# Proforma

## Chris Applegate
## Corpus Christi College

## Eternity: A Durable Anonymous Distributed Storage System

## Part II Computer Science Tripos, 2002

| | |
|---|---|
| Word Count: | 11750 (approx) |
| Project Originator: | Jon Crowcroft |
| Project Supervisor: | Jon Crowcroft |

## Original Aims of the Project

The original aim of this project was to create a distributed data storage network modelled on Ross Anderson's notion of an Eternity Service. The network is designed to stored data anonymously with built-in safeguards to preserve data for long durations. An integral payment system included, and the network was intended to scale well.

## Work Completed

A system was produced which satisfied most of the original criteria. It could be installed on a number of nodes easily, and would store data reliably for the required periods. Data is encrypted and fluidly transferred around the network to increase deniability. Mechanisms for auditing and reporting were provided so that misbehaving nodes could be identified and reported. Some aspects such as scalability have not been fully realised, mainly as other qualities in the network were deemed more important.

## Special Difficulties

None.

# Declaration of Originality

I Christopher Applegate of Corpus Christi 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: May 15th 2002

# Contents

**1 INTRODUCTION ..... 5**

**1.1 Distributed Storage Systems ..... 5**

**1.2 Current State of the Art ..... 6**
1.2.1 File Storage ..... 6
1.2.2 Eternity Networks ..... 7

**2 PREPARATION ..... 8**

**2.1 Early Design Decisions ..... 8**
2.1.1 Data Chunks ..... 8
2.1.3 Client/Server Architecture ..... 8
2.1.3 Java ..... 8

**2.2 Requirements Analysis ..... 9**

**2.2 Satisfying Requirements ..... 9**
2.2.1 Reliability ..... 9
2.2.2 Redundancy ..... 10
2.2.3 Fair Payment ..... 10
2.2.4 Anonymity ..... 11
2.2.5 Scalability ..... 11

**2.3 Miscellaneous Design Decisions ..... 12**
2.3.1 Algorithms and Data Structures Used ..... 12
2.3.2 Software Libraries ..... 13

**3 IMPLEMENTATION ..... 14**

**3.1 The EternityObject ..... 14**
3.1.1 EternityHello ..... 14
3.1.2 EternityPeer ..... 14
3.1.3 EternityChunk ..... 14
3.1.4 EternityLocator ..... 15
3.1.5 EternityPartnerUpdate ..... 15

**3.2 The DigitalCoin ..... 15**

**3.3 The EternityException ..... 15**

**3.4 The EternityServer ..... 15**
3.4.1 Joining the network ..... 16
3.4.2 Receiving uploads and downloads ..... 16
3.4.3 Swapping chunks ..... 16
3.4.4 Auditing chunks ..... 17
3.4.5 Measuring Reliability ..... 18

3.4.6 Making Complaints ..... 18

**3.5 The EternityClient ..... 19**
3.5.1 Uploading ..... 19
3.5.2 Downloading ..... 20

**4 EVALUATION ..... 21**

**4.1 Models and Plans ..... 21**
4.1.1 Engineering Model ..... 21
4.1.2 Test Plan ..... 21

**4.2 Implementation ..... 22**
4.2.1 Network Protocols ..... 22
4.2.2 File splitting and recombining ..... 22
4.2.3 Uploading and Downloading ..... 22
4.2.4 Perfect Network Simulation ..... 23
4.2.5 Imperfect Network Simulation ..... 26
4.2.6 Malicious behaviour ..... 28
4.2.8 Miscellaneous attacks ..... 31

**5 CONCLUSIONS ..... 33**

**5.1 Further Development ..... 33**
5.1.1 Time ..... 33
5.1.2 Server-to-server anonymity ..... 33
5.1.3 Scalability & Network Conditions ..... 33
5.1.4 Misbehaviour ..... 34

**5.2 Final Conclusions ..... 34**

**BIBLIOGRAPHY ..... 36**

**APPENDIX A: RABIN'S INFORMATION DISPERSAL ALGORITHM ..... 37**

**APPENDIX B: TEST NETWORK INFORMATION ..... 38**

**APPENDIX C: SAMPLE SOURCE CODE ..... 39**

**APPENDIX D: PACKAGE AND CLASSES ..... 41**

# 1 Introduction

## 1.1 Distributed Storage Systems

The field of Distributed Systems has become one of the biggest research areas in modern Computer Science. The explosion of domestic computer purchasing and usage, and the dawn of the Online Age have meant that networked systems are now accessible by millions of users.

Storage is one of the most ideal applications, and they are already part of modern life. Today, millions use peer-to-peer Distributed Storage Networks (DSN) such as Gnutella, KaZaA and AudioGalaxy to share music, and increasingly other forms of data such as source code, programs, and video.

However, while a lot of research into distributed storage concentrates on availability, comparatively little has been done in durability and persistence. Yet persistence is a desirable quality. While some data, such as digital copies of popular songs are guaranteed persistency through their popularity, there is no fixed framework that guarantees obscure material will be capably survived. If an obscure Medieval French document like the Domesday Book were published on an existing P2P network, would it be as available as the latest MP3s? Is there any guarantee that it would be available in five years' time, or even six months' time?

Equally important is material of a sensitive nature - whether it's a political manifesto or the recipe for Coca Cola. Node owners can exercise a right to select what they want and do not want, whether by personal choice or due to pressure from external agencies such as governments. Furthermore, if any server does knowingly hold controversial material, then it becomes a target for any powerful organisation or individual to attack, either physically (destruction or damage to server) or legally (through injunctions, prosecutions and the like).

Ross Anderson has given a high-level specification of an 'Eternity Service' [1] that would offer guaranteed storage to any digital material whatsoever. Data storage is offered by a number of servers distributed worldwide, with no point of centralisation. Data is immutable - once a client uploads a file, he cannot delete it, not even under coercion, until that data expires. Data is also encrypted so a server owner cannot know what data it is storing, and no outside agency can tell which server hosts a particular piece of data. To act as an incentive to servers to keep data and act responsibly, Anderson outlines a payment system. This also offsets the risk of attack a server owner undergoes.

This project aimed to take Anderson's specification and create a workable implementation.

## 1.2 Current State of the Art

### 1.2.1 File Storage

File storage is one of the most ideal uses of a distributed application, and so not surprisingly, there are a number of example applications in use today. This summarises some of the best-known and most relevant examples to Eternity:

**Gnutella** [2] - One of the most well known networks, the simple Gnutella P2P protocol has gained widespread use, especially for sharing MP3 music files. Gnutella has little in the way of anonymity mechanisms - data is unencrypted on servers, and transfers are openly conducted. In addition, there is no guarantee data will be persistently stored. The infrastructure can also tend to become centralised and hierarchical, and the methods of network discovery and broadcasting are naive, leading to scalability problems.

Gnutella's ease of use and simple protocol, however, offset its flaws, its mass acceptance works as a substitute for the hard guarantees of other more complicated mechanisms - with millions of servers online, data durability is likely as there will, be a copy of the required file amongst those millions.

**Freenet** [3] - Freenet operates closely to the same principles as Eternity should - it possesses attributes such as publisher anonymity, deniability and decentralisation. Data being uploaded and downloaded is transferred via a series of inter-node hops before arriving at its final destination, to make traffic analysis harder.

There are some differences between Freenet and an Eternity network, however. Freenet locates files using data hashes to identify them, and files are stored on a server whose ID corresponds the most closely to that hash. This means that if a file's hash is known, its location is likely to be on the server with the closest corresponding ID, and so anyone wishing to destroy that file can attack that server.

Freenet also allows files to be updated by clients, which means someone who uploads to the network can destroy the copy of the file later, so the document owner (especially if they have been forcefully coerced) can later delete it. Freenet also contains no mechanism for payment, though the architecture does not preclude such a system being set up on top of it.

**Free Haven** [4] - Free Haven is perhaps the closest current design to an Eternity Network. It takes on board ideas such as splitting files into equal sized shares, which are dispersed over the network - this provides a harder target to aim for, and makes traffic analysis harder. Shares are traded across the network, providing fluidity and making data harder to track. Shares also provide a redundancy mechanism - if some shares of a file are deleted or a server destroyed, then the other shares can still be used to fully reconstruct the file.

Free Haven does not consider remuneration for servers, but apart from that retains many of the properties an Eternity Network needs. Currently the project remains at the design stage.

**Publius** [5] - Publius also employs a splitting principle, but different to that of Free Haven. Publius relies on generating a key, encrypting the file with it, and splitting that key amongst a number of servers - each server receiving a copy of the file and one piece of the key. To retrieve a document, a client retrieves multiple copies of the file, each with a piece of the key, reconstructs the key and decrypts the file. As no one server can decrypt the file on its own, this provides some publisher anonymity.

Publius does not provide many of the guarantees expected of an Eternity Network, however. There is no mechanism for detecting corrupt key shares or files - so malicious servers can publish garbage. And like all the above systems, there is no facility for payment.

**Mojo Nation** [6] - Mojo Nation, unlike most other DSNs, incorporates digital cash and payment. A user splits files into pieces, uploads them to storage servers (paying that server), and storing the hashes of those files. These hashes are stored by a content tracker. The location of pieces is known by a metatracker. To download, a user consults a content tracker with the filename, and receives the corresponding hashes, with which he then asks the metatracker to tell him who stores these pieces.

Servers trade with each other using a currency known as ‘Mojo’ paying each other in Mojo for using each other’s resources, and use a credibility system to ensure that malicious behaviour can be shut out. Mojo Nation is not intended for anonymity or durability (servers can drop unpopular files for more popular ones), nor is it a decentralised P2P system, but it still works as an example of encouraging availability and efficiency through payment.

