A P2P Backup System for Small and Medium-sized Enterprises

Fernando Meira

Kongens Lyngby 2007
IMM-MSC-2007-24

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-MSC: ISSN 0909-3192

Summary

This thesis presents the design and implementation of the All-or-Nothing package transform, presented by Ronald Rivest in 1997 [24], and its subsequently integration in an existing peer-to-peer backup system. This encryption mode provides a third alternate way of ensuring confidentiality, integrity and availability of backups.

Resilia, the existing prototype where this extension is built on, combines peer-to-peer technology with secret sharing and distributed backup algorithms in order to provide a robust, safe and secure environment for backups, required by decentralized and off-site storage of data. The backup schemes used by the application increase the availability of the distributed data and achieve an efficient space and communication usage, if associated with a replication scheme, allowing the reconstruction of backups even in the case of failure of some peers.

The evaluation of the designed and implemented package transform revealed good results, taking 25 seconds to compute and distribute a 10 Mb backup file to 3 peers. Discovering and fixing a tampered share in one of the peers took approximately 7 seconds, considering the time to re-send the backup file to the incorrect peer.

Preface

This thesis was prepared at Informatics Mathematical Modelling department, at the Technical University of Denmark in partial fulfillment of the requirements for acquiring an M.Sc. degree in Computer Science and Engineering.

The thesis deals with the applicability of different secret sharing algorithms or distributed backup in a peer-to-peer system. In particular, it implements the All-or-Nothing package transform proposed by Ronald Rivest [24]. The developed system integrates with an existing safe and secure backup system which implements an information dispersal algorithm and a secret sharing scheme.

Lyngby, February 2007

Fernando Meira

Acknowledgements

First, I thank my supervisor Christian D. Jensen for his support, guidance and good comments throughout the project. And also for believing in me for the second time.

My family, in particular my wife and son, are specially thanked for the love, support and patience for all the time I spend on the computer.

Last, a huge thanks is due to all who helped me throughout the project. To Dagur Gunnarsson and Tore Bredal Nygaard for good comments and motivation during different stages of the project, and to Raghav Karol and Ana Giral for providing good company during the time we shared the office.

Contents

Chapter 1

Introduction

Most small and medium-sized enterprises (SME) operate from a single address, which means that backups are normally kept at the same physical location as the company's computers. This means that fire, flooding or other disasters are likely to destroy both computers and the backups that were meant to ensure the continued operation of the company. The goal of this project is to develop a safe and secure backup system that allows a company to distribute its backup among a number of servers, thereby ensuring availability, without compromising the confidentiality and the integrity of the backup.

Peer-to-peer (P2P) technology is increasingly recognized as an attractive way to distribute information without creating bottlenecks or a single point of failure in the system. This project will therefore investigate the applicability of different secret sharing algorithms or distributed backup in a P2P system. In particular, this project will implement the Package Transform by Ronald Rivest [24].

The project will follow an empirical approach through design, implementation and evaluation of a distributed backup system. The developed system should integrate with an existing safe and secure P2P backup system [19], which implements a secure backup system using Rabin's information dispersal algorithm [23]. This implementation builds on a previous project [21], which implements a very different idea, namely Shamir's secret sharing scheme [25]. The combination of all three protocols into a single backup application should provide system administrators in SMEs with a choice between alternate ways of ensuring the confidentiality, integrity and availability of their backup. It is important that the final backup system is robust and that usability is taken into account when the system is designed.

1.1 Project Goals

As follows from the project definition, the main goals set for this project are:

- extend the existing prototype and application design in order to accommodate a new alternative way of backing-up and distributing data among a peer-to-peer network;
- design and implement the Package Transform by Ronald Rivest [24];
- design a fault-tolerant replication scheme for the new backup scheme that allows to split the backup into different blocks, reducing in this way the communication and storage overhead.

The existent prototype in which this extension should integrate has taken into consideration the security of backups. Therefore, the current extension bases its design on the existent protocols for achieving a similar degree of security. The implemented components and the security constructs used on data structures are still considered efficient and secure, and are therefore used by the new protocols.

The results achieved by this extension are positive. The designed and implemented AON protocol is efficient, robust, performs in acceptable time and provides a good level of availability, integrity and confidentiality to the backup. Improvements of the IDA algorithm also presented positive results, making it outperform the previous version.

1.2 Thesis Organization

With the intention of making this thesis self-contained, some sentences, figures or tables from the two theses [21, 19] which are directly related to the design and implementation of Resilia, are, entirely or partially, re-used in this thesis. To clearly distinguish those parts, the symbols "†" and "‡" are used respectively to refer if that piece of information is taken from Nittegaard-Nielsen's M.Sc. thesis or Meira's Final Year thesis.

This thesis proceeds with a theoretical overview of the state of the art of the technology involved in this extension. Chapter 3 presents a discussion over the existing prototype and an analysis on how the components have been implemented during previous development. It also argues about the problems of the existent designed protocols. Thus, these first chapters form an introductory part, where the context, technologies and aim of the extension is described.

The following chapters form a practical part that is carried out during this project. Starting in Chapter 4 by outlining the design of the new protocols and the changes to the structure of the application and its components. Chapter 5 presents in some detail how the implementation of the protocols and application changes were performed. The design and implementation are thereafter evaluated in Chapter 6, and finally a conclusion over the work done is drawn in Chapter 7, along with some consideration for future work.

CHAPTER 2

State of the Art

This chapter introduces the background theory on which this project is based. It briefly reviews the two schemes that were implemented in the two previous versions. A more in-depth analysis is dedicated to the algorithm that the current project is to implement. An analysis of this algorithm and some alternatives are also presented. Some notions of data replication are introduced, as well with two replication models. The chapter ends with a short presentation of the peer-to-peer platform that is used in this project and by reviewing projects with similar goals.

2.1 Secret Sharing Schemes

A *secret sharing scheme* is a model for distributing a *secret* among a group of participants. The secret information is divided into n shares, where n is the number of participants, in such a way that with any m valid shares, where $m \leq n$, it is possible to reconstruct the secret. Any attempt to reconstruct the secret using up to $m - 1$ shares is unfeasible and discloses no information about the secret. This is mentioned as a (m, n)-*threshold scheme*.

A secret sharing scheme is limited in two aspects: the size of the shares and random bits. Each share needs to be of at least the same size as the secret itself. If this is not the case, an attacker holding $m - 1$ shares will be able to learn some information about the secret. Assuming that the final share is secret, all $m - 1$ shares still reveal some information. This information cannot be secret, therefore must be random.

Shamir's [25] secret sharing scheme (SSS) uses the above concept over a polynomial interpolation, knowing that $n+1$ points are required to determine a n-degree polynomial. Once m shares suffice to restore the data, a $(m-1)$-degree polynomial is then used, which is defined by

$$f(x) = a_0 + a_1 x + \cdots + a_{m-1} x^{m-1}$$

over a finite field. The coefficient a_0 represents the secret, which is the polynomial intersection with y-axes. Any other point of the polynomial is to be distributed to participants. Figure 2.1 presents two examples of the scheme, in a geometrical point of view, for $m = 2$ and $m = 3$, where it can be seen the secret, point (0,s), and other points that would be distributed to participants of the sharing[‡].

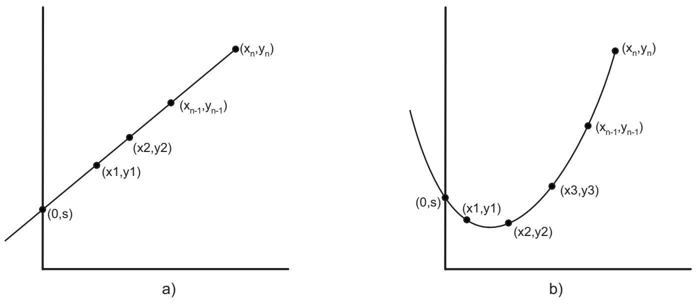


Figure 2.1: Polynomial geometrical representation: a) $(2, n)$-scheme b) $(3, n)$-scheme

This scheme has two flexible proprieties. It allows different levels of control over the secret, by giving more than one point to the same participant. The more shares one participant has, the more control he has over the secret, since less participants will be necessary to reconstruct the secret. The second propriety allows that new participants are added to the sharing after the first distribution has been made, without affecting the existing shares.

2.2 Information Dispersal Algorithm

The information dispersal algorithm (IDA) was proposed by Rabin [23] in 1987. It is a similar technique to SSS, but it allows a reduction of the communications overhead and the amount of information kept at each site, by sending only partial information to sites.

The IDA makes use of a secret key vector, which can be seen as a $n \times m$ matrix, to process the input data. As in the SSS scheme, n stands for the number of participants of the sharing and m the number of required participants to recover the secret. For the main operation in the algorithm, the key matrix is repeatedly multiplied by fix-sized blocks of input data, which can be seen as a $m \times 1$ matrix, until the entire data is processed.

In a formal way, defining the key vector as a set of n vectors, $V_1, V_2, \dots, V_n$, each of length m, ($V_i = (a_{i1}, \dots, a_{im})$), with the condition that any subset of m vectors are linearly independent, and the input data as $b_1, b_2, \dots, b_N$, then the IDA result, c, is achieved by combining values in the following way, for $i = 1, \dots, n$ and $k = 1, \dots, N/m$:

$$c_{ik} = a_{i1} \cdot b_{(k-1)m+1} + \cdots + a_{im} \cdot b_{km}$$

The IDA process adds some bits of redundancy that allows some possibly existing communication errors to be corrected at the reconstruction stage. From each resulting block of a complete key vector computation, that is $c_1, \dots, c_n$, a value c_i is sent to each participant. To note that there is a link between all n key vectors, all n participants and all n values of each resulting block. Participant number two must always receive value number two of each resulting block, which are represented by vector key number two. This link is important at the reconstruction stage.

Thus, each site will store a share of the size $|F|/m$, where $|F|$ is the size of the original data. This represents a total overhead of $((n/m) - 1) \cdot 100$ percentage of the size of the original data[‡].

To reconstruct the original data is required the presence of m valid shares. With less than m shares, it is infeasible to restore the data. The reconstruction of the data is done by

$$b_j = a_{i1} \cdot c_{1k} + \cdots + a_{im} \cdot c_{mk}$$

where $1 \leq j \leq N$. The key vector used for the reconstruction is only composed by the vectors that correspond to the participants' shares used. This results in a $m \times m$ matrix, which is inverted before the reconstruction of the data.

2.3 All-or-Nothing Encryption

An *All-or-Nothing* (AON) encryption paradigm is a mode of encryption that provides the following property: the *entire* ciphertext must be decrypted in order to gain access to *any* piece of the plaintext. The original all-or-nothing scheme was described by Ronald L. Rivest in *All-Or-Nothing Encryption and The Package Transform* [24] in 1997.

In a more formal way, an AON is a bijection function ϕ defined on a finite alphabet A, mapping an input, say $X = (x_1, \dots, x_s) \in A$, to an output, say $Y = (y_1, \dots, y_s) \in A$, where if at most $s - 1$ values of the output Y are known, then any input x_i ($1 \leq i \leq s$) value is unknown, but if all s values of Y are known, all values x_i ($1 \leq i \leq s$) can be found.

