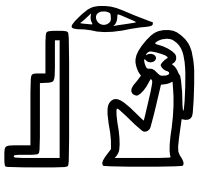
James Bown

Osiris

Secure Social Backup



Computer Science Tripos, Part II Project

St John's College

May 17, 2011

The cover page image is the hieroglyphic representation of the Ancient Egyptian god of the underworld, Osiris. His legend tells of how he was torn into pieces and later resurrected by bringing them together once again.

The font used to generate this image was used with the kind permission of Mark-Jan Nederhof (`http://www.cs.st-andrews.ac.uk/~mjn/`*).*

Proforma

Name:	**James Bown**
College:	**St John's College**
Project Title:	**Osiris – Secure Social Backup**
Examination:	**Computer Science Tripos, Part II Project**
Date:	**May 17, 2011**
Word Count:	**11,841**
Project Originator:	**Malte Schwarzkopf**
Supervisor:	**Malte Schwarzkopf**

Original Aims of the Project

To produce a distributed system enabling mutually beneficial peer-to-peer backup between groups of friends. Each user provides storage space on their personal machine for other users to back up their data. In exchange, they have the right to back up their own files onto their friends' machines. I focus on the challenges of space efficient distribution and fault tolerant retrieval of data. The use of convergent encryption and a strict security policy maintains confidentiality of data.

Work Completed

All success criteria specified in the proposal have been not only fulfilled, but exceeded. I have implemented a concurrent and distributed peer-to-peer backup system that is able to send, retrieve and remove files from the network, recover from node failure or loss, and provide high security supporting convergent encryption. Finally, I have completed a number of additional extensions. They include implementation and integration of an information dispersal algorithm, support for data deduplication, an implementation of a zero-knowledge password proof for authentication, and provision of customisable security levels for each file backed up to the network.

Special Difficulties

None.

Declaration

I, James Bown of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

This dissertation describes the creation of a secure, peer-to-peer, social backup solution, hence forth known as *Osiris*. Its premise is that any user, regardless of technical ability, should be able to back up their encrypted personal data onto their friends' machines with minimal user interaction required.

I focus on the challenges of creating a system with a strong security policy, and ensuring efficient usage of available storage space whilst maintaining high data availability.

1.1 Motivations

Backing up of data is an essential requirement for many people in this modern age. It is of crucial importance to home computer users to protect themselves from loss and corruption of their personal data. However, a survey conducted in 2010 [1] found that of the 6,000 home computer users asked, an incredible 89% do not perform regular backups. Shockingly this figure is 8% higher than the value obtained two years before. Figure 1.1 shows the reasons, according to this study, why users do not backup their data regularly. These problems should be rectified.

Existing backup solutions for home computer users can be put into two categories: *external media* and *cloud storage*. Both of these methods have their disadvantages. External media storage devices, such as USB flash drives or external hard drives, are still expensive despite the cost of storage slowly decreasing. Additionally, they are vulnerable to theft and failure themselves, especially if co-located with the computer.

Cloud storage services, such as Dropbox[1] or Amazon S3[2], allow users to back up their personal data onto their servers. This too has a price if they wish to back up large amounts of data. But more importantly, why should users have to trust these companies with their personal data? It is even possible they will store it within a jurisdiction where their rights to privacy are not protected by law.

Osiris aims to solve these problems by providing a backup solution that is free of charge, off-site and allows the user's data to retain complete confidentiality.

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

[2] `http://aws.amazon.com/s3/`

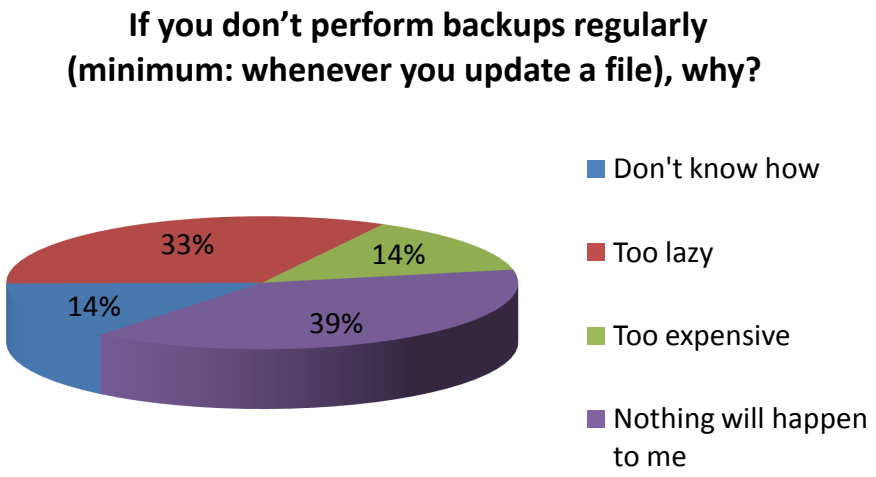


Figure 1.1: Global Data Backup Survey 2010 – obtained from 6,149 home computer users from 128 countries.

The premise of Osiris is that a user will be provided with an interface to back up data onto their peers' machines. Together, a group of friends or colleagues will form a *network*. Each user will make space on their personal machine available to the network. In return for offering this space, they will be allowed to back up their data onto the network. This system is mutually beneficial to all parties. It exploits the fact that the majority of users have excess unused capacity on their personal machines [2].

Initially, the idea of backing up personal documents, music and photos to somebody else's machine may seem unappealing to most. To this end, Osiris works hard to ensure that user data are kept confidential at all times. This is achieved by encrypting all files while they are stored on the network.

A backup system is only as good as its ability to retrieve the backup data. With this in mind, Osiris places a strong focus on ensuring that data recovery is as simple and reliable as the backup process itself. This involves ensuring that user files are available for recovery, given that many clients may be offline at any time. This will be later referred to as, the *availability* of the data.

1.2 Challenges

Primarily, Osiris is a distributed system and so faces all of the difficulties in the field of distributed computing. These include coordinating remote nodes, managing shared resources and accounting for node failure. It must also deal with networking problems such as message loss or out-of-order delivery. To enable distributed operation, Osiris requires a layer of message-oriented middleware to facilitate message passing between different client instances. It must also provide a mechanism for transferring large files across the network.

Additionally, the challenges surrounding this project are deeply entwined in the field of information security. In order to back up files in a secure manner, Osiris needs to ensure that the file data remain confidential at all times. This is accomplished with convergent encryption (see Section 2.2.2). To provide an extra tier of security, all backed up data are anonymised. To meet this goal, the system's metadata must be stored in a location that no client has access to.

The system is also highly parallel, and hence must deal with a variety of concurrency issues. Many of the techniques used in this project draw from widely used, reliable protocols to ensure system stability and database consistency.

1.3 Related Work

A few existing implementations of a peer-to-peer backup system have been attempted, but Osiris aims to do better. A system developed by a group of MIT students, *pStore* [3], does not support an information dispersal algorithm and so relies upon exact-copy replication to maintain data availability. In contrast, an information dispersal algorithm divides and distributes files into chunks, such that only some of those chunks are required to restore the file.

Another commercial peer-to-peer backup system, *CrashPlan* [4], defaults to storing file decryption keys on its central servers. As such, users must trust this service. The only alternative they provide necessitates remembering three separate passwords in order for users to recover their data.

To the author's knowledge, no solution has incorporated both an information dispersal algorithm and deduplication, two prominent features of Osiris. When both techniques are employed at the same time, the resulting system has the potential to simultaneously achieve a low space utilisation and high availability of data.

1.4 Overview of Osiris

By way of an introduction to Osiris, I will describe the high level concepts behind its operation. The justification for these decisions can be found in the rest of this dissertation.

Osiris is a system that is made up of two components: a *client* and a *controller*. An Osiris network consists of multiple clients and a single controller. The controller is involved in all operations of Osiris because it acts as the metadata store for the network. Metadata is colloquially defined as "the data about the data", and in Osiris this refers to all information about backup files on the network, such as their names and locations.

Importantly, the controller is not trusted and as such no backup files ever pass through it. This means it can be run on any machine, even by untrusted third parties such as Amazon or another cloud computing service.

When a file is backed up, it is divided into multiple chunks (for availability and security reasons) before being distributed across the network onto peers' machines. Keeping track of which chunks belong to which files is one of the major pieces of metadata the controller must

store. For added security, keeping this metadata secret is also of crucial importance. A chunk might be stored on multiple clients to provide a greater availability of files and decreased chance of chunk loss. The complexities behind the entire remote backup process are hidden from the user. This includes, but is not limited to encryption, division into chunks, storage allocation policies, network communication and transaction management.

After a user backs up their data to the network, they might have their laptop stolen. To recover their data, they must run Osiris on another machine and then request data retrieval. The controller will calculate whether backup files can be retrieved by observing which other clients are online and which chunks they store. For each file that is available, the client will be able to request all chunks from its peers and then reconstruct it.

Note that if a user has their laptop stolen then all chunks they are backing up for other users will also be lost. This does not go unnoticed by the controller, which will attempt to restore the backup data to their new computer.

Osiris provides a high level of data confidentiality to prevent anybody, even other users on the network, from reading backup files. This is primarily achieved by encrypting all chunks before they are transmitted onto the network. The key for this encryption is a password that the user provides and *must* remember in order to decrypt their data on a new computer. This password is also used to authenticate a user to the network to ensure they are who they claim by way of a zero-knowledge password proof.

Core Principles of Osiris

1. The controller never has access to the backup data.
2. Users need only remember their password in order to restore their data.
3. Backup data are entirely confidential and anonymised whilst on the backup network.
4. Osiris remains operational even if several clients are offline.

Chapter 2

Preparation

Undertaking a project of this size and scope requires a large degree of preparation. Initially, the main bodies of theory relevant for implementation need to be understood. After that a clear design plan needs to be created. In this section, I will discuss the work that was undertaken prior to the implementation of Osiris.

First, I begin with a discussion of the requirements for this project. This is followed by a summary of the relevant theory required to understand Osiris' complexities. Finally, I discuss the practicalities of how Osiris was implemented, including the tools used and the software engineering principles adhered to.

2.1 Requirements Analysis

The first major decision was whether to opt for a pure peer-to-peer model or one with a central coordinator. The main advantages of the purist model are that there is no single point of failure and that the network scales without an increasing demand for resources.

However, Osiris does implement a central coordinator, the *metadata controller*. There were many reasons for choosing this over the pure peer-to-peer model. The primary advantage that a controller provides is that clients do not have access to the file metadata. This means that data can be entirely anonymised from the perspective of each client. No user will be able to determine whose files they have backed up, nor which chunks are required to reconstruct a file. This advantage alone was enough to outweigh the costs of a single point of failure.

Additionally, a controller does provide the network with an improved efficiency since a file can be located in $O(1)$ as opposed to $O(\log n)$ for a metadata distributed hash table [5]. The controller is also privileged to make decisions on behalf of the entire network without being overruled by other nodes. Finally, a point of control means that clients can be easily authenticated before being permitted to join the network.

Despite this decision, Osiris still remains a peer-to-peer solution because all clients communicate directly with their peers whilst sending and retrieving files. Only metadata is passed between clients and the controller. The controller can be run on any machine, even the servers of an untrusted third party since no user data ever passes through it.

Requirement	*Section*
A network should be able to support an average size of 10 clients. The system should scale to support at least 20 clients.	4.3.3
The controller should act as a point of coordination and as a metadata store.	3.2
The client should break files down into chunks and distribute these chunks across the network.	3.6.2
The system should implement a storage algorithm to distribute a backup file across the network, a retrieval algorithm to recover a file from the network, and a removal algorithm to remove a file from the network.	3.6
The system should make use of deduplication to minimise the space used by files.	3.8
The system should make use of an information dispersal algorithm to maximise the availability of files.	3.9.2
If a user deletes files they are meant to have backed up for others, the system should attempt to recover them.	3.11
The system should provide customisable security levels so that a user may select how secure they wish the backup of a file to be.	3.10
The system should be fully customisable to suit different sets of requirements (e.g. prioritise availability over performance or space usage).	3.4
The client should have an intuitive graphical user interface.	3.1

Table 2.1: Requirements analysis.

The requirements for the Osiris network are shown in Table 2.1, alongside the relevant sections describing how they have been satisfied.

2.2 Research

Before implementation could begin, a degree of research needed to be conducted to understand the theoretical basis Osiris relies upon. In the next three sections I summarise some key features of this project: deduplication, convergent encryption and the information dispersal algorithm.

2.2.1 Deduplication

Deduplication is a technique to eliminate redundant data by treating identical files as one and storing only a single copy. This is often used in environments where storage space is a limiting factor, and as such, deduplication is a data compression technique. A recent study performed by Meyer and Bolosky [6] found that deduplication in backup systems can yield very high space savings. This is due to the incremental nature of backups, whereby the majority of file data remains unchanged and only modified chunks need to be stored.

In Osiris, deduplication is used when one or more users upload the same data to the network. The deduplication occurs at the chunk level so to exploit the incremental nature of

backups. The metadata for a file reflects the fact that its chunks might be shared with many other files.

To assess whether two chunks are the same, a collision resistant hash can be used if they are stored in plaintext. However, Osiris offers security guarantees causing the chunks to be encrypted when stored on the network. This poses a real problem for deduplication because identical chunks cannot be easily identified if encrypted under different keys. The solution to this problem is to use *convergent encryption*.

Deduplication was not originally featured in my project proposal, but has become one of the main extensions of my project given the complexities that it introduced into the storage protocol. This is detailed in Section 3.8.

2.2.2 Convergent Encryption

This is a technique originally proposed by the Farsite project [7] for encrypting files for the purposes of deduplication. The following is a summary of the aspects of convergent encryption that were used in Osiris.

The seminal idea for the encryption is that the symmetric encryption/decryption key is generated in a consistent manner from the file data. More specifically, a hash of the plaintext file is used as the key. Because of this, the encryption of two identical plaintext files always yields identical ciphertext. Therefore, just by looking at a collision resistant hash of two ciphertexts, we can identify whether they will yield the same plaintext.

This is secure because the key is only known by those who should have access to the file. Unfortunately, in a backup system we cannot locally store the keys for every file as data loss may also cause the loss of the keys. A desirable scheme for this application is one where a single password can be used to ensure the security of all files.

Convergent encryption solves this problem by storing the decryption key for the file with the file itself. However, this is insecure in the case of Osiris, unless we also encrypt the decryption key itself. This encryption is performed by each user with their own single password. Each user's encrypted decryption key is stored within the file header. The structure of the resulting file is shown in Figure 2.1.

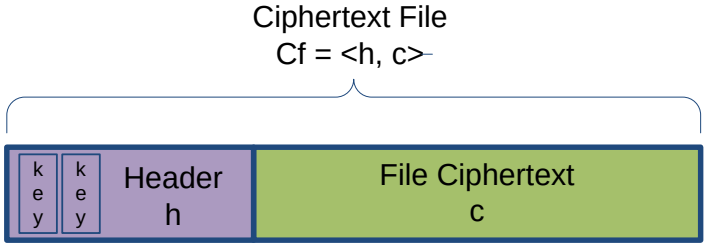


Figure 2.1: Convergent encryption output file structure.

We will now look at how convergent encryption works formally. The following is summarised, for the reader's convenience, from Douceur *et al.* [7]. Table 2.2 shows the terminol-

Symbol	Meaning
$E(x)$	Symmetric encryption function (e.g. AES)
$H(x)$	Cryptographic hash function (e.g. SHA-2)
U	Set of users who own the file
P_f	Plaintext file
p	Plaintext data – $P_f = p$
C_f	Ciphertext file
h	Header of ciphertext file C_f
h_u	Header entry for the user u
c	Ciphertext data
K_u	The single-key for a user u (password based)

Table 2.2: Convergent encryption terminology definitions.

ogy used in the following subsections.

Encryption

We first take the plaintext data p from the file P_f. We calculate a cryptographically strong hash of the data:

$$k = H(p)$$

We will then use k as the key for encrypting the data p.

$$c = E_k(p)$$

We now have our ciphertext data c, but we also need to create a file header containing k. But k cannot be stored in plaintext form, so it must be encrypted. Each user in U must encrypt k with their own secret key K_u. They each store this information as a header entry h_u in h where

$$h = \{h_u \mid h_u = E_{K_u}(k) \wedge u \in U\}$$

The ciphertext file is therefore the tuple

$$C_f = < h, c >$$

Decryption

Given C_f, we must first obtain the key to decrypt c from the user's header entry h_u. We once again call this key k:

$$k = E_{K_u}^{-1}(h_u)$$

And so the plaintext file can be recovered using k as the decryption key:

$$P_f = p = E_k^{-1}(c)$$

The proof of security for convergent encryption can be found in the extended version of the Farsite paper [7].

Summary

To summarise, we can now determine whether two encrypted files are identical by comparing their ciphertext c. We can therefore make use of deduplication to store identical files only once. We do this within the space of a single file plus some small amount of space for the header.

2.2.3 Information Dispersal Algorithm

The information dispersal algorithm (IDA) was initially proposed by Rabin in his 1989 paper [8] dedicated to this topic. The algorithm initially splits any file, F of size $|F|$, into a specified number of chunks, n. Of these n chunks only m chunks are required to reconstruct the entire file. Both n and m can be selected completely arbitrarily subject to the constraint that $m \leq n$. Importantly, we can use *any* m chunks to reconstruct the file.

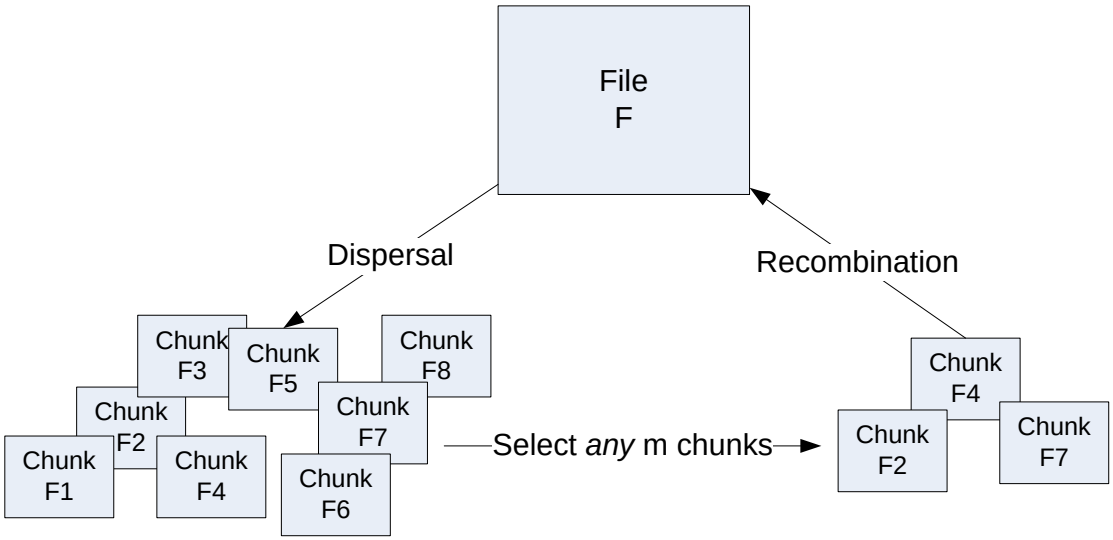


Figure 2.2: Information dispersal algorithm example with m=3 and n=8.

In Osiris, Rabin's IDA is employed to create the chunks of a file before dispersing them across the network. The recombination algorithm of this IDA is then employed if a user wishes to recover their backup data. Clearly, we only need to retrieve m chunks from our clients for a retrieval. This yields a higher file availability, as we can tolerate offline clients.

Each chunk is of size $|F|/m$, leading to a total space usage of $(n/m) \times |F|$. When we compare this to whole-chunk replications we see that the IDA is significantly more space-efficient whilst retaining the same guarantees against loss. For example, we can lose at most $n - m$ chunks and still be able to reconstruct the entire file. This is analogous to providing $n - m + 1$ replicas, if we assume that the entire file is replicated at each location. The space usage for replicas would be $(n - m + 1) \times |F|$ as opposed to the much smaller $(n/m) \times |F|$ for the IDA. This means that the cost of each additional chunk redundancy is $(1/m) \times |F|$ as opposed to $|F|$ with whole-chunk replication.

Since we can set $n = m$ such that $n/m = 1$, the storage space required will be the size of the file, hence the IDA is space-efficient. The IDA achieves the optimal level of space efficiency when its calculations are performed within a Galois Finite Field.

The mathematical explanation of the dispersal and recombination stages of the IDA, as well as an explanation of the Galois Finite Field are provided in Appendix A.