### 1.2.2 Eternity Networks

At the time of writing, there are no full implementations of an Eternity Network. A system proposed by Adam Back called Eternity Usenet [7] uses the Usenet newsgroup infrastructure as a basis of storing files. This publishes documents onto Usenet (using the newsgroup `alt.anonymous.messages`) and uses a special webserver to convert HTTP requests for Eternity documents into NNTP request from Usenet.

This is a clever use of an existing distributed storage network, and provides an easy-to-use interface for users. However, there are still limitations - like Gnutella, it relies there being on a large number of servers to guarantee reliability, rather than making guarantees that servers are individually reliable. Also, the standard Usenet rules on propagation and expiration apply, so the network is not strictly ‘Eternal’, as server owners operate under their own content policy.

# 2 Preparation

I spent the first six weeks of the project conducting research and discussing the project with my supervisor. I thoroughly read Anderson's Eternity paper [1], and storage networks such as those in 1.2.1 (and others) were discussed. I conducted research into other areas of computer science, including cryptography, matrix arithmetic and some basic economics.

## 2.1 Early Design Decisions

A period of design was conducted after the initial research, and a number of pre-implementation design decisions could be made.

### 2.1.1 Data Chunks

Many of the researched systems split files into equal-sized pieces. This confers a number of benefits on the system. First of all, it is no longer possible to use the file size as an indicator as to what that file might be - all data traded is now contained in equal-size pieces, or as they are called in this implementation, chunks.

Splitting also increases efficiency - we no longer have to have multiple copies of the same file - we can just distribute different chunks to different servers. Redundancy can also be added in - by using an algorithm such as Rabin's Information Dispersal Algorithm (IDA) (see 2.2.2).

Chunks must have some identifier attached to them in order for a client to be able to download them. Rather than have user-chosen names for chunks (which may lead to collisions), a hash is taken of the chunk data and used. This also makes other processes such as for auditing (See 2.2.1) easier.

### 2.1.3 Client/Server Architecture

Many P2P storage networks rely on a client-is-the-server principle - i.e. Clients also act as servers. I chose not to follow this principle, instead opting for a hybrid architecture - a network of multiple peer servers, with separate clients to upload and download. The main reason was that of payment - unlike other networks, Eternity involves payment, so there is a clear customer/provider relationship, and it is necessary to keep that distinction. As Eternity is designed to host sensitive or controversial material, those who might wish to upload such material (e.g. political dissenters in oppressive regimes) might not be willing to risk uploading it if it also meant acting as a server and undergoing the risks that it would entail.

### 2.1.3 Java

From the outset Java was the ideal candidate to program the project with. Java has a large API, and is especially proficient with its wide-ranging networking capabilities. Its platform-independence is also an advantage, meaning it can be used over a wide variety of platforms with no additional work needed, and thus increasing potential network coverage.

## 2.2 Requirements Analysis

Following this research, I was able to break down the requirements for an Eternity Network under several headings:

- **Reliability** - Data must persist on the network. Servers cannot choose which data they can and cannot store - they must store whatever valid data they have. Mechanisms must exist to check they have the data that they have promised to store.
- **Redundancy** - A file stored on the network must be retrievable, even if the network suffers attack - either by a server being taken offline, or by a server maliciously destroying data.
- **Fair Payment** - Servers must be fairly compensated for storing the data, and this must be structured so that the servers that offer more space and store more reliably receive more money.
- **Anonymity** - Publisher anonymity is required - so that it cannot be discerned from a stored file who created or uploaded it. Server anonymity is also required - so that it cannot be discerned which server is holding which piece of data.
- **Scalability** - As the network is intended to be a large-scale global distributed system, the design must scale without risking server traffic overload.

## 2.2 Satisfying Requirements

With these requirements noted, I could make some further design decisions.

### 2.2.1 Reliability

Eternity must include well-defined mechanisms to ensure data is kept reliably - relying on the data existing because of the size of the network like Gnutella or Eternity Usenet offers no guaranteed security. All servers are designed to take any valid data (files are encrypted before uploading by the client, so no server can determine what data it is keeping.

Servers should also be auditable. This is where one of the major problems implementing an Eternity network comes in. Ideally an Eternity Network should be one of zero knowledge - no one other than the server owner knows the identity of the data that server is holding. However, if this principle remains in place, then there is no way of making sure that a server is reliable and actually possessing the data it has been given, as no other server knows what data it is supposed to host.

So I took a compromise and implemented a ‘partner’ scheme similar to that of Free Haven’s. A chunk on server P has a partner, hosted on a server Q. P can challenge Q (and vice-versa) to prove it has that data, by demanding Q send it a copy of the data. P can check the hash of that data to prove it is the data requested and Q has not maliciously tampered with it. Q has previously signed that data so that P cannot falsely claim that Q possesses the data when it does not. This breaks zero-knowledge but only adds one extra entity while providing plausible security.

### 2.2.2 Redundancy

I had already taken the decision to break files into chunk. One of the benefits of this is the ability to use redundancy methods such as Rabin's Information Dispersal Algorithm (IDA). The IDA takes a file, and by multiplying specially constructed matrices, create a set of $n$ chunks, with any $m$ (where $m \leq n$) chunks randomly selected from that set are required to reproduce that file. Such a defence means that servers can steal data or be removed from the network, and the file can still be resurrected, as long as $m$ distinct chunks are available across all servers.

There are other algorithms out there that also do this, such as Shamir's secret-sharing algorithm, but Rabin's is more efficient in space terms (Shamir's scheme produces chunks that are the same size as the original file).

More information on Rabin's IDA is provided in Appendix A.

Furthermore, to provide a 'moving target', I decided to swap chunks of data around the network randomly. This means that even the location of a chunk is never known for definite, even if the server that the chunk was initially uploaded to was known.

### 2.2.3 Fair Payment

This is one of the hardest problems to deal with. Most payment systems rely on a trusted third party or some other centralised payment facility, but this would become a vulnerable point of attack in a real-world system.

Initially, I proposed that payment be made at the same time as the data upload. Distribution was a major issue here. Who decides to whom the money should be distributed? If I allowed just a single server to do so, then the system is instantly prone to fraud. So I thought of distributing payment amongst many servers at once.

But there still has to be some incentive to keep the data, so I decided to delay payment until the data had expired - using digital coins with an expiry date, which makes them act more like post-dated cheques. But this still doesn't fully address incentives - even by delaying payment, what incentive is there for them to store the data and carry on behaving well, now that they are guaranteed money? There could be some sort of client authentication process at the expiration date, but what if the client is absent when the data expires? And what if the client has uploaded 'fake' or corrupt coins?

The solution I finally adopted is a simple one in theory, if harder in practice. Since a digital coin is just data, then it can be attached to the file data as well. The partner system can be used to guarantee integrity, by taking a hash of the data and coin and getting the owner of the chunk's partner to check and sign the hash.

This brings about an implementation of a 'data equals money' philosophy, creating an extra safeguard on the integrity of the network. Server owners now have a responsibility to maintain data integrity on the network - if they don't take care, then someone else is taking their money.

As well as acting as an incentive, the digital money system is also inherently fair - the more available and reliable a server is, the more data it can expect to be uploaded, and the more money it will receive.

### 2.2.4 Anonymity

The Free Haven paper [4] outlines various forms of anonymity. Two of the most important are Server Anonymity: Given a file, an adversary cannot tell what server it is stored on; and Document Anonymity: Given a server, an adversary cannot tell what files it stores.

These are important qualities, and eternity's design takes both of these into account - documents are encrypted with a secret key before uploading so servers cannot know what they are taking. All inter-server communications are encrypted and signed using a public-key system, so eavesdroppers cannot detect what data is being transported. This, of course, trusts that keys will not be compromised - no cryptographic system in the world is safe if keys are carelessly managed.

### 2.2.5 Scalability

Initially, the project proposal included a requirement to include guarantees that the network would be scalable.

One of the most common methods in P2P systems to save bandwidth is to use intelligent network routing. This takes advantage of the fact that many servers are often close together. If server A wants to send a message to servers B, C and D, and they are all on the same LAN, then A can just send a message to B, and B will forward it on to C and D, requiring just one message to the LAN rather than three.

However, Eternity's principles make this harder. As public key cryptography is being used for communications, the separate messages sent to B, C and D are now different, so no saving is made. Furthermore, in a system such as Eternity, where this a degree of mutual mistrust between servers (at least, by default), then A may become reluctant to send messages to C and D through B for fear of eavesdropping, tampering, or man-in-the-middle attacks.

I decided to take a step back on this matter and take a more broad view, as although scalability is considered a paramount quality in many distributed storage networks, Eternity is different from them in a number of ways.

Given the nature of Eternity, with servers separate from clients, and putting themselves under some risk, the number of servers on the network is going to be low compared to the scale of other distributed networks. In fact, in many networks where clients also act as servers, the number of nodes hosting data is minimal (On Gnutella, for example, 1% of the network hosted over 50% of the files).

In addition, the likely technical capability of such a machine is going to be high, as servers are essentially going to be businesses. Businesses will not want to rely on 56k modems as their connection to the Internet - servers are likely to have high-speed connections which will cope with higher bandwidth demands.

So scalability is not as pressing an issue as originally outlined. Despite this, it is obviously desirable to save bandwidth and runtime wherever possible. I designed the servers to operate so that regardless of the number of nodes on the network, their output from standard operation (i.e. runtime - swapping and auditing of data) would be constant. This means an O($n$) amount of data being traded on the network during standard operation, which scales well.

Unscalable aspects of the network include joining and broadcasting, which are of O($n^2$) complexity. I initially put forward the idea that such broadcasts are only made by a node when first joining the network and that it is assumed it is online from then on. However, as the network ages and more and more nodes leave permanently, then this leads to more and more failed connections, which makes it just as wasteful as the occasional broadcast to indicate when servers are logging on and off - which should not be that often, as servers are foreseen to be semi-business ventures and will be online for long sustained periods.