The primary motivation behind AON is to make brute-force attacks more difficult. For example, when using a cipher-block chaining (CBC) or an electronic codebook (ECB) encryption mode, among others, an attacker can search the whole key space and try each key against only one cipher block. Holding one cipher block, being that the first or any other, an attacker can keep decrypting it using all possible keys until the result makes any sense. An encryption mode that has the property of obtaining one block of plaintext by decrypting *just* one block of ciphertext is known as *separable.* The AON paradigm guarantees a *strongly non-separable* property. In other words, it makes it infeasible to obtain one block of plaintext without decrypting *all* the ciphertext blocks. This property slows down a brute-force attack by a factor equal to the number of ciphertext blocks.

Some block-encryption algorithms, such as DES, are vulnerable to exhaustive key-search attacks due to their key size restriction. However, DES and similar algorithms are standards and still used in many applications and are therefore implemented in several hardware devices. Using the AON scheme as a pre-step of the encryption, it is possible to extend the life of these algorithms and reduce the urgency for replacing all the cryptographic hardware devices in activity.

2.3.1 Rivest's package transform

The scheme presented by Rivest introduces a "package transform", hence the name *All-or-Nothing Transforms* (AONT), as a prior step to an ordinary encryption. For an input message with plaintext blocks $x_1, x_2, \ldots, x_s$ and a random key K' of the same size of the blocks, this package transform computes the output sequence $y_1, y_2, \ldots, y_{s+1}$ in the following way:

$$y_i = x_i \oplus E(K', i) \qquad \text{for } i = 1,\ 2, \ldots,\ \text{s}$$

and

$$y_{s+1} = K' \oplus h_1 \oplus h_2 \oplus \cdots \oplus h_s,$$

where

$$h_i = E(K_0, y_i \oplus i) \qquad \text{for } i = 1,\ 2, \ldots,\ \text{s}$$

and K_0 is a fixed, publically-known encryption key. The computation of h_i can be seen as performing one-way hash function over the exclusive-or of the output encrypted block and the value i. The $E(K', i)$ represents an encryption function for a block cipher, for example CBC, where K' is the secret key and the value i the data to be encrypted. It is important that the key K' is randomly generated, so that a known message does not yield a known output. This scheme is displayed in Figure 2.2.

Since the input message should be reconstructed at some point, it is important that the AONT is invertible. This is shown as follows:

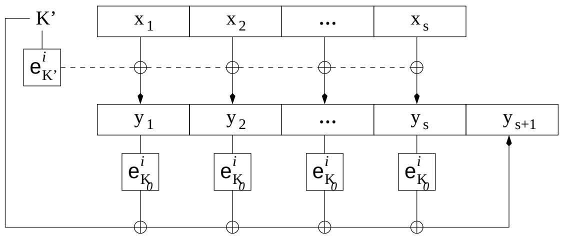


Figure 2.2: Rivest's AONT

$$K' = y_{s+1} \oplus h_1 \oplus h_2 \oplus \cdots \oplus h_s$$

and

$$x_i = y_i \oplus E(K', i) \qquad \text{for } i = 1, 2, \ldots, s.$$

It can be seen that if *any* block y_i is not present, it is infeasible to compute K' and therefore *any* message block. Moreover, modifications to the ciphertext, such as permuting the order of two blocks or duplicating blocks, will likely result in computing the wrong key K'. While adding an extra layer of security, increasing the time required to search for the correct key by a factor of n, where n is the number of blocks in the ciphertext, this scheme adds twice the cost of the actual encryption to the total cost of the process. To be noted that the key K' is part of the pseudo-message. Thus, an AONT cannot be seen as an encryption process, since there is no secret key involved.

As seen above, the main idea behind an AONT is to ensure that *all* blocks of a message are present at the decryption point. However, this has a problematic downside. If *any* of the blocks is unavailable or corrupt, the message cannot be reconstructed correctly. To cope with this error-propagation property, one is advised to transmit the ciphertext blocks in a reliable way. In order to allow the detection of corrupted ciphertext blocks, one can append one block of redundancy to the message before the AONT takes place. This block can be, for example, the sum of all the previous message blocks. In fact, the AONT can be seen as a n-out-of-n secret-sharing scheme, where all the n blocks are required to reconstruct the original message.

Another drawback of this model is that its security factor depends on the size of the ciphertext. Hence, short messages will present a lower security factor than larger messages.

2.3.2 Model discussion

Following the original AONT presented by Rivest [24], some papers were subsequently published addressing the security of the model and providing a formal description of the scheme.

Stinson [27] provides a formal analysis of the AONT restricted to unconditional secure types of transforms, as compared to Rivest's computationally secure scheme. That allows him to prove that if any output block of size n is unknown, then any input block can have 2^n possible values. However, Stinson's transforms are of a linearity and non-randomizing character, which should not be used over plaintext blocks in order to prevent an attacker to learn information departing from the linear dependencies between different input blocks.

Stinson's construction uses the function $\phi(\mathbf{x}) = \mathbf{x}M^{-1}$, where $\mathbf{x}$ is a vector of s message blocks in the Galois Field for some q prime value, $GF(q)$, and M is an invertible s by s matrix over $GF(q)$, such that no entry of M is equal to 0. He then proves that function ϕ is a linear (s, q)-AONT, since $\mathbf{x} = \phi(\mathbf{x})M$.

Boyko [4] pointed out that Rivest's and Stinson's definition of an AONT were not precise regarding the amount of information, in terms of bits, needed to be learned by an adversary to prevent any leak of the input message. He studied the *semantic security model* of the scheme, that is, the ability to hide *all* the information about the input whenever *any* part of the output is missing. Rivest only considers two cases: when the whole output is known—allowing one to compute the input—and when all but one *complete* block of the output is known—making it infeasible to compute the input. In the semantic security model, Stinson's definition is not secure, because it is possible to learn linear relations among the elements of $\mathbf{x}$ by looking at elements of $\phi(\mathbf{x})$. Rivest's definition is considered secure along with strong assumptions about the block cipher.

2.3.3 Model variants

In Boyko's [4] analysis of an AONT, he showed that Bellare and Rogaway's *Optimal Asymmetric Encryption Padding* (OAEP) [2] yields an AONT. However, this was proved to be true in the *Random Oracle model*, which provides only a limited security in real-life schemes. The main idea behind the random oracle model is that all protocol parties, that is, legitimate users and adversaries, have access to the random oracle. This model simplifies the problem formulation and provides analysis to security concerns. However, it has the disadvantage of being an unrealistic assumption, and therefore replaced by a hash function in real implementations.

The OAEP was introduced for constructing semantically secure public-key cryptosystems. Along with parameters n, as the length of the message, and k, the security parameter, it uses instances of the random oracle, the "generator" $G : \{0,1\}^k \rightarrow \{0,1\}^n$ and the "hash function" $H : \{0,1\}^n \rightarrow \{0,1\}^k$, for defining the transform OAEP: $\{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^{n+k}$ as:

$$\text{OAEP}(x, r) = x \oplus G(r) \mid\mid r \oplus H(x \oplus G(r)),$$

where $||$ denotes concatenation, x the input message and r a random string.

The input message can be recovered only if *all* the output y is known,

$$y = x \oplus G(r) \mid\mid r \oplus H(x \oplus G(r)),$$

and referring to the first half of the OAEP as y_L and to the second half as y_R, one can compute:

$$\begin{aligned} x' &= H(y_L) \oplus y_R \\ &= H(G(r) \oplus x) \oplus (H(G(r) \oplus x) \oplus r) \\ &= r \end{aligned}$$

and by knowing x', one can compute the input message x as

$$x = y_L \oplus G(x') = (G(r) \oplus x) \oplus G(r).$$

As it is shown, the recovering process requires only two XOR operations.

Boyko provides formal definitions and proofs about the polynomial and semantic security for AONTs, formulated in terms of a single random oracle. He shows that the proposed OAEP is an AON that satisfies these definitions with bounds nearly optimal. That is, an adversary facing an OAEP is in no better place than against an exhaustive key-search. Furthermore, he concludes that no AON construction can achieve better results than the OAEP. With respect to Rivest's package transform, he shows that it does not fulfill the security definitions in a probabilistic setting. For sake of simplicity, Boyko's definitions and proofs are not described here. Those interested may consult Boyko's paper [4].

Rivest [24] refers that the package transform can be seen as a special case of the OAEP, where

$$G(x) = E(x, 1)||E(x, 2)|| \cdots ||E(x, s)$$

and

$$H(x) = \bigoplus_{i=1}^{s} E(K_0, x_i \oplus i).$$

Canetti et al. [6], while studying the problem of partial key exposure, proposed an *Exposure-Resilient Function* (ERF) as an alternative to AONT, which they refer to as much more efficient in settings such as private-key cryptography. They define an ERF as a deterministic function whose output appears random even if almost *all* the bits of the input are revealed. The main difference between this alternative and the AONT is the "secret" element. In AONT, the secret $\mathbf{x}$ is stored as $y = \phi(\mathbf{x})$, whereas in ERF the secret $f(r)$, a (pseudo) random value, is stored as r. Since the value r is shorter than the secret $f(r)$, it allows the ERF to be performed in a more efficient way.

Formally, a l-ERF is defined as a function $f : \{0, 1\}^n \rightarrow \{0, 1\}^k$ for $l < k \leq poly(n)$, where l is the security parameter. The function f is defined as

$f(r) = M \cdot r$, where M is a k by n matrix and $r \in \{0,1\}^n$. They provide a construction of an AONT using ERF in following way:

$$T(x;\ r) = \langle r, f(r) \oplus x \rangle$$

where

$$T : \{0,1\}^k \to \{0,1\}^n \times \{0,1\}^k$$

and $x \in \{0,1\}^k$ is the input message. The secret key K' from Rivest notation is here in the form of r and the public component h_i as $f(r) \oplus x$.

The above construction was shown secure in the standard model, that is, without random oracles. However, Desai [12] refers to it as an impracticable option due to efficiency issues.

A new characterization of AON is given by Desai, which provides more efficient ways of instantiating the AON paradigm. He uses the notion of the Shannon Model block cipher [26] to prove it secure. That is, an exhaustive key-search is slowed down by a factor equal to the number of blocks in the ciphertext, as initially mentioned by Rivest. Desai's definition considers information as in blocks and does not regard Boyko's [4] critic on information leaked from one particular block. He argues that this type of formalization weakens the efficience of encryption modes.

The construction provided by Desai [12] is based on the counter mode of encryption (CTR). The transform CTRL of block length l uses a key length k where $k \leq l$. The algorithm is similar to Rivest's package transform with one difference, the "hash" pass is skipped altogether:

$$y_i = x_i \oplus E(K', i) \qquad \text{for } i = 1,\ 2, \ldots,\ \text{s}$$

and

$$y_{s+1} = K' \oplus y_1 \oplus y_2 \oplus \cdots \oplus y_s.$$

This algorithm has a cost of only two greater than the encryption mode, whereas Rivest's AONT has a cost of three. Even with a weaker definition, Desai proves it secure based on his characterization. He bases his proof on the fact that as long as at least one block of output is missing, one cannot reconstruct the key K'. Under Boyko's [4] security sense, the CTRT is insecure, but secure under Rivest's [24] sense and in the Shannon model.