2.3 Tools

2.3.1 Languages

Java

The main programming language that was used to write Osiris was Java. More specifically, I made use of the Oracle JDK6 SE[1]. This decision was made for a number of reasons relating to my requirements.

- Given the specifications of my project, it was sensible to select a high-level, object-oriented programming language.
- My system needs to support concurrency, and Java provides support for locking through use of `wait()`, `notify()` and `synchronized`.
- The nature of Osiris is that it is a distributed system and Java has very good support for serialisation of objects and input/output.
- I already have some experience in writing Java code.
- Successful peer-to-peer and database libraries are available for Java.
- The Java Cryptography Extension is part of its standard library which provides implementations of symmetric encryption functions and cryptographic hash functions.

SQL

The controller of my system requires a database to store the network's metadata. The library that I chose to use (see Section 2.5.2) uses SQL to interact with this database and so I needed

[1] `http://download.oracle.com/javase/6/docs/index.html`

to refamiliarise myself with the basic constructs of this query language. I also needed to learn how to use some more advanced transactional SQL commands which allowed me to create, isolate, commit, and rollback transactions.

2.3.2 Development Tools

Integrated Development Environment

Any project of this scale can quickly become large and unmanageable without the aid of an integrated development environment. The one I selected for implementation and testing was Eclipse 3.5 (Galileo)[2].

It provides some very advanced refactoring tools and builds automatically so that any compile errors will be reported in-situ before I attempt to run the program. Eclipse also has an inbuilt debugger which allows conditional breakpoints and step-through of programs.

Revision Control

In a project as large as this there are times when changes to a file need to be reverted or a file needs to be recovered entirely. For this, I made use of the revision control system Subversion[3]. With this I was able to ensure all machines had the latest version of the code base when working with multiple operating systems.

Backup Strategy

Given the nature of my project, it would be somewhat ironic if I had not employed a competent backup strategy to protect my files. This strategy was devised to ensure that I had a working copy of my files at most one day out of date given any potential failure. My primary backup source is on the Public Workstation Facility (PWF). This is where my SVN repository is based. I also perform daily SVN updates onto my personal machine and onto my external hard drive.

My SVN repository is also mirrored onto the Student Run Computing Facility[4] using the `svnsync` utility. This serves as an additional backup source which is at all times identical to the repository state on the PWF.

2.4 Software Engineering

Java is an object-oriented programming language and as such there are clearly defined good practices for software development. In my project I attempt to adhere to as many of these as possible, leading to maintainable, self-explanatory code. Java provides many opportunities for high levels of abstraction, inheritance and subtype polymorphism.

[2] `http://www.eclipse.org/galileo/`
[3] `http://subversion.apache.org/`
[4] `http://www.srcf.ucam.org/`

Java's use of modifiers also provides opportunity for high levels of encapsulation. All of these techniques, when used correctly, lead to a system with low coupling and high cohesion between its classes.

Since this project had a time scale of eight months with an unmovable deadline and a set of requirements that was unlikely to change throughout development, I decided that the best software lifecycle model to use would be *the waterfall model*. This is a non-iterative solution which dictates that every stage should be 100% complete before moving onto the next (see Figure 2.3).

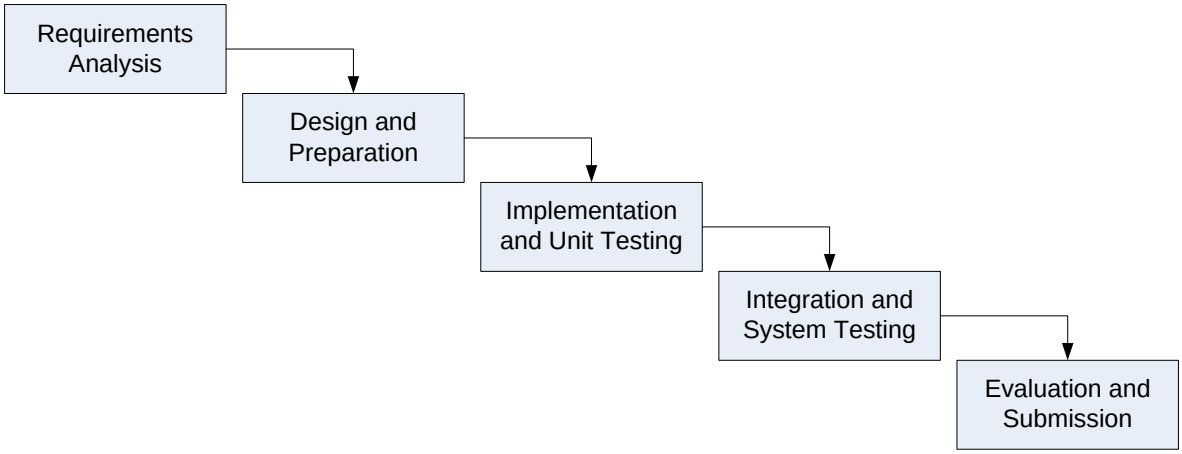


Figure 2.3: Waterfall model used over the course of this project.

In practice, iterative and agile development methodologies were also suitable in the implementation stage of the project. Namely, I practiced feature driven development, which was used frequently in order to ensure that my system was in a working state as often as possible. This allowed progress to be observed at regular intervals and enabled earlier testing of the systems components.

2.5 Software Libraries

2.5.1 FreePastry

As mentioned in Section 1.2, Osiris requires a layer of message-oriented middleware. Also, by its design as a peer-to-peer application, each node on the network requires the ability to communicate with every other node. To facilitate this, I decided to use FreePastry[5], which is an implementation of Pastry [9]. Pastry is described by its creators as a "generic, scalable and efficient substrate for peer-to-peer applications"[5].

Each node on a Pastry network has its own unique identifier in a circular 128-bit space. Message passing is performed through key-based routing where each node can be reached in $O(\log n)$ hops. However, to increase the security of my system, I chose to send messages directly, making use of the node handles (essentially an IP address and port number) that

[5] `http://www.freepastry.org/`

FreePastry provides. This decreases the ease with which a client can eavesdrop on communications. All node handles are provided by the controller as part of Osiris' transactions. The node handle of the controller itself is received upon joining the network.

One other important use of FreePastry was in opening a socket between the sender and receiver of a file. Files were sent direct between nodes to avoid flooding the network with too many large messages.

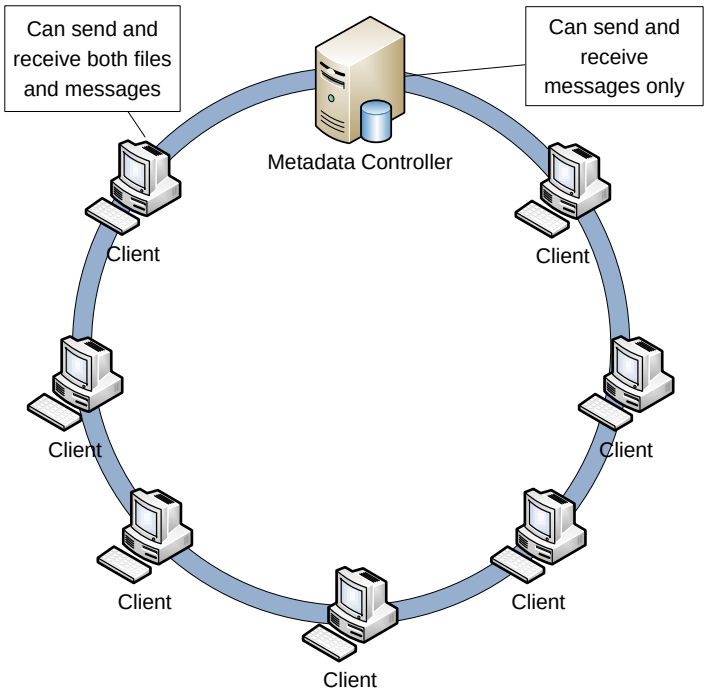


Figure 2.4: Illustration of the methods of communication between nodes on the Pastry ring.

To become familiar with the API, the developers had provided a tutorial to learn the basics of FreePastry. Unfortunately, the documentation of the API is rather poor and many hours were spent over the course of the project trawling through web forums and the code itself for answers. The most difficult part of this was in tweaking the network settings so that it was able to communicate with the network through firewalls and NATs.

2.5.2 Hyper SQL Database

My metadata controller required a database to store details of nodes, files, chunks and their locations. To deal with this, I needed a database solution that could interact with Java's own database connectivity (JDBC), was fully multithreaded, supported transactional SQL, and had customisable isolation levels. I found that the relational database manager HSQLDB[6] fulfilled each of these criteria.

[6] `http://hsqldb.org/`

2.5.3 SWT

Although a graphical user interface was not required for this project, I decided that it would be the most intuitive way for a user to interact with the client. Since I also designed Osiris to be a cross platform solution, I needed a graphical widget toolkit that supported this. The two best options were Swing and SWT[7].

The Eclipse Foundation says "The primary design goals [of SWT] are high performance, native look and feel, and deep platform integration."[7] Since this matched the goals of what I was trying to achieve, I decided to use the unfamiliar SWT despite having some previous experience with the Swing toolkit.

Summary

In this chapter I have discussed the substantial amount of work undertaken prior to the implementation of Osiris. The tools, libraries, and development methodology were described and their uses in Osiris made clear. An explanation of deduplication, convergent encryption and Rabin's IDA was given so that the reader will understand many of the implementation decisions in the next chapter.

[7] `http://www.eclipse.org/swt/`

Chapter 3

Implementation

Osiris currently exceeds 12,000 lines of code across 127 classes. Both client and controller are completely functional and achieve all objectives set out for the project.

This chapter discusses how the requirements in Section 2.1 have been implemented. First, I discuss the basics of inter-node communication, before describing the transactional protocols which underpin the system. I then move on to a discussion of the complexities behind deduplication and the information dispersal algorithm extensions. I conclude with explanations of client authentication, customisable security, and recovery from node data loss.

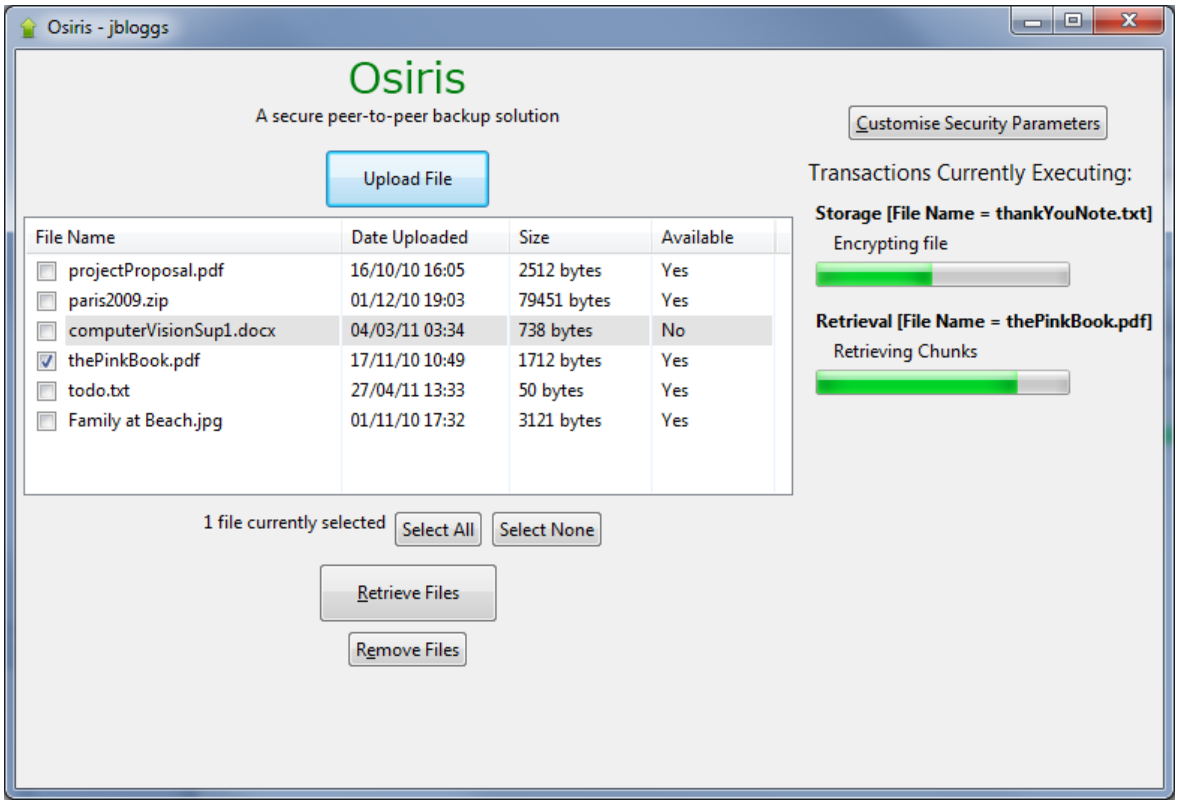


Figure 3.1: The graphical user interface for the main screen of the Osiris client.

A typical use case for Osiris is shown in Figure 3.2. This use case illustrates the interaction between all parties during the backup of a file to the network. Alice, Bob and Charlie are a group of friends who are each running the Osiris client application. The external cloud computing service (e.g. Google App Engine) is running the metadata controller for the network.

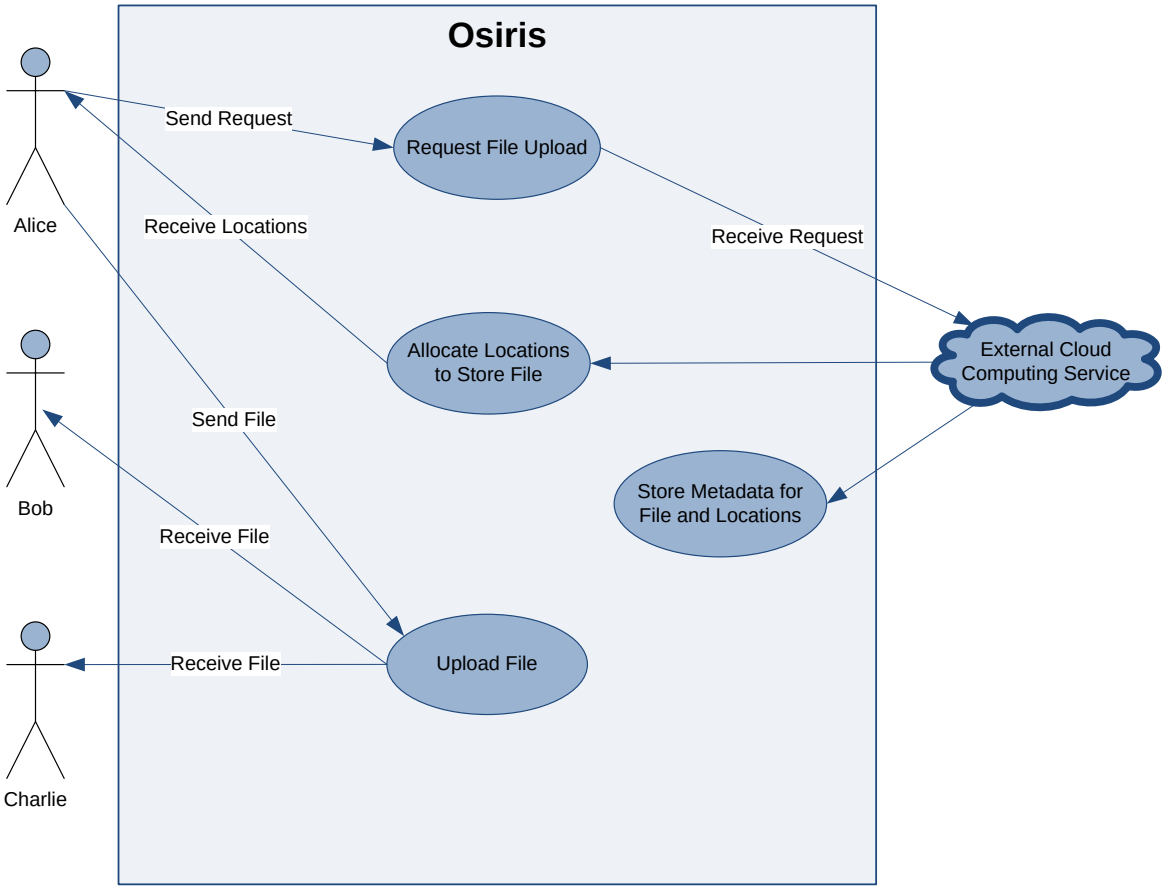


Figure 3.2: Osiris use case diagram – file storage.

Alice wishes to back up a file and so she informs the controller. The controller tells her to back up her file to both Bob and Charlie. It records this decision in its database. Alice then sends these files to Bob and Charlie for storage. The full protocol for this can be found in Section 3.6.2.

3.1 Client

The Osiris client is the program which users of the system run on their home computers. This program communicates with the controller and other clients to facilitate the backup and retrieval of files. The user interacts with the client through a graphical interface (Fig-

ure 3.1). It has been designed to be self-explanatory and provide the look-and-feel of the native operating system.

The user interface is updated by the various transactions and by the message delivery code when new information arrives from the controller. The user interface is an abstract concept and as such a limited console based interface was also implemented to enable scalability testing of my system.

The client performs most of the intensive processing required for Osiris, such as encryption, running the information dispersal algorithm and file transfer. This leaves the controller free to handle lightweight tasks permitting the network to scale without demanding greater resources from the central coordinator.

3.2 Controller

An Osiris network is comprised of multiple clients and a single controller. The always-on controller has two main functions within Osiris: it acts as a point of control for the transactions and as a metadata store.

The controller plays a part in each transaction as they all need to read or update the metadata store. The controller permits this to occur concurrently such that the store remains in a consistent state at all times. A significant portion of this is achieved through the database isolation model and also through inbuilt concurrency primitives.

The controller's role in a transaction is to process the file metadata to allow a client to achieve its goal. For example, in a retrieval transaction, the controller must find which chunks are part of the file and where they are stored. It then observes which nodes are currently online and sends only those locations which are available back to the client.

The controller is a single point of failure in this design, since no transaction can occur without communicating with the controller. However, in case of failure, I of course wanted the network to be recoverable. This crucially requires ensuring metadata durability. I have chosen to use a database for this, and explain it in depth in Section 3.5.

The controller does provide a command line user interface but this is mainly for the benefit of maintenance and debugging. Under normal operation the controller does not require any input beyond that of its properties file. Because of this the controller can be run silently on any server, even those of an untrusted third party. This is possible because, while I trust the controller to store the metadata for files, I do not trust it with user data. This means that no backup files ever pass through it – whether encrypted or not. The only data it receives is network status information and file metadata. However, it should be noted that I do trust the controller to store and retain this metadata.

Liveness Controller

The controller provides a `LivenessController` thread running at all times. This manages which clients are currently online and how much space they each have available to back up

chunks. It periodically checks which nodes remain online using FreePastry's API. After a node has gone offline it is not used in any subsequent transactions.

So that clients always have the latest file availability information, the liveness controller calculates which files are available on the network whenever a node appears or disappears from it. This latest information is then passed on to the clients.

LivenessController.java

```
public synchronized boolean hasSpace(Id nodeId, int changeBy) {
    if (nodeId.isAlive()) {
        long availableSpace = nodeId.availableSpace;
        if (availableSpace + changeBy >= 0) {
            // There is space for operation
            nodeId.availableSpace = availableSpace + changeBy;
            return true;
        } else { // No space for operation
            return false;
        }
    } else { // Node not alive
        return false;
    }
}
```

Figure 3.3: Java code to check if a node has sufficient space to store a chunk.

A storage transaction running on the controller must allocate chunks to be stored on clients. As part of this, it must check that a particular client has available space to store a chunk. The liveness controller provides a method to do this. Since multiple transactions may be running simultaneously, this operation must be atomic to avoid interleaving errors. This is achieved using a concurrent programming technique similar to *test-and-set*. Some Java style pseudo-code shows how this works in Figure 3.3.

3.3 Communication

Osiris makes use of the FreePastry software library for peer-to-peer communication between all nodes on the network. All communication is sent directly to the destination node without passing through any intermediate nodes along the way. It is more efficient to use directed traffic to the recipient node, with the added benefit that message interception becomes more difficult for rogue clients.

There are two types of communication used in Osiris: message passing and file transfers. A detailed discussion of these and an explanation of my protocol to limit the number of simultaneous file transfers can be found in Appendix C.

