6	1	0	1	$0.01 * 0.99^2$
s_7	1	1	0	$0.01 * 0.99^2$
s_8	1	1	1	0.99^3

Table 5.3: Probability of being in the different states with 3 hosts, each host has availability of 0.99. A state is a unique combination of available and unavailable hosts.

Table 5.3 show the probability of being in the 8 different possible states, assuming every host has an availability of 0.99. Four states exist where restoration is possible (s_1, s_6, s_7 and s_8). The probability of being in one of these four states is $3 * (0.01 * 0.99^2) + 0.99^3 = 0.999702$, a significant higher availability compared to using regular duplicity with one host providing 99% availability. However, this result is only valid if it is certain every part was successfully backed up to the corresponding storage host.

Consider the following example:

- At the time of backup, H_1 is unavailable (state s_4).
- Accordingly, H_2 and H_3 must be available for restoration to be possible (state s_4 and s_8).

This example shows some of the complexity that comes with a distributed backup solution. With the scenario above, only state 4 and state 8 enables restoration. The probability of being in state 4 or state 8 at time of restoration is $0.01 * 0.99^2 + 0.99^3 = 0.9801$. Obviously, the probability of being in state 4 at backup and not state 4 or state 8 at restoration is quite low, but shows the complexity a distributed backup solution has when it comes to considering availability.

The scenario with 3 hosts, and a $(3, 2)$ erasure code scheme is analysed further. The probability/availability of different scenario/availability pairs is quite easily calculated, especially in this example where every host has the same availability. Table 5.4 shows the four different scenarios at time of restoration. The second column shows the probability of being in that scenario (which depends on available hosts at

Scenario	Probability of scenario	Availability of restoration
$Sc_1 : \frac{3}{3}$ backed up	$P(s_8) = 0.970299$	$s_4 + s_6 + s_7 + s_8 = 0.999702$
$Sc_2 : \frac{2}{3}$ backed up	$P(s_4 \cup s_6 \cup s_7) = 0.029403$	$s_8 + s_{backup} = 0.9801$
$Sc_3 : \frac{1}{3}$ backed up	$P(s_2 \cup s_3 \cup s_5) = 0.000297$	0
$Sc_4 : \frac{0}{3}$ backed up	$P(s_1) = 0.000001$	0

Table 5.4: Different scenarios with 3 hosts and a $(3, 2)$ erasure code scheme. Each host has an availability of 0.99. s_i notates the states given in Table 5.3

time of backup). The third column shows the calculated availability of restoration given that you are in a specific scenario.

Because the decoder requires a minimum of $k = 2$ file parts, restoration is impossible in the last two scenarios since less than 2 parts are backed up. The total availability of the system, where probability of the different scenarios is taken into account is quite simple:

$$A_{total} = P(Sc_1) * A_{Sc_1} + P(Sc_2) * A_{Sc_2} = 0.998827731 \tag{5.1}$$

The total availability, A_{total}, is still significantly higher than the one host scenario previously compared against, but not as high as the initial analysis where it was assumed every part was successfully backed up.

The possible states and scenario combinations increase greatly for higher values of n, increasing the complexity of calculating the total availability. Luckily, the total availability may also be calculated through a series of mathematical calculations.

Formula 5.2 calculate the availability of a specific scenario with x parts backed up, and y hosts available for restoration. It is based on Weatherspoon and Kubiatowicz availability formula in "Erasure Coding Vs. Replication: A Quantitative Comparison" [61].

$$A_{x,y} = \sum_{i=0}^{x-k} \frac{\binom{n-y}{i} \cdot \binom{y}{x-i}}{\binom{n}{x}} \tag{5.2}$$

The formula is tweaked slightly to take into account both hosts down at time of backup, and hosts down at time of restoration. $A_{x,y}$ is the probability that a block is available. x is the number of hosts available at time of backup. y is the number of hosts available at time of restoration. n is total number of hosts in the system, and k is the number of parts required for reconstruction.

The availability of the scenario is equal to the number of ways the unavailable parts can be arranged on unavailable hosts, multiplied by the number of ways the available parts can be arranged on available hosts, divided by the number of ways every backed up part can be arranged on every host.

$A_{x,y}$ is used to calculate the availability for $x \in \{0, 1, ..., n\}$ and $0 \leq y \leq n$, i.e. the availability when it is certain x parts are backed up, independent of how many hosts y that are available hosts at time of restoration. Formula 5.3 calculates the Availability when the backup was successfully uploaded to x hosts.

$$A_x = \sum_{y=0}^{n} A_{x,y} \cdot \binom{n}{y} \cdot A_h^y \cdot (1 - A_h)^{n-y} \tag{5.3}$$

A_x is calculated by summarizing $A_{x,y}$ multiplied by $\binom{n}{y}$ (Possible combinations with y hosts) multiplied by $A_h^y * (1 - A_h)^{n-y}$ (the probability of y servers being available and $n - y$ being unavailable) for $0 \leq y \leq n$.

The final step is to calculate A_{total}, the total availability of the system. Formula 5.4 sums the availability with x backed up, A_x, multiplied by the probability of x servers being backed up and the possible combinations with x hosts.

$$A_{total} = \sum_{x=0}^{n} A_x \cdot \binom{n}{x} \cdot A_h^x \cdot (1 - A_h)^{n-x} \tag{5.4}$$

	$n = 3$	$n = 4$	$n = 5$	$n = 6$
$k = 2$	0.9988277312	0.9999689481	0.9999992284	0.9999999816
$k = 3$	0.9414801494	0.9976865143	0.9999235276	0.9999977219
$k = 4$	0	0.9227446944	0.9961951721	0.9998493334
$k = 5$	0	0	0.904382075	0.9943680915
$k = 6$	0	0	0	0.8863848717

Table 5.5: Total availability for system with (n, k) erasure code scheme and $n = k+r$ hosts. Each host has the availability of 0.99.