Desai's scheme is displayed in Figure 2.3.

Canda and Trung [5] propose a new mode of using AONT that is faster than Rivest's design and provides a security gain for short messages. Their work is motivated on Rivest's design security being dependable on the message length. They suggest to apply the AONT *after* the encryption. This allows the use of a non-randomized AONT which results in a performance improvement. Their

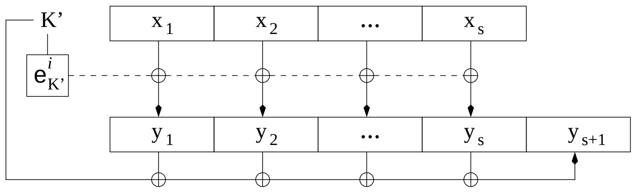


Figure 2.3: Desai's AONT

AONT function can be defined as follows, being $y = (y_0, y_1, \ldots, y_s)$ the output of a CBC encryption over a message $x = (x_1, x_2, \ldots, x_s)$:

$$\phi(y) = z = (z_0, z_1, \ldots, z_s)$$

where

$$z_i = y_i \oplus y_{i-1} \qquad \text{for } i = 1, \ldots, s$$

and

$$z_0 = y_0 \oplus \lambda \cdot y_s$$

where $\lambda \neq 1$ is a constant. This transformation ϕ is a Stinson-like linear AONT. The last block of the output z_s is masked with a pseudo-random constant K', derived from secret key K by a *w-slow one-way function* $f_w : \{0,1\}^n \rightarrow \{0,1\}^n$. This function computes K' as $f_w(K) = K_{w+1}$, where K_i is computed as

$$K_i = E\Big(K_{i-1}, (K_{i-1} \oplus K_{i-2})\Big) \qquad \text{for } i > 1$$

with $K_0 = 0$ and $K_1 = K$. This scheme is displayed in Figure 2.4.

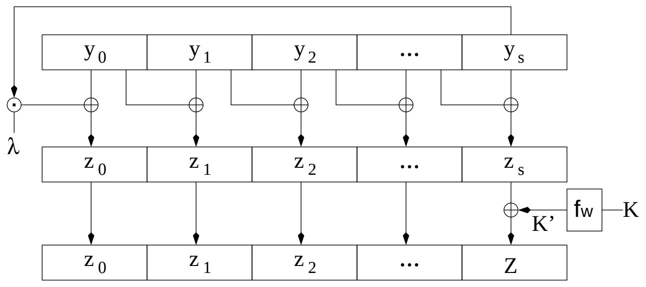


Figure 2.4: Canda and Trung's AONT

The inverse transformation is computed in the following way:

$$y_0 = \frac{1}{1 \oplus \lambda} \cdot (z_1 \oplus \lambda \cdot \sum_{i=2}^{s} z_i)$$

and

$$y_i = z_i - yi - 1 \qquad \text{for } i = 1, \dots, s.$$

This design slows down an exhaustive key-search by a predefined factor w. This brings an advantage over Rivest's model by allowing an adjustable security factor, independent of the message length. To illustrate the performance gain when using this scheme, the Table 2.1, taken from [5], presents the number of operations that are executed depending on two factors: the number of blocks s of the message and the security gain w.

Table 2.1: Comparing operations to be executed between Rivest's and Canda and Trung's mode

s	w	Rivest's mode	Canda and Trung's mode		
10	10	30 $\times e_K$	20 $\times e_K$	10 $\times$ add	1 $\times$ mul
100	100	300 $\times e_K$	200 $\times e_K$	100 $\times$ add	1 $\times$ mul
1000	1000	3000 $\times e_K$	2000 $\times e_K$	1000 $\times$ add	1 $\times$ mul
10	10000	30000 $\times e_K$	10010 $\times e_K$	10 $\times$ add	1 $\times$ mul
100	10000	30000 $\times e_K$	10100 $\times e_K$	100 $\times$ add	1 $\times$ mul
1000	10000	30000 $\times e_K$	11000 $\times e_K$	1000 $\times$ add	1 $\times$ mul

The first part of Table 2.1 presents the case when the number of blocks in the message is equal to the desired security gain. This scheme outperforms Rivest's by requiring only $2 \times s$ encryptions. Although it also requires s additions and 1 multiplication, these operations are executed faster than e_K. Thus, this scheme will perform better, specially for long messages.

The second part of Table 2.1 presents the case when the desired security gain is bigger than the number of blocks in the message. Rivest's scheme needs to pad the message to a total length of w blocks and encrypt each one 3 times. Canda and Trung's scheme only needs $s + w$ encryptions and therefore is significantly faster.

2.4 Data Replication

Data replication consists of distributing copies of data, called *replicas*, among different storage sites. The motivations for replicate a piece of information are: to improve the performance, by allowing users to access nearby replicas and avoiding remote network access; to enhance information availability, that is, access to the data even when some replicas are unavailable; and to make a system fault-tolerant, guaranteeing a correct behavior, or access to correct data, despite a certain number and type of faults. If up to f of $f + 1$ sites crashes, there is still one remaining copy of the data. In the presence of up to f Byzantine failures, a group of $2f + 1$ sites can provide a correct behavior [8].

Having data replicated on different locations has advantages and disadvantages. It provides reliability and fault tolerance to a system, or data, at the cost of data redundancy, storage space and consequently communication overload. Having several copies stored in different sites, provides reliability in a way that even if some of the sites become unavailable, and some other sites become corrupt, there will still be some sites storing a correct copy of the data. It increases the possibility of obtaining the correct data when a disaster strikes, by natural reasons or not. It also provides reliability in the sense of discovering what is the correct data in the case of having sites storing different copies of it. This assumes that most of the sites work correctly and hold a valid copy of the data. One can then compare all copies and assume as the most common one to be the correct.

The reasons are very similar for achieving fault tolerance as a result of replication. A fault tolerant system is a system that can continue to operate correctly in the event of the failure of some the components or services. Having replicated components or services allows one system to use a backup component or service instead of the faulty one. The same idea stands behind fault tolerance of data. Having malfunctioning or unavailable sites, or corrupt data in some sites still allows one to access a correct copy of the data if it is replicated in more sites.

There are two replication models for fault tolerance: *passive*, or *single-master*, and *active*, or *multi-master*. In the former model there is only one writer, the *master*. All updates originate at the master and then are propagated to other replicas, or *slaves*. This model provides limited availability and relatively large overheads. On the other hand, the active model allows that updates are submitted by multiple sites independently. All masters have equivalent roles. Although it provides higher availability, this model is more complex. It has to manage conflicts and operations scheduling. An increased conflict rate can also limitate the scalability of an active model.

Data replication models can still be divided into two categories: *pessimistic* and *optimistic*. Pessimistic approaches, also called *traditional*, are focused on maintaining a single-copy consistency. For achieving that, pessimistic algorithms synchronously coordinate replicas during accesses and block access to them during an update. An optimistic approach focus on providing an enhanced performance and availability to the data, but sacrificing temporally the consistency of all replicas. It allows users to access data without a prior synchronization to verify if the data is up to date. In summary, each approach take one side in a trade-off between availability and consistency, typical in distributed systems.

2.4.1 Design Choices

When designing a replication scheme, there are a few key-points to take into consideration. Although there is one main goal to maintain data consistency, by keeping several replicas spread among different sites in the most similar state of consistency, different systems use different designs.

Besides the number of masters, one has to take into consideration how the operations are transfered between sites, how to schedule these operations in a way to produce equivalent states on all replicas, how to handle conflicts, how modifications are propagated and what guarantee of consistency is appropriated.

There are two types for defining the operations: *state transfer* or *operation transfer*. A state transfer system will restrict an operation either to read or to overwrite the entire object. It is simple, because the consistency is maintained by sending the newest replica to all replicas. In contrast, an operation transfer system will transfer the operations required to arrive to the newest replica. In this way, all sites will receive a set of operations that will allow them to alter the replica to its newest state. This allows a more flexible conflict resolution, once all sites hold a history of operations.

The scheduling of operations can be done in a *syntactic* or *semantic* manner. A syntactic scheduling sort all operations based on time and owner properties. A semantic scheduling is based on semantic properties, such as commutativity of operations. Although syntactic scheduling is simpler, it may cause unnecessary conflicts.

To be accepted, operations need to satisfy their preconditions. Otherwise, a conflict takes place. A pessimistic algorithm prevents conflicts by blocking or aborting operations. A single-master system avoids conflicts by accepting updates from only one site. A system can also ignore conflicts and let a newer operation overwrite it. The remaining systems can be divided in *syntactic* and *semantic* handling conflicts policies. A syntactic policy relies on the timing of an operation. A semantic policy exploits the semantic knowledge of operations.

When an operation is executed at one site, it must be propagated to all other sites. In a *pull*-based system, sites will poll other sites for new operations. In a *push*-based system, the site that performs the operations sends them to all sites. Generally, the quicker the propagation, the more consistent the replicas will be, and therefore the less the rate of conflict. But at a cost of more complexity and overhead.

There are a few difficulties that a pessimistic replication scheme needs to withstand. It needs to ensure the correctness of all replicas. All copies of the same logical data must agree on exactly one current value – a correctness criteria named *single-copy serializability*. At the other end of the spectrum, in a optimistic algorithm, the guarantee is only that the state of replicas will eventually converge – called *eventual consistency*. In this case, it may be observed sites working on replicas in different states, or even incorrect.

2.4.2 Replication Models

There exist different replication models, each one more appropriate to some type of system. Following are presented two models that suit the nature of this project.

2.4.2.1 Read-one Write-all

In this simple model [3], proposed by Bernstein, Hadzilacos and Goodman in 1987, read operations are done from only one site, the local one to minimize the communication cost, while write operations are performed at all sites.

A more real-life model that do not require that all sites will be available to perform an update is called *read-one write-all-available* (ROWAA).

2.4.2.2 Majority Quorum

A quorum system is defined as a set of subset of sites, or quorum, with pair-wise non-empty intersections. The main idea behind this model is that a quorum of sites will take decisions on behalf of the whole system, and still guarantee overall consistency.

There are different variations based on this idea. Here is mentioned only the version that best relate to the scope of this project. In a majority quorum (MQ) [29], read and write operations must fulfill the following constrain. A read operation must be performed from at least half of the system sites, when there are an even number of sites, or by a majority if there are an odd number of sites. A write operation must be performed by any majority.

2.4.2.3 Models Comparison

Jiménez-Peris et al. [17], presented a comparison between some replication models (including the above mentioned), according to scalability, availability and communication overhead. Their results conclude that ROWAA model offers a good scalability, very good availability and a reduced communication overhead, specially in networks working with multicast operations. It has also the advantage of being simple to setup and implement.

The MQ model can be a better choice in terms of scalability if the system is very write intensive, close to write only.