3.4 Configurability

It is good practice to create a system that is fully configurable. As such there are two configuration files in Osiris, one containing user properties and the other containing system properties. All clients and controller have a copy of both which each user can edit to suit their preferences.

Many properties can be changed on a per user basis, such as whether to use encryption and the strength of the encryption, username and name, controller address, and the amount of space to make available to the network.

There are also a number of global properties which the controller can set, such as whether to use deduplication, whether to use the IDA and its values of m and n, the replication factor for chunks, the size of the chunks and the storage allocation policy.

The effects of these different customisations upon the network are evaluated in Chapter 4.

3.5 Database

The metadata for Osiris is stored in an SQL relational database on the controller. The backend database is provided by the HSQLDB library discussed in Section 2.5.2. JDBC connections are used to run SQL statements against the database. HSQLDB is fully multi-threaded and provides transactional SQL. A transaction commits or rolls back its database updates when it terminates.

An entity relationship diagram for this metadata database is shown in Figure 3.4. It is clear to see that a file is owned by a single node, a file is made up of many chunks, a chunk can be shared between many files (with deduplication), and a chunk is stored in many locations.

Most of the fields are self-explanatory, but I will discuss some of the ambiguous ones here. The `chunkNumber` field refers to the order in which the chunks must be reconstructed to recreate a file. The `authenticationValue` is the Y value used in authentication, discussed in Section 3.7. The `encryptedHash`[1] is a hash of the encrypted chunk (excluding its header). This hash is used to uniquely identify a chunk for deduplication.

Many Osiris transactions follow the general pattern: 1) update the database, 2) transfer files, 3) commit. However, Osiris requires an isolation model that locks database tables as little as possible to retain a high level of concurrency. A *two-phase locking* concurrency control model could not be used because conflicts are common, and so database tables will be locked often, while files are transferred.

Instead I chose to use lock-free *multiversion concurrency control* with *snapshot isolation*. With this model, multiple versions of the database exist at any one time. When a transaction begins, it takes a snapshot of the database in its present state. This ensures that its view will remain in a consistent state for the duration of the transaction. However, its updates will not be seen by other transactions until it commits.

[1] This hash of the ciphertext is *not* the same as the hash of the plaintext used as the key for encryption.

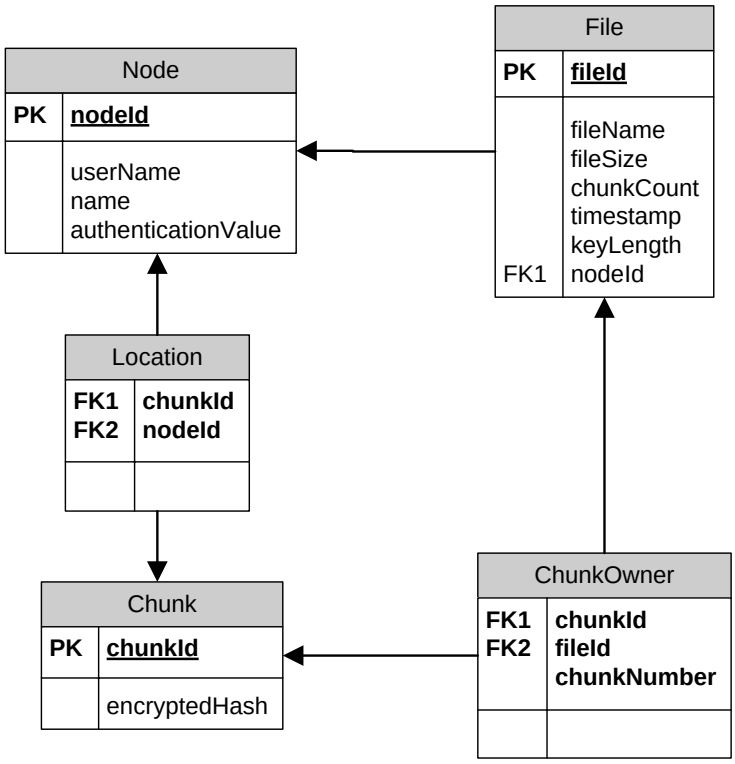


Figure 3.4: Metadata Entity Relationship Diagram.

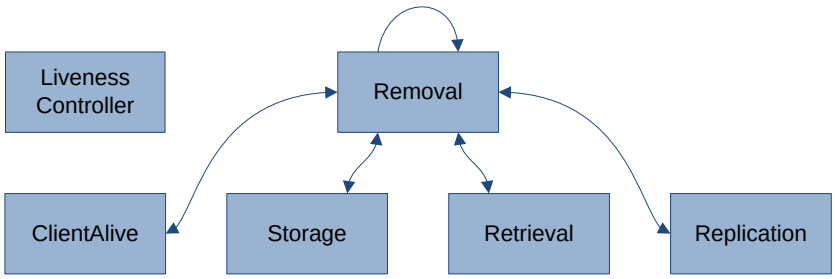


Figure 3.5: Conflicts between classes that access the database.

However, this model means that interleaving transactions may at times have an out-of-date view of the file data. This is prevented programmatically by the transaction manager to avoid running conflicting transactions at the same time. Figure 3.5 shows the conflicts between all classes that access the database. It can be seen that removals must be run in isolation of all other transactions.

Amdahl's law tells us to optimise the common case, so while it is not ideal that removals can only occur when all other transactions have terminated, it is more efficient to do this than introduce complex database locks which will block storage and retrieval transactions (which are the common case).

In order to decrease the overhead of opening and closing database connections for short lived transactions, I have implemented a database connection pool. When a transaction wishes to access the metadata store, it must first wait for a database connection to become available. This is achieved through the use of the *wait-notify* paradigm on the database connection pool. To avoid deadlock, it is important that a transaction only requests a database connection after it has been given permission to run by the transaction manager.

3.6 Transactions

Most user operations in Osiris, such as removing a file from the network, spawn a new transaction. This section discusses the design of the protocols that underpin these transactions. Both the client and controller have their own unique implementations for their part in the following transactions. Table 3.1 shows the transactions currently supported by Osiris. Note that I only explain the storage transactions in detail here, while details of other transactional protocols can be found in Appendix B.

Transaction	*Function*	*Section*
Client Alive	Run when a client logs on to the network.	B.1
Storage	Allows a file to be backed up onto the network.	3.6.2
Incoming Storage	Created when a client is asked to store backup data. This is part of the storage protocol but specifies the implementation of the client receiving chunks. The client receives a chunk and then stores it in a temporary directory until asked to commit the transaction.	3.6.2
Retrieval	Allows the recovery of a file from the network in case of loss.	B.2
Removal	Allows the removal of backup copies from the network if they are no longer required or to gain more space.	B.3
Replication	Replicates an existing backup chunk to a new location.	B.4

Table 3.1: Explanation of Osiris' transactions.

Each of these implements my `Transaction` interface and so is obliged to define at least the following three methods: `run()`, `commit()` and `rollback()`. Osiris creates and runs new transactions as they are required. Since many (but not all) transactions also implement the `Runnable` interface, some way of limiting the number of active threads running in the JVM is necessary. Fortunately, Java provides an implementation of a fixed size thread pool which my transactions are submitted to. If no threads are available, they are fairly queued and run when a thread becomes available. To avoid deadlock, a separate thread pool is used to execute a `ChunkTransfer` (which also implements `Runnable`) since these are started within other transactions.

Each transaction registers itself with the transaction manager upon startup, requesting permission to run. The transaction manager will then at some point in the future either commit or roll back this transaction.

3.6.1 Transaction Manager

The transaction manager's primary job is to keep track of which transactions are currently executing and to ensure that all transactions eventually terminate.

Java does have a transaction API (JTA), but I decided not to make use of it because it is inherently aimed at distributed transactions with JDBC database connectivity. Since Osiris only has a single database instance in one location, JTA it is an overkill solution. Instead, I created my own lightweight transaction manager that is thread-safe and provides all features necessary for Osiris transactions. Each client runs its own manager which looks after all uncommitted transactions that the client plays a part in.

Transaction synchronisation between nodes occurs through message passing. Of special note is the `EndTransactionMessage` which specifies whether to commit or roll back the transaction. This decision is made through ideas borrowed from the atomic *two-phase commit protocol* [10]. This protocol comprises of two phases:

The Commit Request Phase All participants in a transaction are asked whether they would like to commit or roll back the transaction.

The Commit Phase The coordinator of the transaction collects the responses from the participants. If and only if they all vote to commit is the transaction committed. The coordinator notifies all participants of the decision. Note that the coordinator is also a participant and so must vote.

The coordinator for this protocol is different for each type of transaction[2]. For example, in the *storage* transaction it is the client who requested the storage. Also, the voting is often implicit. For example, again in the *storage* transaction, if a client storing backup chunks acknowledges that it has received all chunks, then it is implied that it is ready and waiting to commit. If there were to be an error in file transfer, the coordinator would be notified by FreePastry's API. This would be counted as a vote to roll back the transaction if it was unrecoverable.

Because there are so many opportunities for error in a distributed system, rollbacks are not discussed in much depth in this dissertation. However, they are handled effectively and cleanly for all transactions. A rollback restores the system to the state prior to the transaction's execution. This often requires rolling back database updates, deleting temporary directories and informing the user of the error.

In the worst case, my transaction manager also operates a timeout facility. If the time elapsed for a transaction exceeds the timeout value, then the transaction manager votes to rollback the transaction. This is required in case of node failures or if a message goes astray.

It is sometimes the case that a transaction is able to complete successfully even when necessary participants are offline. This is implemented by temporarily storing messages in a message queue. Before a client is allowed to participate in any new transactions, it must receive each of its queued messages one-by-one and in the order that the transactions sent them. An example use case is if a chunk needs to be removed from the network, a

[2]The transaction coordinator is not the same as the controller.

`ChunkRemovalMessage` can be held in a message queue and sent to the client when it next appears online.

3.6.2 Storage

This is the most significant transaction which the system implements on both the client and controller side. This transaction allows a user to backup a file to the network completely autonomously.

An instance of this transaction is created when a user clicks the upload button and then selects a file (or files) from the presented file dialog. Figure 3.6 shows the underlying protocol for this transaction. The three parties are the client uploading the file, the metadata controller allocating storage locations for the file, and those clients requested to store the file.

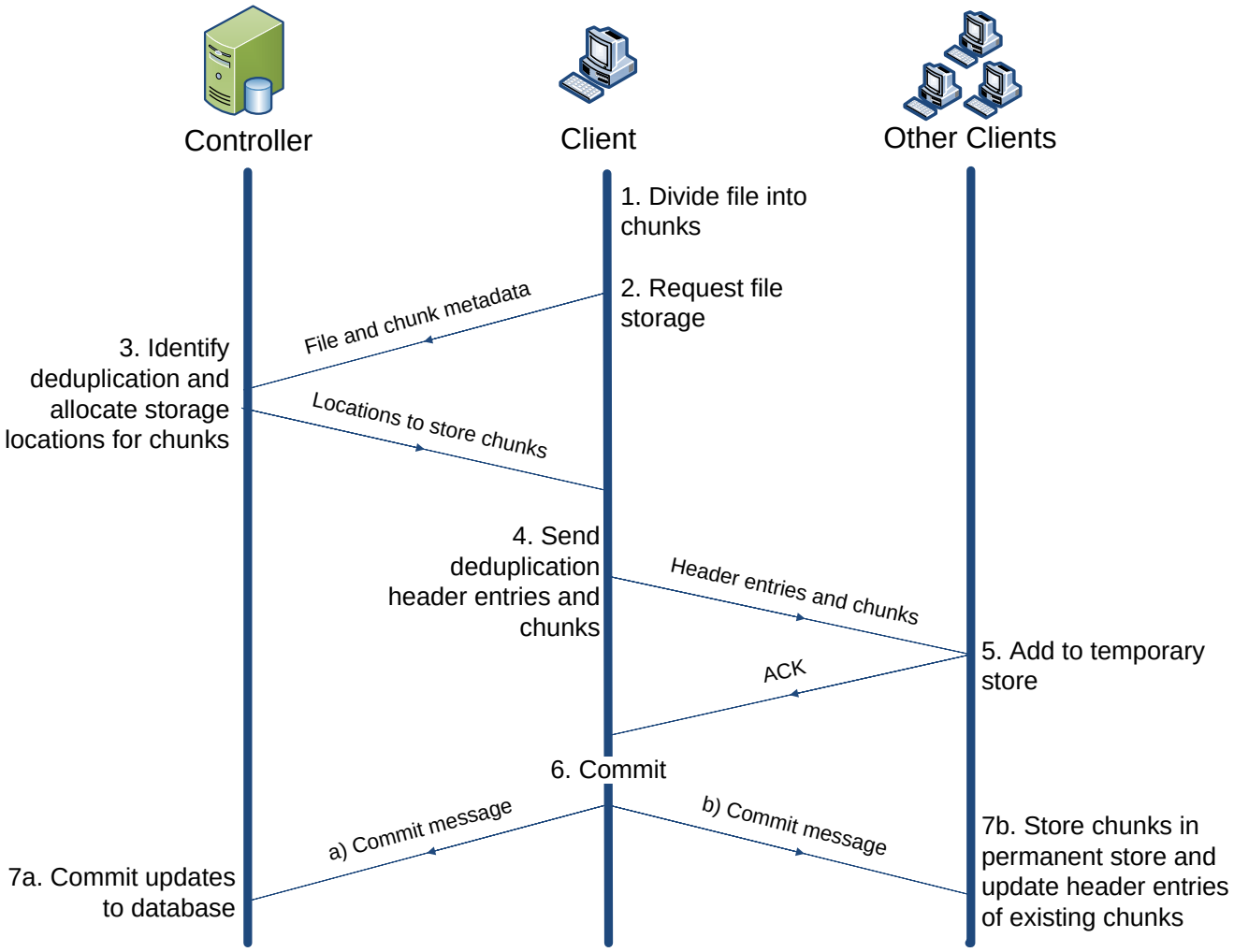


Figure 3.6: Storage Transaction Protocol.

1. The storing client initially must divide the raw plaintext file into chunks. According to the configuration settings, the client will either divide the file into sequential fixed size chunks, or use the IDA to create the chunks. Next, the chunks will be individually encrypted by convergent encryption (if required by the user).

2. The client will send the file and chunk metadata to the controller. This includes the chunk sizes and the hashes of their encrypted chunks.

3. Upon receipt of this metadata, the controller will allocate storage locations using one of the storage policies described below. To do this allocation it must consider a number of network properties.

   Replication Factor If the network replication factor is n, then each chunk must be stored in n separate locations. To satisfy this, the controller must be able to ensure there are at least n clients online, with sufficient storage space. The liveness controller's `hasSpace()` method is used to reserve this storage space for each chunk.

   Deduplication If deduplication is switched on, the controller must be able to detect whether any of the new chunks already exist on the network. If they do, then the existing chunks are associated with the new file in the `Chunk Owner` table of the database (Section 3.5).

   The controller detects whether two chunks are identical by comparing a hash of their ciphertext (which must be identical due to convergent encryption). The hashes of the existing chunks are stored in the `Chunk` table of the database and the new hashes are sent with the chunk metadata in step 2.

   In the case that two chunks are identical, rather than storing new locations of the chunk, a new header entry must be added to the existing chunk locations to indicate this chunk has a new owner. This header entry is the decryption key for that file, encrypted by the user's password. Step 4 describes how these header entries are added.

   Further complexities that deduplication introduces into the storage protocol are discussed in Section 3.8.

   Block Lists In Section 3.10, I discuss the need for preventing chunks from being stored on particular nodes. This constraint is handled by the controller when allocating locations for chunks.

   Finally, the controller sends the location data back to the client. For each chunk, this tells whether it should be added to the network in specified locations, or whether a new header entry should be added to existing locations.

4. Upon receipt of this location data, the client processes it so to only send one `IncomingTransfer` message to each recipient client. Each message consists of: the number of chunks the client is requesting to send and the header entries for each existing chunk the recipient client holds.

5. The recipient client adds the chunk header entries (if there are any) to a temporary store, not actioning them until transaction commit. It then follows the file transfer protocol (described in Section C.2) to permit the storing client to send its chunks, if there are any to send.

6. The commit for this transaction follows the two-phase-commit model described previously. In this transaction, the storing client is the coordinator. Once it has received an ACK from each of the recipient clients, stating that all chunks and header entries have been successfully received, it commits the transaction.

7. (a) The metadata controller commits all the new changes to the database. This includes the new file uploaded, the new chunks added to the network and the locations of these chunks.

   (b) The recipient clients move all the received chunks from temporary store to permanent store and then add the new header entries to the correct existing chunks.

After the transaction has completed, the graphical user interface is updated to add the newly uploaded file to the file table.

Storage Policies

The controller side of the Storage transaction must allocate which chunks should be stored on which nodes. It does this by making use of one of the following storage policies (according to which one has been selected in the properties file).

Round Robin Allocates chunks to nodes in a round robin fashion. This means that each online node should receive approximately the same number of chunks in a single transaction.

Random Allocates chunks to nodes in a random fashion. No guarantees are made regarding the distribution of chunks.

Most Space Allocates chunks to nodes based upon the available space a node has. Firstly, recall from Section 3.2 that the liveness controller keeps an up-to-date record of the available space each node has. The online nodes are here stored in a priority queue with a comparator ordering them by available space. When the storage transaction wishes to get the next node to allocate to, it simply takes it from the priority queue.

3.7 Authentication

To prevent any random Osiris user from joining any network, and to protect the system against attackers who spoof node IDs, clients must authenticate themselves to the controller before being permitted to communicate with other nodes. If a client has not been authenticated, all of the messages it sends are dropped by the receivers.

My original ideas for authentication used a challenge-response protocol between controller and client that required some shared secret. However, a strength of my system is that a user can lose an entire computer and reinstall Osiris on a new computer. Then, with just their username and password combination, they can gain access to their backup data again. This means that the only secret a client has is the user's password.

Furthermore, I cannot permit the controller to have access to the password in any way because this password is used for file decryption and the controller is an untrusted party. Even a hash of the password, stored with the controller would make brute forcing the password significantly simpler, since it provides a way to check whether the password is correct with minimal computation.

Instead, a way to authenticate without revealing any information about the password to the controller was required. This is made possible with *zero-knowledge password proofs*.

3.7.1 Zero-Knowledge Password Proof

A zero-knowledge password proof is a method for a prover (a client) to prove to the verifier (the controller) that it knows the value of a password without revealing anything other than the fact that it knows that password to the verifier.

This is an ideal solution to my problem as the controller need never know anything about the password, other than the fact that the client has it. A well known example explaining some of the ideas behind zero knowledge proofs can be found in a paper published by Quisquater *et al.* [11]. This section concentrates on the sigma protocol defining a ZKPP which I have implemented within Osiris. The protocol implemented is borrowed from Lum Jia Jun [12] and Camenisch [13].

Initialisation

First, a cyclic group G with some generator g is selected by the controller and shared with all clients. A cyclic group has the property that every element of the group is some power of g.

Registration

When a client first joins the network, it is registered with the controller. Part of this registration requires the client to provide an authentication value

$$Y = g^x$$

where x is a cryptographic hash of the user's secret password. The controller stores Y in the database.

Authentication

First, the controller generates a random value a which it sends to the client. The client then computes $Y = g^x$ and then selects a random element

$$r \in G$$

Using this element, the client then calculates

$$Q = g^r$$

Finally, the client produces two values (c, z) to send to the controller where H is a cryptographic hash function and

$$c = H(Y, Q, a)$$
$$z = r - cx$$

Upon receipt of this tuple, the controller calculates its own value of Q using

$$Q = Y^c g^z$$

Note that this alternative equation for Q does not require the secret x. Finally, it can then verify that $c = H(Y, Q, a)$. If this property holds, the client is authenticated.