## 2.3 Miscellaneous Design Decisions

Before starting actual coding, I was able to make some more implementation-oriented decisions, such as what software libraries to use.

### 2.3.1 Algorithms and Data Structures Used

Encryption is provided by the Java framework, using 1024-bit RSA encryption for key exchange and signature. I initially used DSA keys for signature but found that RSA performed faster, especially for signature checking (The difference can be as much as 40 times according to [6]), which is what the majority of public key operations were. The estimated time for breaking a 1024-bit key is $3 \times 10^{11}$ MIPS Years according to [6].

Servers themselves use 168-bit Triple-DES three-key encryption for storing data. Triple-DES is also used by clients to encrypt files before uploading to the network. To crack a key, half the keyspace ($2^{167}$) has to be searched, so on average, which will take $5.9 \times 10^{36}$ MIPS years, assuming a new test key can be generated, tried and tested per instruction. SSL Communications use RC4, which has a 128-bit key, and would take $5.4 \times 10^{24}$ MIPS years to crack under the same conditions.

Hashing is done using the MD5 hash algorithm. The hash produced is 128 bits long. A birthday attack ($2^{64}$ trials) will take an average of $5.8 \times 10^5$ MIPS Years to find a string of data that creates a known hash, assuming a new test string can be generated and hashed per instruction.

I consider these figures safe enough to use in Eternity, the levels of encryption being well above levels that can be cracked with distributed systems. The implementation is fairly open-ended

Most of the data structures passed around the network used are of my own design, each message and piece of data in the network being written from scratch. These heavily use the standard Java primitives and arrays, as well as structures in the

java.util package, such as Vector (a variable-size array implementation) and Hashtable.

### 2.3.2 Software Libraries

Java provides a framework for most of the algorithms used here, and most of the libraries used were directly from the Java API.

I wrote the Information Dispersal Algorithm code based on Rabin's paper [11], but it relies on matrix multiplication. I originally used the NIST Matrix library [7], which provides a framework for the representation of matrices and provides fast methods for inversion. That library did not provide any modular arithmetic functions, so I modified and re-compiled the source code to provide such functions, but the NIST work was still an invaluable foundation for my work.

The Java libraries do not fully support some cryptographic processes thanks to US export licences. Therefore I use the freely available Cryptix [8] free library for some functions such as RSA encryption. However, where possible, to make future expansion easier, I use the standard Java security libraries.

Later on I adapted the X.509 Certification system into the network, thus making it compatible with modern authentication and online commerce models. Java provides no support for direct X.509 Certificate production, so I used a free evaluation library of X.509 software from Wedgetail [9]

# 3 Implementation

The Eternity Network runs as a set of EternityServer RMI (Remote Method Invocation) objects. Each of these sends an EternityObject over secure communications channels whenever it wishes to send a message to another server. EternityExceptions are passed if an error occurs. EternityClients connect to the network to upload and download files.

## 3.1 The EternityObject

This is an abstract object, and forms the basis of every message transacted between servers and clients. Although servers communicate many different types of message, each message has a unique ID (to prevent replay attacks) and identifiers marking the sender and receiver (To prevent man-in-the-middle and masquerading attacks). The EternityObject acts as a simple parent class to all the different objects used, and allows a generic object handling method to handle all incoming communications. It also implements Java's `Serializable` interface, making all subclasses convertible into bytestreams, which means they can be signed and encrypted, which is an essential part of the network's operation.

The most important objects are detailed below; most of the rest are mentioned in other parts of this chapter.

### 3.1.1 EternityHello

Used by servers for joining and leaving the network an EternityHello is just a simple ID/sender combination, with a random nonce as the message ID. A server operating correctly should decode the EternityHello and send another EternityHello back with the nonce + 1.

### 3.1.2 EternityPeer

This object describes an EternityServer node, and is in effect its 'calling card'. It includes its ID number, public certificate and IP address. It can be used by servers to announce themselves to the network, or can be published online for clients so that they know what servers they can upload to.

### 3.1.3 EternityChunk

This contains the chunk and digital coin (see 3.2) and is what is traded around the network between servers. It contains the data, coin and two hashes (one of both coin and data, the other of just the data) for identification and verification. There is a similar object, the EternityDataChunk, which just contains the data and its hash with no coin, and is used for sending to clients when they request a file.

EternityChunks also contain details of a chunk's partner - another chunk somewhere on the network. The hash of the partner, the network ID of the server hosting that partner, and a signature from that partner certifying it does host that chunk, to prevent other servers from claiming that someone else hosts a chunk.

### 3.1.4 EternityLocator

This object contains all the information needed to download a file. It contains all the hashes of chunk data that form part of the file, plus a hash of the entire file, so that the final file can be checked for integrity. It can also optionally include the decryption key for the file, although public publishing of this is dangerous, as it destroys the server anonymity protection discussed earlier. EternityLocators can be saved as files and circulated by users wishing to share files.

### 3.1.5 EternityPartnerUpdate

This is sent whenever a chunk moves, and it needs to inform its partner that it has moved. The update contains the chunk's name, it's new host's name and a timestamp, all signed by the new host. The update is sent by the old host to guarantee that the partner is notified. The update is also used to reply back to mistaken audits about a chunk that has moved to another node.

## 3.2 The DigitalCoin

Although some efforts have been made, most notably by David Chaum[10] on creating digital coins, the methods so far often fall short of a fully functional digital replacement for real money. I deemed implementing a proper digital coin to be outside the scope of this project, and so a simple implementation has been used - a DigitalCoin object merely consists of a value and maturation date, after which that coin can be spent. The project assumes some sort of bank (or better still, a distributed network of banks) will exist to honour the coins once they mature.

## 3.3 The EternityException

An EternityException is an extension of the standard Java exception, and is thrown whenever an Eternity process meets an error. This can include such things as bad data being transmitted, signatures not being correct, communications initiated by an unknown server and so forth. Exceptions are thrown both internally, and across the network from one server to another. Such network Exceptions are signed and also record the number of the transaction that caused them to be thrown.

## 3.4 The EternityServer

This module acts as the server-running program. It utilises the Java Remote Method Invocation (RMI) model, which means that a server can be instantiated as an object by other servers, and so by executing that object's methods, they can perform such actions as swapping, auditing and complaining to each other.

Eternity Servers listen for connections, whether from other servers or clients. They use encryption and digital signatures to maintain integrity.

Initially, communications were on unencrypted channels, with every message signed and then encrypted into a bytestream. This bytestream is then transmitted over the network and passed as an argument to the appropriate object method.

However, this has disadvantages - although the data being transmitted is encrypted, the overlying RMI headers are not - an eavesdropper can tell what type of transaction is being made, and thus can make detailed traffic analysis. In addition, using high-strength RSA encryption for communications is very slow, so slow that any server receiving many requests could suffer major performance problems, thus making it an easy target for Denial of Service attacks.

So I switched to using the SSL (Secure Sockets Layer) model of encryption, with RSA used for just key exchange encryption method, and a symmetric stream cipher (RC4) being used for actual transmissions, which is much faster. The optional digital signature part of SSL was not used, instead signing and verification was done at the application level, with Java SignedObjects were passed along the encrypted channel. This means that the server applications can easily extract and store the signed data and use them for informing other servers.

Servers' operations can be split into several distinct areas, all of which are explained in further detail below.

### 3.4.1 Joining the network

A new network node has to first find other nodes. With no central server, a server cannot 'log in' as such. Instead the system is designed so that when a user downloads the EternityServer software, it will also download some EternityPeer files and retrieve information about other available nodes from that peer. It can then contact one peer, an 'introducer', which may verify the newcomers' identity, and then sign its public key. This new server can now freely contact all the other servers on the network (provided by the introducer) with an EternityJoin object, and be accepted as a trusted member of the network.

A server joining the network will contact the peer(s) it knows and invoke the sayHello() method on each. This sends an EternityHello object to its peer, which then should reply a signed encrypted reply of the EternityHello with its nonce/ID value incremented.

Servers keep track of all peers they have ever been in contact with, as well as a subset, which covers peers they know that are online now.

### 3.4.2 Receiving uploads and downloads

Both processes are described in more detail in sections 3.5.1 and 3.5.2.

### 3.4.3 Swapping chunks

All servers actively do swapping at certain intervals. The interval time is random (to prevent timed attacks), somewhere in the range of 1-3 minutes, 2 being the average.

The protocol runs as follows:

1. Server P initiates a swap, picking a random server Q and proposing a swap of a certain value with it.

2. If Q agrees, then P sends Q a chunk $X_1$ of that value.

3. P sends Q a chunk $X_2$ of the same or similar value back, and a signature $S_Q(X_1)$ certifying it now has $X_1$.

4. P informs R, the server that hosts $X_1$'s partner, of the change of $X_1$'s host and associated signature $S_Q(X_1)$.