2.5 JXTA

JXTA [20] is a peer-to-peer platform developed by Sun Microsystems that provides standardize functionality to enable developers to create interoperable services and applications. The JXTA protocols were designed to be independent of any programming language, transport protocols and deployment platform, to allow that any digital device on a network to be able to communicate and collaborate as a peer.

JXTA provides an easy way to network devices discover each other, self-organize into groups, advertise and discover network services, communicate with each other and monitor each other.

2.5.1 JXTA Architecture

JXTA is composed of six protocols that provide all the functionalities to create a decentralized peer-to-peer environment. The *peer resolver protocol* is used to send a query to any number of other peers and to receive a response, the *peer discovery protocol* is used by a peer to advertise its own resources and to discovery other peers contents, the *peer information protocol* is used to obtain diverse peer information, the *pipe binding protocol* is used to create communication paths, by means of pipes, between peers, the *peer endpoint protocol* allows a peer to discovery routes between peers and finally the *rendezvous protocol* is used to propagate messages in the network [15].

The JXTA architecture comprises three layers, as shown in Figure 2.5.

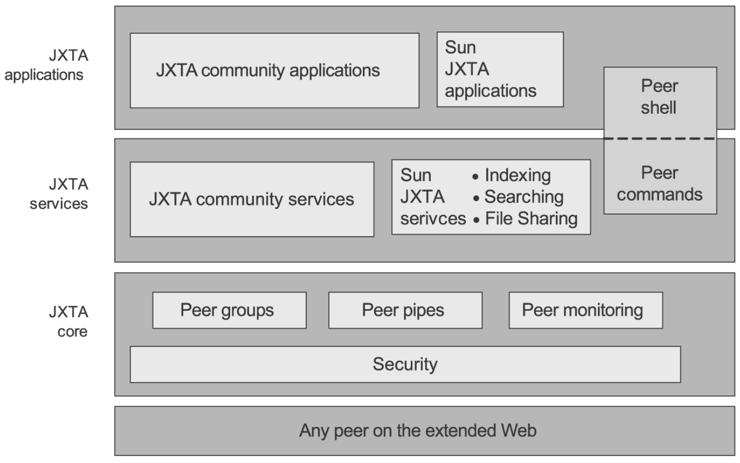


Figure 2.5: JXTA architecture

In the *core* layer can be found the implemented protocols. It is through the protocols that all functionalities are provided to a peer. This layer serves as foundation to services and applications. In the *services* layer can be found the services that use the implemented protocols to accomplish a task. For example, the searching for resources or documents, or authenticating a peer. In the *application* layer are the developed applications that use JXTA capabilities.

2.5.2 Peers

A peer is any networked device that implements one or more of the JXTA protocols. Each one is identified by a unique ID. It can be of four types. A *minimal edge peer* can send and receive messages, but does not cache advertisements or route messages to other peers. They are normally devices with limited resources. A *full-featured edge peer* can send and receive messages and cache advertisements. It replies to discovery requests but does not forward discovery requests. They are the most common type of edge peers. A *rendezvous peer* is like any edge peer, but also forwards discovery requests to help other edge peers to discover resources. An edge peer when joining a peer group seeks for a rendezvous peer. If one is not found, it will become a rendezvous peer for that peer group. This type of peers maintain a list of other rendezvous peers. A *relay peer* maintains information about the routes to other peers. When a peer cannot find a route information in its own local cache will query relay peers for it. Relay peers also forward messages on the behalf of peers that cannot access another peer, due to firewall or NAT problems.

2.5.3 Peer Groups

A peer group is a collection of peers that have agreed upon a common set of services. Each one has a unique ID and can be open to all peers or protected, requiring some type of credentials to join. When a peer is initiated it is by default joined to a common group called Net Peer Group. Furthermore, a peer can be part of different peer groups simultaneously.

A peer group provides different useful services. Discovery services, that allows peers to search resources within the group, membership service, controlling the membership application of each peer, pipe services, used to create and manage pipe connections between peers in the group, among others.

2.5.4 Pipes

Pipes are the message transfer mechanism used for communication in JXTA. They bound two peers allowing them to communicate. The endpoints of a pipe are referred to as the *input pipe*, the receiving end, and the *output pipe*, the sending end.

Pipes provide two types of communication. *Point-to-point* pipes connect two pipe endpoints together, one input pipe to one output pipe. *Propagate* pipes connect one output pipe to multiple input pipes. A secure version of a point-to-point pipe, providing a secure and reliable communication channel is called *secure unicast pipe*.

2.5.5 Advertisements

JXTA resources are represented by advertisements – XML documents used to describe and publish the existence of peer resources. Advertisements are published with a lifetime associated to the availability of the resource. There are different types of advertisements depending on the resource, such as peer, peer group, pipe and rendezvous advertisement, among others.

2.6 Related Work

There are a few projects presenting alternative ways of performing backups using a peer-to-peer platform.

pStore combines peer-to-peer systems with techniques for incremental backup systems [1]. It includes support for file encryption and versioning. pStore allows insertion, update, retrieve and delete of backup files. Files are split into equal-size data blocks and stored in a distributed hash table (DHT). Efficiency is achieved by updating only modified shares, especially when different versions present minor changes. It shares the same goals as Resilia, but off-site storage is fully replicated, requiring higher resource-usage, and its security relies on ownership tags. Furthermore, pStore is a pure research project, with no implementation[‡].

DIBS is a freeware backup system that performs incremental backups, hence handling versioning [18]. Like pStore, unchanged files or shares are not updated for the sake of bandwidth. It uses Gnu Privacy Control (GPG) to encrypt and sign transactions in order to achieve confidentiality and authenticity. Robustness is guaranteed by using Reed-Solomon codes, a type of error correcting code that provides resilience against a limited number of communication errors[‡].

Pastiche [10] is a cooperative backup system where selective nodes share a significant amount of data. Similar peers are identified through the use of fingerprints, in a way of predicting the amount of data in common between them. The owner of remotely stored data performs periodically checks to its status. If the check fails, a new replica replaces the old one. Pastiche provides mechanisms for confidentiality, integrity and detection of failed or malicious peers[‡].

Samsara [11] enforces a fair peer-to-peer storage system without requiring trusted third-parties. Peers willing to store data in samsara have to guarantee that they can provide the same space amount for other peers. It ensures availability and durability through replication, and is used as punishment mechanism for cheating nodes, that have eventually lost data. Samsara was designed as an extension of Pastiche[‡].

The *CleverSafe Dispersed Storage* [7] is an open-source application that is able to disperse a document to 11 storage locations throughout the world. It is implemented in C++ programming language and uses a version of Rabin's

IDA to disperse the information. The storage locations are part of a grid, which keeps data private and safe from natural disasters.

CleverSafe IDA, also named CleverSafe Turbo IDA, disperses the data into 11 slices, each one stored in a different node of the grid. To retrieve the information, at least 6 nodes need to be available. This setup is fixed and cannot be altered by a user. A grid is already setup and available to users, although it is possible that users setup their own grid.

The advantage of restricting the IDA setup to a 6-out-of-11 scheme for all users is mainly in the hability to optimize the algorithm for this specific case. The optimized algorithm outperforms significantly the general implementation of Rabin's IDA. On the other hand, it is inflexible for users. Although a 6-out-of-11 scheme represents a good balance between availability and storage overhead, it is not allowed to shift this balance to either side. In other words, it is not possible to increase the availability of an important backup, nor reducing the amount of space used to store a not so important but large backup.

Comparing it to Resilia, CleverSafe provides an already setup grid and a mechanism to store backups without choices to users. Thus, in the point of view of simple end-users, it is an easy-to-use application that does not require to know how the balance of a scheme changes the availability and reliability of the backup.

Resilia provides a more flexible application, also oriented to simple end-users, but requiring some knowledge on how the backup schemes work. Users need to setup their peer-to-peer network, or have access to an existing one, and specify the parameters for the algorithm to be used. Moreover, Resilia offers two other algorithms than IDA to backup data.

CHAPTER 3

Analysis of Resila

This chapter presents an analysis of the design and implementation of the protocols in the existing system.

The existing prototype of Resilia, first developed during Nittegaard-Nielsen's Master Thesis project [21] and later extended during Meira's Final Year project [19], is a working application that allows any group of users to setup a P2P network, to establish peer groups and to distribute, restore, delete and update their files in a secure way. A file backup can be performed in two different manners. In a high level point of view, both protocols differ mainly in what relates to performance and communication-storage overhead. While the SSS outperforms the IDA, mainly because it works only over the fixed-size secret of 2048-bit and not over the whole file itself, the IDA is able to reduce the communication-storage overhead down to approximately the size of the distributed file. In a security point of view, the SSS shares the secret among peers and requires that part of them, m, value defined by the user, must be available to recover the backup. But, it sends the entire backup file in an encrypted form to all peers. The IDA sends the secret to all authorized peers, called *masters*, and computes smaller blocks of the file to be dispersed among all peers, requiring also that m peers must be available to be possible to recover the backup.

That means that an attacker holding the *share data* —the backup file or part of it— can perform a brute-force attack over the file if backed-up using the SSS mode but not if it was used the IDA scheme. Regarding the *share* itself —the metadata of the backup—, the SSS provides a good security by computing the secret-sharing shares and distributing them to peers. The IDA provides the complete share to all master peers and an incomplete share to other peers. That means that all master peers have the complete information about the backup file, including the symmetric key used to encrypt the file and the vector key used to perform the IDA operation, whereas any other non-master peer will only hold information about the settings of the backup, such as the ID and name.

The following sections present a review of the status of the prototype and introduce the different components that make part of the application.

3.1 Application Structure

The structure designed for the first version was maintained on the second version. The application is divided into six packages that group classes belonging to the same category, such as main core classes, objects, platform specification, graphical interface, send and receive classes.

When selecting an algorithm to perform a backup, the protocol takes care of preparing the backup, running the algorithm over the data, creating the shares and sending them to selected peers. However, by using this beginning-to-end process, a user cannot run both SSS and IDA algorithms on the same backup. Once these algorithms work on different types of information of the backup, they could be used together. That would allow to combine the advantages that both schemes offer, but not necessarily eliminate their disadvantages. A more modular structure would then suit better a system that desires to combine distribution schemes and would as well improve the easiness for future extensions. The modular approach is mainly an implementation consideration that would allow to run one algorithm and keep the result at the local peer, allowing in this way that another protocol be used over the same backup. When all desired algorithms have been used, then the protocol can proceed with the distribution of shares and data to peers.

Combinations of algorithms that can be used together are the SSS + AON and the SSS + IDA. The AON + IDA also makes sense but it can result in a very time consuming operation. A more efficient combination of these two algorithms would be by performing the AON over the entire backup file and then the IDA over the last block of the resulting AON protocol operation. This combination would provide two layers of protection, first by using the AON protocol over the backup and second by dispersing the key block in a way that provides a good degree of security and availability.