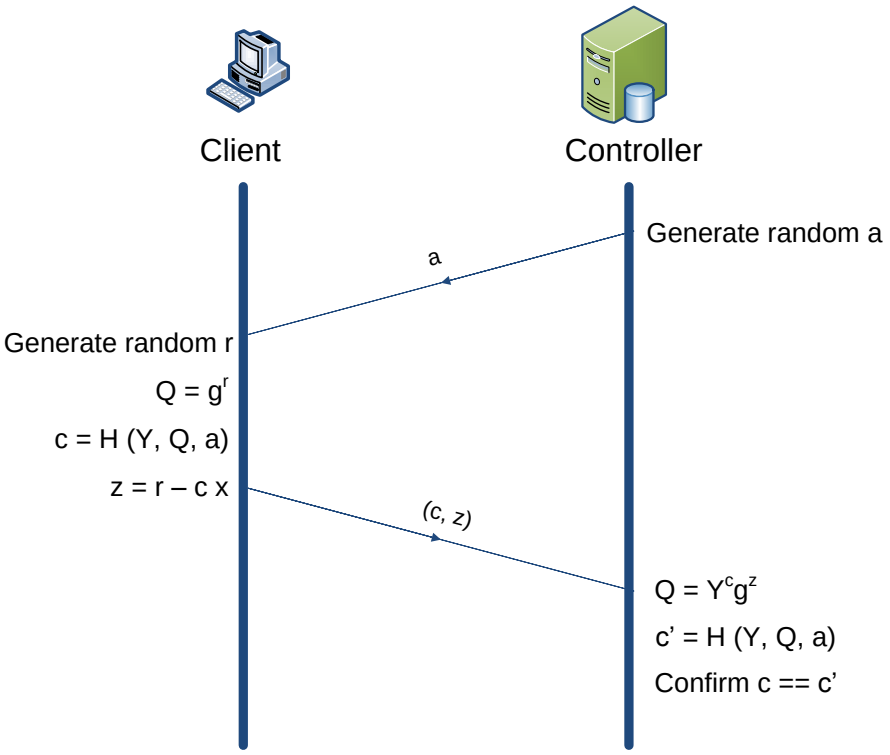


Figure 3.7: Zero-Knowledge Password Proof Authentication Protocol.

My implementation of this sigma protocol makes use of Java's `BigInteger` class which allows integers of an arbitrary size. This is useful when dealing with large groups and performing arithmetic on 512-bit hashes.

Communication occurs through message passing and the authentication value Y is contained within the `ClientAliveMessage`, as this process is part of the Client Alive protocol. Figure 3.7 shows the communication that occurs between the client and the controller during authentication.

3.7.2 Proof of Concept

Again from Lum Jia Jun [12], by considering the two definitions of Q we wish to prove that

$$g^r = Y^c g^z$$

Proof. During registration we defined $Y = g^x$ and during authentication we defined $z = r - cx$. By substitution

$$\begin{aligned} g^r &= Y^c g^z \\ g^r &= (g^x)^c g^{r-cx} \\ g^r &= g^{cx} g^{r-cx} \\ g^r &= g^{cx+r-cx} \\ g^r &= g^r \end{aligned}$$

□

3.7.3 Proof of Security

A cyclic group has infinitely many solutions of $x = \log_g Y$ because g^x is periodic. My value of x is a SHA-512 hash of the user's password. This large value, together with a large cyclic group ensures the discrete logarithm problem is very hard to solve. The cyclic group that my system implements is $(\mathbb{Z}/p\mathbb{Z})^\times$ with generator g. This group is known as the multiplicative group of integers modulo p.

My choice of a large p and x stems from the proven result that $\Omega(\sqrt{p})$ is a lower bound for solving the discrete logarithm problem. This was shown by Shoup [14]. From this, finding an x such that $Y = g^x$ is computationally hard.

3.8 Deduplication

Since deduplication lies at the heart of Osiris, the implementation details of it, and how it interacts with convergent encryption, can be found throughout this chapter. Important details behind its operation are provided in Section 3.6, where the transactional protocols are described.

Deduplication is a space conserving technique which requires very little additional processing when storing a file to the network. In fact, it is almost always significantly more

efficient because duplicate files do not need to be transferred over the network twice. The rest of this section concentrates on the problems deduplication introduces and the solutions I have implemented to fix them.

Consistency of Chunk Headers

Deduplication has introduced a great deal of complexity into the storage and removal algorithms. Before, handling the metadata was simple; nothing was shared. With deduplication, strong consistency of the database became very important because chunks were shared with multiple files. But it is not just the metadata store that must reflect the fact that chunks are shared. The headers of all the (encrypted) chunks must now contain an entry for each owner[3]. This means that the chunk headers and metadata store must be kept synchronised.

One problem introduced by deduplication occurs when a chunk is being deduplicated and so the new owner needs to add its header entry to all existing chunk locations. But, at least one node storing the chunk might be offline at the time of this storage transaction. The solution implemented in Osiris is to store the header entry temporarily with the controller. When the offline client next appears online, the chunk header entry will be sent using the transaction manager's offline message queue implementation. Figure 3.8 visualises this interaction.

Unfortunately, this solution requires an additional pair of messages between client and controller, requesting the necessary header entries to send in the messages. No security is lost in temporarily storing the header entries with the controller, because it never sees any associated file data. Hypothetically, if it were to gain access to the file data, it would then have access to the entire header anyway.

The Self-Backup Problem

I will now discuss another problem that deduplication introduces. First, recall that the replication policy of the network is how many replicas of each chunk there should be.

If Alice is uploading a chunk that already exists on the network, then it must be deduplicated. This means she needs to become an owner of all the existing replicas of that chunk. However, Alice might be holding one of those existing replicas herself. As such, she would be effectively backing up data to her own machine. To resolve this problem, a new replica needs to be created on a new node. However, just storing the chunk to be backed up on a new node (which would be simple) is not possible because it would end up missing the pre-existing chunk header entries.

The solution chosen in this project relies on the fact that, since Alice is already a backup source for this chunk, she must also have the most recent header data for it. Instead of transmitting the new chunk that is being backed up, Osiris transmits the existing copy of that chunk from Alice's backing store.

[3]Recall that a chunk header contains the chunk's hash encrypted under each user's specific key.

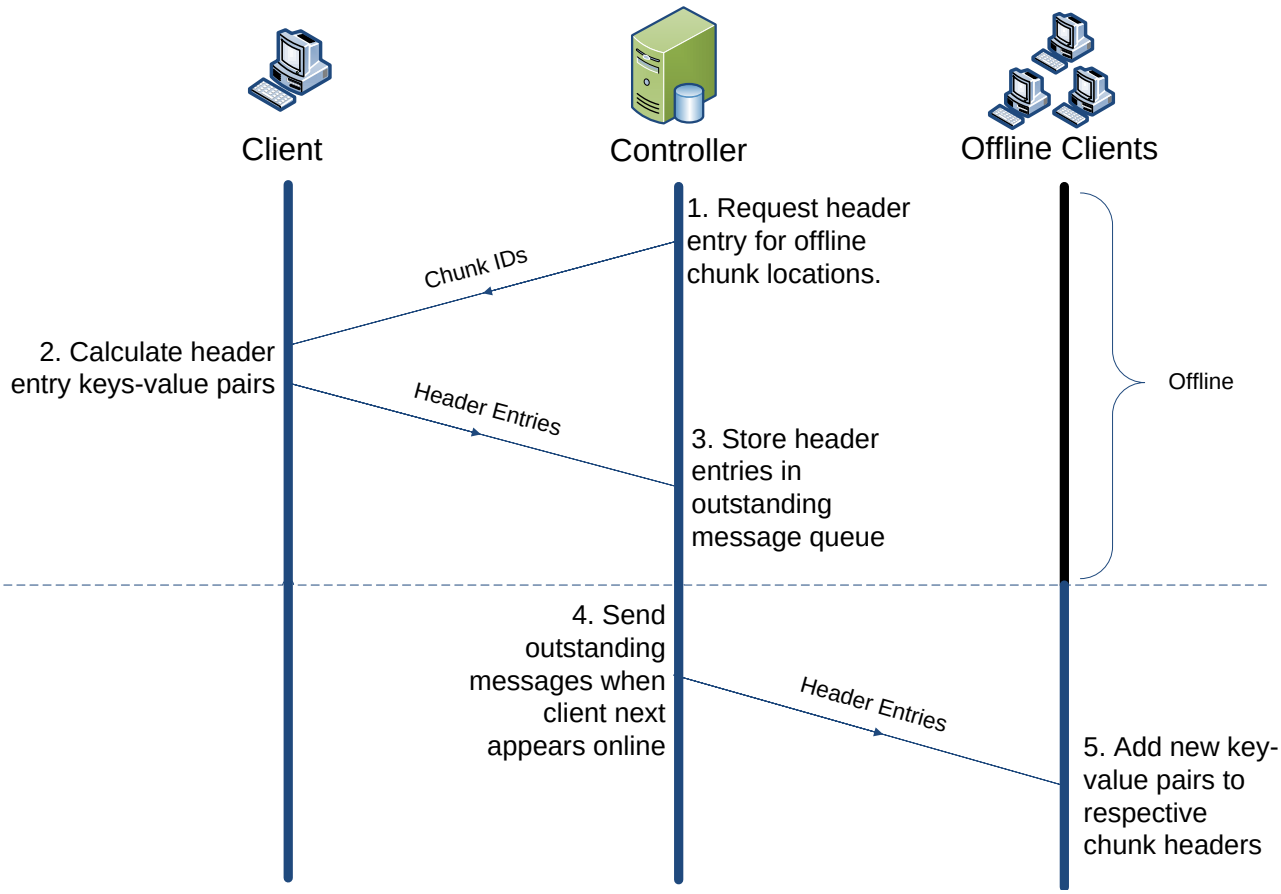


Figure 3.8: Deduplication – Updating chunk headers of offline clients.

Chunk Header Security

Deduplication also introduced some interesting security concerns. Specifically, if a chunk is deduplicated, a header entry exists for each owner of that chunk. Upon retrieval, a client must be able to calculate which header entry is its own. Storing this information with the controller would be difficult, because multiple clients storing the same chunk might update their header entries in a different order due to network delay.

The owner of each header entry should therefore be stored within the header itself. My implementation uses a serialized `HashMap`, mapping from the owners identity i to its encrypted decryption key k. However, the value of i must be chosen so not to reveal any information about the chunk's metadata. For example, setting $i =$ `nodeId` would reveal all owners of that chunk to an attacker.

Instead, Osiris sets $i = H(\texttt{nodeId}, Y, \texttt{chunkId})$, where Y is a function of the user's password introduced in the zero-knowledge password proof in Section 3.7.1. A node uploading/retrieving a file will know all three of these values if it is an authorised owner of that chunk. This is secure because, although an attacker can know the `chunkId`, and even possibly the `nodeId`, he has no way of knowing the value Y. The value of Y is specific to each

client and an attacker could only gain access to Y if he knew the client's password or had compromised the controller (see Section 4.4.2).

3.9 File Availability

Osiris offers whole-chunk replication or an information dispersal algorithm in order to ensure a high level of file availability while keeping space utilisation as low as possible. This section discusses these techniques and how the IDA is used with deduplication.

3.9.1 Replication

Whole-chunk replication is a method Osiris uses to ensure a high availability of data in the system. For example, without replication, if a node goes offline containing a chunk a user requires, they will be unable to retrieve that file. However, if two or three replicas of each chunk are stored, he is more likely to be able to restore the file.

Replication provides reliability and fault tolerance to the system at the cost of increased network communication and storage space. However, an ideal solution would provide us with all the benefits of replication without the significant space usage costs. For this we look at Rabin's Information Dispersal Algorithm.

3.9.2 Information Dispersal Algorithm

I initially implemented the IDA and Galois fields as per their descriptions in Appendix A. However, I ran into some difficulties in ensuring that the potentially huge matrices did not all need to be present in memory at once. To help me with this, I found an existing implementation of the algorithm written in Python [15]. Since I was unfamiliar with this language, and due to Python's lack of explicit types, I spent some considerable time porting its ideas into Java.

After completing the IDA implementation, I found it to be operational yet inefficient. I installed the *YourKit*[4] Java profiler to analyse where the problems resided. The profiler allowed me to find and optimise many inefficiencies of Osiris but one significant optimisation was the introduction of file buffers for the IDA algorithm. These allowed multiple rounds of IDA to occur before any bytes were written to the file chunks. Similar buffer logic was written for the retrieval algorithm.

My implementation of the IDA is easily extensible for use with any field (not just $GF(2^{16})$) and is fully compatible to use alongside deduplication, as discussed in the next section.

Given that the main purpose of the IDA is to achieve a high availability and fault tolerant solution to file storage, it is most effective when the storage policy is *round robin*. This policy ensures that the number of nodes receiving chunks is maximised.

[4] `http://www.yourkit.com/`

Client-side, the storage protocol was modified to provide two methods for generating chunks. The original method was by splitting them into fixed length chunks and the new method is creating them with the IDA. The retrieval protocol was changed to provide analogous behaviour for file reconstruction. It also required editing to request just m chunks as opposed to all n.

3.9.3 Deduplication of IDA Chunks

Now that IDA and deduplication have been implemented separately, I will consider how to use both of them together in order to minimise the space used while maintaining a high availability of data. First, consider the requirements:

1. The convergent encryption process must occur after the IDA, since the IDA creates the chunks and each chunk should be able to be encrypted.

2. The IDA must be deterministic, that is, for two identical files, the IDA should produce two identical sets of chunks. This is necessary so that if multiple clients upload the same file, the chunks will be identified as the same by deduplication.

The first is easily achieved through by restructuring the storage protocol. The second is slightly harder. Three factors influence the IDA protocol: the values of m and n, the alpha values and the file itself. These must each be constant for identical files to produce identical chunks. Ensuring this for m and n is easy, since they are already preset by the controller configuration file. The backup file itself is trivially a constant for identical files. Therefore only the alpha values remain variable.

Previously to this the alpha values chosen for the IDA were selected at random. This produced values that were *almost surely* linearly independent [8]. However, they are clearly not deterministic. Algorithm 1 shows a scheme Rabin suggested [8] for producing alpha values in a deterministic way, which he has proven to always result in a non-singular matrix.

Algorithm 1 Rabin's scheme for deterministically producing alpha values

First, create sets $x_1 \dots x_n$, $y_1 \dots y_m$ such that

$$\forall i, j \cdot x_i + y_j \neq 0$$

$$\forall i, j \cdot i \neq j \rightarrow x_i \neq x_j \wedge y_i \neq y_j$$

then our alpha matrix, with rows as alpha vectors, is defined as

$$A = \begin{pmatrix} \frac{1}{x_1 + y_1} & \cdots & \frac{1}{x_1 + y_m} \\ \vdots & \ddots & \vdots \\ \frac{1}{x_n + y_1} & \cdots & \frac{1}{x_n + y_m} \end{pmatrix}$$

Since my implementation performs arithmetic in $GF(2^{16})$ the operation $x + y$ is defined as $x \oplus y$. To satisfy the first condition we require $\forall i, j \cdot x_i \neq y_j$. To satisfy the second condition we require unique values within the set x and y. One such solution I have implemented is

$$x = \{1, 2, \ldots n\}$$
$$y = \{n+1, n+2, \ldots n+m\}$$

We now have constant alphas vectors and hence deduplication can be used in combination with the IDA.

3.10 Customisable Security

Customisable security was the third extension implemented for my system. It provides a way for users to specify on a file-by-file basis how secure they want their backup to be.

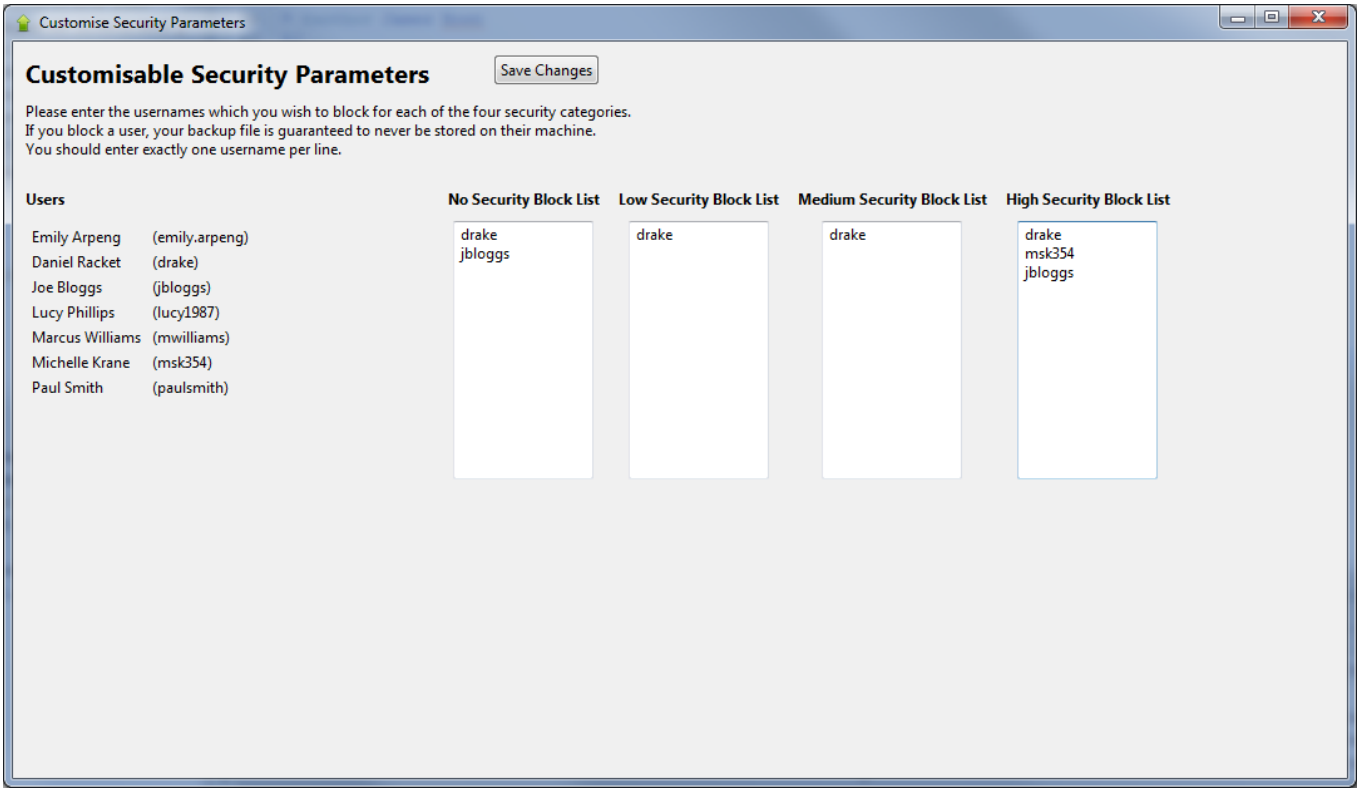


Figure 3.9: Customisable security graphical user interface.

For example, Alice might not care whether her backed up Christmas wish list is encrypted or not. However, she does want her spreadsheet of financial details to be very strongly encrypted. Bob might be an untrustworthy character and so, even with strongly encrypted data, Alice would like to prevent her backup files from being stored on his computer, even though he is on the same social backup network.

Customisable security solves both of these problems by providing four levels of security.

None The backup files are not encrypted.

Low The backup files are encrypted with 128-bit encryption.

Medium The backup files are encrypted with 192-bit encryption.

High The backup files are encrypted with 256-bit encryption[5].

Additionally, each of the four levels provides a *block list* where users can specify which users they do not wish to backup their files to. Figure 3.9 shows the graphical user interface controlling the block lists. The block lists persist between user sessions by reading and writing to the `user.properties` configuration file.

3.11 Recovering from Failure

With any client application, like Osiris, the users are not to be trusted. With this in mind, this section discusses how my system recovers if a user accidentally deletes all of the chunks they are supposed to be backing up for others.

The loss of backup files is detected within the *Client Alive* protocol. The client sends the controller a list of all chunks it currently has and the controller compares those to the metadata store to see if any chunks are missing. If so, the user is asked whether they are able to relink the chunks (for example, if the user has replaced their computer and forgotten to copy over the old directory). If this is the case then the user is refused log in since they are not allowed to enter the network in an inconsistent state. If the files are lost forever then action must be taken to restore them, wherever possible.

For each chunk lost, the controller identifies whether this was the last remaining location of that chunk or not. If so, the chunk is declared as lost forever (this should be very rare, however). All files which that chunk was a part of are removed from the network by running a (slightly modified) removal transaction. If the chunk does exist in another location then a replication transaction is spawned to recover the lost chunk at the earliest convenience. For more details on how these replications occur see Section B.4.

[5]Due to import-control restrictions, Java's unlimited strength cryptography policy files must be installed in order to encrypt at 192 or 256-bit levels.

Chapter 4

Evaluation

To evaluate my project, I required access to multiple machines that could run my client application. I decided to use Amazon Web Services Elastic Compute Cloud[1] (EC2). This service provides virtual machines, each with their own unique IP, to run clients on. I used a custom-built Ubuntu Server 8.04 image for my virtual machines. This contained Java and an SVN working copy of the Osiris client.

Both Osiris client and controller are fully cross platform. They have been tested thoroughly on Windows 7 and Ubuntu 10.10. They have also been successfully run on Mac OS X 10.6.7 and Windows XP.