5. Q then replies to P, sending $S_P(X_2)$, the signature certifying it now has $X_2$.

6. Q now informs S, the server that hosts $X_2$'s partner, of the change, along with signature $S_P(X_2)$.

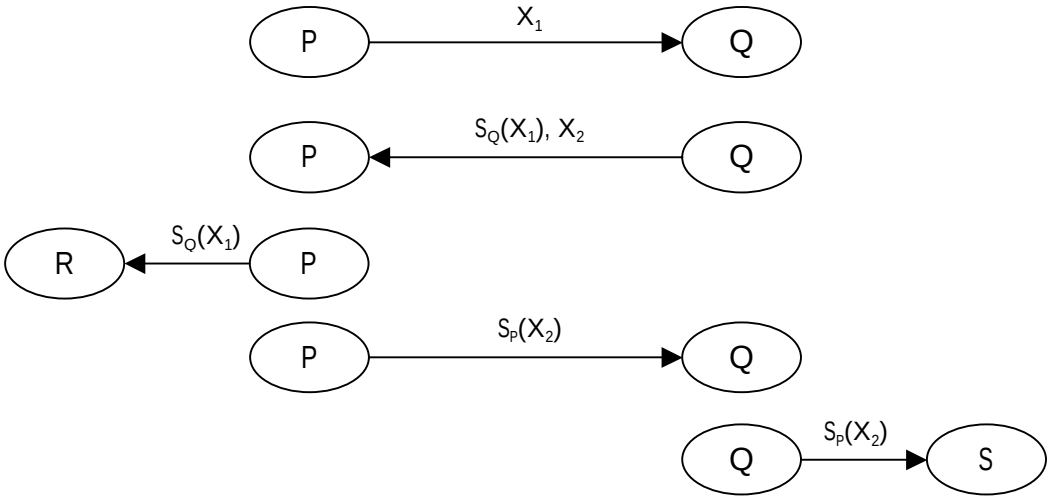


*Fig 3.1: Swap between hosts P and Q. These diagrams correspond to steps 2-6 of the process.*

If any part of the trade is considered faulty, such as when an EternityChunk does not match its hash, signatures are incorrect, or a chunk of insufficient value is traded, then the failure is noted and the other server's rating is suitably modified (see 3.4.5).

If updating the partner hosts fails (e.g. if either partner host is not online) then the server stores the EternityPartnerUpdate containing those details in an array and waits for the partner host to come online again.

To prevent the partner system from being compromised, neither A nor B can send a chunk whose partner is hosted on the other, or else a server would then control both partners at one point in time and thus be able to alter both without any outside monitoring.

### 3.4.4 Auditing chunks

Auditing is also managed by the main runtime thread, and can be initiated by any server. The process runs as follows.

1. Server A picks a random chunk X, and contacts B, the host of that chunk's partner, sending the hash of X as proof of reliability.

2. B looks up Y, X's partner and checks to see if the hash of X is correct. B sends Y back to A.

3. A calculates the hash of Y and checks it against the hash X has recorded.

For rating purposes, the audit is logged as a trade with the value of the chunk audited, and suitably marked as being successful or unsuccessful.

### 3.4.5 Measuring Reliability

Ratings are a simple but fundamental part of the Eternity network. They enable servers to tell exactly how trustworthy other servers are, and how reliable they can store the data.

Each server has two values recorded in the EternityPeer object that describes its reliability - one representing the total value of all chunks sent to it, and the other the total value of all valid chunks received from it. The rating is then simply: $\frac{\text{Total Valid}}{\text{Total Sent}}$

The values are incremented by the suitable amounts whenever a trade of chunks is made. When an audit is conducted, it is treated as a swap of the same value as the chunks being audited.

Ratings are used to discriminate when trades are made. When a server makes an active trade, it can choose not to pick any other server that has a rating beneath a certain threshold. This threshold in testing was 0.9, but it should be as high as 0.95 (meaning effectively, that a maximum of 5% of your money is susceptible to destruction). If it receives an offer to trade from a server it thinks is untrustworthy, it can return an EternityException refusing to trade with it.

Audits are not so strictly regulated - being audited does not cost a server anything in terms of data or digital coins, unlike a swap. So a server may receive audits from another server regardless of the sender's rating.

### 3.4.6 Making Complaints

A server will in two circumstances alert its peers to a misbehaving node. One is if an audit request it makes is not satisfied, and the other is if a swap it receives is invalid in some way - bad data, or an unsatisfactory amount of money is returned. Both transactions involve signed messages, so proof of misdemeanour is a lot more reliable.

To prevent bandwidth overload a complaint is not immediately made after a misdemeanour - or else a DoS attack could be made against the network by a rapid succession of bad transactions, resulting in a rapid number of corresponding complaints across the network, and a snowball effect ensuing. Instead complaints (if any) are stored when the bad transaction is made, and then transmitted at certain intervals of time. There is a limit on the number that can be made (10 in this implmentation), to avoid overloading.

## 3.5 The EternityClient

The EternityClient is a class that handles the conversion of files into chunks hosted on the network, and the download and recombination of chunks. It can be run off the command line, or through a GUI implementation.

### 3.5.1 Uploading

The upload process is slightly long-winded, for reasons described below. The protocol runs as follows:

1. The client initially contacts one known server on the network, P.
2. P replies, with EternityPeer details of other online trusted servers.
3. The client contacts each of these servers requesting space.
4. Each server replies saying how much capacity it can offer.
5. If there is enough capacity on the network, then the client encrypts the file with a secret key, and turns it into a set of chunks.
6. The client allocates chunks to servers, the number per server proportional to the amount of free space offered, and then pairs partners up.
7. The client uploads to each the chunks allocated but with no details of partners (which thus makes chunks initially unusable). It asks in return for signatures certifying it hosts those chunks.
8. The client finishes the uploads by sending each server the signatures from the servers that host the partners of that server's chunks. The chunks are now complete and viable.

This is illustrated in the diagram below:

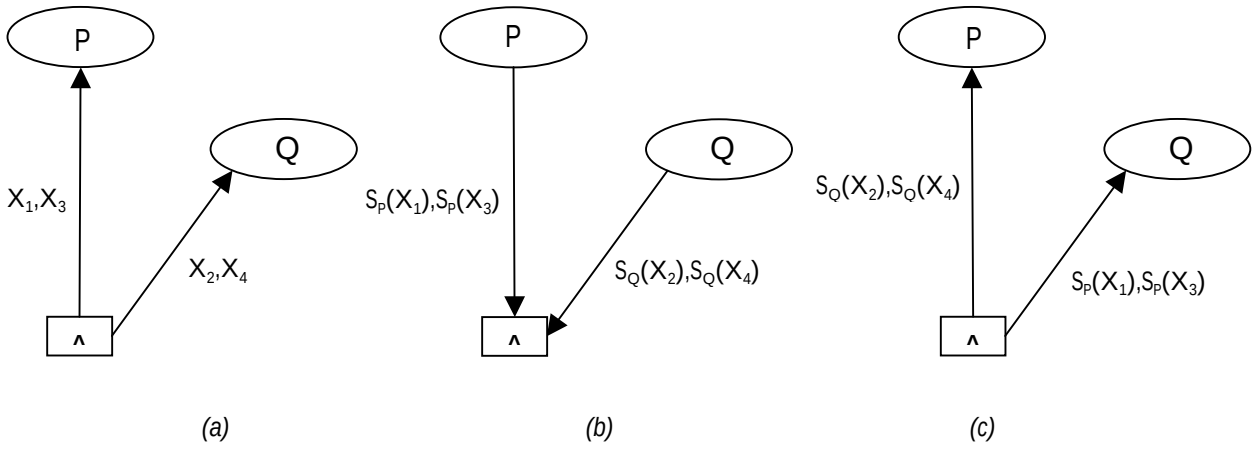


*Fig 3.2: Example of an upload from client A to servers P and Q. In this case, $X_1$ & $X_2$ and $X_3$ & $X_4$ are pairs of partners. (a) $X_1$ & $X_3$ are uploaded to P, $X_2$ and $X_4$ to Q. (b) Signatures $S_P(X_1)$, $S_P(X_3)$ and $S_Q(X_2)$,$S_Q(X_4)$ certifying ownership are sent back to the client. (c) The client forwards the signatures to the other host.*

The client will abort if there's not enough space on the network, if there are less than two servers available, or if one server is allocated more than half the chunks available - that way the integrity of the partner system is kept intact.

This process is quite time and bandwidth-consuming - the client has to make three separate connections and queries to each server - one to get the allocation, one to upload data and received signatures, and one to complete the upload. It also wastes bandwidth contacting each server separately, rather than relying on intelligent connections and multi-hop queries. But this the deliberately safe approach with many precautions against network unreliability.

An alternative considered was to use one server as an agent, to take the entire upload at once, and redistribute. However, this places all the digital cash at one point in the network temporarily, even if the coins were encrypted or otherwise protected, the agent node could just maliciously delete the money being uploaded, or it could become a target for attackers.

There is also the question of chunk and partner allocation - a single node on the network might not be trusted to allocate partners. The client is the most trustworthy part of this transaction, as it is the only one that is paying money - a 'customer is always right' mindset, perhaps.

### 3.5.2 Downloading

Downloading is a lot simpler than uploading. In this case, as it is merely copies of data being handled with no coins and thus no threatened damage to integrity, the process can be done through one server acting as an agent. A client sends a download request to the agent, consisting of the chunk hashes generated and a randomly-generated RSA public key, which it distributes to any online servers it knows about.

Each server responds to the agent with data chunk copies that match the hashes, encrypted with the client's public key, so the agent cannot tell which chunks are actually contained. Random data can be added to fool traffic analysis. The agent then forwards the data to the client, which reassembles the chunks into a file, and decrypts them if possible.

# 4 Evaluation

## 4.1 Models and Plans

### 4.1.1 Engineering Model

The implementation of the project can be broken down into five distinct tasks.

1. Network Protocols - Basic communications, network discovery
2. File Splitting & Recombining - Implementing the IDA
3. Uploading and Downloading - Transferring data from Client to Server
4. Perfect Network Operation - Transferring data from Server to Server
5. Imperfect Network Operation - Coping with errors and malicious behaviour

Task 1 and 2 can be independently written, but 3, 4 and 5 all depend on the preceding tasks being completed. With this need for building on past efforts required, I adopted an iterative/spiral model of engineering, with each stage being regarded as a prototype for the next step. Each stage was separately tested to exhaustion before moving onto the next, minimising the possibility bugs persisting throughout the development.