Table 5.5 shows the calculated availability for for 3 to 6 hosts, and different values k. As expected, the availability increases for lower values of k. The increased availability comes at the cost of increased storage capacity required, further discussed in section 5.4.4. Setting $k = n$ is in general a bad idea, as the encoding will give the same functionality as splitting the data into k equal parts. If $k = n$, every part needs to be successfully backed up, and every part is required to be available for restoration to reconstruct the original data, making the total availability worse than when backing up a full backup to a single storage provider.

In the case $k = 1$, the result of the system is the same as regular replication. Because a single partial is enough to reconstruct, the single partial is required to contain all original data.

Listing D.5 in Appendix D may be used to further analyse the availability properties of different (n, k) values, and help the user gain the number of nines he desires.

Some of the calculations were also verified by a simulation script. The simulator uses python's random generator to determine backed up hosts and restorable hosts. The hosts are used to check if restoration is possible or not. With a large number of trials, the results of the simulator give availability values that differ with such small values from the calculated availability that it is concluded that the calculations above are correct. The simulation script is listed in Listing D.6 in Appendix D.

5.4.2 Availability with sub-optimal XOR scheme

The $(10, 6)$ XOR erasure code scheme is not optimal, and does not promise reconstruction with $n - k = 6$ data parts available. The erasure code scheme is able to promise reconstruction with up to two missing parts.

The set of solvable equations with specific parts missing determine if reconstruction is possible. With three parts missing, the scheme is able to reconstruct in 110 of the 120 possible ways to choose 7 parts from 10, i.e. the probability of reconstruction with three parts missing is $\frac{110}{120}$. If four parts are missing, the scheme is able to reconstruct in 125 of the 210 possible ways to choose 6 parts from 10 parts, i.e. the probability of reconstruction with four parts missing is $\frac{125}{210}$.

Parts missing	Probability of reconstruction
[0,1,2]	1.0
3	$\frac{110}{120}$
4	$\frac{125}{210}$
[5,6,7,8,9,10]	0.0

Table 5.6: Probability of XOR scheme being able to reconstruct the original data

The availability calculation is done with a similar line of though as with the MDS codes preciously analysed.

The first calculations find the probability of reconstruction for specific x and y values, namely $A_{x,y}$, where x is number of hosts available at time of backup, and y is number of hosts available at time of restoration. Because the XOR scheme provide uncertainty of reconstruction capabilities with different number of missing parts, Formula 5.5 is a bit different than the previous $A_{x,y}$ formula. The probabilities of successful reconstruction with parts missing are defined in Table 5.6.

$$A_{x,y} = \sum_{i=0}^{\alpha} P(2n - x - y - i) \cdot \frac{\binom{n-y}{i} \cdot \binom{y}{n-x-i}}{\binom{n}{x}} \begin{cases} \alpha = 10 - x \text{ for } x > y \\ \alpha = 10 - y \text{ for } x \leq y \end{cases} \tag{5.5}$$

The formula sums the different combinations of missing pieces multiplied by the probability of reconstruction with that amount of missing pieces. The formula matches every possible way for the unavailable hosts at time of backup to be the same as unavailable hosts at time of restoration. If 8 parts were backed up, and 9 hosts are available for restoration, the maximum number of available parts is 8, and minimum number of available parts is 7, i.e. one missing part may be the same at both backup and restoration. With the described scenario, the formula produces:

$$A_{8,9} = P(3) \cdot \frac{\binom{10-9}{0} \cdot \binom{9}{10-8-0}}{\binom{10}{8}} + P(2) \cdot \frac{\binom{10-9}{1} \cdot \binom{9}{10-8-1}}{\binom{10}{8}} = 0.933333333333 \tag{5.6}$$

The first part of the equation determines the probability that 3 parts is missing, and multiplies it with the reconstruction probability with three parts missing. The second part determines the probability that 2 parts are missing, i.e. that the part not available for restoration is one of the parts that was not backed up, and multiplies it with the probability of reconstruction with 2 parts missing.

Further calculations required to find A_{total} is equal Formula 5.3 and Formula 5.4. The formulas first sums the result for any value of y hosts available at time of restoration, and then makes the final calculation for any x and y.

The total availability A_{total} of the system when using the $(10, 6)$ XOR scheme is 0.999918998669. Accordingly, The total availability of the system with zfec, and the same storage overhead is 0.999999276582. As expected, the system utilizing the XOR scheme offers lower availability than the system using zfec with $n = 10$ and $k = 6$. While Reed-Solomon based codes achieve a higher availability, the XOR scheme is faster, and sub-optimal schemes should be considered in storage systems to increase computation performance.

Every $A_{x,y}$ calculation was verified by a python script testing if reconstruction is possible for every combination of backed up hosts and restorable hosts. The script is listed as Listing D.7 in Appendix D.

5.4.3 Performance of zfec and XOR implementation

Generally, the primary objective of XOR-schemes is to provide faster encoding and decoding of partial files. The implemented scheme is programmed in Python, and would likely be faster if implemented in a lower level programming language such as C.