4.1 Testing

Testing of any system is highly important to ensure the quality of the software produced. The testing process employed is tightly coupled to the development methodology used for development. Earlier, it was mentioned that I made use of feature-driven development during implementation. This led me to the testing cycle shown in Figure 4.1.

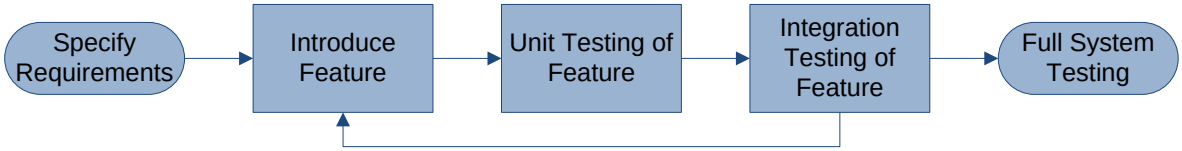


Figure 4.1: Testing Cycle for Osiris.

Throughout the development of this project, many of the components were developed and then tested in complete isolation from the rest of the system. This unit testing approach was an especially valuable tool for me to ensure that my `ConvergentEncrypter` and `IDA` classes were working as expected. After introducing a feature such as the IDA, it then needs to be integrated into the rest of the system. With this comes the integration testing. This

[1] http://aws.amazon.com/ec2/

ensures that the new feature works well with the system and especially that it has not broken any existing functionality.

For my full system testing, I created a testing framework which operates Osiris in a purely random mode and observes failures. The controller initially sends a message specifying the system parameters for the test to each online client. These clients then spawn random transactions at short random time intervals ensuring that some transactions overlap and others do not.

After a period of time, the test ends and the metadata store is compared with all client's chunks on the network to ensure that the two are consistent. Finally, the headers of all encrypted chunks are checked to ensure they contain the correct entries. This testing framework also helps me to evaluate the performance of Osiris by recording the time taken for the transactions that occur.

4.2 Success Criteria

Prior to undertaking this project, I specified success criteria so that in the future I would be able to determine whether my system achieved its goals. I now state these again with an explanation of how my system attains them.

1. *Decentralised peer-to-peer backup-file transfer and storage. This is shown to be working by successfully transferring a file to the network.*

   The system takes a file and initially divides it into chunks. It then transmits these to selected client nodes directly, without the chunks ever passing through a central server.

2. *Centralised or distributed meta-data storage describing status of online nodes, availability and location of data. This is shown to be working by successfully retrieving a file from the network.*

   The controller implements a Hyper SQL database for storing the network's metadata. It also provides a liveness controller for monitoring which nodes are currently online to allow it to determine the availability of data. By coupling the database and liveness controller, a controller can obtain the available locations of the necessary chunks. This information is passed to the awaiting client who then requests the chunks from the nodes to recreate the file.

3. *Confidentiality of backup files using convergent encryption. This is shown to be working by confirming that a file transferred to a remote node is stored as ciphertext. Then an attempt to retrieve it will once again yield the plain text.*

   Convergent encryption has been successfully implemented and all files stored to the network are stored as ciphertext. The symmetric key encryption function used is Java's implementation of AES. The system also abides by a strong security policy to protect users' files and metadata. Customisable security allows convergent encryption to work with multiple security levels. It also provides block lists to prevent storing a user's files with other users they do not trust.

4.3 Performance

This section analyses Osiris' performance in terms of time, space and availability. Recall that key goals of this project were in ensuring an efficient space utilisation whilst providing high file availability. The evaluation concentrates on showing that Osiris is an effective backup solution and reasoning how the performance is affected by the properties of an Osiris network.

4.3.1 Time and Space

For this evaluation, I use a 1MB chunk size. This selection is a trade-off between performance and security. With a large chunk size, the time taken to distribute and reconstruct files would be less. But, with a small chunk size, less file data is available to an attacker of the system. A scheme that produced variable sized chunks dependent on the file size could have been implemented. However, this would not work in conjunction with deduplication because chunks need to be identical to be deduplicated.

Throughout this evaluation, the *Round Robin* controller storage policy is used. Both the *Random* and *Most Space* policies were also tested but they delivered less favourable results for file availability by their very nature because the chunks for a single file are not necessarily distributed evenly across the network.

The results for convergent encryption and replication (below) were gathered by running 15 clients on 15 EC2 instances and then requesting each of them to upload files of varying sizes (between 10 bytes and 100 MB) to the network. Of the 15 trials for each file size, an average was taken to discount any variation in network latency and the averages are plotted in Figures 4.2 and 4.3.

Convergent Encryption

Figure 4.2 is plotted on a logarithmic scale to show the effects of the chunk headers on file size. However, as the size of a chunk increases, the small header size becomes negligible. Note that the space used to store a file grows linearly and is approximately equal to the size of the file uploaded (for single copy storages).

Figure 4.3 shows that convergent encryption significantly affects the time taken to store a file. An encryption storage transaction takes approximately four times longer than an unencrypted one. This predictable result is acceptable in order to maintain a user's data confidentiality. Additionally, a linear correlation can be observed between file size and the time taken for a transaction. The fact that larger file sizes do not cause performance degradation is an achievement of my system.

The reason that the 192-bit storage takes longer than the 256-bit storage is down to the underlying implementation of the AES algorithm and not a property of my implementation.

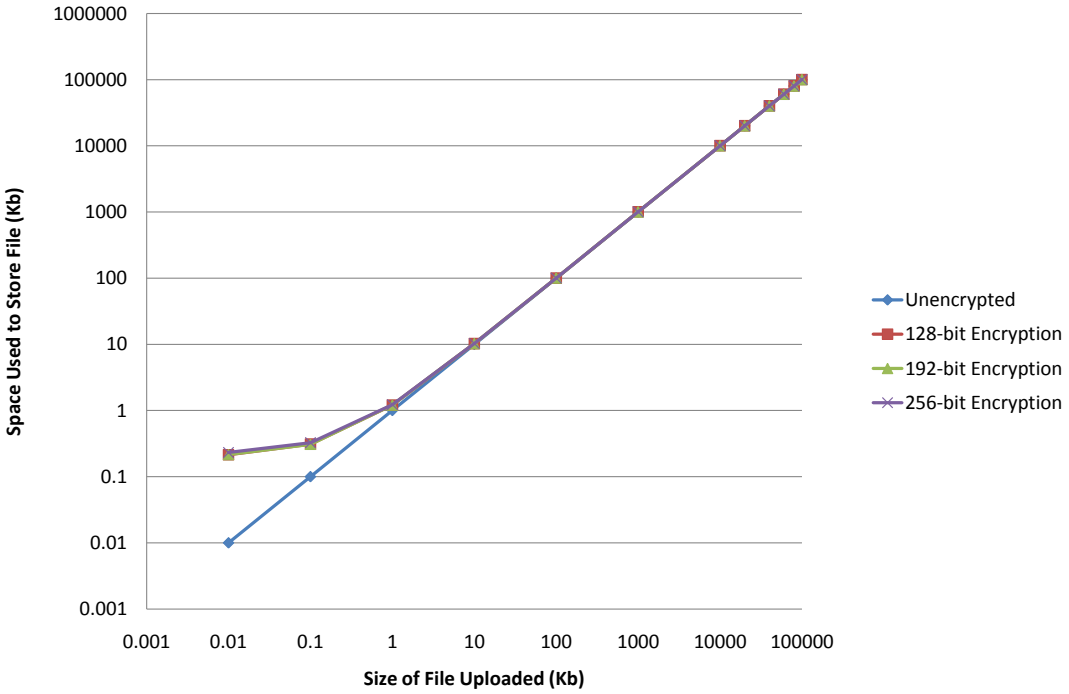


Figure 4.2: Performance of Osiris in space for varying levels of encryption.

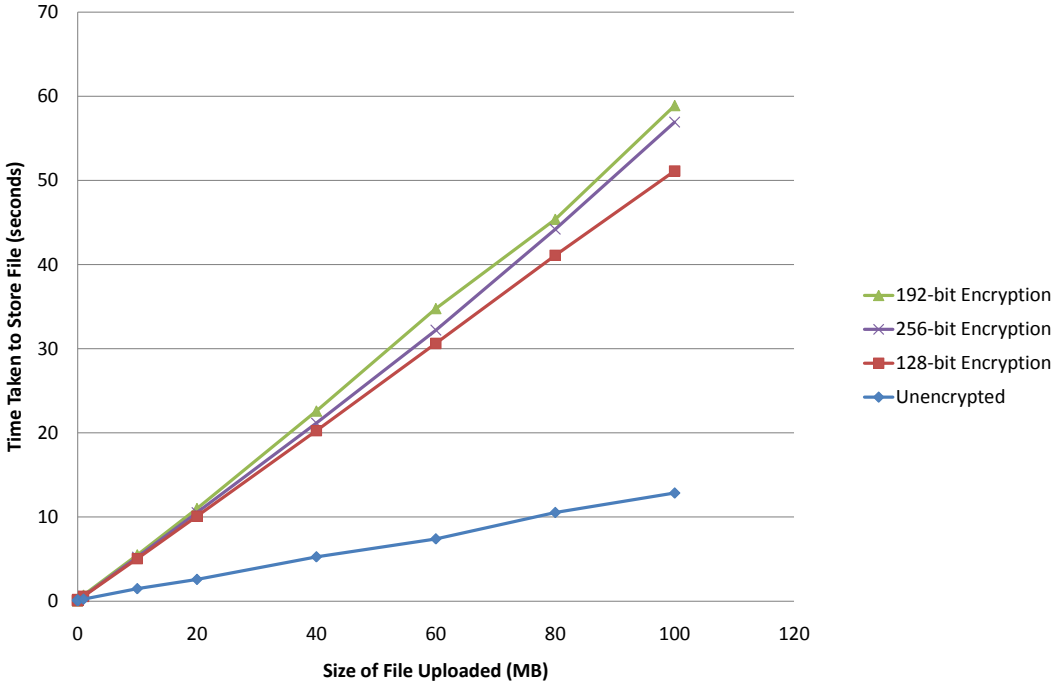


Figure 4.3: Performance of Osiris in time for varying levels of encryption.

Deduplication

Figure 4.4 shows how Osiris performs with and without deduplication enabled and with and without convergent encryption. The results were gathered by asking 15 EC2 clients to upload the same file with these varying parameters.

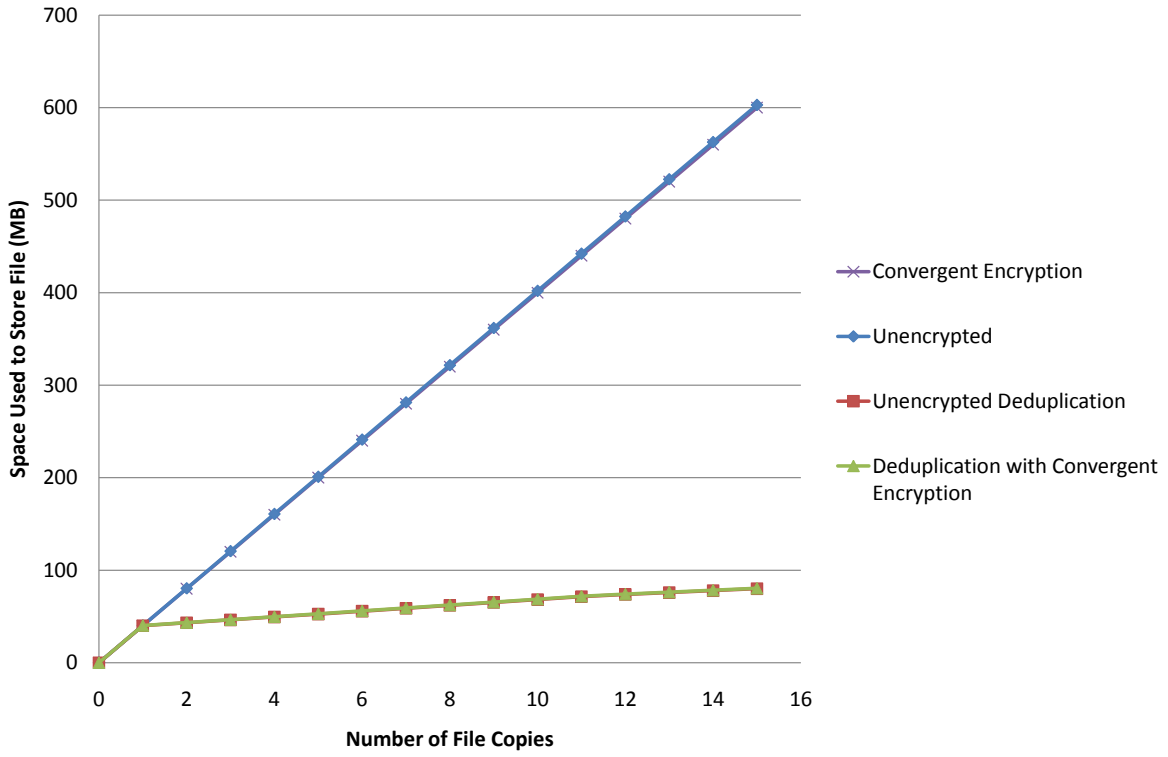


Figure 4.4: Performance of Osiris in space with deduplication.

Observe that the space usage required to upload the first copy of a file is identical with and without deduplication. Beyond that however, the space usage of whole-file replications increases linearly whereas that of deduplication remains fairly flat.

Sometimes, a client must necessarily produce a replica of an existing chunk, if it is the backup source for that chunk (see Section 3.8). Because of this, the gradient of the deduplication line is not zero. However, the space costs can be no greater than a replication factor of 2, regardless of how many copies are stored onto the network.

Figure 4.5 shows that deduplication does reduce the time taken for a storage transaction. The saw-tooth pattern shown is caused by the varying network latencies of the 15 different clients uploading files. Admittedly, the time reduction due to deduplication appears insignificant compared to the time reduction caused by not encrypting the data. However, this is due to Amazon EC2 network latency being very low such that the cost of sending a chunk header is not significantly different from sending an entire file. On a real world Internet connection, the benefits of deduplication would be a lot clearer.

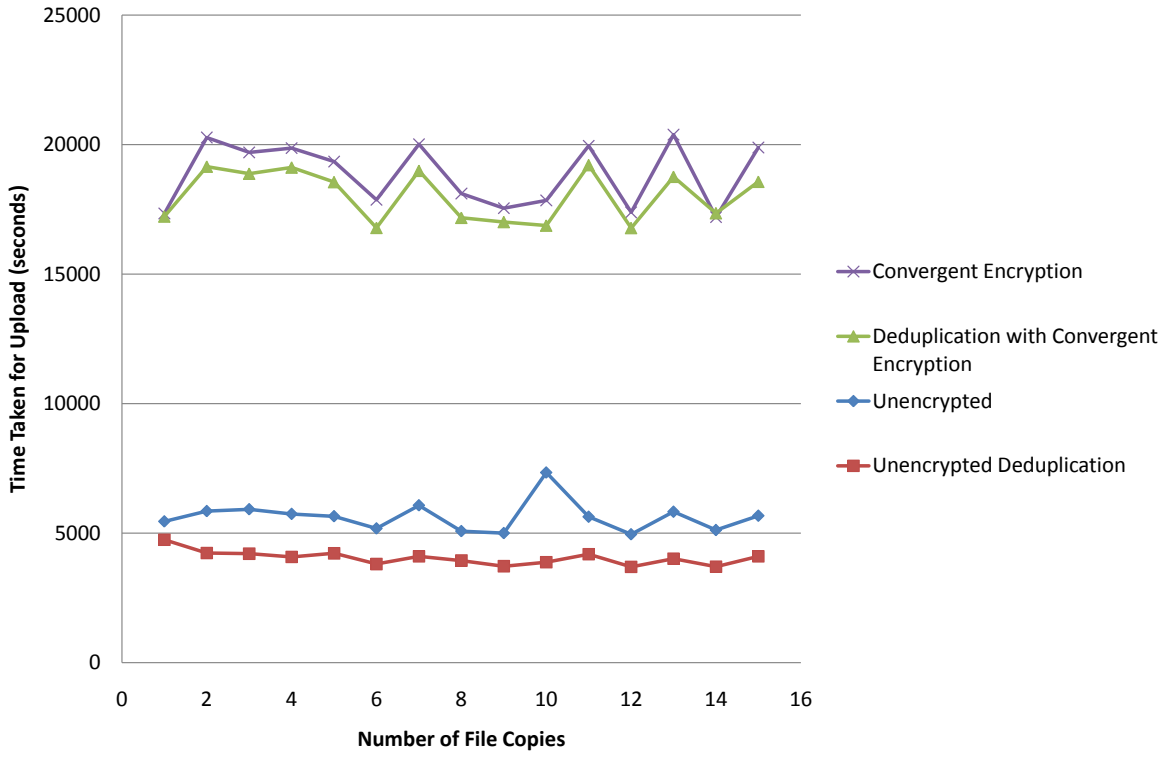


Figure 4.5: Performance of Osiris in time with deduplication.

Replications

Since Osiris needs to maintain a certain degree of file availability (see next section), there is a certain amount of redundancy in the data storage. Figure 4.6 shows how Osiris performs with transactions of varying replication factors. The cost and performance of using the IDA is also shown.

In the IDA series of these graphs, $n = 15$ and $m = 10$. These two values are of course fully customisable and here they are a fair representation of a working system with 15 clients that wishes to provide a redundancy factor of 6. When discussing whole-file replication, it is clear to see a linear correlation between the replication factors in terms of both time and space. However, when the IDA is introduced the time taken is significantly larger, due to its additional processing. But the major benefit of this is that it significantly outperforms the strict duplications of files, in terms of space required. Observe that the IDA provides the same redundancy of data as the *6 copies* data series.

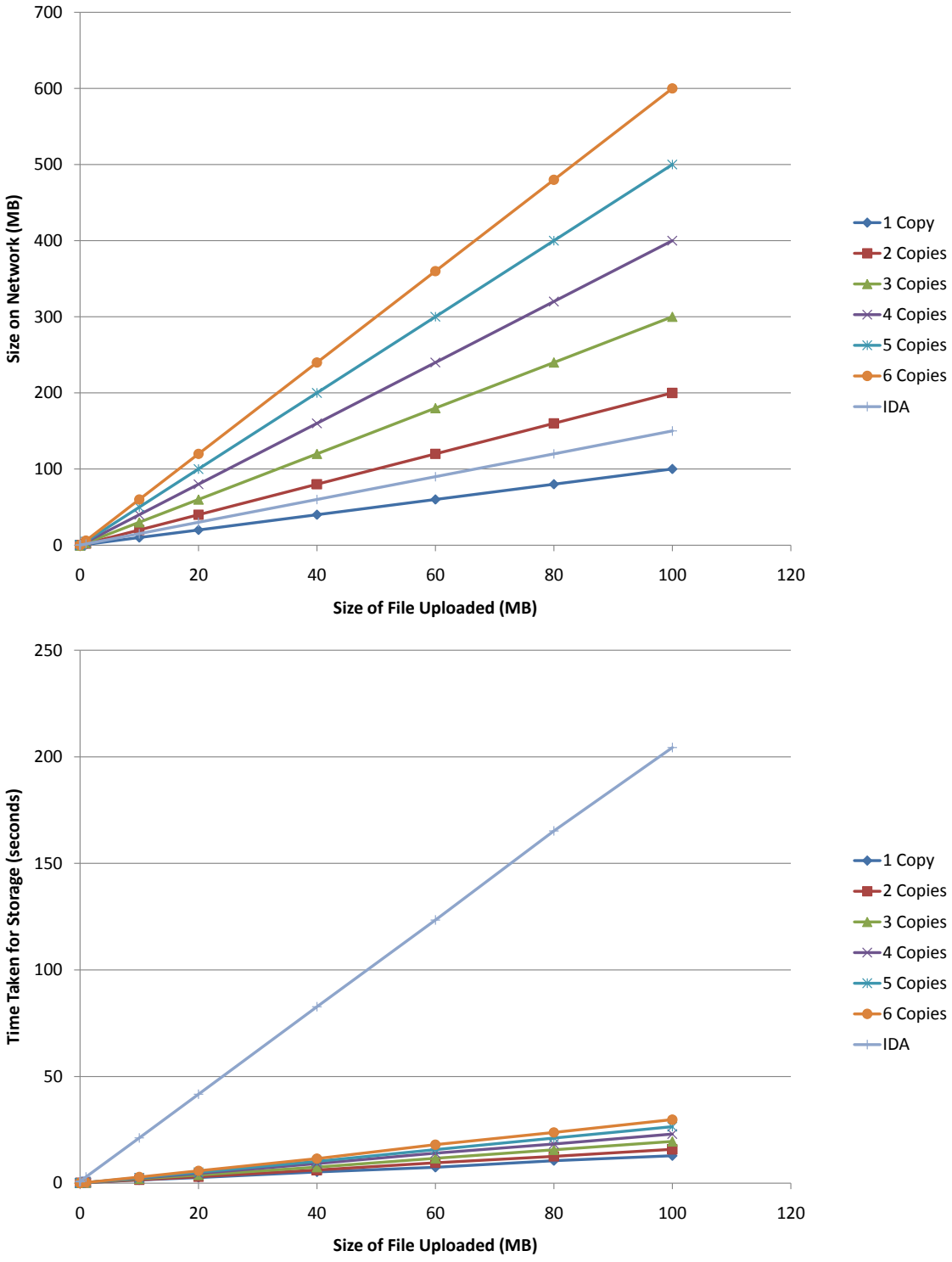


Figure 4.6: Performance of Osiris in space and time for varying replication factors and the IDA.