### 4.1.2 Test Plan

After taking into account the requirements and design, I was able to come up with a test plan, detailing what test criteria the network should meet in order to be a satisfactory implementation:

1. In normal operation, any uploaded files must be downloaded intact 100% of the time.
2. Uploads and downloads must scale well with file size.
3. Nodes must audit each other correctly - i.e. for a particular chunk, the node that chunk should be on is always audited. This means that swaps in particular must work - the right partner needs to be informed of the change.
4. If some of a file's chunks are lost on the network, then the file should still be available as much as possible, as long as the proportion of servers lost is smaller than the redundancy built in to the split chunks (i.e. smaller than $\frac{n-m}{n}$)
5. If a server acts maliciously, then the other servers must drop it from the network quickly.

Tests 1-3 could be made on a normal, uncorrupted network (thus stage 4 of the implementation). Test 3 needs additional testing on an imperfect network, i.e. when the final implementation was complete. Tests 4 and 5 also need to be done at this time.

## 4.2 Implementation

### 4.2.1 Network Protocols

The basic RMI protocol was implemented on two machines, each acting as client and server. This was followed by the addition of SSL cryptographic and authentication procedures so that machines could swap “Hello World” messages securely. The EternityObject class and some of its basic descendents (such as EternityHello) were also created, with the simple network joining protocol implemented successfully.

Some rudimentary packet sniffing software was used to make sure the machines were genuinely transmitting encrypted data - it would not do to rely on unknown software libraries without making sure.

### 4.2.2 File splitting and recombining

After I could guarantee a viable network protocol could be produced. I could start to implement the IDA algorithm. This did not rely on using a network connection, so this module could be coded over the Christmas holiday at home.

Writing the IDA module took more time than expected, as although the algorithm worked on small amounts of data (up to 256 bytes), when applied to realistic file sizes (up to 100k) problems with performances and overflow started occurring.

This was due in part to my over-reliance on existing software libraries for matrix operations (Which IDA relies on, see Appendix A). Sun’s own simple Matrix class in the Java API, used for 3D modelling, was slow and unsuitable for working on matrices representing filestreams. A faster implementation from NIST [7] was used, but this was not in modular arithmetic. Eventually I wrote my own ModularMatrix representation, based on NIST’s source code.

Testing of IDA was done on files resident on a single machine, with no network operation involved, to minimise any bugs appearing. First the ModularMatrix was tested, trying to invert large and very large matrices of random byte values.

Once I was satisfied it could invert and multiply 5000x5000 matrices without error, I implemented the system for IDA. I tested it on 20 files, ranging from 1 byte to 100k in size, and of different types (Testing on ASCII would not be rigorous enough, as ASCII only has a limited range of byte values, not all 255). IDA was successful in splitting and recombining every single one.

This work took longer than expected - I had a delay of approximately two weeks in trying to get the Matrix and ModularMatrix implementations fully working.

### 4.2.3 Uploading and Downloading

Once this had been achieved the network protocol and IDA modules could be combined to produce a simple network for uploading and downloading. Client software that took care of splitting the file and creating EternityLocator objects was written, while servers had proper upload and download methods added.

As client software is meant to be used by anyone, a simple GUI interface was built using Java Swing. This was not too complex, just having two functions - Upload and Download - but incorporated step-by-step guides to uploading a file for the average user to understand.

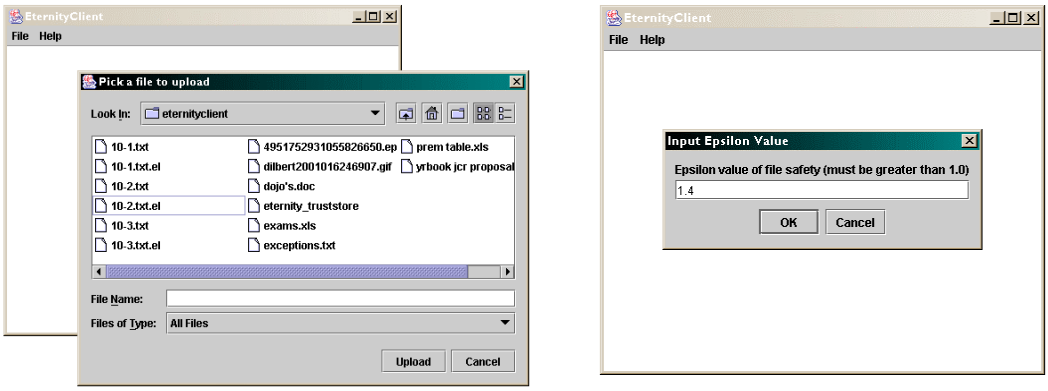


*Fig 4.1 Sample screenshots of client software in upload mode. The left hand screen is asking the user to pick a file. The right hand screen is requesting an epsilon (n/m) value of file safety.*

Thanks to the successfully tested and completed modules beforehand, this stage was very quick to write and implement, and required little testing and bug correction.

At the end of this a semi-complete storage network was available. Servers could join the network, and clients could upload data to the servers and download it later successfully. With this infrastructure in place and operational, the network could be expanded, with it being tested up to six extra nodes with no problems.

### 4.2.4 Perfect Network Simulation

After making sure the network had reliable data on its nodes, then the processes for chunk auditing and swapping were added. Although the protocols had been designed beforehand, writing the actual code for them allowed me to see deeper into them, and gain a better appreciation into how they worked. As a result I was able to think about attacks on the system and so refined the design to become more secure - the addition of destination fields and timestamps in EternityObjects is one example.

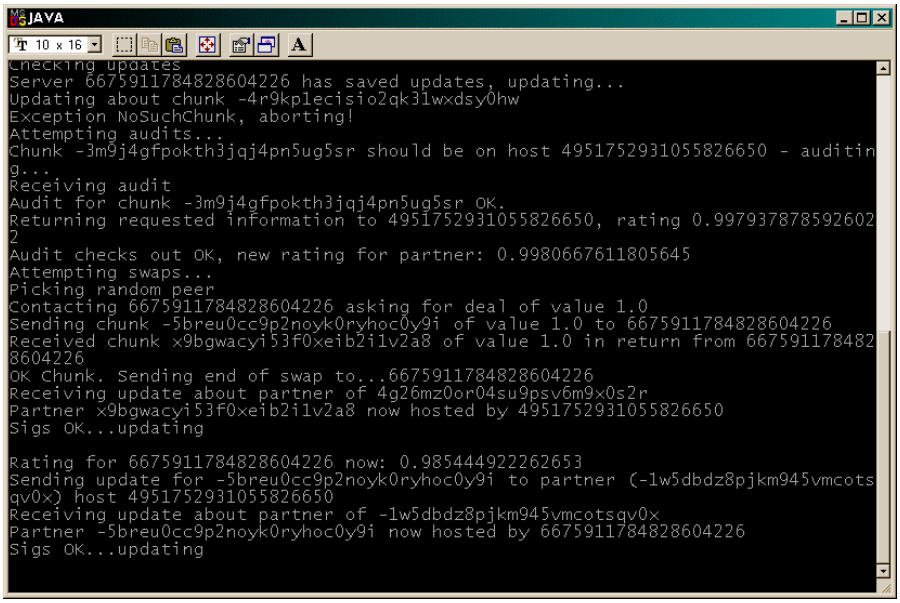


*Fig 4.2 Server in operation, performing audits and updates in normal runtime operation*

An intense period of testing followed. The auditing is simpler, and so was the first to be implemented and tested. Afterwards, the swapping protocols were implemented, before the two combined together.

Now it was possible to simulate proper network operation. The first test was to satisfy requirements of file integrity. Over 500 chunks were uploaded to the network over a period of 24 hours. The data consisted of a variety of file types (e.g. ASCII text, MS Office, GIF, JPEG) and varied in size from 2k to 80k. The network was left to run on its own for 48 hours, and then the data was re-downloaded.

As a hash of the entire file is taken before uploading and stored, it is easy for clients to check integrity. Every file re-downloaded returned a correct hash, so the file correctness rate is 100%, which means Test 1 is satisfied.

With the large amount of widely shared data on the network, this also provided an opportunity to fully test the swapping and auditing methods (Test 3) to see if they worked under proper network conditions. Servers kept logs of any errors that occurred. After inspecting the logs, I found no such errors. The protocols had worked 100% with ‘proper data’ as well, which means that the test passes in perfect operation.

Next we can measure upload and download performance. It is obviously important to have a network that is not only easy to use but also relatively fast and convenient for clients.

The main factor in speed of downloading/uploading is the number of chunks - the more data to be transferred, encrypted, signed and processed, the more time it will take. Transferring is obviously linear, an O(*n*) time process, where n is the number of chunks. Encrypting and signing (which is based on hashing) are also O(*n*) processes.

Processing involves matrix inversion and multiplication, which depends on the product of the different matrix dimensions, which are (From Appendix A) *n*, *m* and *b*,

the block size. *m* and *n* are linearly related, and *b* is constant, so this just becomes $O(n^2)$.

To test this, I uploaded various file sizes, from 10 to 100 chunks to the network of servers, and used the client to time how long uploads and downloads would take. Five sample downloads and upload were taken.

| File size | 10 | 20 | 40 | 60 | 80 | 100 |
|---|---|---|---|---|---|---|
| **Upload** | 4.46 | 6.42 | 10.80 | 16.62 | 22.02 | 29.56 |
| **Download** | 2.58 | 3.95 | 5.28 | 8.57 | 11.81 | 16.31 |

All times in seconds.