Applying the SSS to the secret and then dispersing the backup file using the IDA would add extra security and availability to the data share and reduce the communications and storage overhead of the backup. At the same time, the share would have the security propriety that the SSS provides, which would enhance the protection of the share, but reduce its availability, in a way that more master peers would be required to be available so that the recovering would be possible. This drawback could be easily overcome by sending the complete share to all peers instead of to only master peers.

Replacing the IDA by the AON would also provide a similar type of enhancement due to the fact that the SSS does not add any security component to the file itself. If used along with a replication scheme, it can reduce the communications and storage usage.

3.2 JXTA Setup

The first time that Resilia is started, it is necessary to configure the JXTA platform. Users need to input a name for the peer, which does not need to be unique, it serves as an identifier for other peer users. If the peer is directly connected to the Internet, then this is all that a user needs to do. However, not all peers are directly connected to the Internet, for example peers behind a private network which is using Network Address Translation (NAT) or just behind a firewall. Peers under these conditions will need to setup JXTA to be able to contact the outer network. For that they need to be connected to a relay and a rendezvous peer. A peer behind a firewall needs to contact an outside peer and use it to forward its messages. Peers outside of a firewall are, normally, unable to contact peers inside unless they are replying to a request.

The current way of setting up a peer is through the JXTA graphical configurator. It allows to setup HTTP and/or TCP connection to rendezvous and relay peers.

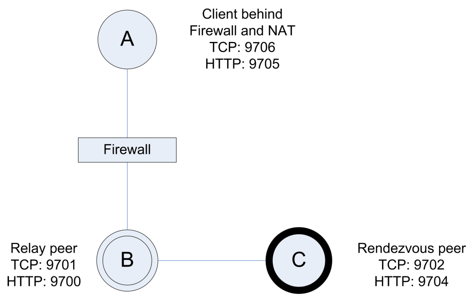

A
Client behind Firewall and NAT TCP: 9706 HTTP: 9705
Firewall
Relay peer TCP: 9701 HTTP: 9700
B
C
Rendezvous peer TCP: 9702 HTTP: 9704

Figure 3.1: Example of client behind firewall and NAT

Figure 3.1 illustrates an example of how peers behind firewalls and in networks using the NAT service can be configured. Both relay and rendezvous peers are directly connected to the Internet. Although all port values differ, they could be 9701 for TCP and 9700 for HTTP, which are the default for JXTA. They are only required to be different if the peers are running on the same machine, for avoiding the use of duplicated addresses. Naturally, the relay peer is configured to act as a relay peer, which is an option when setting up JXTA for the first time, and the rendezvous peer is setup to act as a rendezvous. The relay peer is also setup to use peer **C** as rendezvous, which is not required to be the only rendezvous that the relay peer is connected to. This setup will make both peers connect to each other and share information when they start up.

When the client peer starts, it will perform discovery services to obtain local and remote information about resources, such as peers and pipes. However, once it is behind a firewall, in a private network that is using NAT and it is the only active peer at the moment in that network, it will not succeed in discovering any resource. For that reason, it must be configured for using the relay peer **B**. By providing the correct IP address and HTTP port of the relay peer, the client will be able to connect to it and receive resource information and forward its own messages. The reason to setup the HTTP port is that firewalls normally block traffic which is not HTTP. Also, it may be necessary to setup the peer to use port 80.

3.3 Application Control

In the core of the application sits one class that binds together the functionality of all threads. It is at this central class that all input arrive and from where all jobs depart to be handled by more specialized classes. The input to this class is gathered and queued by a monitor, which will then feed each request to the class. Figure 3.2‡ illustrates the distribution of jobs arriving to and departing from the application main class.

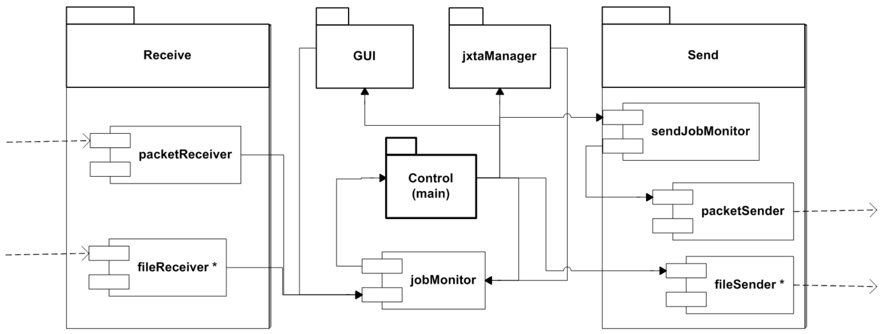


Figure 3.2: `Control`'s job monitor

3.4 Monitors

Monitors are used to provide assistance in managing jobs. Components that are likely to receive many jobs use a monitor to gather and queue them. The queuing strategy used is First-In First-Out (FIFO). They facilitate the handling of requests to a component, allowing in this way that the component performs one entire task without being interrupted queuing or handling other requests.

In Figure 3.2[‡] can be seen two types of monitors, the monitor of the `Control` class and the monitor of the `packetSender` class. All monitors follow the same structure and each one only accepts jobs related to the class they feed, in order to avoid providing to the component a misplaced job that could raise a failure in the application.

3.5 Managers

Managers administrate sections of the application. They run in their own threads, for the sake of efficiency and availability. In this way, components can be focused in special tasks. For example, the platform manager, named `jxtaManager`, manages in an independent manner the platform-related events, such as advertisement discoveries or joining a peer group. All inputs collected by this manager are passed along to the control class where they are processed.

In more detail, the `jxtaManager` implements methods to create, join and leave groups, using an auxiliar `peerGroupTool` class. New groups are created inside the default backup-system group, which are nested inside the common JXTA NetPeerGroup. In this group are published other groups advertisements, so that any peer can be aware of the existence of such groups. Whenever joining a group, the peer creates a `packetReceiver` and set it to listen to incoming messages from that group. Predictable pipe IDs based on peer information are used so that any other peer can easily create a pipe connection without requiring the recipient's pipe advertisement[‡].

The prototype uses three other managers to handle specific tasks. A `share-Manager` to administrate activities related to backup-sharings, a `sendManager` to control the communication of data packets and a `peerManager` to handle the information and methods related to peers.

The `shareManager` implements methods to backup, restore and delete backups for each algorithm. Methods for the integrity check and update shares (only available for the SSS algorithm) operations have their own class. This manager also contains all auxiliar functions common to both algorithms, such as functions to find, load and save shares, among others. Shares are stored inside the local folder `mySharings`, inside one single file for each algorithm, that is, `sharings.ssb` for SSS backups and `sharingsIDA.ssb` for IDA backups. Data shares are stored in the hard disk of each peer, inside of the `encryptedFiles/` folder, using methods from the `fileHandler` class. They are saved under the backup file name, file extension and algorithm extension. For example, `book.pdf.ida` represents the data share processed using the IDA algorithm of the file `book.pdf`.

The `sendManager` takes care of sending packets to other peers. It makes use of a monitor, `sendJobMonitor`, to queue all packets that are then sent by the `packetSender` class, which establishes and maintains connection to one other peer. Thus, this manager holds a list of all `packetSenders`, one for each known peer, created at the moment of the first communication between both peers. *Known peers* are all the online peers that a peer is aware of, and that belong to the same peer group. Packets are communicated between peers using jxta sockets[‡].

The `peerManager` manages peer-related information about the peer running the application and all other known peers to it. It implements methods to add, find and remove peers and peer information. The manager also controls the certificates and operates the keystore, controlling the access to the asymmetric key-pairs, authenticating and validating peers[‡].

3.6 Data Structures

Several data structures are seen as objects in the application, and hence are located in the Objects package, with the exception of the main sharing data structures that are located in the Main package. They can hold information about a peer or a share. Others encapsulate these data objects in order to be used in communication. The following sections introduce the different types of data structures used in the application.

3.6.1 Peer Structures

Every peer node gathers information about all others present in the same peer group. This information comprises the peer ID, name, pipe advertisement and peer certificate, among others. All this information is essential to be able to communicate and authenticate with any other peer in the system.

The information about a peer is stored in a peer structure called `peer`, which is managed by the `peerManager`. Collections of peer structures which belong to a backup process are called *sharing peers*.

Any peer node needs to hold a X.509 certificate signed by a peer group trusted Certificate Authority. The trust in the certificate will decide if another peer will communicate with the peer holding the certificate.

3.6.2 Sharing Structures

Sharing structures are created at the end of the backup stage, or in a update shares process. They include all the information of a backup process and a copy is sent to each one of the sharing peers.

There is a sharing structure for each backup protocol, `sharing` and `sharing-IDA`, because the information required for each one differs. In common, both sharing structures hold information about the sharing peers and about the backup properties, which are comprised in other structure called `sharingData`. This data structure is encapsulated by the sharing structures.

For the case of the IDA scheme, two more sharing structures are used. The `idaShare`, which holds information about each piece of the data share that is sent to a destination peer, and the `receipt`, which serves as a proof ticket for future operations on that backup, holding specific data for that.

3.6.3 Communication Packets

There exist several types of packets in order to carry the appropriate information and transmit the desired request. All jobs, requests, messages, confirmations and accusations are implemented in the form of packet objects. Each one contains the specific data to their task.

Examples of packets are: the `controlJob`, which holds information concerning a job for the `Control` class, the `netData`, which holds information to be sent from one peer to another, as the packet type and the packet time, and the `sharingNetData`, which extends the `netData` packet providing more information to the packet.

These data packets are then encapsulated in communication packets, which can be of two types: a *network data packet* and an *encrypted network packet*, both illustrated in Figure 3.3[‡]. The former type includes the data packet to be transmitted and, along with it, the request type ID and a timestamp of the creation of the packet.

The encrypted network packet includes the network data packet in an encrypted form and cryptologic information to preserve confidentiality and integrity of the data and to provide non-repudiation. To achieve that, the packet contains the symmetric key, used to encrypt the network data packet, encrypted with the asymmetric public key of the receiver, the digital signature of the sender over the hash code of the data packet and the sender's own public key.

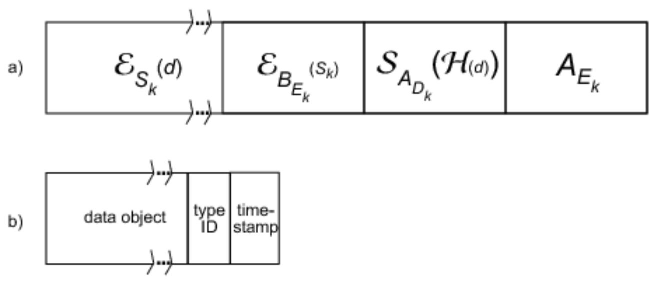

a)
$\mathcal{E}_{S_k}(d)$
$\mathcal{E}_{B_{E_k}}(S_k)$
$\mathcal{S}_{A_{D_k}}(\mathcal{H}(d))$
A_{E_k}
b)
data object
type ID
time-stamp

Figure 3.3: Network packets: a) encrypted network packet b) network data packet

In more detail, the encryption algorithms used in these security operations are‡:

- AES with 128-bit key, for data and packets encryption;
- RSA with 2048-bit key, for session keys encryption,
- SHA-256, for hash code generation;
- SHA1withRSA, for digital signatures.