4.3.2 Availability

The cumulative distribution function in Figure 4.7 shows the probability that a particular file will be available from the perspective of the owner of that file.

This data was gathered by permitting a single client to upload 50 files to the network, under various different configuration settings. After each run of this evaluation, the power set of all 15 clients on the network was found. For each set s within the powerset, I calculated how many of those 50 files would be available if all nodes in s were online. For each length l of s, an average was taken before being plotted.

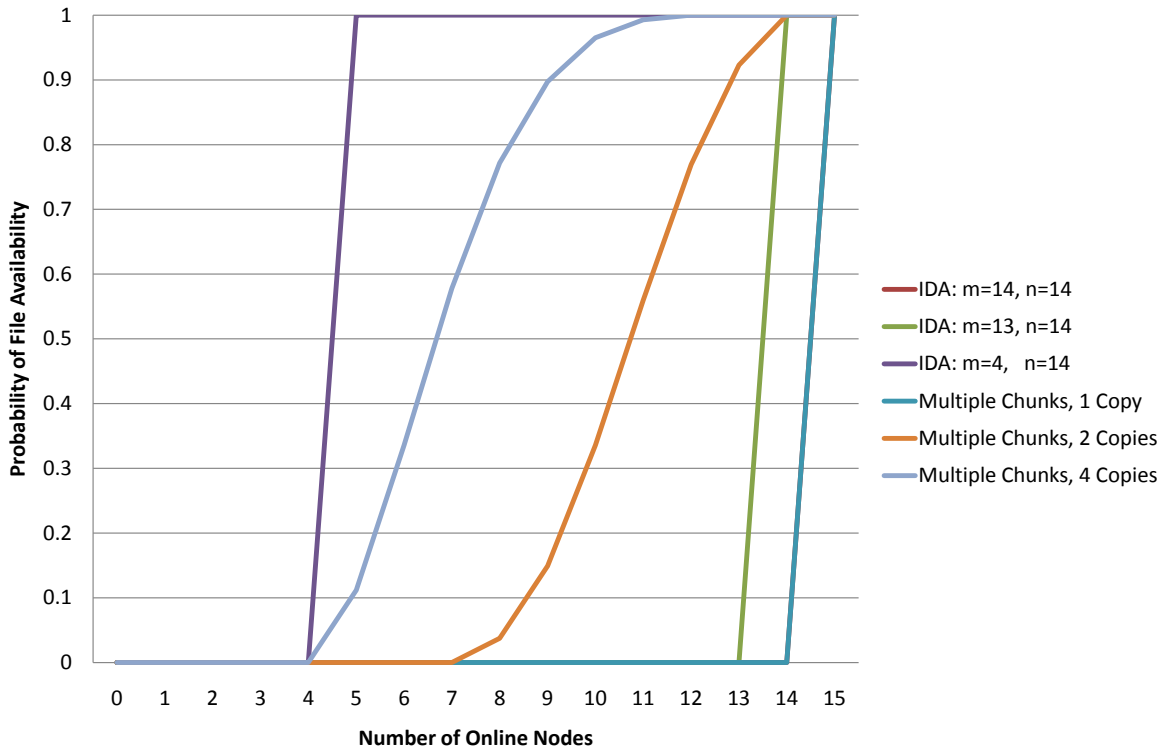


Figure 4.7: Probability distribution of file availability. Note that the IDA curve for $m = n = 14$ is identical to that for a single copy.

The results show, as predicted, that the IDA does produce a very high file availability, while retaining a low space usage. For example, setting $m = 14$ and $n = 4$ provides the same level of redundancy as a *10 copies* data series would. However, it uses less network storage space than the *4 Copies* data series does.

With whole-chunk replications, to recover a file, the correct nodes holding the correct chunks need to be online, whereas with the IDA, the probability of a file being online is binary. The IDA requires *any* m chunks to be available and as they are distributed in a round robin fashion, this is analogous to requiring m nodes to be online[2].

[2]Do note that this is only true if n is the number of nodes on the network.

It can be seen that setting $m = n = 14$ provides the same availability probability as the *1 Copy* data series. They both require all nodes to be online for a file to be available and so this is not a realistic setting to choose except in networks where node failure is not tolerated.

Although not shown here, uploading entire files, rather than chunks, provides a better file availability than the multiple chunks series. However, Osiris does chunk files with a round robin storage policy to ensure that no single node will have access to all file data necessary to reconstruct a file.

4.3.3 Scalability

To test the scalability, the testing framework was run with 20 clients each spawning transactions at random intervals. On average, a new storage, retrieval or removal transaction was introduced into the system every 50ms. This was repeated for 2200 transactions with zero rollbacks occurring. When the test was repeated with deduplication, five rollbacks occurred due to the few corner cases where the system does not perform perfectly. These corner cases are caused by concurrency problems in a distributed environment. However, despite this, transactions do rollback cleanly after they have timed out, allowing Osiris to remain fully operational.

Although the space and availability results for the scalability tests are comparable to those of the analysis of previous sections, the average time per transaction increases. This can be attributed to various limits imposed on the concurrency of the system.

Transaction Limiter Limits the number of concurrent transactions on both client and controller.

Thread Pool Limiter Limits the number of threads per instance of Osiris client and controller.

Database Pool Limiter Limits the number of open database connections on the controller.

Processing Power A client performing encryption on multiple files simultaneously must share its resources between multiple threads.

Hard Disk Performance Reading and writing files to hard disk simultaneously is very demanding and slow.

Database Isolation Conflicting transactions cause one to wait for another to commit before being able to execute.

Socket Limiter Limits the number of open incoming and outgoing sockets for clients to transfer files upon. These are required because the network bandwidth and latencies are the major contributing factor to time delay of an Osiris transaction.

This is not a failing of Osiris, instead it is a significant accomplishment that simultaneous transactions can, for the most part, function seamlessly despite an increased delay.

4.4 Security

4.4.1 Breaking File Encryption

All encryption and cryptographic hash functions are performed using the secure *Java Cryptography Extension*. The encryption standards used in Osiris are AES-128, AES-192 and AES-256. They are all believed to remain secure into the foreseeable future due to a lack of computing power to break them. The cryptographic hash functions used are SHA-256 and SHA-512 for which no collisions have ever been found.

Breaking the encryption security within Osiris is by design very complex. First, I assume that the attacker does not know the user's password. Since the decryption key is derived from the password using the PBKDF2[3] standard [16], I assume that the attacker does not know the decryption key either.

To break the encryption, the attacker might first try password cracking. One popular solution to this is a dictionary attack. The procedure for password cracking is to first generate a password and then check it. Osiris does not make it easy to check if a password is correct. First, no hash of the user's password is ever transmitted across the network or stored anywhere. This keeps this information as secure as the storage of the password itself. So an attacker must check it by other means.

The first method would be to decrypt a file and check if successful. But for Osiris there is an increased constant complexity for each calculation due to convergent encryption (see Section 2.2.2). An attack would require the following procedure:

1. Generate the key from the password using PBKDF2.
2. Decrypt a file header entry to gain access to the file decryption key.
3. Decrypt the file using this file decryption key.
4. Check if the file has been successfully decrypted.

This is computationally hard.

Alternatively, an attacker could attempt password cracking by repeatedly trying to authenticate itself against the controller using the zero knowledge password proof scheme. But this is not an offline attack and as such has significant network communication overhead, so is also infeasible. A compromise of the controller would permit this attack to be performed offline but that too would require solving the computationally hard discrete logarithm problem (see Section 3.7.1).

The final option left to an attacker is to brute force AES. This involves trying every possible key combination. For the lowest grade encryption (128-bit) there are over 10^{38} possible combinations, which again makes a brute-force attack computationally infeasible.

[3] Password-Based Key Derivation Function

4.4.2 Security Threat Analysis

Now that I have shown that decrypting a file without possessing the password/key is computationally difficult, I will discuss how Osiris manages this and other risks through a threat analysis of data confidentiality.

Given the scope of this project, I do not consider it a failure that the occasional leakage of metadata can occur. The important factor is that an attacker that possesses metadata, might not possess the necessary chunks to make use of it. Even if he also has those, without the decryption keys, the files are useless to him.

Eavesdropper

An eavesdropper to transmissions could gain a certain degree of knowledge about the metadata for a file that is being backed up. By observing which clients are sending chunks to who they could infer which chunks belong to which files. However, they would not know in which order the chunks would need to be restored to reconstruct the file. Also, they have no access to the file decryption keys since these are never transmitted across the network.

Secondly, an observer could theoretically infer which nodes store which chunks, and which chunks make up which files by intercepting file availability messages. These messages are sent whenever a client joins or leaves the network. However, for this to work, the observer must be able to detect which node has left the network and fortunately, only the controller is privileged to this information. So this attack would require a controller compromise.

Client Spoofing

Client spoofing is where an attacker uses the same user name as an existing client and pretends he is that user. Client spoofing is impossible in Osiris without first breaking the zero-knowledge password proof that is used to authenticate a client to the network. If the zero-knowledge password proof is broken, then the attacker has also broken the user's decryption key and as such will be able to gain access to any file and decrypt it. However, solving the discrete logarithm problem, as previously discussed, is computationally hard, so the system remains secure.

Alternatively, if a client had managed to gain access to the network password[4], they could create a new client on the network. This new client would then become a source of backups for other clients encrypted files. Fortunately, the network password yields no information about the passwords of individual users, so the chunks remain confidential.

Client Compromise

A client compromise is where an attacker gains access to the data of an active Osiris client and is able to control it.

[4]This is a password a client requires for registration to the network on first login.

The attacker would have access to all the chunks that have been backed up to this client. Each chunk file is named by its ID, but because chunk IDs are allocated randomly, this yields no information about which chunks form which files. Each chunk is made up of a header and the encrypted data. Without the key, the encrypted data are useless to an attacker.

An attacker here would be able to read any of the chunks which have been stored unencrypted. However, the distribution of chunks across the network makes it unlikely that a particular client would be able to reconstruct an entire file. Nor would the attacker know which chunks were required or what order to put them in.

Controller Spoofing

This is where a man-in-the-middle intercepts all traffic intended for the real controller and instead runs his own controller in place. The attacker would have no pre-existing metadata but would gain the metadata for all new files being backed up to the network. However, this attack would immediately be detected by users when they observe that all their backed up files are missing from the user interface.

I could prevent controller spoofing by performing two-way challenge-response authentication. However, this would require the client retaining some prior information to authenticate the controller. This is obviously undesirable as one of the four principles of this backup system (see Section 1.4) is that a client need only possess their username and password to login anywhere.

Controller Compromise

A controller compromise is highly unlikely since, in a high security network, the controller is assumed to be running on a third party computing service and not a user machine.

However, in the case of a compromise, an attacker has full access to the metadata of the network. As such, it would be possible to gain access to any chunks stored on the network. This can be achieved by:

1. Editing the metadata store to make a new client an owner of all required chunks.
2. Running this new client.
3. Authenticating this new client.
4. Running the retrieval transaction for the desired file.

However, this is equivalent to the previously described case of a client compromise, except that the client holds all chunks now. The client can view unencrypted files, but encrypted ones remain secure.

Full System Compromise

If an attacker has multiple clients collaborating and has compromised the controller, he will be able to gain access to all files as before. But again, he will still not have the decryption

key for that file, since the user's password is not stored nor transmitted at any time. Osiris does not even transmit a hash of the password for the purposes of authentication. Instead, a zero-knowledge password proof is used.

Summary

The results from the evaluative runs on Amazon EC2 not only show that my system works as described, but also that it does so in a very efficient manner. When Osiris is run with deduplication and the IDA, it provides a solution which is both space efficient and has a high availability of data.

Osiris provides a two-pronged security model for maintaining data confidentiality. First, it conceals the file metadata with a strict security policy to prevent an attacker from being able to gain access to the necessary chunks in the first place. Second, it protects against gaining any meaningful data from the chunks through use of proven encryption methods. Ultimately, an attacker must break both the security policy and the file encryption to access a user's raw data. Without the decryption keys this becomes computationally infeasible.

Chapter 5

Conclusion

5.1 Achievements

Overall, Osiris is a great success. It has exceeded its success criteria providing several additional extensions including support for an IDA, zero-knowledge password proof authentication, customisable security, deduplication of data and convergent encryption.

The resulting system is a high performance, concurrent and distributed, peer-to-peer backup solution. It also provides a great deal of security to its users by protecting both their metadata and the file data itself with a strict security policy and encryption of data.

On the controller side, I have created a highly efficient, stable, scalable piece of software for managing the metadata and transactions of Osiris. And on the client side, I have created a cross-platform, fully customisable application with an intuitive graphical user interface for the storage and retrieval of backup files.

When the IDA is used with parameters determined by the network size, the probability that a particular file will be available remains high, even with only a few other clients online. Furthermore, Osiris is able to maintain this file availability while keeping the space usage of the network low. My implementation of deduplication further helps to keep the system both space and time efficient, eliminating unnecessary data redundancy.

To my knowledge, this is the first peer-to-peer backup system that has combined both the information dispersal algorithm and deduplication with convergent encryption. The result of this is a highly secure, fault-tolerant backup system which requires minimal space to perform secure backups.

Throughout this project, I have gained a great insight into many different fields of computer science, not just that of distributed computing. My newly found knowledge has greatly benefited me academically and will prove valuable in the future.

5.2 Lessons Learned

With the benefit of hindsight, there are a few areas which I might have approached differently when designing Osiris. First, I would consider my choice of peer-to-peer library more

carefully. FreePastry does not provide me with all of the features I might desire for my system, e.g. an inbuilt security layer that allows me to send encrypted messages. Also, although some of the tutorials provided are quite good, I found the API documentation to be poor or non-existent in many places.

Whilst creating Osiris, I found that every small decision I made had significant security implications. Prior to this project, I had never implemented any application with an emphasis on security, so at first I found this very challenging. However, as the weeks went by, I gained a detailed understanding of how to approach such decisions in order to enhance user security.

Many of the weaknesses in the security of my system are due to the fact that the controller is untrusted. If the controller were a trusted entity then all encrypted files could be sent through it. This would mean that no receiving client could detect who the original sender was, but it would make the controller even more essential to the system's operation, and fail to differentiate the system from existing solutions.

Although previously warned of the challenges of designing a system that is both concurrent and distributed, I found it to be a highly challenging, yet rewarding experience. I was, however, not expecting to spend as much time as I did writing code accounting for rollbacks due to transactional failures.

Given the opportunity to start again, I would change very little, for this project has provided me with the opportunity to be fully creative and innovative whilst learning a great deal about software engineering along the way.

5.3 Future Work

Although Osiris has exceeded its success criteria, there will always be scope for future improvement. I outline a few of my ideas below.

Controller Redundancy The controller is currently a single point of failure in Osiris in order to maintain the strong security primitives it has. A nice extension would be to provide redundancy of the controller so that multiple controllers are running on the network at once. Clients can communicate with any active controller and each controller has a mirror of the database.

Incremental Backup Perform regular backups such that only changes made to a file are stored to the network with each new update. The challenge would be to do this whilst maintaining a high availability of files. This also opens up the possibility of providing a revision control feature.

File System Integration Currently, a user must specify which files they wish to backup to the network. A useful improvement would be for a user to specify a folder they wish to keep backed up at all times. Osiris would be able to detect when changes have been made to a file and back it up accordingly. This idea would perfectly complement a revision control feature.

Bibliography

[1] Consumer Statistics. Global data backup survey results. `http://www.consumerstatistics.org/global-data-backup-survey-results/`, accessed on 18-04-2011.

[2] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '99, pages 59–70, New York, NY, USA, 1999. ACM.

[3] Christopher Batten, Kenneth Barr, Arvind Saraf, and Stanley Trepetin. pStore: A secure peer-to-peer backup system. Technical Memo MIT-LCS-TM-632, Massachusetts Institute of Technology Laboratory for Computer Science, October 2002.

[4] Code 42. Crash Plan. `http://www.crashplan.com/`, accessed on 05-05-2011.

[5] Hari Balakrishnan, M. Frans Kaashoek, David R. Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. *Communications of the ACM*, 46(2):43–48, 2003.

[6] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In Gregory R. Ganger and John Wilkes, editors, *FAST*, page 6. USENIX, 2011.

[7] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *International Conference on Distributed Computing Systems*, pages 617–624, 2002.

[8] M.O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2):335–348, 1989.

[9] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[10] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency control and recovery in database systems, chapter 7. 1987.

[11] Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michal Quisquater, Louis C. Guillou, Marie Annick Guillou, Gad Guillou, Anna Guillou, Gwenol Guillou,

Soazig Guillou, and Thomas A. Berson. How to explain zero-knowledge protocols to your children. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 628–631. Springer, 1989.

[12] Brandon Lum Jia Jun. Implementing zero-knowledge authentication with zero knowledge (ZKA_wzk). In *Proceedings of PyCon Asia-Pacific 2010*, 2010.

[13] J. Camenisch. *Group signature schemes and payment systems based on the discrete logarithm problem*. Citeseer, 1998.

[14] Victor Shoup. Lower bounds for discrete logarithms and related problems. In *Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*, EUROCRYPT'97, pages 256–266, Berlin, Heidelberg, 1997. Springer-Verlag.

[15] Personal communication with Malte Schwarzkopf; Python implementation initially by Steven Hand.

[16] B. Kaliski. PKCS# 5: Password-based cryptography specification version 2.0. Technical report, RFC 2898, september, 2000.

[17] Harald Niederreiter Rudolf Lidl. *Finite Fields (Encyclopedia of Mathematics and its Applications)*. Cambridge University Press, 1997.

[18] J. Bacon and T. L. Harris. Operating systems: Concurrent and distributed software design. 2003.

[19] Dropbox. `http://www.dropbox.com`, accessed on 02-05-2011.

[20] Amazon Simple Storage Service. `http://aws.amazon.com/s3/`, accessed on 02-05-2011.

[21] Bouncy Castle Cryptography API. `http://www.bouncycastle.org/`, accessed on 14-10-2010.

[22] The Cryptix Project. `http://www.cryptix.org/`, accessed on 14-10-2010.

[23] Java Cryptography Extension. `http://download.oracle.com/javase/1.4.2/docs/guide/security/jce/JCERefGuide.html`, accessed on 02-05-2010.

[24] Pastry – a substrate for peer-to-peer applications. `http://research.microsoft.com/en-us/um/people/antr/pastry/`, accessed on 02-05-2011.

Appendix A

Information Dispersal Algorithm

A.1 Algorithm

This algorithm was proposed by Rabin [8] and has been implemented within Osiris. The important aspects of the algorithm to understand are summarised in the following sections for the benefit of the reader.

A.1.1 Dispersal

I will now discuss the algorithm for creating n chunks from our initial file, F. Firstly, let

$$F = b_1, b_2, b_3, \ldots b_N$$

where b_i denotes some character from our file. Note that N is the number of characters we have in our file. For simplicity, I will state that $0 \leq b_i \leq 255$ so that each b_i represents a single byte. However, do note that characters b_i can represent any possible range of values at the implementer's discretion.

Now, we select some prime number p such that $255 < p$. We can now consider F as a string of residues modulo p. All subsequent calculations will be performed modulo p and as such each value might exceed 255. For example, if we selected $p = 257$, then our calculations might yield some values as high as 256. In this case, we have a one bit excess for each b_i. For a remedy of this problem of storing extra information, see Section A.2.