These results produce the following graph:

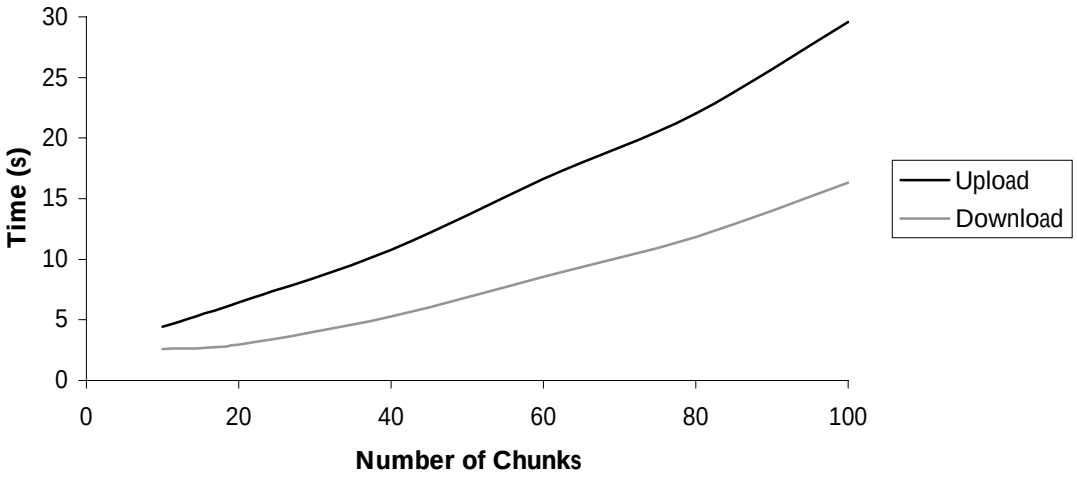


*Fig 4.2: Upload and Download time depending on file size*

As can be seen, the rate of change in both graphs increases, suggests a correspondence with the $O(n^2)$ prediction, however, this rate of change is quite slight, suggesting that even large files don't have a large overhead in reassembly - the linear costs of connecting, signing and encrypting appear to dominate.

No tests were done comparing upload time with network size. Firstly because I don't believe that there is much difference in performance - from my own experience of working the network, connection times are very quick and so whether 2 or 8 connections are needed will probably not have much effect. In addition, the nodes all had very different processors and hard drives, and some of the more demanding processes, such as encrypting and signing data, varied wildly from machine to machine, meaning a fair measure of performance purely based on the number of machines was impossible.

What must borne in mind from these successful results is that they only simulated 'perfect' network behaviour and did not deal with real-life problems, such as disconnections, node destruction and malicious behaviour.

### 4.2.5 Imperfect Network Simulation

To complete implementation, the complaints methods and ratings systems were added and implemented, and full network testing could begin.

First of all, a completion of Test 3 was made. On the standard network of 8, a server would go offline for an hour, and then return to the network. It should be automatically updated with all the information about the movements of its chunks partners when it logs back on, and so should not make any mistaken audits to the wrong server.

Initially this went wrong, which got very frustrating, as it would mean having to reset all the servers on the network and getting rid of the 'bad' data. However, after redesigning the methods so that mistaken audits were given the appropriate pre-saved EternityPartnerUpdate object in reply, and the introduction of timestamps to prevent out-of-date updates interfering, I managed to get a working implementation up.

Next I tested for Test 4, which checks for file integrity when the network undergoes partial damage. This requires a little extra analysis. In theory, if there are *n* chunks uploaded to a network of *k* servers, then as allocation is random, the number of chunks on a server follows a binomial distribution, so there will be $n/k$ chunks per server, on average. There is no better policy for allocation than random, as we cannot have servers divulge information about what they are storing to each other.

But in a random allocation, the number of chunks can vary, as the number of chunks is under a binomial distribution. The standard deviation $\sigma$ of this distribution for a single server is:

$$\sigma = \sqrt{n\frac{1}{k}\left(1-\frac{1}{k}\right)} = \sqrt{n\left(\frac{1}{k}-\frac{1}{k^2}\right)}$$

This can be approximated to: $\sqrt{\frac{n}{k}}$ as k is large enough for $\frac{1}{k^2}$ to be insignificant.

But there are *k* servers on the network, so by taking all *k* samples we divide the standard deviation by $\sqrt{k}$ to obtain the standard deviation for the number of chunks on each server over the entire network:

$$\sigma_k = \sqrt{\frac{n}{k^2}} = \frac{\sqrt{n}}{k}$$

The samples of the number of chunks on each server will follow a normal distribution, with mean *n/k* and standard deviation $\sigma_k$. An average scenario, where each server has between 0 and $n/k$ chunks, will occur 50% of the time under this distribution. If *l* servers are lost, this leads to a maximum of $nl/k$ chunks being lost, so if *m* chunks are needed to resurrect the file, then the file is viable as long as:

$$m > n - nl/k \qquad (1)$$

We can also create more pessimistic scenarios. Each server has between 0 and ($n/k$ + $\sigma_k$) - i.e. one standard deviation more than the mean - 80% of the time. The file stays intact as long as:

$$m > n - (n + \sqrt{n})\frac{l}{k} \qquad (2)$$

Which will hold 80% of the time.

A 'very worst' case scenario is where each server has between 0 and ($n/k$ + $2\sigma_k$) chunks. This will occur 95% of the time, and means *m* has to satisfy the condition:

$$m > n - (n + 2\sqrt{n})\frac{l}{k} \qquad (3)$$

Which will hold 95% of the time. Thanks to the random nature of the network there is no guarantee of 100% availability, although in (3) we get quite close.

By inputting the appropriate variables into equations (1), (2) and (3) and rearranging, the values of *n* for different values of fixed *m* required were computed as follows:

| *m* | **50%** | **80%** | **95%** |
|---|---|---|---|
| **10** | 14 | 15 | 16 |
| **20** | 27 | 29 | 31 |
| **40** | 54 | 56 | 59 |

*Table 4.1*

To test this, I set up a network of 8 nodes, and uploaded three sample files, of m size 10, 20 and 40, each with the three corresponding *n* sizes from the table above. After a suitable long period of network operation (24 hours in this case). I randomly disconnected 2 of the nodes to simulate attack, and attempted to re-download the nine files. These were repeated ten times at 12-hour intervals, and the results given below.

| *m* | **Successful Attempts out of 10** | | |
|---|---|---|---|
| **10** | 7 | 8 | 9 |
| **20** | 6 | 8 | 10 |
| **40** | 7 | 8 | 10 |

*Table 4.2*

After accounting for error of $\pm 0.5$, the failure rate is roughly as expected, in fact actually slightly better for the 50% model. This may be due to the approximations - as the figures in Table 4.2 have to be rounded up to the nearest whole value, they are 'better' than the actual floating-point figure to achieve their respective percentages.

Random error may have also played a part. However, more extensive testing was not possible, because of the long allotted time between samples. Alternatively, my model

could be flawed and too pessimistic, but being pessimistic is not a crime when concerned with an inherently paranoid network such as this.

This analysis does give one useful conclusion - and that is that files are more secure when they are split up into many chunks rather than few. For example, to guarantee 95% coverage, if a file is 10 chunks in original size, then you need an *n* of 16 (a factor of 1.6). A file of 40 chunks in size needs an *n* of only 59, (a factor of 1.475).

Small files, thus, should either be padded out (which is an expensive waste of data), or chunk sizes be made variable so that there are many chunks available for small files. A minimum chunk size of 512 bytes means 10k file is now 20 chunks in size rather than 10. This relatively uncomplicated feature has been easily incorporated into the client design.

The most pleasing result that only a few extra chunks are needed to be created for a file to go from 50% availability to 95% availability - approximately a 10% increase in the above tests - which is good for both clients and servers

This means that there is only an approximately 50% guarantee that a file of size *m* split into *n* chunks will withstand a loss of a proportion of the network $\frac{n-m}{n}$ in size. But an increase of *n* to $^{11}\!/_{10}\,n$ will guarantee 95% availability, which is good enough to satisfy Test 4.

### 4.2.6 Malicious behaviour

#### 4.2.7.1 Data Destruction

To check for Test 5 - the network's tolerance to malicious behaviour - I constructed a server that would automatically destroy one-fifth of any chunks uploaded to it. This joined up to form part of a network of eight, and the number of audits made by each node before rejection was recorded. This was repeated five times, each with a 'brand new' malicious server joining.

The nature of the auditing system would mean that one-fifth of audits will fail. So as audit after audit is made, the rating will be suitably altered until the average rating falls below 0.9. To avoid teething troubles when a node first joins (It may not have any chunks to start off with) a minimum of five trades with it are needed before its rating is taken into consideration.

How can we calculate this? The process can be mapped as a stochastic problem like the Gambler's Ruin. We initially start at position 0 on a line. For every successful audit we mover forward one place. For every failed audit we move backward nine places. The ratio between successes and failures has to be at least 9:1 for the server to be deemed trustworthy, if this ratio is any smaller then we fall backward more places than forward - i.e. if the position becomes 0 or less.

A simple program was made to simulate this behaviour to reach a numerical solution - the average result over millions of trials is approximately 5.93, which means servers

should take an average 6 audits (and thus, 6 swaps, assuming swaps and audits are at the same rate) before realising the malicious server is no good.

Initial tests, however, showed that servers were not noticing such intransient behaviour at all, and would continually keep a rating of approximately 0.94 or more for that server. The reason being that the runtime method also makes and receives swaps. For every audit A makes to B, on average, A will send a swap to and receive a swap from B. If both of these swaps are valid, then these will add to B's rating. These two extra steps make the above problem into one of three steps forward and seven steps backward, which will very rarely (if ever) return to zero.

In fact this could lead to an attack, whereby B could continually bombard A with valid swaps, which it cannot refuse, to mask its misbehaviour when it came to invalid audits. To combat this problem, I doubled the penalty for a bad audit, and extended the swap feature so that after every incoming swap from B, A would give B another audit.