Although the encryption strength is reduced throughout time, with the invention of newer and more powerful machines, able to compute more operations during the same period of time, the current configuration of the encryption algorithms is still up-to-date, and is therefore assumed as secure. The security algorithms used are provided by the open-source Java Bouncy Castle Crypto APIs from the Legion of the Bouncy Castle [22].

3.7 Communication Protocols

This section introduces how both backup schemes are implemented in the prototype. Since explaining in detail all the steps of all the operations would be cumbersome for the reader, here is only presented an overview of the main operations. This should however be enough to provide a good idea on the proceedings of the protocols. A more in-depth description can be found in the previous reports [21, 19].

3.7.1 Peer Startup

The startup of a peer is composed of two stages: authenticating the user in the application and joining the peer to a peer group.

In order to a user authenticate in the application, he needs to hold a certificate from a trusted Certificate Authority (CA). The authentication is validated by the input of the password for the key associated with the user's certificate. If the user does not hold a certificate, the application will present a form in order to create a certificate request, which the user should afterwards send to the CA. The reply from the CA should include the user and the CA certificates.

After a successful authentication, the `jxtaManager` is started and it creates and joins the default group for Resilia users. Once inside this group, the manager starts the group discovery and the results are passed through to the interface and presented to the user. This protocol is illustrated in Figure 3.4.

Depending on the intentions of the user, he can create a new peer group or join an existing one. If a new peer group is to be created, the user is required to input the name for the new group and a password to protect the access to it. The `jxtaManager` takes care of the group creation and of publishing the group advertisement so that other peers get aware of its existence. A receive thread is

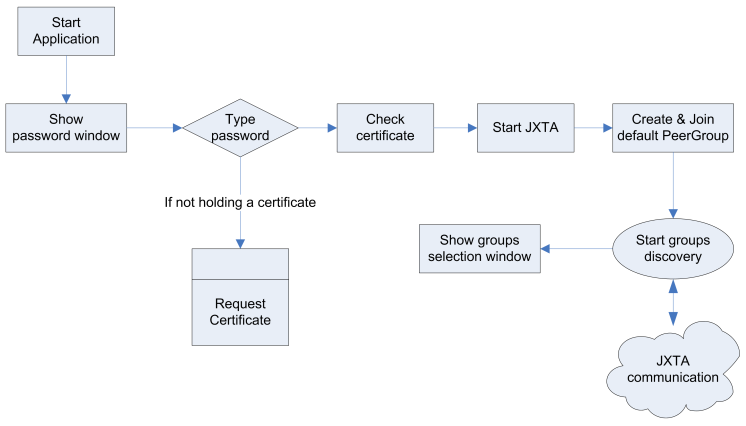


Figure 3.4: Protocol for peer startup

started for the peer group in order that the peer can receive packets from other peers. The user is then presented with the application's main window and able to run the protocols available, if the requirements for that are met.

When the user prefers to join an existent peer group, he selects one from the list of existent groups that the application presents to him. He needs to input the password that gives him access to the group. Again, `jxtaManager` takes care of authenticating the user in the group and, if valid, joining the peer to the group. Before the user is presented with the application's main window, a receive thread is initiated for the joined group.

3.7.2 Peer Authentication

Once inside a peer group, the `jxtaManager` continuously searches for other peers within the same group. For each peer discovery, two packets are sent, one to authenticate itself and other to request the authentication of the other peer. These packets are sent in a non-encrypted form, once the peers still do not know each other's public-key. They include the certificate chain and peer identification, such as peer ID and name. A peer receiving this packet first checks the validation of the received peer certificate and, if trusted, adds it to its own certificate chain and the discovered peer to the known peers list. Then, it replies with an identical packet, allowing that the other peer validates itself as well. Figure 3.5 illustrates this protocol.

3.7.3 The SSS Backup

This protocol represents how the application proceeds when a user chooses to backup a file using the SSS algorithm. It is illustrated by Figure 3.6.

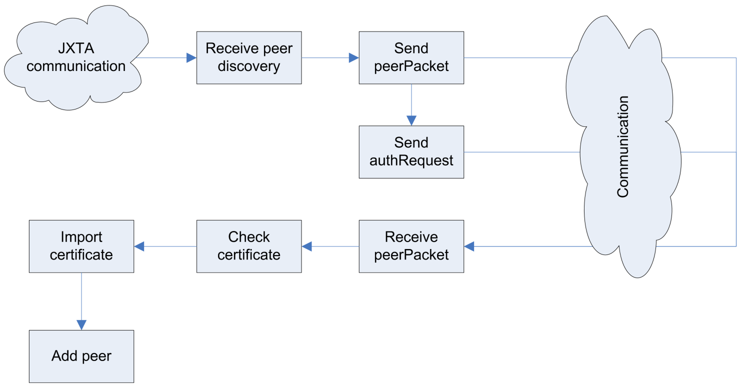


Figure 3.5: Protocol for peer authentication

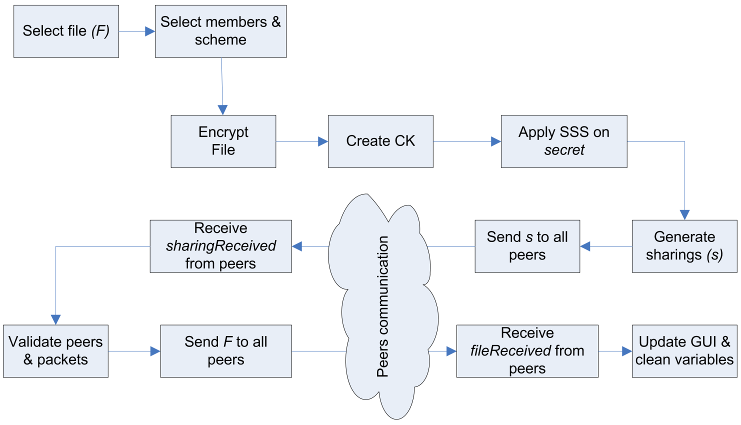


Figure 3.6: The SSS backup operation

The user provides the file to backup and selects the scheme settings, that is, he selects the peers that will store the shares, n, and how many of them will be required to reconstruct the secret, m. Then the application takes over and starts by generating the secret value (2048-bit) and the AES symmetric key (128-bit) from the secret. The key is then used to encrypt the file and a hash value is computed of the encrypted form of the file for later integrity check. Then the process is delivered to the `shareManager` so that the shares are created. The manager computes the SSS algorithm over the secret value and returns the created shares for each sharing peer, which are handed to the `sendManager` and sent to all selected peers.

When receiving a share, each peer validates it, by verifying the authenticity of the packet and the sender, and starts a `fileReceiver` thread in order to be able to receive the encrypted backup file. It replies with a confirmation of the reception of the sharing. The peer distributing the backup will then send the file to each peer that replied a correct confirmation.

All file recipients will verify the integrity status of the file, to ensure it was not altered during the transmission, store the file and reply with another confirmation status message.

3.7.4 The IDA Backup

The IDA backup protocol differs from the SSS protocol because it runs the algorithm over the file itself and not the secret, and it sends file pieces to all sharing peers as soon as they get ready. The protocol is illustrated by Figure 3.7.

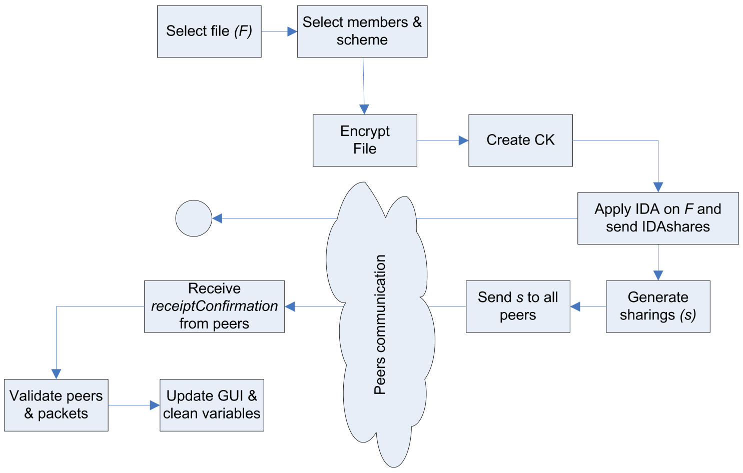


Figure 3.7: The IDA backup operation

The initialization of the protocol is equal, with the exception that the user now selects the IDA algorithm to compute the backup. Thus, the user selects the file, the sharing peers and the number of required peers to restore the backup. Then, the application generates the AES symmetric key, which is used to encrypt the file and hands the file to the `shareManager`. This manager computes the IDA algorithm over blocks of the file and when finish passes them to the `senderManager` so that they can be sent to all sharing peers, along with a sharing structure `idaShare`. The reason for that is to avoid that higher than an amount of the file is loaded in the peer memory, possibly causing out-of-memory issues.

All sharing peers will temporarily store all the IDA share pieces until they receive a packet with the `sharingIDA`. This sharing provides information about all the pieces that each peer should hold, allowing in this way each peer to verify that they have all the correct pieces. The sharing also contains a `receipt` which is signed by the recipient peer and sent to the sender peer in a way of proving that it received correctly the backup. Then, the sender peer collects all receipts, validates and stores them.

This protocol also adds two possibilities that are not allowed in the SSS backup protocol. It allows to coexist more than one owner of the backup, or in other words, it allows the owner of the backup to specify master peers with the same privileges over the backup as himself. This feature is convenient for environments where exist shared backups, such as a document that has more than one author, allowing all of them to restore the backup and perform other operations over it. However, this propriety brings some natural complications to the application, such as handling conflicts. The protocol also allows that the owner of the backup excludes himself from the sharing peers. This is useful when the peer that performs the backup has not enough hard disk space to store the data share, or in the case where the owner prefers not to have any copy stored in his local peer or network, making the backup in this way completely off-site.

3.7.5 The SSS/IDA Restore

Although the algorithms performed are different, the protocols for restoring a SSS or an IDA backup are equal. Figure 3.8 illustrates both cases.

The user starts by selecting the backup file he wants to restore. The file must be selected from the `My Shared Files` list in the application main window. The application then verifies if the user has enough credentials and if there exist enough peers available to perform the restore operation. If these conditions are met, then the peer sends a restore request packet to all available sharing peers and waits for the share replies.

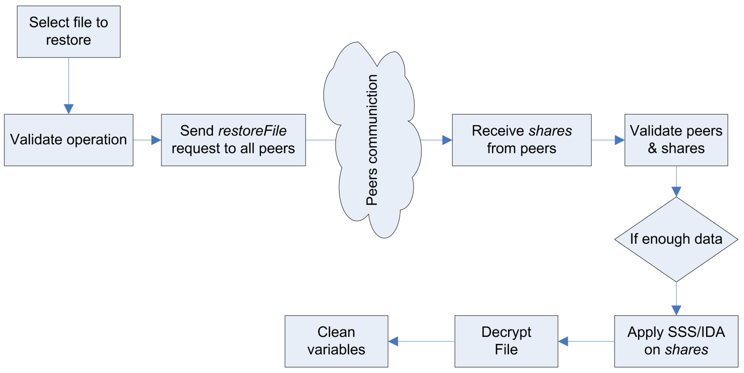


Figure 3.8: The SSS/IDA restore operation

A sharing peer, when receiving a restore request, starts by validating it. The requesting peer must be part of the known peers, of the backup sharing peers or owner of the backup and be allowed to perform the operation, which means that it is the backup owner for the SSS case or a master peer for the IDA case. If the requesting peer is considered authentic and authorized to restore the backup, then the receiver peer reply with the data share it holds.