Filesize	**Mode**	**XOR**	**zfec**
100KB	Encode	0.065s	0.138s
	Decode	0.040s	0.152s
1MB	Encode	0.068s	0.140s
	Decode	0.049s	0.171s
1GB	Encode	5,663s	9.474s
	Decode	4.912s	5.172s

Table 5.7: Average execution time of encoding and decoding with the implemented XOR scheme and zfec with parameters $n = 10$ and $k = 6$. Decoding is done with two of the data parts missing.

Table 5.7 shows the result of an execution time test done with the XOR implementation and the zfec python library. As expected, the XOR library performs better than the Reed-Solomon based library. The difference in execution time for smaller file sizes is very small, likely because a lot of the execution time stems from other tasks than the actual encoding and decoding. The memory was purged between each test, and each test was executed by running a minimalistic python file. The similarity of the results with 100KB and 1MB is probably due to python specific tasks and library imports.

It should be noted that zfec includes a lot of different error handling not found in the XOR implementation. Subsequently, the effort to increase the reliability of the software is responsible for some of the increased execution time.

5.4.4 Storage overhead of encoded data

Encoding of files with erasure codes comes at the cost of increased required storage to store all the partial files.

The zfec library used as part of this prototype, provides n partial files, where k is required for reconstruction. Each partial file is $\frac{1}{k}$ times the size of the original data. The original data is encoded into n partial files, making the total storage capacity required to store all the partial files $\frac{n}{k}$. In addition, each partial file also uses one to four bytes of additional storage for metadata added to the start of the share file [56]. Correspondingly, a use case with 3 hosts and a $(3, 2)$ erasure code scheme will require

$\frac{3}{2} \cdot$ original size. Similarly, a $(4, 2)$ scheme would require twice the amount of storage as a single copy.

At the same time, a replication scheme with two copies at two different storage host also require twice the amount of storage capacity. The total availability of backing up to two different hosts is 0.99960399, one number of nines less than the optimal $(4, 2)$ erasure code scheme achieves.

The implemented XOR scheme always outputs 4 redundancy parts in addition to the 6 data parts. Similarly as above, the required storage capacity to hold the encoded files is $\frac{10}{6} \cdot$ original size.

5.5 Concluding Remarks and Future Work

This chapter has shown how distributed backup is possible through the use of Duplicity and erasure codes. Erasure codes enable the possibility of restoration of backed up data, even though a subset of the storage host are unavailable at the time of backup or restoration. The proposed system resulted in a proof a concept prototype, solving the key tasks to achieve increased availability and reliability.

Existing open source software is utilized in the prototype to show how easily a distributed backup solution may be implemented. While the proof of concept prototype has some clear limitation such as error handling and decoding failure detection, it shows how such a solution may be used to safely store sensitive backup in an encrypted archive in the cloud. Future work should focus on error detection and error handling, as it is of high importance to a solution who's main purpose is to safeguard data.

Mathematical analysis was performed on the system to investigate the offered availability. The calculations show that availability may be greatly increased through using the proposed solution. The storage overhead of the encoded data is lower than when several copies of the same data are redundantly stored to increase availability, one of the key objectives of the solution.

Chapter 6 Conclusion

This project has investigated the open source software Duplicity.

Duplicity is an advanced solution, suitable in many different use case scenarios. The software may have a somewhat difficult learning curve for users that are not used to using the shell of their Unix operating system.

The project consisted of different, quite fragmented phases. The first phase consisted of setting up Duplicity, and enable testing with different settings and options. The tests uncovered how different settings may greatly influence the execution time of Duplicity backups and restorations.

Chapter 3 gave background information on Duplicity, and show how to install and set up Duplicity for usage with SSH/SFTP, Google Drive, Dropbox, OneDrive and Amazon S3. Throughout the chapter, the fragmentation of software requirements is identified as one reason Duplicity may be difficult to use for new users. Different software libraries are required for different file transfer protocols and cloud storage providers, and limited documentation exist on how to set up Duplicity with different types of storage.

Performance evaluation of different use case scenarios

The introduction to Duplicity is followed by a series of different scenario tests in chapter 4. The execution time of backups and restorations with Duplicity is measured with a wide variety of options and storage hosts. The goal of the chapter was to gain some insight into how features such as encryption and compression affect the performance of Duplicity. The chapter concludes that the confidentiality of backed up data is worth much more than the performance gain from disabling encryption. If a user wishes to speed up Duplicity backups, he/she should rather analyse the data, to see if any performance may be gained through setting the compression level to something more suitable. The experimental option enabling asynchronous upload also has the ability to greatly decrease the backup execution time.

The chapter also investigates incremental backups, and show how the restoration time increases greatly when the backup snapshot is at the end of an incremental backup chain. Users of Duplicity should limit their incremental backup chain, mainly to decrease the risk of data corruption or restoration failure, but also to reduce restoration execution time. By limiting backup chains, additional storage will be required at the storage host. The additional storage may be mitigated through decreasing the frequency of old restorable snapshots.

The last tests done in chapter 4 study Duplicity's performance with different cloud storage providers. The tests show a significant difference in execution time with the different cloud providers, with some options spending twice the time as the best options. Amazon S3 provides by far the best result at both backup and restoration execution time. The result is important because it shows a major performance difference between what may seem like similar products.

Distributed backup system with Duplicity

Chapter 5 designs, implements and analyses a distributed backup system. The system utilizes erasure codes, Duplicity and directory traversal to enable distribution of partial files to different storage hosts. The major benefit gained through using the system is increased availability due to the ability to successfully restore, even if parts of the backup is unavailable or corrupted.