These modifications to the model now mean it makes one step forward (if audit is successful) or eighteen steps backward (if audit is not successful). There are also two extra possible steps forward, each with a $1/(k\text{-}1)$ probability (simulating two possible swaps that might occur in this round between this server and the bad one - one initiated and one received), and a further repeat of the audit after that.

This, in the simulator (with $k$=8), makes the average number of swaps 6.82, which means an average of 7 are now needed before such a server will regard a node as being malicious.

A second set of tests on the network with such modifications meant a definite rejection of the malicious server for all the nodes, as illustrated in the results below:

| | No. of swaps with bad server before cutting off | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **Mean** |
| **1** | 7 | 3 | 8 | 4 | 11 | 5 | 3 | 5.86 |
| **2** | 3 | 9 | 2 | 12 | 2 | 6 | 4 | 5.42 |
| **3** | 5 | 3 | 10 | 5 | 3 | 6 | 9 | 5.86 |
| **4** | 9 | 3 | 4 | 6 | 5 | 6 | 10 | 6.14 |
| **5** | 4 | 9 | 14 | 3 | 3 | 5 | 5 | 6.14 |

*Table 4.3*

The results are slightly better than the model predicted (an average number of 5.88 swaps to stop). It is noticeable that there are one or two nodes in each test that have remarkably worse numbers than the rest. This is because as the malicious server runs out of partners to trade with, it will focus more on the few remaining nodes that are still willing to trade, and make plenty of swaps with them, so they will take longer for their audits to bring down

But if we now combine in the complaints system, then we should be able to pre-empt this. With two servers downgrading the malicious server for every bad audit, we increase the ratio of audits to swaps, and distribute the audits more evenly across the network. The results are as follows:

| | No. of swaps with bad server before cutting off | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **Mean** |
| **1** | 4 | 5 | 2 | 3 | 3 | 2 | 3 | 3.14 |
| **2** | 2 | 3 | 3 | 4 | 1 | 2 | 3 | 2.57 |
| **3** | 2 | 2 | 2 | 2 | 3 | 2 | 2 | 2.14 |
| **4** | 1 | 2 | 1 | 3 | 3 | 5 | 3 | 2.57 |
| **5** | 2 | 2 | 1 | 3 | 2 | 3 | 2 | 2.14 |

*Table 4.4*

This reduces the average to 2.51, over a 50% reduction, which is a marked improvement. This shows that the complaints facility is definitely worth having to speed up network-wide detection of malicious behaviour.

Test 5 demanded a ‘quick’ response to malicious behaviour. 2-3 swaps is particularly quick (It corresponds to about 30 minutes of real time), and approaches as short as possible a time a node can know another before making a judgement on its behaviour.

#### 4.2.7.2 Server Victimisation

However, Test 5 may not just apply to malicious behaviour to everyone on the network. One particular form of attack is if a malicious server attacks one server with corrupted swaps while behaving perfectly well with other members.

One server A on the network of eight was set up to swap bad data chunks when offered good ones to another (B), while behaving perfectly normally to the others. B would send complaints to C, D,..., H whenever it was treated badly, and would continue to do so until B’s rating of A dropped below the threshold.

However, the complaints system was only of limited use. B would typically stop trading with A after 6 or 7 bad trades, and so the complaints didn't continue. As the complaints were dispersed over the entire network, their effect was quite negligible.

One modification in testing was to send all the complaints to just one server (e.g. C). Then server C would indeed downgrade A but often not enough to take it beneath the 0.9 threshold before B stopped trading with A and stopped making complaints. This is because A was returning enough good trades and audits to C to keep its rating up.

Raising the threshold would make it easier for C to stop talking to B, but would also reduce the number of complaints B could make before stopping itself.

The only way to give C enough complaints about A would be for B to carry on trading with A even after the threshold had been crossed, but B is unlikely to do this - it can just cut its losses and ignore A forever. The self-interest aspect of a network's behaviour means it is unlikely to risk more data and money to alert other servers.

In conclusion, the complaints system, while good theory, is less useful in practice when concentrated attacks on individual servers are made.

Despite this, I feel the system is good enough to pass Test 5, although alerting abuse of individual servers to the entire network definitely needs further research.

### 4.2.8 Miscellaneous attacks

There are a number of other attacks possible on the system, and it was not possible to test all of them. However, I have put some thought into how the system could be attacked.

#### 4.2.8.1 Denial of Service

A node could be bombarded with swaps or audits, causing it to overload and go offline. This is a potential problem as there is some fixed overhead in each connection.

Such a DoS attack would have to be carefully constructed - transactions are refused if the chunk involved is invalid, and audits have to come from the proper partner host. So it would have to come from servers who have validly worked for a period of time until they have enough resources to attack.

The network is designed to prevent DoS attacks from snowballing - if lots of bad transactions are made, then the complaints they generate are stored and only sent when the next complaint send is due. Complaint traffic is kept constant within maximum limits. The worst possible knock-on effect is when a swap occurs - as the new hosts have to send updates to their partners. But at most only two updates have to be sent, they are relatively small compared to chunks, and can go randomly to any node on the network. Also this means the attacker has to use proper chunks and coins, which means a lot of work in collecting them beforehand if it is to mount a sustained attack.

Some primitive anti-DoS attack precautions could be added at the application level (e.g. refusing an audit/swap within ten seconds of the last one from a host), but this would be less effective against distributed attacks from many sources. Also, this would mean that a SSL connection would be set up regardless, which takes up a lot of the overhead in a transaction. A better solution is to implement proper anti-DoS measures at the network level with established anti-DoS software.

#### 4.2.8.2 Hoarding

Servers are not required to move specific pieces of data, they could do with any. This may mean that low-value chunks are rapidly moved around the network, while a greedy server will hoard high-value ones.

However, this is not as bad as it seems. For the server hoarding the expensive chunk will still be audited regularly for it, so this does not impact availability. All this does is increase the risk the server has of having that chunk detected on it. This risk is offset by the extra value of the chunk possesses.

#### 4.2.8.3 Withholding Chunks From Clients

A server could well provide correct answers to all audits from its peers, but give nothing when demanded to download, effectively cutting off the data it hosts. One solution this would be for each server to check what chunks in a download request are partners of its own chunks, and for it to demand the partner host to send a copy. However, this would result in $O(n^2)$ connections being set up between servers each time a download was requested, which would not scale very well. With more intelligent connection this cost could be reduced, possibly.

# 5 Conclusions

In all, I am satisfied that the network produced was an effective implementation of an Eternity Network. The system satisfied the tests that I planned for, although my work only scratches the surface of what could be achieved - safe distributed storage is a complex subject matter.

## 5.1 Further Development

There are further areas that time and resources did not permit me to properly research, but could be borne in mind for future development.

### 5.1.1 Time

The system relies on timestamps to make sure that, for example, updates are fresh. However, there is no unified timing system on the network, and so the reliance is ad-hoc. With the small number of network nodes in the test belonging to the same time zone and with accurately set clocks, this problem was not really apparent, but I came to realise that for a large network on machines with less reliable clocks, over many time zones, discrepancies can occur, and may even become security faults - if an attacker knows a node's clock is set several hours slow, it can resend out-of-date updates which will still be valid on the faulty node and change its perception of the network.

Attempts were made to find some synchronization software using NTP (Network Time Protocol), a system of servers worldwide that accurately return the current time . Although this gives the network some degree of centralization and thus vulnerability to attack, there are enough NTP servers around the world to make it secure enough.

However, no suitable Java backend applications that could integrate with Eternity could be found, and there was not enough time to create my own NTP implementation.

### 5.1.2 Server-to-server anonymity

Although all communications are encrypted, the fact that a particular IP address is an EternityServer is not a secret. As servers are known, the owner may come under attack for running it, regardless of whether the data it holds. The financial incentive counters the risk involved, but it may still not be enough. Further extension of Eternity could possibly involve steganographic methods or anonymising protocols such as Onion routing, which could hide any evidence of Eternity's existence in the first place.

### 5.1.3 Scalability & Network Conditions.

Due to limited resources I could only test the network on a maximum of eight nodes, all run within my college Ethernet. Although the design assumed that Eternity was meant to be a (relatively) small-scale network, it could easily expand to a large-scale network.

If so, then more efficient network architectures would have to be implemented, and efforts made for more efficient routing. This might mean some of the underlying protocols would have to be altered or even dropped- RMI and SSL are point-to-point protocols, and trying to implement them as hop-by-hop protocols would be demanding and raise lots of questions on guarantees of security and reliability.

In addition, little testing could be done under more demanding conditions - on computers separated across the large geographical distances, on networks with large lag times or high packet drop rates.

### 5.1.4 Misbehaviour

The complaint communication between nodes is perhaps the biggest shortfall in the system. Although it does work to speed up a server being rejected because of malicious behaviour, it only does so if that server maliciously misbehaves to many servers at once.

But the problem when only individual servers are the target of bad trading remains. Rather than sending complaints to the just one server, a complaining server could send it to many or even the entire network, but then this becomes a Denial of Service weakness - one server's misbehaviour could create a large amount of traffic as the network becomes flooded with complaints.

A 'real-world' parallel could perhaps be drawn here - a crime with only one relatively insignificant victim is hardly going to be as much concern to a society as one that is perpetrated against many members, a harsh but true fact of life. However, it is still desirable to stop as much 'crime' as possible on the network.

If progress was made on scalability as outline in 5.1.3, then a more efficient/intelligent network architecture could enable broadcasts to become efficient, meaning that such behaviour could be exposed without spamming the network.

## 5.2 Final Conclusions

I enjoyed working on this project, and feel I have created a useful and workable application for distributed file storage. The network is (relatively) easy to set up and run for a node operator, although it does demand a fairly fast processor (For the heavy crypto workload) and a fast, persistent Internet connection for servers. If Eternity, as Anderson outlines, becomes some sort of business venture, this should not be a problem. The client software has an easy-to-use GUI and requires no more capability than a normal home PC and modem.