Next, select values of m and n such that $m \leq n$ and so that we have the desired space usage/fault tolerance level. The file is then divided into sections of length m.

$$F = (b_1, \ldots b_m), (b_{m+1}, \ldots b_{2m}), \ldots$$

Note that if N is not a multiple of m, we must pad our b_i with zeros. And so, for brevity, consider the sequences of F as

$$F = S_1, S_2, \ldots S_{\lceil N/m \rceil}$$

To create our chunks F_i, I must first introduce the *alpha vectors*. Each chunk has an alpha vector, $a_i = (a_{i,1}, a_{i,2}, \ldots a_{i,m})$, associated with it, so $1 \leq i \leq n$ because there are n chunks. It

is required that *any* subset of m alpha vectors is linearly independent. A discussion of how this was achieved in Osiris is included in Section 3.9.2.

Now that we have alpha vectors and the file divided into sequences of bytes we can construct our chunks, F_i where $1 \leq i \leq n$ and

$$F_i = c_{i,1}, c_{i,2}, \ldots c_{i,\lceil N/m \rceil}$$

where

$$c_{i,j} = a_i \cdot S_j$$

This algebraic explanation can be hard to follow and is more easily observed in its matrix equivalent form. Let A be the matrix of our alphas, B the matrix of our file data, and C our output matrix.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix} \cdot \begin{pmatrix} b_1 & b_{m+1} & \cdots \\ b_2 & b_{m+2} & \cdots \\ \vdots & \vdots & \cdots \\ b_m & b_{2m} & \cdots \end{pmatrix} = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots \\ c_{2,1} & c_{2,2} & \cdots \\ \vdots & \vdots & \cdots \\ c_{n,1} & c_{n,2} & \cdots \end{pmatrix}$$

We now have everything we need to create our chunks. F_i is the i^{th} row of the matrix C. The alpha vector (the i^{th} row of A) is also stored with the chunk F_i.

A.1.2 Recombination

The recombination stage of the IDA is very much a reversal of the dispersal stage with the difference that we only deal with m chunks.

As such, we must first arbitrarily choose m chunks from the n chunks created by the dispersal. By design, any m chunks contain all the information required to reconstruct our original file F. We first obtain the alpha vectors associated with each chunk. These are stored with the chunk.

The recombination lends itself to the simplest explanation in terms of matrices. Let A' be our matrix of alphas. Given that we now only have m chunks, we only have m alpha vectors, each of length m. It follows that our matrix is of dimension $m \times m$. Also, we selected our alphas such that any m of them are linearly independent. Now, m linearly independent vectors arranged into a matrix have a non-zero determinant. Since our matrix is both square and non-singular, it has an inverse A'^{-1}.

Let B be the unknown data for the file we wish to reconstruct. Let C be the data from our chunks where each row C_i is the data from a chunk. We then have

$$A' \cdot B = C$$

and hence

$$B = A'^{-1} \cdot C$$

As with the dispersal algorithm B now contains our data from our original file such that

$$F = b_{1,1}, b_{2,1}, \ldots, b_{m,1}, b_{1,2}, b_{2,2}, \ldots$$

Since B might have been padded with zeros in the dispersal stage, only the first N elements should be accepted as part of the file.

A.2 Galois Field

In the previous two sections all calculations have been performed modulo p. However, with $p > 255$ it is possible that our calculations may yield a value greater than > 255. As such, each calculation on an 8-bit byte requires potentially more than 8 bits to store the result. This increases the file storage space for each of our chunks significantly. Since one of the primary goals of Osiris is to ensure efficient space usage, a solution for this issue is required.

A Galois field (or finite field) is a field which contains a finite number of elements. The field $GF(p^n)$ is of order p^n and contains this number of elements. The prime p is the characteristic of the field and n is the dimension of the field. Arithmetic performed within a finite field results in an element within that field. This property will allow us to calculate the IDA without requiring additional storage space.

A single element within the field $GF(2^8)$ would represent a single byte within our file. Similarly an element of $GF(2^{16})$ would represent two bytes. This latter field is the one which has been chosen for this project. The result of this is that each calculation upon two bytes from the input file results in two bytes in the resulting output chunks.

The details behind finite field arithmetic are not discussed in this dissertation but they have been implemented, and explanations can be found in standard textbooks [17].

Appendix B

Transactional Protocols

This appendix describes each of the protocols implemented by Osiris, with the exception of the *storage* protocol, which can be found in Section 3.6.2.

B.1 Client Alive

This transaction is initiated by every single client upon startup and can be observed in Figure B.1. It is used to register the client with the controller, authenticate the user (as per Section 3.7) and ensure that the client's data is in a state consistent with the database. If it is not in a consistent state then action must be taken to to recover lost chunks (see Section 3.11).

Next, the client is sent all of its outstanding messages from transactions that occurred whilst it was offline. The client processes each of these messages in order and acknowledges that this has been completed to the controller once complete.

Finally, the client is permitted to join the network. When this occurs all other clients are made aware that this client has been authenticated and so messages received from it should be actioned. Also, the new client is added to the liveness controller which then sends all clients their latest file availability information (see Section 3.2).

If client has not been seen before on the network it is instead authenticated against the network password, shared between the group of friends. If the user possesses this information then he must now register his own authentication data with the controller for future use.

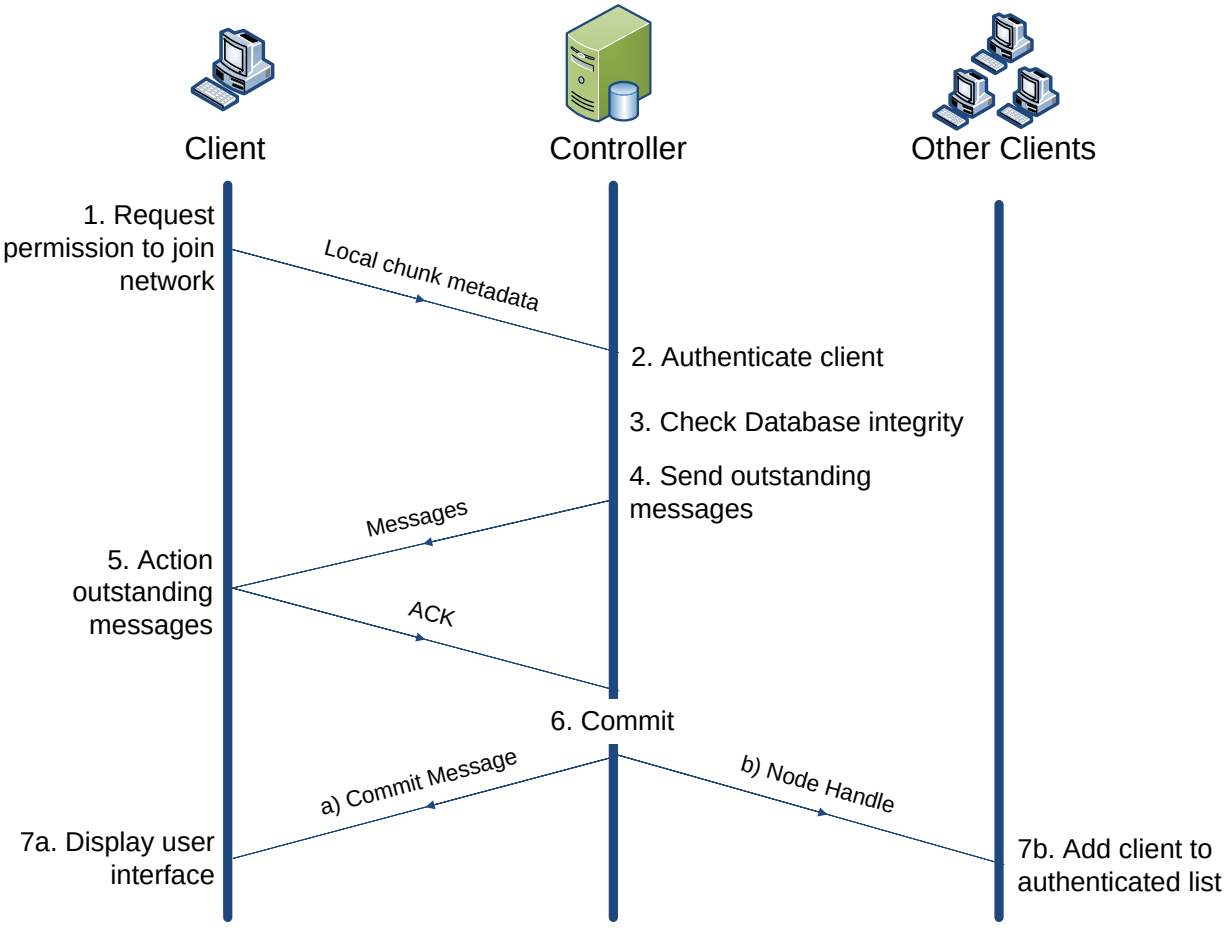


Figure B.1: Client Alive Transaction Protocol.

B.2 Retrieval

The retrieval transaction is created when a user wishes to recover a file from the network. In step 2, the controller determines whether the file is available by observing where the necessary chunks are stored and which clients are online. It then passes on this location information to the requesting client.

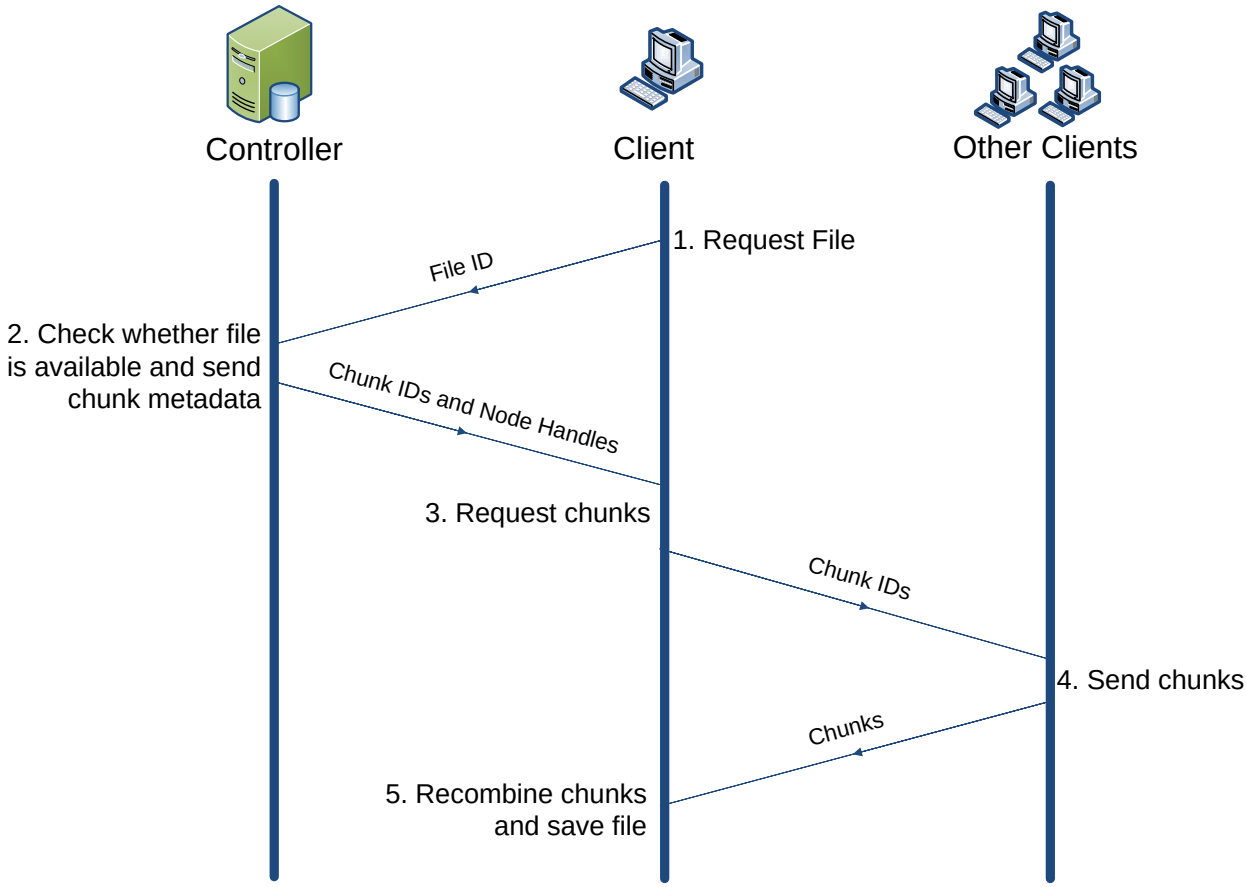


Figure B.2: Retrieval Transaction Protocol.

B.3 Removal

The removal transaction is created by a user when they opt to remove one of their backed up files from the network. To achieve this, the controller analyses the metadata database to determine which chunks make up the file being removed. Some complex logic is then used to determine whether a header entry should be removed from the chunk or whether it should be deleted entirely. In step 3, the controller requests the header identifiers for the header entries that should be removed. Finally, the clients storing chunks receive their instructions to remove header entries or delete the chunks.

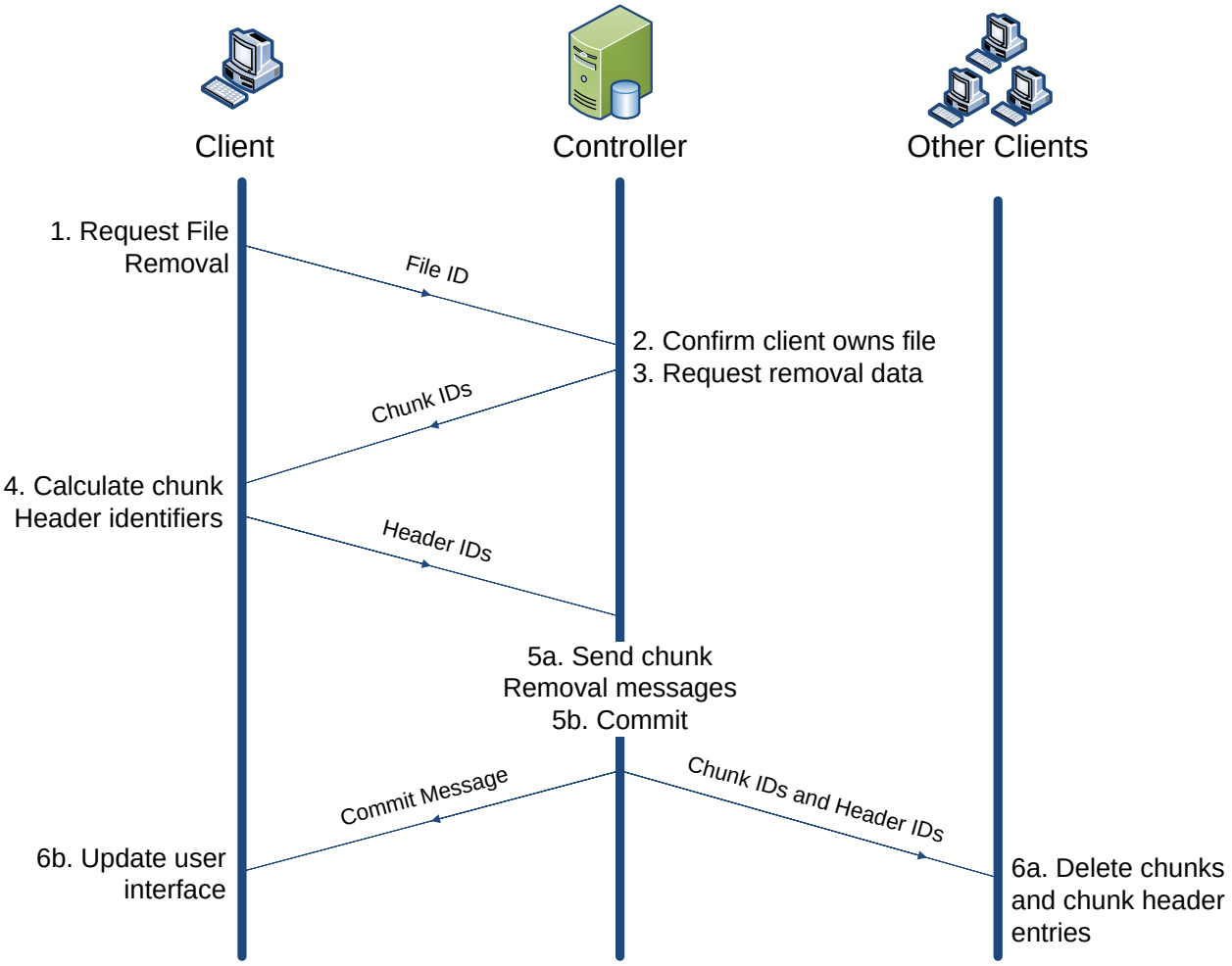


Figure B.3: Removal Transaction Protocol.

B.4 Replication

This transaction describes how files are copied from one client to another. Replications are spawned by the controller when a client has lost chunks it is required to have backed up. Section 3.11 describes how and why replications are essential in this case.

The controller is used in this protocol to inform a client that it is missing chunks and where they exist elsewhere on the network (if they do). Once the replication is complete, the controller commits the new location to the metadata database.

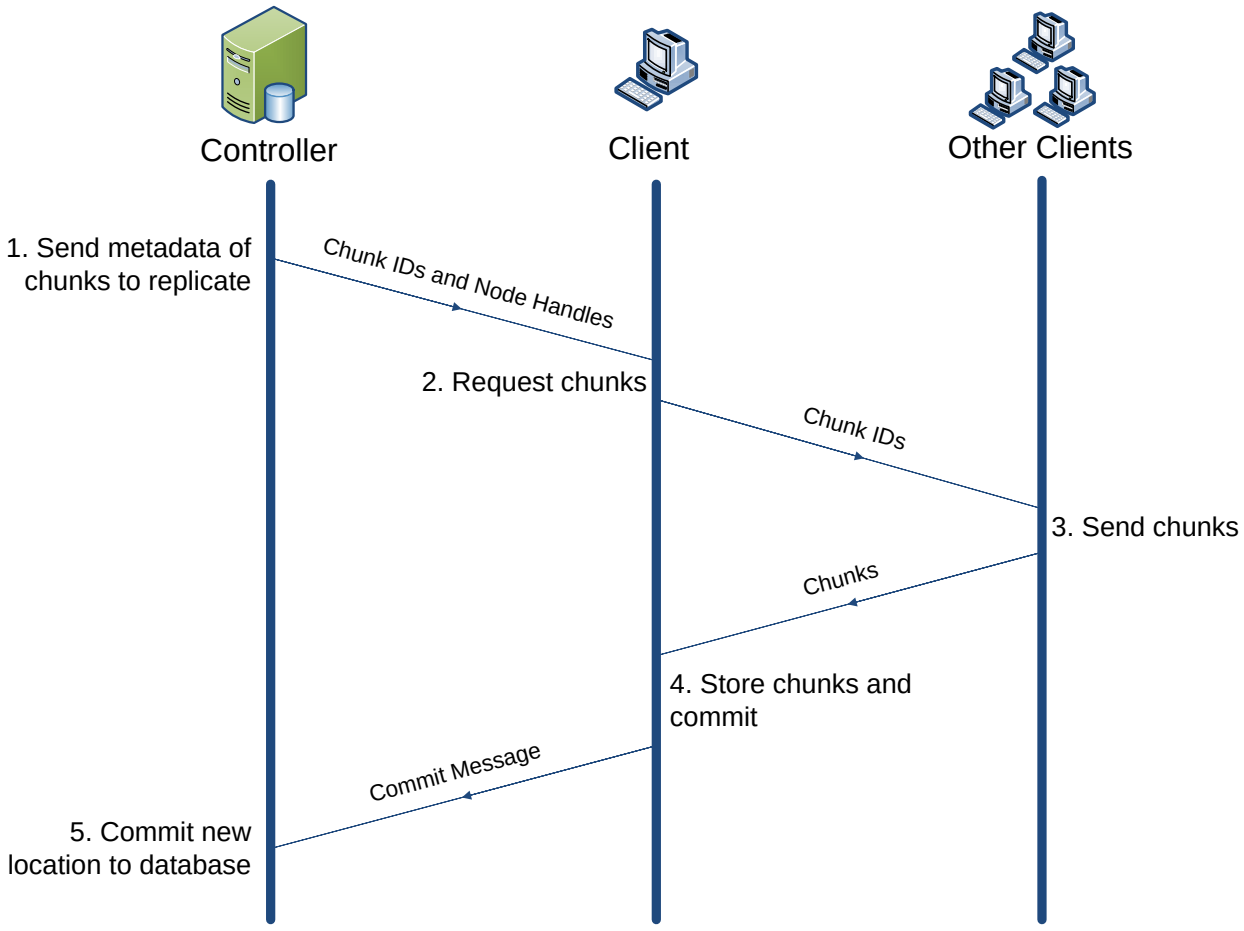


Figure B.4: Replication Transaction Protocol.