The last part of chapter 5 does a statistical analysis of the systems availability. Through a series of calculations, the total availability of the system is calculated. The calculations consider both hosts unavailable at time of backup, and hosts unavailable at time of restoration.

The chapter shows that a distributed backup system using Duplicity is a viable scheme for users in need of increased reliability and availability. The proof of concept prototype show how simple an implementation may be, but a proper implementation should taker greater consideration to error-handling, and investigate if other security mechanisms are required. Scheduling of restoration is another possible addition to the system, based on the properties of the system, it may be an advantage if specific hosts are prioritized to increase overall system performance.

References

[1] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud Computing: State-of-the-Art and Research Challenges,” *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[2] “Proofs of Ownership in Remote Storage Systems, author=Halevi, Shai and Harnik, Danny and Pinkas, Benny and Shulman-Peleg, Alexandra, booktitle=Proceedings of the 18th ACM conference on Computer and communications security, pages=491–500, year=2011, organization=ACM,”

[3] A. Tridgell and P. Mackerras, “The rsync algorithm,” *Joint Computer Science Technical Report Series*, vol. TR-CS-96-0, p. 6, 1996.

[4] A. Tridgell, *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, The Australian National University, 1999.

[5] A. Tridgell, P. Mackerras, *et al.*, “Rsync - an utility that provides fast incremental file transfer..” https://rsync.samba.org. Accessed: 2015-02-15.

[6] M. Pool *et al.*, “librsync.” GitHub repository: https://github.com/librsync/librsync. Accessed: 2015-02-17.

[7] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, “Inside Dropbox: Understanding Personal Cloud Storage Services,” in *Proceedings of the 2012 ACM conference on Internet measurement conference*, pp. 481–494, ACM, 2012.

[8] B. Escoto, K. Loafman, *et al.*, “Duplicity.” http://duplicity.nongnu.org/. Accessed: 2015-02-14.

[9] M. Pool, “librsync Programmer’s Manual.” http://librsync.sourcefrog.net/doc/librsync.html. Accessed: 2015-02-17.

[10] M. Pool, “rdiff - Linux man page.” http://linux.die.net/man/1/rdiff. Accessed: 2015-02-20.

[11] W. Koch *et al.*, “The GNU privacy guard.” https://gnupg.org. Accessed: 2015-02-27.

[12] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," RFC 4880, RFC Editor, November 2007. http://www.rfc-editor.org/rfc/rfc4880.txt.

[13] Free Software Foundation, Inc., "GNU General Public License." http://www.gnu.org/copyleft/gpl.html. Accessed: 2015-02-27.

[14] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols," *Computer Communication Review*, vol. 27, no. 2, pp. 24–36, 1997.

[15] J. S. Plank, "Erasure codes for storage systems: A brief primer," *;login: the Usenix magazine*, vol. 38, December 2013.

[16] J. Lacan and J. Fimes, "Systematic MDS Erasure Codes Based on Vandermonde Matrices," *IEEE Communications Letters*, vol. 8, no. 9, pp. 570–572, 2004.

[17] K. M. Greenan, X. Li, and J. J. Wylie, "Flat XOR-based Erasure Codes in Storage Systems: Constructions, Efficient Recovery, and Tradeoffs," in *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pp. 1–14, 2010.

[18] J. J. Wylie and R. Swaminathan, "Determining Fault Tolerance of XOR-Based Erasure Codes Efficiently," in *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, pp. 206–215, 2007.

[19] A. Shokrollahi, "Raptor Codes," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

[20] N. Alon, J. Edmonds, and M. Luby, "Linear Time Erasure Codes with Nearly Optimal Recovery (Extended Abstract)," in *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pp. 512–519, 1995.

[21] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the Society for Industrial & Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[22] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage," in *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings*, pp. 253–265, 2009.

[23] J. A. Cooley, J. L. Mineweaser, L. D. Servi, and E. T. Tsung, "Software-Based Erasure Codes for Scalable Distributed Storage," in *IEEE Symposium on Mass Storage Systems*, pp. 157–164, 2003.

[24] J. Luo, M. Shrestha, L. Xu, and J. S. Plank, "Efficient Encoding Schedules for XOR-Based Erasure Codes," *IEEE Trans. Computers*, vol. 63, no. 9, pp. 2259–2272, 2014.

[25] K. M. Greenan, E. L. Miller, and T. J. E. Schwarz, “Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications,” in *16th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008), Baltimore, Maryland, USA, September 8-10, 2008*, pp. 257–266, 2008.

[26] D. Salomon, *A Concise Introduction to Data Compression.* Undergraduate Topics in Computer Science, Springer, 2008.

[27] B. Escoto, K. Loafman, *et al.*, “Duplicity readme file.” http://duplicity.nongnu.org/README. Accessed: 2015-02-15.

[28] A. Chervenak, V. Vellanki, and Z. Kurmas, “Protecting File Systems: A Survey of Backup Techniques,” in *Joint NASA and IEEE Mass Storage Conference*, 1998.

[29] B. Escoto, K. Loafman, *et al.*, “Duplicity manual.” http://duplicity.nongnu.org/duplicity.1.html. Accessed: 2015-02-14.

[30] “Duplicity features list.” http://duplicity.nongnu.org/features.html. Accessed: 2015-02-20.

[31] “GnuPG Manuals.” https://www.gnupg.org/documentation/manuals.html. Accessed: 2015-05-28.