I have learnt quite a lot in constructing this program, especially on networking and cryptography. My research also taught me a lot about the state of the art and current research in the field. In bringing the project from concept to reality, I've learnt that it is possible to make elegant ideas come to life, but that it's also a lot of hard work. The biggest part of this project was thinking about managing the network when things go wrong - when they go right it is far simpler - and I feel I've been taught an awful lot about how to deal with faults and errors.

I am glad I managed the foresight to allocate spare time at the end, as some of the trickier parts of implementation (notably the IDA and the tricky swap protocol) took more time than expected.

In retrospect I should have been more analytical when it came to design - although I managed to get the correct requirements for the project, it wasn't until actual coding did I get a full grip of the algorithms and protocols involved, and thus wasn't able to fully think about attacks on the network. I therefore had to keep evolving the design against each of the new attacks I came up with, at the implementation stage, which made the work more frustrating.

Despite this, I managed to get the implementation fully working only a couple of weeks behind schedule, and I feel I fulfilled the specification to a more than satisfactory level. The technology and concepts I have encountered are at the cutting edge of computer science, and I feel especially pleased that I've been able to contribute to such a field in this way.

# Bibliography

1. Ross Anderson, *The Eternity Service*, 1996.
http://www.cl.cam.ac.uk/users/rja14/eternity/eternity.html

2. Gnutella
http://www.gnutella.org

3. The Free Network Project
http://freenet.sourceforge.net

4. The Free Haven Project
http://www.freehaven.net

5. Publius
http://www.publius.net

6. Bruce Schneier, *Applied Cryptography*, Second Edition, 1996

7. National Institute of Standards & Technology (US)
JAMA: A Java Matrix Package
http://math.nist.gov/javanumerics/jama/

8. Cryptix 3.2.0
http://www.cryptix.org/products/cryptix31/index.html

9. Java Crypto & Security Implementation
http://www.wedgetail.com/jcsi/2.2/base/index.html

10. David Chaum, *Online Cash Checks*, 1989
http://www.chaum.com/articles/Online_Cash_Checks.htm

11. Michael O. Rabin, *Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance*, 1989

# Appendix A: Rabin's Information Dispersal Algorithm

Rabin's IDA takes a file $F$ and splits it into $n$ pieces, such that any $m$ pieces where $m < n$ can be used to reconstruct $F$.

First a prime $P$, where $P > 256$ is taken. For the Eternity service $P = 257$.

Determine a block size $B$ (e.g. 1024 bytes), and pad the file so that $|F| = mB$. Fill a $m \times B$ matrix $S$ with the data so that each continuous block of data occupies one row.

Find a value $k \geq 1$ which is the margin of error for the file - e.g. if you want the file to be retrievable even if half the dispersed chunks are lost, then $k = 2$. Find integer $n = \lceil km \rceil$

Create an $n \times m$ matrix $A$ such that each row of the matrix is linearly independent of all the others. Such a matrix can be constructed by assigning each element in $A$ the value:

$$A_{ij} = i^{(j-1)} \bmod P$$

Find the matrix product $C = A.S \bmod P$ (i.e. standard matrix multiplication, but with all elements mod $P$). This is an $n$ x $B$ matrix, each row representing one chunk. These chunks can now be dispersed. Each chunk records its row number $i$ with it, for reasons as described below.

To resurrect, we take a collection of any $m$ chunks and place them in an $m \times B$ matrix, $C'$. Now construct a m $\times$ $m$ matrix $A'$, with each element assigned the value:

$$A'_{ij} = c_i^{(j-1)} \bmod P$$

Where $c_i$ is the original row number for the $i$th row of $C'$, which was stored earlier.

We have basically reconstructed $A'$, the submatrix of $A$ that when multiplied with $F$, produced the submatrix $C'$. i.e.

$$C' = A'.F \bmod P$$

Which means that:

$$F = (A')^{-1}.C' \bmod P$$

So we find the modular inverse of $A'$, and perform the above equation. The modular inverse of a matrix is just like the inverse of a matrix, but instead of dividing each element by the determinant you multiply it by the inverse of the that element with respect to the determinant mod $P$. As by definition $A'$ only contains linearly independent elements, then there exists as determinant, and therefore inversion is guaranteed to have a definite result.

# Appendix B: Test Network Information

The test network used for this project consisted of between 2 and 8 network computers, all located on the Corpus Christi College Ethernet. For convenience and efficient work, the majority of work was done on a ‘mini network’ of two computers (Both residing in my room), for initial code testing, before the network was extended to the other 6 nodes for full-blown testing.

The nodes were all Intel machines, running Java JRE 1.4 on either Windows or Linux environments. Each has a 10Mb/s network connection. The computers varied in speed and capability, from a reconditioned Pentium 100 Mhz with 64Mb RAM to a Athlon 1.5Ghz with 512Mb RAM.

The client was typically run from one of my own computers, a PIII 500 with 128Mb running Windows.

# Appendix C: Sample Source Code

This is the active auditing method for the server. The runtime thread calls this at irregular but frequent intervals throughout its operation.

```
/* Audits other servers. It picks a random server and sends
 * a challenge to provide the data from a chunk. It then hashes
 * that chunk and checks it for integrity
 */
private void doAudits() {

  // If I host at least once chunk, and I know of more than
  // one peer (i.e. someone other than me)
  ;audit:if (chunkstable.size()>0 && alivePeers.size()>1) {

    /* Pick a random chunk with an alive peer. If after 10 tries it cannot
     * find such a chunk, then abort. */
    EternityPeer ep = null;
    EternityChunk ec = null;

    ;forLoop: for (int i=0; i<10; i++) {
      ec = getChunk(null);
      ep = alivePeersLookup(ec.getPartnerHost());
      if (ep==null) break forLoop;
    }
    if (ep==null) {
      return;
    }


    // Connect to the server, if unable, abort
    EternityServerInt esi = connectTo(ep);
    if (esi==null) {
      return;
    }


    // Make a request object for this chunk's partner
    long l =  sr.nextLong();
    EternityAuditRequest ear = new EternityAuditRequest(IDno, l,
                    ec.getPartnerHost(), ec.getDataHash(), ec.getPartner());

    SignedObject so = signObject(ear);

    ep.addOffered(ec.getCoin().getValue());
    writePeers();
    /* Audit is treated as a trade of same
     * value as the chunk being audited */

    // Send this chunk on and try to get a reply.
    SignedObject so2;
    try {
      so2 = esi.audit(so);
    }

    // Handle connection error
    catch (RemoteException re) {
      removeFromAlivePeers(ep);
      return;
    }

    // Handle Eternity error
    catch (EternityException ee) {
      String m = ee.getMessage();
```

```
      if (m.equals("BadPartner") || m.equals("NoChunk")) {
        EternityAuditComplaint eac =
            new EternityAuditComplaint(IDno, sr.nextLong(), l,
                                       ec.getPartner(), ec.getPartnerDCHash(),
                                       ec.getChunkSig(),
                                       so, ee);
        addtoComplaints(eac);
        ep.addOffered(ec.getCoin().getValue());
        writePeers();          // Double the penalty paid for by ep
      }
      else {
        processException(ep, ee);
      }
      return;
    }


    // Extract a reply
    EternityAuditReply reply;
    try {
      reply = (EternityAuditReply) forceObject(so2);
    }
    catch (EternityException ee) {
      processException(ee);
      return;
    }


    // Check the hash of the reply, if good then credit it
    // with a successful 'trade'

    byte[] replyHash = makeHash ( new Object[] { reply.getData() } );

    if (Arrays.equals(replyHash, ec.getPartner()) &&
        Arrays.equals(reply.getDCHash(), ec.getPartnerDCHash())) {

      ep.addTraded(ec.getCoin().getValue());
      writePeers();
    }

    // Else create a complaint...
    else {
      EternityAuditComplaint eac = new EternityAuditComplaint(IDno,
                            sr.nextLong(), l,
                            ec.getPartner(), ec.getPartnerDCHash(),
                            ec.getChunkSig(),
                            so, so2);
      addtoComplaints(eac);
      ep.addOffered(ec.getCoin().getValue());
      writePeers();          // Double the penalty paid for by ep
    }
  }
}
```

# Appendix D: Package and Classes

`EternityServer`
`EternityServerInt`
The Eternity Server, and the RMI Interface it implements

`EternityParameters`
Used to hold private parameters (ID number, store size etc.) for a server

`EternityClient`
`EternityClientGUI`
Command-line and GUI implementations of the Eternity Client

`DigitalCoin`
Simple Digital Coin Implementation

`EternityChunk`
Data chunk with coin

`EternityDataChunk`
Data chunk with no coin

`EternityLocator`
List of a file's chunks and (optionally) the decrypt key

`EternityDownloadRequest`
Sent by client to servers to download chunks

`EternityException`
Signed Exception for Eternity-specific errors

`EternityObject`
Base class for all objects sent between servers

- `EternityJoin`
  Used by server joining network for first time

- `EternityHello`
  Used by server reconnecting to network

- `EternityPeer`
  Record of node's public key and rating

- `EternityAuditRequest`
  `EternityAuditReply`
  Messages used in an audit

- `EternitySwapQuery`
  `EternitySwapStart`

`EternitySwapReply`
`EternitySwapFinal`
`EternityPartnerUpdate`
Messages used in a swap

`EternityAuditComplaint`
`EternitySwapComplaint`
Complaint objects

`IDA`
`LUDecomposition`
`ModMatrix`
Implementation of the Information Dispersal Algorithm, and supporting mathematical tools

`RMISSLClientSocketFactory`
`RMISSLServerSocketFactory`
Custom socket creation classes for server and client