The requesting peer collects all replies and when has at least m starts the restore process. While restoring, all shares are checked for errors by matching them against previously computed hash values, available in the sharing information. If an error is found, that share is discarded and another is used instead. Finally, the inverse of the algorithm is applied over the data and the encrypted file restored. The AES key is used to decrypt it and the original file is placed inside the `restoredFiles/` folder.

3.7.6 The SSS/IDA Delete

The delete protocol is also very similar for both alternative backup schemes. Figure 3.9 illustrates this protocol for both cases.

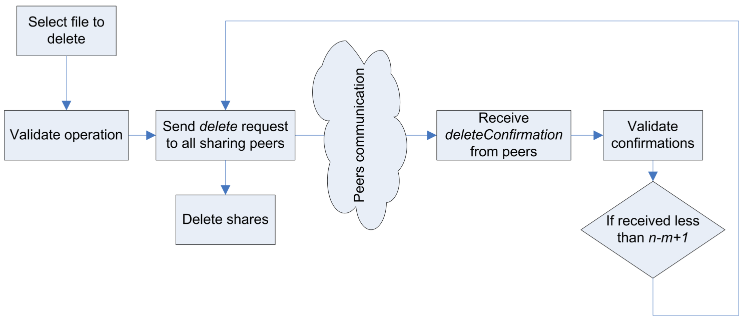

Select file to delete
Validate operation
Send *delete* request to all sharing peers
Delete shares
Peers communication
Receive *deleteConfirmation* from peers
Validate confirmations
If received less than *n-m+1*

Figure 3.9: Protocol for delete shares operation

A backup can be considered as deleted when at least $n-m+1$ shares of it are deleted. For both schemes, any subset composed of $m-1$ shares do not suffice for reconstructing the original backup file. Even though, when performing a delete operation, the delete requests are sent to all available sharing peers and not only to $n-m+1$. It is obvious that if all shares are deleted, the system will not be storing parts of non-existing backups, which would only represent a waste of storage space.

The validity of the operation is checked by the requesting peer application, ensuring that the user can in fact perform it, and by each sharing peer that receives the request. All peers, when receiving a valid request, delete their shares and data associated with the backup.

For the IDA scheme case, all sharing peers reply to the requesting peer with a deletion confirmation status. These confirmations are used to ensure that the minimum number of peers $(n-m+1)$ have in fact succeeded to delete their shares. The SSS scheme does not require the verification. It assumes that the probability of not deleting enough shares is low and therefore not critic†.

3.7.7 The SSS Update of Shares

This protocol is only available to use with backups processed with the SSS scheme. It has the purpose of distributing a new set of shares of the same secret to all sharing peers. For that, this protocol is based on the Proactive Secret Sharing (PSS) scheme proposed by Herzberg, Jarecki, Krawczyk and Yung [16].

A PSS scheme allows one to generate a new set of shares for the same secret using the old shares without having to reconstruct the secret. This is a very useful scheme for refreshing shares in a secure way, once the secret is never reconstructed and therefore, never at risk of being exposed to an attacker. One attacker that has been collecting shares related to one backup, will see his effort ruined if the shares are refreshed before he is able to collect m shares‡.

A peer requesting an update of shares will first send the request to all peers, which will validate the request and ensure that the shares were not updated for some time, in order to avoid too frequent updates. If all peers accept the request, they then send a confirmation message to all other sharing peers. In this way, all sharing peers are aware that all other peers will run the update of shares protocol†.

Each peer computes its update shares and sends them to every other sharing peers. When a sharing peer has all update shares, or a timeout is triggered, it sends the computed collection to all sharing peers. Then, each one will process the update of shares, by finding the correct values using a majority rule, since it is assumed that at least half of the peers are working properly and are not compromised. Peers that sent incorrect values will be accused of *cheating* and will have to request a new correct share. When all update shares have been verified and all accusations sent, all peers send a validation packet to cheating peers. This packet will serve as ticket to authorize the overwritten of the old shares by the new shares†.

3.7.8 The SSS/IDA Integrity Check

Being able to check the integrity of a backup is a very important feature for a backup application. It allows users to be aware of the status of their backups, permit to fix errors and prevent an unwanted scenario to happen, such as loosing the backup.

Both schemes use a similar protocol to check the integrity of the distributed peer shares. The protocol is illustrated in Figure 3.10. A user selects the backup file to be checked at the interface level and then the application takes over by handing in the process to the `integrityProtocol` class. As in all other operations, the application verifies that the user is allowed to perform such operation before proceeding.

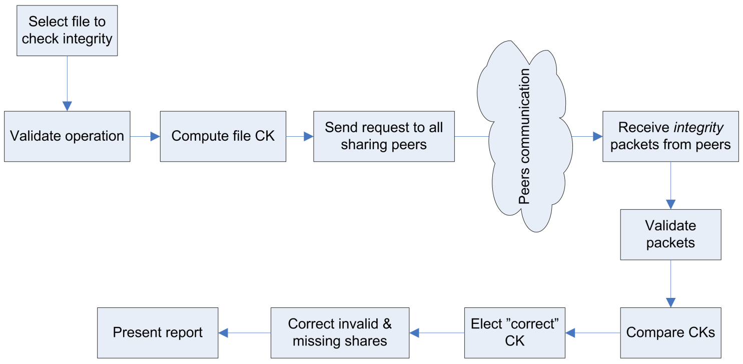

Select file to check integrity
Validate operation
Compute file CK
Send request to all sharing peers
Peers communication
Receive *integrity* packets from peers
Validate packets
Compare CKs
Elect "correct" CK
Correct invalid & missing shares
Present report

Figure 3.10: Protocol for integrity check operation

The idea behind this protocol is to collect all shares' hash values from all sharing peers and compare them, in order to discover incorrect shares. In the case of the IDA scheme, the hash values are in respect to all share blocks, since they are dispersed in blocks. When receiving an integrity request, a sharing peer computes the hash value over its share, or share blocks, and sends it back to the sender.

The integrity check operation occurs then when all sharing peers have replied, or when a timeout triggers it. The requesting peer matches all hash values that correspond to the same share, or share block, and analyses the result. It is assumed that the value in majority is correct, and therefore, used to sort correct and incorrect shares. All peers holding incorrect shares will receive new shares to overwrite the faulty ones.

3.8 Security of Backups

Resilia aims to perform backups in a simple and clever way, distributing them to other peers that can be located any where in the world. Is therefore of great importance to protect the backups when stored on peers that by any reason can be easily compromised. Thus, Resilia takes into consideration the following proprieties in order to consider a backup secure:

- Data must be kept *confidential*. To achieve confidentiality, Resilia makes use of powerful cryptographic algorithms, that are believed to protect the data against unauthorized access. Backup data should only be decrypted when restored, in an authorized manner.

- Damaged or tampered data must be easily detected, by proper *integrity* verification methods. A backup is useless if it cannot be recovered. Resilia replicates the backup data in a fault tolerant manner, which associated with sharing schemes, provides a secure way to recover a backup even in the case of some damaged shares.

- A backup is also useless if it is not *available* to be restored when needed. A secure backup should always be available. Resilia distributes the backup among a user-defined number of peers, which are assumed to provide a reasonable good availability to the shares they hold.
- It must be possible to *verify* the distributed shares for their integrity and have the ability to reconstruct correctly the backup. Resilia provides a protocol to check the integrity of all distributed shares.

CHAPTER 4

Design

This chapter discusses the design of the protocols and alterations to be performed on the existing system.

4.1 Adjusting the Application Structure

Incorporating another algorithm in the existing system requires implementing special objects to hold data and new functions to handle specific tasks. Although some of these objects and functions are specific of the algorithm to be implemented, others are common to both existing algorithms. Therefore, a small re-organization of the structure is appropriated. It brings advantages related to the performance of the application, making it faster, and to the size, making it smaller, composed by less classes.

The class that handles the creation of shares is the `shareManager`. To better combine the three algorithms, in order to achieve a cleaner organization, an interface class should stand in between the shares manager and the different sharing object classes. In this way, all the common proceedings can be implemented in the manager and the application control class as one common function. In other words, the `share` interface binds all sharing types so that they are transparent to the processes handling them.

When all shares are seen equal in the application core, it is also possible to eliminate the differences, seen from the outside the application, between backups that used different algorithms. This regards the name of the backup data share that is used when it is stored on each peer and the use of different files to store the shares under `mySharings/` folder. Hence, files are stored using the file name and the file extension added with a new common extension "`.enc`". For example, `book.pdf.enc` represents the data share of the backup of the file `book.pdf` without yielding the algorithm that was used to process the backup. All shares are stored in one single file `sharings.ssb`. In this way, a peer does not yield what type of backups it is storing just by looking at the name of the files.

Figure 4.1 presents the classes that form the application core, the share interface and some of the objects used.

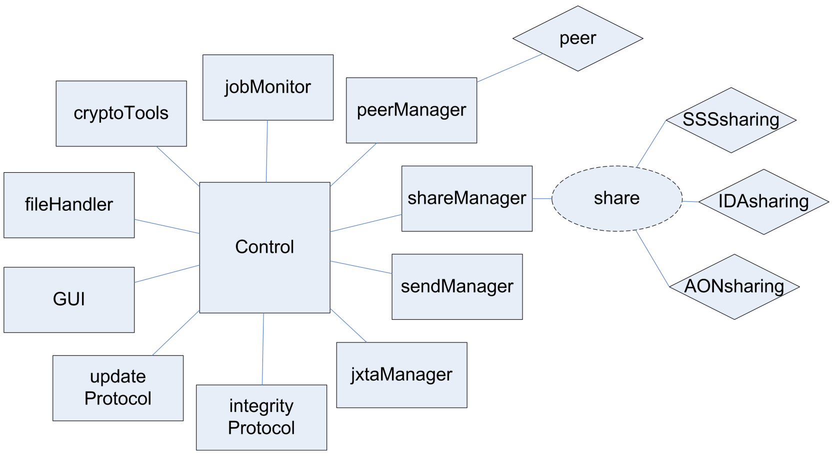


Figure 4.1: Organization of the classes

To maintain a consistency regarding the naming of classes, the sharing classes were renamed to `SSSsharing`, `IDAsharing` and `AONsharing`.

The methods implemented in the `shareManager` class handle backup, restore and delete operations, along with some auxiliar functions. These operations are performed using just one sharing algorithm. To allow a combination of algorithms, as discussed in Section 3.1, it is logic to move the sharing algorithm to a different class so that the protocol can include any number of algorithms. In other words, the `shareManager` methods will not be repeated to accommodate all possible algorithms combinations, but a common method will be able to request the use all possible different sharing algorithms.