[32] “pip - The PyPA Recommended Tool for Installing Python Packages.” https://pip.pypa.io. Accessed: 2015-03-01.

[33] “PyPA Tool Recommendations.” https://python-packaging-user-guide.readthedocs.org/en/latest/current.html. note = Accessed: 2015-04-15.

[34] D. J. Barrett and R. E. Silverman, *SSH, the Secure Shell: The Definitive Guide.* O’Reilly Media, 2001.

[35] OpenBSD Project, “OpenSSH.” http://www.openssh.com/. Accessed: 2015-03-15.

[36] R. Pointer and J. Forcier, “paramiko: Python SSH module.” GitHub repository: https://github.com/paramiko/paramiko. Accessed: 2015-02-18.

[37] J. Wu, L. Ping, X. Ge, Y. Wang, and J. Fu, “Cloud storage as the infrastructure of cloud computing,” in *Intelligent Computing and Cognitive Informatics (ICICCI), 2010 International Conference on*, pp. 380–383, IEEE, 2010.

[38] S. Landau, “Highlights from making sense of snowden, part II: what’s significant in the NSA revelations,” *IEEE Security & Privacy*, vol. 12, no. 1, pp. 62–64, 2014.

[39] M. Garnaat *et al.*, “Boto - Python interface to Amazon Web Services.” GitHub repository: https://github.com/boto/boto. Accessed: 2015-03-01.

[40] Amazon.com, Inc., “Amazon Web Services: Regions and Endpoints.” http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region. Accessed: 2015-03-29.

[41] Google Inc., “GData Python Client.” GitHub repository: https://github.com/google/gdata-python-client. Accessed: 2015-03-02.

[42] K. Reitz, “Requests: HTTP for Humans.” http://docs.python-requests.org/. Accessed: 2015-03-12.

[43] K. Reitz, “Requests-OAuthlib.” GitHub repository: https://github.com/requests/requests-oauthlib. Accessed: 2015-03-12.

[44] Dropbox Inc., “Dropbox Python SDK.” https://www.dropbox.com/developers/core/sdks/python. Accessed: 2015-03-03],.

[45] A. Santos and J. Bernardino, “Open Source Tools for Remote Incremental Backups on Linux: An Experimental Evaluation,” *Journal of Systems Integration*, vol. 5, no. 3, pp. 3–13, 2014.

[46] The OpenSSL Project, “OpenSSL rand tool.” https://www.openssl.org/docs/apps/rand.html. Accessed: 2015-03-05.

[47] P. Rubin, D. MacKenzie, and S. Kemp, “dd(1) - Linux man page.” http://linux.die.net/man/1/dd. Accessed: 2015-03-05.

[48] B. escoto, K. Loafman, *et al.*, “Duplicity Source Code: gpg.py, line 388.” http://bazaar.launchpad.net/~duplicity-team/duplicity/0.7-series/view/head:/duplicity/gpg.py#L388. Accessed: 2015-03-20.

[49] C. Coager, “[Duplicity-talk] no-compression still compresses?.” Archived email thread: https://lists.nongnu.org/archive/html/duplicity-talk/2013-03/msg00011.html. Accessed: 2015-03-20.

[50] A. Khan, M. Shahriar, S. K. A. Imon, M. D. Francesco, and S. K. Das, “PeerVault: A Distributed Peer-to-Peer Platform for Reliable Data Backup,” in *Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings*, pp. 315–329, 2013.

[51] F. Morcos, T. Chantem, P. Little, T. Gasiba, and D. Thain, “iDIBS: An Improved Distributed Backup System,” in *12th International Conference on Parallel and Distributed Systems, ICPADS 2006, Minneapolis, Minnesota, USA, July 12-15, 2006*, pp. 58–67, 2006.

[52] D. Slamanig and C. Hanser, “On Cloud Storage and the Cloud of Clouds Approach,” in *7th International Conference for Internet Technology and Secured Transactions, ICITST 2012, London, United Kingdom, December 10-12, 2012*, pp. 649–655, 2012.

[53] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, “RACS: a Case for Cloud Storage Diversity,” in *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pp. 229–240, 2010.

[54] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: the least-authority filesystem," in *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pp. 21–26, ACM, 2008.

[55] "Tahoe-LAFS: The least-Authority File Store." https://www.tahoe-lafs.org. Accessed: 2015-02-27.

[56] Z. Wilcox-O'Hearn *et al.*, "zfec: a fast erasure codec." https://pypi.python.org/pypi/zfec. Accessed:2015-03-30.

[57] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[58] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The Structure and Value of Modularity in Software Design," in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, (New York, NY, USA), pp. 99–108, ACM, 2001.

[59] L. Torvalds *et al.*, "Git: Free and open source version control system." http://git-scm.com/. Accessed: 2015-04-25.

[60] H. N. Matland, "Distributed Duplicity Prototype." GitHub repository: https://github.com/hmatland/distributed_duplicity. Accessed: 2015-06-7.

[61] H. Weatherspoon and J. D. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison," in *Peer-to-Peer Systems*, pp. 328–337, Springer, 2002.