Appendix C

Communication

C.1 Message Passing

FreePastry provides a message passing overlay which is used to maintain the peer-to-peer network the controller and clients join. The library also permits access to the overlay for applications to use to communicate with nodes on the network. In Osiris, this is the primary method of communication between clients and the controller. Messages are most commonly used for:

- requesting data from a node.
- returning requested data to a node.
- synchronising transactions (e.g. commit and rollback).
- synchronising file transfers (e.g. acknowledgement messages).
- refreshing client information (e.g. files which are currently available).
- maintaining the overlay network (FreePastry).

For example, a transaction might wish to request the location of a file from the controller. It will send a `FileRequestMessage` containing the file ID to the controller. This transaction will then wait for a reply from the controller. The controller will reply with a `ChunkLocationMessage` containing the locations of the chunks necessary to reconstruct the file.

My implementation contains a Java package of containing all messages that might be passed around the network. Each class implements FreePastry's `Message` interface. However, to avoid duplicate code, each message is wrapped in an outer message object which contains the metadata for the communication. This includes the IDs and node handles of the sender and recipient of that message.

My implementation also provides a queue-based mechanism to send messages to offline nodes when they next appear online. This is discussed in more detail in Section 3.6.1.

C.2 File Transfer

Due to the nature of Osiris, the majority of communication (by traffic volume) will be file transfers between clients. Sending files using the message passing overlay is possible, but not a sensible choice due to the size of the messages. The overlay uses a single TCP connection and so large messages are likely to interfere with FreePastry maintenance traffic, thereby causing network degradation.

Fortunately, FreePastry does provide a mechanism for sending and receiving large files using sockets. However, I found that it did not provide the fine grained control needed to scale to many simultaneous file transfers. The most significant problem I encountered was that no method is provided to inspect or close a socket to another client.

Instead, I decided to implement my own file transfer classes whilst still making use of the FreePastry network, which is pre-configured to bypass NATs and firewalls where possible.

Limiting Simultaneous Transfers

Since Osiris is inherently multi-threaded, any client can be transmitting or receiving multiple chunks at any time. With large files and a fixed size chunking strategy, the number of chunks can be very large. A large number of simultaneous transfers is unlikely to make best use of the available networking bandwidth and will slow concurrent transactions to a crawl.

To limit the number of open sockets, a transmitting client queues its chunk transfers until the number of open sockets falls below the maximum specified in the properties file.

The more complex scenario is when a receiving client must control incoming chunks from multiple sources. My implementation uses a solution that is analogous to the *producer-consumer* paradigm [18] to accomplish this. Each client permits a certain number of chunks to be transmitted to it simultaneously. This is its number of *resources*. When a client wishes to send chunks to another client, it sends a `RequestToSend` message. The receiving client adds this request to a queue and then invokes the `consumeRequestToSend()` method shown in Figure C.1.

Once the chunk has been received, an acknowledgement is sent to the client. The single resource used to send it is freed to other requests, through the produce method. Finally, the consume method is called again in case there are other requests in the queue, waiting for permission to send chunks.

ChunkTransferReceiver.java

```
private static int availableResources = maxResources; //Member field
public static void consumeRequestToSend() {
    synchronized (guard) {
        if (!requestQueue.isEmpty()) {
            ChunkSender cs = requestQueue.peek();
            int requestCount = cs.getChunksToReceive();
            int chunksToSend;
            if (requestCount <= availableResources) {
                chunksToSend = requestCount;
            } else {
                chunksToSend = availableResources;
            }
            cs.decrementChunksToReceive(chunksToSend);
            availableResources -= chunksToSend;

            // Send client permission to use new resources
            cs.sendPermissionMessages(chunksToSend);

            //Remove request from queue if fulfilled
            if (requestCount - chunksToSend == 0) {
                requestQueue.remove();
            }
        }
    }
}
```

Figure C.1: Java code for consuming resources.

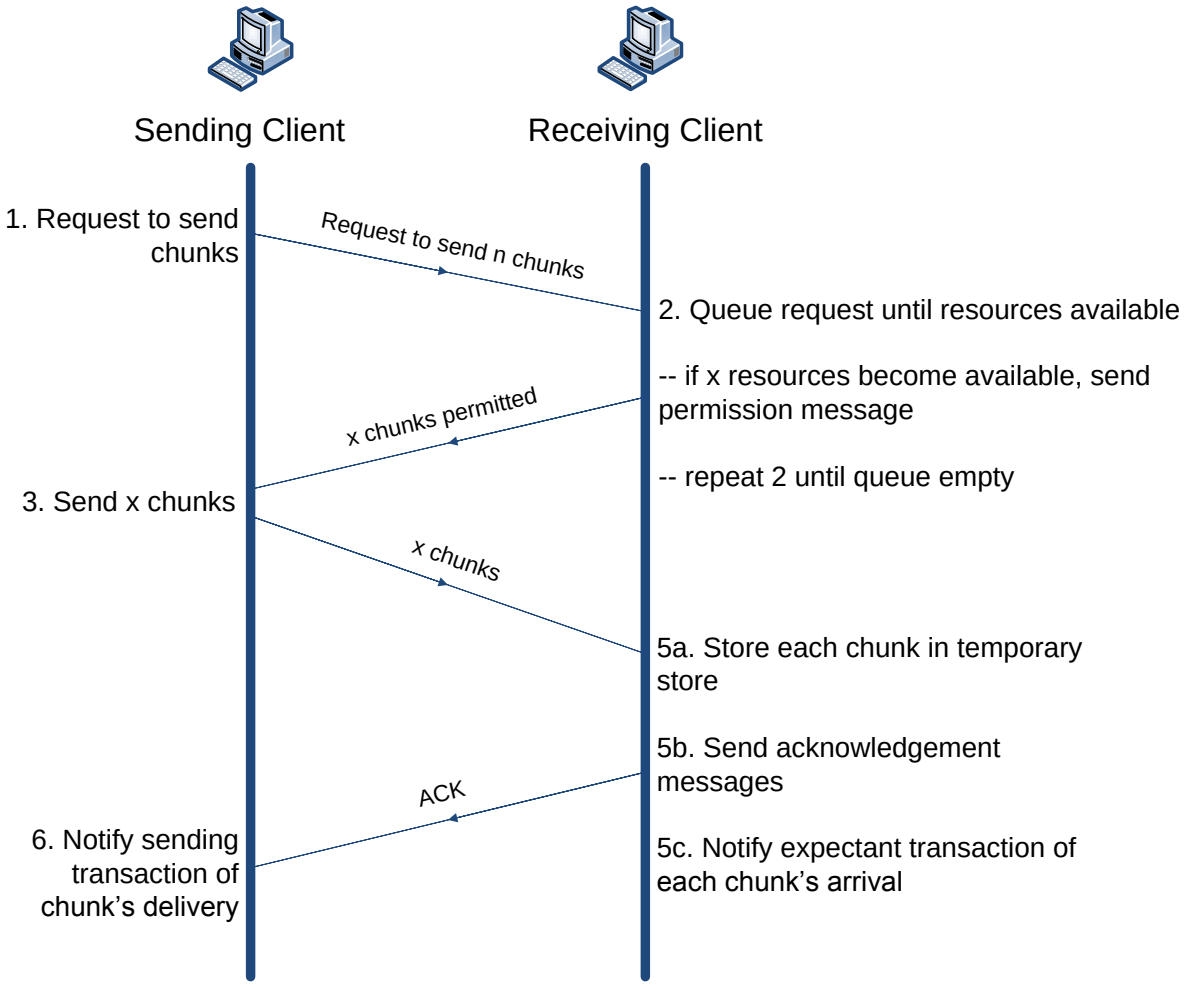


Figure C.2: File Transfer Protocol.

Appendix D

Project Proposal

Osiris – Secure Peer-to-Peer Backup

James Bown

October 2010

Introduction

In today's modern age the majority of computer users have a substantial amount of personal electronic data which they wish to preserve. It has always been necessary to backup this data in case of loss or corruption. In this proposal I expose a number of problems with traditional backup methods and describe how my project aims to address them with the creation of an alternative peer-to-peer backup model.

The first existing method of backup is onto an *external storage device*, for example a DVD or USB key. A problem with this is that you must take good care of the backup device and be sure not to lose it. Another requirement for this method of backup is that the storage device is located off-site. That is, in a different location to the device holding the original data, in case of natural disaster or theft.

A second existing method of backup is with a *cloud storage service* (for example Dropbox [19] or Amazon S3 [20]). However, this requires entrusting your personal data to a cloud provider, who is often an unknown entity to the user. There may also be issues with storing data in a country with a jurisdiction that might not protect their privacy. Furthermore, the cost of subscribing to a cloud storage service remains prohibitively expensive.

This project addresses these problems by creating a system which provides peer-to-peer backup between groups of friends. Users will backup their data to one or more friend's machines who are also running the system. This utilises the fact that the majority of users have a substantial quantity of free space on their machines which could be used by others to backup data. It also assumes that users are more likely to be willing to trust their encrypted data with a group of friends than with anonymous cloud storage services. However, friends are not completely trusted since the proposed system will ensure the security of all files using a security mechanism developed by Microsoft Research, known as *convergent encryption* [7].

A user and their friends will form a *'network'*. Each user will provide some space on their machine and in return will be provided space on the network to backup their personal data. The complexity involved in distributing the backup files will be hidden from the user. This includes data location, data encryption, replication policies and the division of files into multiple chunks.

Substance and Structure of the Project

This project can be clearly divided into two core components, a *controller* and a *client*. In the traditional manner, there will be many client instances and one controller instance. The untrusted controller acts as a central control point for all communication within the system without any of the encrypted backup files ever passing through it. This means that the clients will also need to communicate with one another directly, forming a peer-to-peer network, to transfer the backup files. Both the client and controller will be written in Java.

As mentioned, the controller only has access to the meta-data, not the backup data itself. It manages all aspects of the network of users including node status (online or offline), node location and amount of storage space available on each node. It will also be the central store of the meta-data for the backup files, including file location and file size, meaning that the client nodes act as block storage devices.

When a client wishes to backup some files, it makes a *store request* to the controller, providing it with the file meta-data. The controller then analyses the network of nodes currently online and determines how this file will be divided up and where it will be stored. It passes this information back to the client who then initiates the transfer in direct negotiation with the other clients.

When a client wishes to recover a file from the network it makes a *retrieve request* to the controller. The controller, again, analyses the network and determines if and where this file is available. It passes this information back to the client who then initiates the transfer.

The client is the program which will run on each of the nodes of the network. This is where a user will be able to initiate backup and recovery of files to and from the network. The client will make the controller aware that it is online whenever it is running (as explained further below). The client requests various pieces of information from the controller including the current amount of free and used storage space it has available on the network and which file chunks it has backed up.

The client must also be able to initiate connections to other clients in a peer-to-peer manner to transfer files once it has obtained permission to do so from the controller.

Finally, the client must be able to encrypt and decrypt files using convergent encryption [7] to maintain the confidentiality of the files whilst they are backed up on the network. This encryption mechanism works by creating an initial hash of the file to be encrypted. This hash is then used as the key to encrypt the file itself. This method of encryption is to be used for this project to encrypt all backup files stored within the network. Compared to traditional encryption methods, it has the advantage that compromising the security of one file will not compromise the security of all other files. A policy decision must also be made about a suitable way in which to store the private keys (i.e. hashes) for the decryption of files. One possible idea is to store them encrypted with the controller.

A number of challenges present themselves in this project by way of how to deal with distributed data which may not always be available. The protocols which are likely to be implemented are outlined below.

Node Status The controller must have some way to determining which nodes are online.

This is to be done using a heartbeat protocol or gossiping state protocol.

Addressing and Storage When a user wishes to backup a file, the controller must decide the best place in which this backup should occur. A few policy ideas for this include:

- Earliest response time to controller.
- Most storage space available.
- Node storing fewest files.
- Random.

It should be noted here that, because some files may be particularly large, and to improve security, that each file will be split into multiple chunks which may or may not be stored on the same node. The controller will need to find a way to do this whilst maintaining a high availability of data. The controller must also specify some replication policy. That is, how many copies of the backup files (or chunks thereof) do we store to maximise the availability of files when some nodes are offline.

Retrieval When a node wishes to retrieve its files from the network, it will ask the controller where it can find them. The controller will then find an online node for each chunk of the file and pass this information back to the client.

Backup Space Policy A policy will be created that specifies how much backup space each user should provide and how much backup space is available for them to use. The tradeoff to be considered here is one between fairness and efficient usage of space on the network.

The main units of work for this project are:

1. Familiarisation with Java, its standard libraries and the external libraries to be used for this project. Familiarisation with the software tools to be used to create the system. Research and careful planning of all distributed communication protocols to be used. A plan describing how all class files will interact with one another.
2. Implementation and setup of the controller.
3. Implementation and setup of the client. This includes creating both a client-server model of communication between it and the controller, and a peer-to-peer method of communication between all nodes on the network.
4. Implementation of the security constructs behind the system including file encryption and ensuring that the controller is untrusted with personal data.
5. Evaluation and testing of system.
6. Writing the dissertation.

Resources Required and Backup

Additional computers will be used to successfully test my system under both normal and stressful conditions. Each machine will act as at least one node on the network.

I intend to undertake the majority of the project work on my personal machine and on the PWF workstations. The storage and backup policy for this project provides a number of levels of protection against loss and corruption of data. The entire project will be an stored within an SVN repository on the PWF. Additionally, my personal machine is synced with the online backup service *Dropbox*. Finally, I will perform regular backups onto external USB keys.

Starting Point

- Recent work by Microsoft Research includes a project known as *Farsite*. This project implements a serverless distributed file system which avoids using unnecessary space by identifying and removing duplicated files in a secure manner. To facilitate this, a file encryption mechanism has been created known as convergent encryption [7]. This project makes use of this encryption mechanism to encrypt each backup file.
- Java libraries providing hashing, encryption and decryption functions. Candidates for usage include Bouncy Castle [21], Cryptix [22] and the Java Cryptography Extension [23].
- Java libraries providing peer-to-peer networking capabilities. The use of a library should provide capabilities for NAT-travesal and firewall bypassing to ensure that nodes can make the direct connections required for data transfer. One likely candidate to be used is Pastry [9,24].
- Building upon existing knowledge of Java from Part IA and IB practical classes and from my summer internship at Deutsche Bank.

Success Criteria

The creation of a backup system supporting peer-to-peer transfer and storage of data between a group of friends who have formed a *network*.

This system must provide:

- Decentralised peer-to-peer backup-file transfer and storage. This is shown to be working by successfully transferring a file to the network.
- Centralised or distributed meta-data storage describing status of online nodes, availability and location of data. This is shown to be working by successfully retrieving a file from the network.

- Confidentiality of backup files using convergent encryption. This is shown to be working by confirming that a file transferred to a remote node is stored as ciphertext. Then an attempt to retrieve it will once again yield the plain text.

I will evaluate this project by analysing the effect that changing the following parameters will have on the performance of the system: number of nodes online, number of nodes in the network, replication factor of backup data and strength of the encryption.

The implementation of any of the extensions detailed below would make the project a greater success, but they are not essential.

Extensions

Android Application Creating an application for Google's mobile operating system, Android. This application will be able to retrieve files from your network on any Android device.

Distributed Meta-Data Instead of relying on an untrusted central server to control where data is transferred to and fetched from, we attempt to make the system entirely peer-to-peer. This requires distributing the meta-data for the files to all clients using some distributed communication protocols to ensure this remains up to date.

Information Dispersal Algorithm The IDA [8] provides a way to reconstruct an original file without all of the original blocks of data being available on the network at any one time (e.g. only m out of n chunks of data available would reconstruct the original file). This would improve the reliability/availability of data on the system and the efficiency of storage space used on the system compared to a copy-based replication policy on the network. Furthermore, it would allow us to ensure that no single node on the network holds enough blocks to restore the file in the event of a hash compromise.

Security Levels The strength of the encryption of your personal files can be adjusted on a per-user basis. This provides a mechanism to have some friends within your network more trusted than others with your data. This provides the trade-off between security of data against system performance.

Timetable and Milestones

In order to complete the project before the deadline it is necessary to be very organised and to stay on schedule. This is why I have developed a timetable for the project below. A number of phases also specify milestones which I aim to meet by the end of the relevant phase.

Phase 1: 07/10/10 – 27/10/10

Michaelmas: Week 1–3

- Project Proposal written, submitted and approved by overseers. Meanwhile, familiarisation with Java, software tools and libraries can occur. This includes setting up a subversion repository within Eclipse and importing all libraries to be used. This time frame will also include suitable research of appropriate distributed and security algorithms.
- To familiarise myself with the software libraries, I will write small test programs to understand how they work before using them in the core of my system.

Milestone: Preparation completed.

Phase 2: 28/10/10 – 17/11/10

Michaelmas: Week 4–6

- I will at this stage begin to code my system, starting with the controller. First, this will require implementing a database structure in which to store data about the node and file locations. Note especially that the files will be split into multiple chunks possibly on different nodes.
- The next part of the controller to be implemented will be the addressing and storage policy. This must decide the best place in which to store a given file when it is created.

Phase 3: 18/11/10 – 08/12/10

Michaelmas: Week 7–8+

- From here on the controller and client will be developed side by side to facilitate the necessary interactions and testing which must occur whilst development progresses. The first thing to do in this phase is to create some skeleton code for the client.
- Using the necessary peer-to-peer library, code must be written to transfer files from one node to another and also to retrieve files at a later time.
- The controller and client must together implement a heartbeat algorithm so that the controller is able to detect which nodes are online.

Milestone: Node status, storage algorithms and retrieval algorithms implemented.

Phase 4: 09/12/10 – 29/12/10

Christmas Holidays Part I

Over the Christmas holidays there should be plenty of time to test and refactor the existing code. Along with this, some significant parts of the code base must be completed.

- The security constructs that ensure the confidentiality of files should be implemented. This means integrating the necessary cryptography libraries and coding the convergent encryption mechanism.
- A mechanism for the controller to provide permissions for a node to request files from another node will be implemented.

Phase 5: 30/12/10 – 19/01/11

Christmas Holidays Part II

- Some store for the private keys to the encryption must be created. These may be stored on the controller or elsewhere on a separate external device. Whichever solution is chosen, the confidentiality of the user's files must be maintained.
- The policy for ensuring there is enough space on the network for all users to backup will be implemented.

Milestone: All core functionality of the system completed.

Phase 6: 20/01/11 – 09/02/11

Lent: Week 1–3

- This is a period in which to bring the core functionality of the system together into a fully working and usable system.
- The debugging of existing code and modular testing.
- Prepare and submit a progress report to the overseers of this project and prepare a presentation to accompany it which will be presented at the very start of the next phase.
- Time should also be available to start making initial notes on the content of the dissertation.

Milestone: Progress report submitted and presentation prepared.

Phase 7: 10/02/11 – 02/03/11

Lent: Week 4–6

- Completion of the implementation of the system should occur within the first week of this phase, leaving the remainder of the phase for testing.
- If time permits, the implementation of any extensions to the project.
- Integration testing can begin by looking at the system as a whole under a variety of different conditions.

- Detailed notes on dissertation content and any necessary research to take place.

Milestone: Implementation completed.

Phase 8: 03/03/11 – 23/03/11

Lent: Week 7–8+

- Completion of full system tests under normal and stressful conditions.
- System evaluation to take place; analysing the performance of the system whilst changing a number of different parameters as detailed earlier in the proposal.
- Writing of the dissertation can begin.

Milestone: Evaluation and testing completed.

Phase 9: 24/03/11 – 13/04/11

Easter Holidays Part I

The entire focus of this phase will be to complete a first draft of the dissertation in its entirety. Once complete I will submit this working draft to my supervisor and director of studies for comments and corrections. Meanwhile, small tweaks to the system to improve its efficiency, readability and maintainability can occur.

Phase 10: 14/04/11 – 27/04/11

Easter Holidays Part II

I will make the necessary adjustments to my dissertation based upon the comments received from my director of studies and supervisor. Once completed I will submit both my dissertation and code base to the examiners.

Milestone: All Part II Project work completed and submitted.

Phase 11: 28/04/11 – 20/05/11

Remaining Time to Submission Deadline

The intention is that this time will not be used for any Part II Project work. I plan to spend it preparing and revising for my examinations in June. However, this time is available to correct any persisting bugs or to catch up on project work in the unlikely event I have fallen behind my timetable.