Appendix A
Test File Creation

Listing A.1 Bash script used to create the test directory used in chapter 4

```bash
#!/bin/bash
for l in {0..9}
do
  #Create 1 empty folders named dir_0 through dir_9
  mkdir full1/dir_$l
  for i in {0..9}
  do
    #Create 10 empty folders, named dir_0 through dir_9
    mkdir full1/dir_$l/dir_$i
    for j in {0..199}
    do
      #In each of the 10 folders, create 100 files approx. 1KB each
      openssl rand -out full1/dir_$l/dir_$i/1KB_$j 768
      dd if=/dev/zero count=1 bs=256 >> full1/dir_$l/dir_$i/1KB_$j
    done
    for k in {0..44}
    do
      #In each of the 10 folders, create 45 files approx 100KB each
      openssl rand -out full1/dir_$l/dir_$i/100KB_$k $(( 2**10 * 75 ))
      dd if=/dev/zero count=25 bs=$((2**10)) >> full1/dir_$l/dir_$i/100KB_$k
    done
    for m in {0..4}
    do
      #In each of the 10 folders, create 5 files approx 1MB each
      openssl rand -out full1/dir_$l/dir_$i/1MB_$m $(( 2**10 * 768 ))
      dd if=/dev/zero count=256 bs=$((2**10)) >> full1/dir_$l/dir_$i/1MB_$m
    done
  done
done
exit 0
```

Appendix B
Timing test between openssl rand and dd if=/dev/urandom

Listing B.1 Timing test script between openssl rand and dd with interface /dev/urandom

```
echo 'dd with 1024: 1KB'
time dd if=/dev/urandom count=1 bs=1024 of=test
echo 'openssl with 1024: 1KB'
time openssl rand -out test2 $(( 1024  ))
echo 'dd with 1024*10: 10KB'
time dd if=/dev/urandom count=10 bs=1024 of=test13
echo 'openssl with 1024*10: 10KB'
time openssl rand -out test14 $(( 1024*10  ))
echo 'dd with 1024*128: 128KB'
time dd if=/dev/urandom count=128 bs=1024 of=test3
echo 'openssl with 1024*128: 128KB'
time openssl rand -out test4 $(( 1024*128  ))
echo 'dd with 1024*512: 512KB'
time dd if=/dev/urandom count=512 bs=1024 of=test5
echo 'openssl with 1024*512: 512KB'
time openssl rand -out test6 $(( 1024 * 512  ))
echo 'dd with 1024*1024: 1MB'
time dd if=/dev/urandom count=1024 bs=1024 of=test7
echo 'openssl with 1024*1024: 1MB'
time openssl rand -out test8 $(( 1024*1024  ))
echo 'dd with count:1024*100, bs=1024: 100MB'
time dd if=/dev/urandom count=$((1024*100)) bs=1024 of=test9
echo 'openssl with 1024*1024*100: 100MB'
time openssl rand -out test10 $(( 1024*1024*100  ))
echo 'dd with count=1024*1024, bs=1024: 1GB'
time dd if=/dev/urandom count=$((1024*1024)) bs=1024 of=test11
echo 'openssl with 1024*1024*1024: 1GB'
time openssl rand -out test12 $(( 1024*1024*1024  ))
```

Appendix C
Duplicity Patch to Set Compression Level

This appendix describe how duplicity was patched to enable compression-level setting through command line options.

In **commandline.py** the following two line were added to enable modification of compression level through the command line options:

Listing C.1 The new option is accessible as an argument when executing Duplcity from shell.

```
# Compression-level used by gpg and GzipWriteFile
parser.add_option("--compress-level", type="int",
↪  metavar="compress_level")
```

The new option needs a corresponding value in **globals.py**. This is also were the default value used when the –compress-level option is not used. The default value is set to 6.

Listing C.2 Variable for storing the parsed compression level option. Default value is 6.

```
#Compression level used with both GPG and GzipWriteFile.
compress_level = 6
```

The changes that actually utilize the new option possibility is found in **gpg.py**. The following lines are added to the **GPGFile** class.

Listing C.3 The newly created `--compress-level` option is not used if compress level is set through GnuPG options

```
if('--compress-level' not in globals.gpg_options):
  gnupg.options.extra_args.append('--compress-level='
↪   +str(globals.compress_level))
```

The if-statement avoids problems if the compression level is also set through the use of −gpg−options. −−gpg−options is given precedence.

The hardcoded compression level in the **GzipWriteFile** function of **gpg.py** is replaced with the newly created `globals.compress_level`.

Listing C.4 New global variable used instead of hardcoded value 6.

```
#gzip_file = gzip.GzipFile(None, "wb", 6, file_counted)
gzip_file = gzip.GzipFile(None, "wb", globals.compress_level,
↪  file_counted)
```

Appendix D Code From Distributed Duplicity prototype

This appendix contain various source code listings from the distributed backup system prototype. The full project may be viewed or downloaded from the authors GitHub repository. [1]

Listing D.1 shows the `Duplicity_distributer` class. It is the class responsible for setting up the different hosts according the the settings set in `duplicity_conf`. The code use the Pool class from python's multiprocessing library to execute several Duplicity instances simultaneously.

Listing D.2 contains the source code for the $(10, 6)$ XOR encoder described in section 5.3.2. The implemented decoder is shown in Listing D.3 and D.4.

Listing D.5 and D.6 show two availability scripts that are used to calculate and simulate the availability of the system using the zfec library.

[1] Distributed-Duplicity at GitHub: https://github.com/hmatland/distributed_duplicity

Listing D.1 duplicity_distributer.py with the Duplicity_distributer class

```
from subprocess import call
import os
from multiprocessing import Pool
import duplicity_conf as conf

def execute_duplicity_command(host):
  print 'Running: '
  print host
  call(host.split())

class Duplicity_distributer:
  def __init__(self, path):
    self.path = path

  def set_up_hosts(self):
    source=os.path.join(self.path,'distdup')
    self.hosts = [None]*len(conf.remote_hosts)
    #Format every duplicity command based on configuration variables
    for i in xrange(0,len(conf.remote_hosts)):
      self.hosts[i] = 'duplicity --allow-source-mismatch {options} {source_path}{i}
       ↪  {host}/{remote_path}/distdup{i}'.format(source_path=source,
       ↪  host=conf.remote_hosts[i], remote_path=conf.remote_paths[i],
       ↪  options=conf.remote_host_options[i], i = i)

    #Set up any environment variables set in conf
    for env_var in conf.env_vars:
      os.environ[env_var[0]] = env_var[1]

  def run_backup(self):
    Pool().map(execute_duplicity_command, self.hosts)

  def remove_env_var(self):
    #Delete any environment variables set in conf
    for env_var in conf.env_vars:
      del os.environ[env_var[0]]
```

Listing D.2 The XOR encoder for the $(10, 6)$ systematic erasure code scheme.

```
import Crypto.Util.strxor
from os import path, stat
from math import ceil

def sxor(s1, s2):
  s1_len = len(s1)
  s2_len = len(s2)
  if s1_len > s2_len:
    s2 = s2.ljust(s1_len,' ')
  elif s2_len > s1_len:
    s1 = s1.ljust(s2_len,' ')
  return Crypto.Util.strxor.strxor(s1,s2)

def encode(data):
  y7 = sxor(sxor(data[2],data[4]),data[5])
  y8 = sxor(sxor(data[1],data[3]),data[5])
  y9 = sxor(sxor(data[0],data[3]),data[4])
  y10= sxor(sxor(data[0],data[1]),data[2])
  return [y7,y8,y9,y10]

def create_part_files(part_path):
  files = [None]*10
  for i in xrange(0,10):
    files[i] = open(part_path+'.'+str(i)+'_10.partial','a')
  return files

def split_file(path, path_size, files):
  with open(path,'rb') as f:
    part_size = int(ceil(path_size/6.0))
    for i in xrange(0,6):
      part = f.read(part_size)
      files[i].write(part)

def encode_file(inf, to_path):
  data = None
  filename = path.basename(inf)
  part_path = path.join(to_path, filename)
  path_size = stat(inf).st_size
  part_size = int(ceil(path_size/6.0))
  files = create_part_files(part_path)
  split_file(inf, path_size, files)

  for f in xrange(0,6):
    files[f].close()
    files[f] = open(part_path+'.'+str(f)+'_10.partial','r')

  while part_size > 0:
    array = ['']*6
    for f in xrange(0,6):
      array[f] = files[f].read(4096)
    data = encode(array)
    for f in xrange(0,4):
      files[6+f].write(data[f])
    part_size -= 4096
  for f in files:
     f.close()
```

Listing D.3 The XOR decoder for the $(10, 6)$ systematic erasure code scheme.

```
import Crypto.Util.strxor
rec_equation = [None] * 10
rec_equation[0] = [[10,2,3],[9,4,5]]
rec_equation[1] = [[10,1,3],[8,4,6]]
rec_equation[2] = [[7,5,6],[1,2,10]]
rec_equation[3] = [[2,8,6],[1,9,5]]
rec_equation[4] = [[3,7,6],[1,4,9]]
rec_equation[5] = [[3,5,7],[2,4,8]]
rec_equation[6] = [[3,5,6]]
rec_equation[7] = [[2,4,6]]
rec_equation[8] = [[1,4,5]]
rec_equation[9] = [[1,2,3]]

def sxor(s1, s2):
  s1_len = len(s1)
  s2_len = len(s2)
  if s1_len > s2_len:
    s2 = s2.ljust(s1_len,' ')
  elif s2_len > s1_len:
    s1 = s1.ljust(s2_len,' ')
  return Crypto.Util.strxor.strxor(s1,s2)

def merge_files(outfile, files):
  for x in xrange(0,6):
    for f in files:
      if int(f.name[-12]) is x:
        byte = f.read(4096)
        while byte != '':
          outfile.write(byte)
          byte = f.read(4096)

def restore_partial_file(outfile, files, eq):
  req_files = []
  # print eq
  for i in eq:
    for f in files:
      if int(f.name[-12]) is (i-1):
        req_files.append(f)
  for f in req_files:
    f.seek(0,0)
  b_a = req_files[0].read(4096)
  b_b = req_files[1].read(4096)
  b_c = req_files[2].read(4096)
  while b_a != '' or b_b!='' or b_c!='':
    res = sxor(sxor(b_a, b_b),b_c)
    outfile.write(res)
    b_a = req_files[0].read(4096)
    b_b = req_files[1].read(4096)
    b_c = req_files[2].read(4096)
  for f in files:
    f.seek(0,0)
```

Listing D.4 Continuation of the XOR decoder for the $(10, 6)$ systematic erasure code scheme.

```
def decode_from_files(outfile,files,verbose):
  files_indices = [None]*10
  for f in files:
    files_indices[(int(f.name[-12]))] = True
  if(not None in files_indices[0:6]):
    merge_files(outfile, files)
  else:
    previousIndices = -1
    while(None in files_indices[0:6]):
      indices = [i for i, x in enumerate(files_indices) if x == None]
      if(previousIndices==indices):
        break
      for missing in indices:
        for eq in rec_equation[missing]:
          possible = True
          for i in eq:
            if files_indices[i-1] == None:
              possible = False
          if possible:
            filepath = list(files[0].name)
            filepath[-12] = str(missing)
            filepath = ''.join(filepath)
            with open(filepath,'w') as tempfile:
              restore_partial_file(tempfile, files, eq)
            newfile = open(filepath,'r')
            files.append(newfile)
            files_indices[missing] = True
            break
    if None in files_indices[0:6]:
      print 'Reconstruction of file :' + str(outfile.name) + ' not possible. Too few
        ↪ partial files available.'
    else:
      merge_files(outfile, files)
```

Listing D.5 Script that calculates A_{total}, the total availability of the system, when Reed-Solomon is used with number of hosts equal n, and every host has the same availability

```python
from operator import mul
from fractions import Fraction
import math
import argparse

#Parser of arguments left out for readability in pdf version

def nCk(n,k):
  return int( reduce(mul, (Fraction(n-i, i+1) for i in range(k)), 1) )

def calculate_availability(k,n, avail_hosts):
  avail_tot = 0
  for x in xrange(0,n+1):
    avail_x = 0
    for y in xrange(0,n+1):
      prob_restoration_xy = 0
      for i in xrange(0,x-k+1):
        prob_restoration_xy += nCk(n-y,i)*nCk(n-(n-y),x-i)
      prob_restoration_xy = prob_restoration_xy/float(nCk(n, x))
      avail_x += prob_restoration_xy * nCk(n,y)*(avail_hosts**y *
       ↪  (1-avail_hosts)**(n-y))
    avail_tot += nCk(n,x)*(avail_hosts**x)*((1-avail_hosts)**(n-x)) * avail_x
  return avail_tot

k= args.k
n= args.n
availability_of_hosts = args.a

if k == -1:
  for k in xrange(2,n):
    print 'Availability for system with optimal (' +str(n) +','+str(k) + ') erasure
     ↪  code scheme'
    print 'One partial part per host. Each host has availability of ' +
     ↪  str(availability_of_hosts)
    print calculate_availability(k,m,availability_of_hosts)
else:
  print 'Availability for system with optimal (' + str(n)+','+str(k) + ') erasure
   ↪  code scheme'
  print 'One partial part per host. Each host has availability of ' +
   ↪  str(availability_of_hosts)
  print calculate_availability(k,n,availability_of_hosts)
  print calculate_availability(k,m,availability_of_hosts)
```

Listing D.6 Script that simulates availability of hosts at backup and restoration, and checks if restoration is possible with the current combination. A large number of trials should be done to get a good estimate of the total availability. The script is when MDS erasure codes, as provided by the zfec library, is used.

```python
parser = argparse.ArgumentParser(description='Simulate availability of distdup prototype')
parser.add_argument('-k','--k', type=int, help='Number of blocks necessary to reconstruct the original data', required=True)
parser.add_argument('-n','--n', type=int, help='Total number of total blocks', required=True)
parser.add_argument('-a','--a', type=float, help='Availability of the hosts', required=True)
parser.add_argument('-t','--trials', type=int, help='Number of trials', required=True)
args = parser.parse_args()

k= args.k
n= args.n
availability_of_hosts = args.a
trials = args.trials

def restoration_possible(backed_up_hosts, restorable_hosts, k):
  counter = 0
  for server in restorable_hosts:
    if server in backed_up_hosts:
      counter += 1
  if counter >= k:
    return True
  else:
    return False

failed_counter = 0
for i in xrange(0,trials):
  backed_up_hosts = []
  restorable_hosts = []
  for host in xrange(0,n):
    if random() < availability_of_hosts:
      backed_up_hosts.append(host)
    if random() < availability_of_hosts:
      restorable_hosts.append(host)
  if not restoration_possible(backed_up_hosts, restorable_hosts, k):
    failed_counter += 1

print 1-(float(failed_counter)/trials)
```

Listing D.7 Script that calculates A_{total}, the total availability of the system when the $(10, 6)$ XOR scheme is used with 10 hosts, every host has the same availability

```
import itertools
from operator import mul
from fractions import Fraction
import math

p= [1,1,1,110.0/120,125.0/210,0,0,0,0,0,0,0,0,0,0,0,0]

def nCk(n,k):
  return int( reduce(mul, (Fraction(n-i, i+1) for i in range(k)), 1) )

def probabilityOfRestoration(backed_up_hosts, restorable_hosts):
  counter = 0
  for server in restorable_hosts:
    if server in backed_up_hosts:
      counter +=1
  if counter >= 8:
    return 1
  elif counter == 7:
    return 110.0/120
  elif counter == 6:
    return 125.0/210
  else:
    return 0

def calculateAvailability(n, availability_of_host):
        hosts =[]
        for i in xrange(1,n+1):
                hosts.append(i)
        avail_total = 0
        for x in xrange(0,n+1):
                avail_x = 0
                backed_up_combinations = None
                for y in xrange(0,n+1):
                        backed_up_combinations = itertools.combinations(hosts, x)
                        restorable_servers_combinations = itertools.combinations(hosts,y)

                        possible_counter = 0
                        total_counter = 0

                        for backed_up_servers in backed_up_combinations:
                                for restorable_servers in restorable_servers_combinations:
                                        total_counter += 1
                                        possible_counter += probabilityOfRestoration(backed_up_servers, restorable_servers)
                        avail_x_y = float(possible_counter)/(total_counter)
                        avail_x += avail_x_y * nCk(n,y)*(availability_of_host**y * (1-availability_of_host)**(n-y))
                avail_total += nCk(n,x)*(availability_of_host**x)*((1-availability_of_host)**(n-x)) * avail_x
        return avail_total
print calculateAvailability(10,0.99)
```