To better organize this, it is introduced a new package, named `factory`. It comprises classes that implement the sharing algorithms and any other auxiliar component. Classes in this package make use of the sharing structures and are handled by the `shareManager`.

Furthermore to these changes, a new package `util` is added to the application with the purpose of containing utility classes. For example, a class comprising all global variables, `Constants`, so that it is easy to modify global parameters of the application or the algorithms. In this new package should also be present classes related to the configuration of the JXTA platform, the application environment, the logging structure and timers.

4.2 The AON protocol

Throughout Section 2.3 are presented different approaches on how to apply the All-or-Nothing algorithm to a message. The main goal of this project is to design and implement Rivest's [24] package transform as an alternative way of creating and distributing backups. However, due to the tight similarity between Rivest's and Desai's [12] schemes, both are considered in the design of the protocol. Canda and Trung [5] approach is not considered, although the authors refer that it performs better and provides an adjustable security factor, independent of the message length. However, it is due to the difference of the algorithm that it is not taken into consideration. The algorithm is applied after the file is encrypted, contrarily to Rivest's algorithm, and requires that the security factor w is defined for each run. This would require an extra input from the user, which may not choose a good value and therefore fail to provide the expected degree of security.

There is a small difference between Rivest and Desai approaches. Desai improves the performance of the algorithm by avoiding to compute the hash code value for each block of the message. By doing that, the additional block y_{s+1} is then obtained without requiring that all s blocks are computed in an ordered manner, which adds a security feature to Rivest's scheme. In practical cases, this feature may not be of great importance, while the performance gains of about 30-40% are more important.

In terms of implementation, Rivest's scheme can be implemented completely as it is, with an added optional bypass for the hashing operation, allowing in this way to run Desai's approach, if wanted.

4.2.1 Basic Algorithm Structure

Besides the specific steps of the AON algorithm, any AON-related operation follows identical proceedings as the other algorithms implemented in the application. Thus, the structuring of the AON protocol is based on those.

An AON operation, such as the backup operation as shown in Figure 4.2, is composed of a few steps. The backup data to be dispersed is first read into the system and converted into an appropriated data type, such as a byte array. Then, shown by just a box, the AON algorithm is processed over the data. This step includes the creating of the key, computing sequentially the exclusive disjunction of 128-bit blocks of data with the encrypted block counter, creating the hash of each block, if using Rivest's scheme, and finally creating the last data block. The resulting data is written back to the hard disk and the hash code of the file is calculated so that it can be verified for integrity later on.

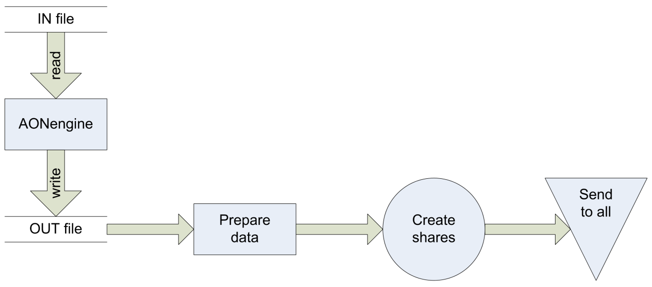


Figure 4.2: Overview of the AON backup operation

After this step all the information required for future operations is gathered and all shares for all sharing peers are created. It includes encrypting the AON resulting file with a different key as the one used in the AON algorithm and creating a new hash code of the encrypted form of the file, so that all sharing peers can easily confirm the validity of the file they receive and during future integrity check operations. The final steps, such as the process of creating and sending shares and data files to all peers, are equal to the way that it is done for the other implemented algorithms.

Using two different keys, one for the encryption within the AON algorithm and another for the encryption of the file resulting from the AON algorithm, is not critical, but advised. When using the same key for both processes, an attacker can try all different keys over the encrypted file until the key that he obtains from the AON algorithm is equal to the one used to decrypt the file. In this way, the attacker obtains a confirmation of correctness and does not have to complete the AON inverse algorithm to reconstruct the data and verify if he used a correct key or not. Therefore, using the same key can be seen as a reduction of the tasks that an attacker has to perform when breaking a file. Even though, the advantage of using two different keys is not that significative.

4.2.2 The AON Engine

Like the IDA, the AON also works over the entire file data. This means that the file cannot be read entirely to memory, in order to avoid out-of-memory issues when large files are to be processed.

Figure 4.3 presents the structure of the AON engine, which is composed of three components. The `fileReader` reads sequential parts of the input file and feeds the `AON` component that will perform the AON algorithm over that block of information. When done, this block is passed to the `collector` component that will regroup all blocks in one output file and store it. All three components should run on independent threads and be synchronized so that the AON component has always a block to work on.

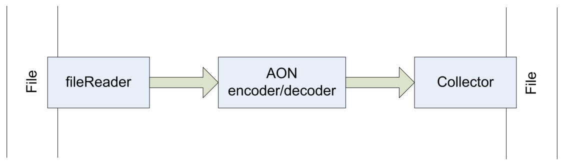


Figure 4.3: The AON engine structure

Using this structure, only part of the file is loaded to memory at each time, without affecting the performance. Furthermore, it is possible to run AON components in parallel, which can possibly speed up the process, specially if it is running on a multithreaded or multiprocessor machine.

The AON engine only comprises the functions for backup and restore data. The remaining operations do not make use of the AON algorithm. Throughout the following sections are designed all the operations available using the AON protocol. Contrarily to the way that all operations protocols were introduced in the previous reports [21, 19], in a very detailed UML diagram, this report makes use of an informal data flow diagram to illustrate the main steps of each of the protocols.

4.2.3 The Backup Operation

Figure 4.4 presents the main steps of the AON backup operation, from the point of view of the peer that is performing the operation.

The backup operation is initiated by the user through the graphical interface, by selecting the file to backup and the scheme, AON in this case. With this information, the algorithm takes the file in and applies the AON transform to it.

The protocol follows with preparing the data, by encrypting it and creating security fields, and sending the shares to all members. Each sharing peer will validate the backup request and reply if ready to receive the backup, or ignore the request otherwise. The request is considered valid if the sender is a valid and known peer and if the share is correctly formed, for example, a share containing an already existing sharing ID is not valid.

There is a difference between which type of destination peer is the one to receive the share. In a similar way as it is done in the IDA protocol, if the recipient is a master peer, that is, a peer that can perform any operation over the backup, such as restore and delete it, then the share will be sent complete. Otherwise, the share will be incomplete, not providing non-authorized peers with the key-information to the share data.

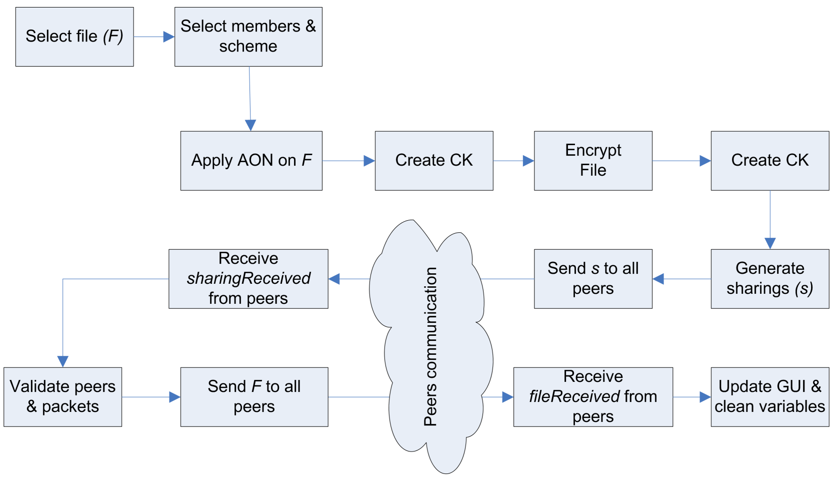


Figure 4.4: The AON backup operation

The encrypted file is then sent to all peers that reply to the backup request. The sharing peers verify the integrity of the received file and, if correct, store it and reply the status of the file reception, if incorrect, request a new file transfer. Finally, all sharing peers update the GUI, so that it lists the new backup file. Since the requesting peer waits for all sharing peers replies, a timer must be associated with the backup process in order to finalize the operation in case of some of the peers do not reply.

The AON sharing structure includes the sharing data, which holds all data common to all algorithms, and some specific AON information. An example of the latter type of data is the checksum of the file after applying the AON to the original file and the AON keys.

4.2.4 The Restore Operation

The restore operation can be seen as an inverse simplified version of the backup operation. Figure 4.5 presents the data flow diagram illustrating it.

This operation starts by a user selecting a file from the list of shared files through the interface. The application verifies if the operation is allowed, that is, if the user has enough credentials to restore the file and if there exist enough peers available to reconstruct the original file.

Besides the owner of the backup, all master peers of the backup are allowed to restore it. Users of master peers will be able to find the backup file name in both share files and received files lists of the application main window. Users of normal peers that belong to the sharing peers of the backup will only find the backup file name in the received files list.

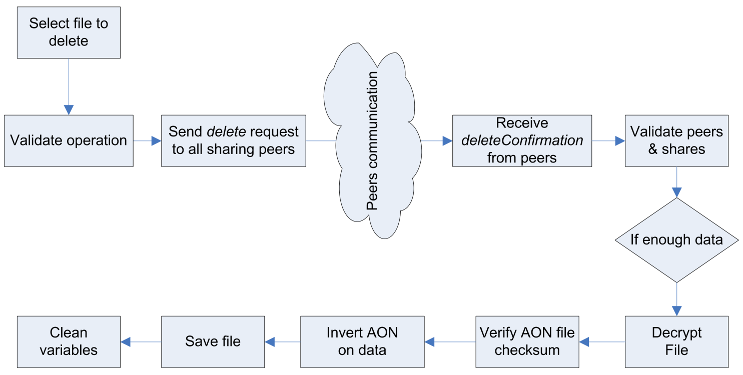


Figure 4.5: The AON restore operation

Although the protocol comprises the design for a replicated AON sharing, the standard AON backup operation sends the entire backup file to all sharing peers. In this way, to restore the file, the requesting peer first checks if it holds the file itself. If so, it is not necessary to send any request to other sharing peers. It is also possible that the operation requester does not hold the data, or the data it holds is invalid. In those cases, it needs to request the file from other peers.

If it needs to request shares and/or data shares to other peers, then the replies will be collected and validated. The validation includes peer, share, and file validation. As soon as sufficient shares are available, the backup inverse process can begin. It starts by decrypting the file with the symmetric key and reconstruct the original file, by running the inverse of the AON transform algorithm. By last, the file is stored in disk, under the `restoredFiles/` folder, which is the default location for storing restored files.

4.2.5 The Delete Operation

In order to delete a backup, the information dispersed through all sharing peers needs to be erased. In special, the share owner and all share masters, since they are the only ones able to reconstruct the backup. Figure 3.9 illustrates this operation.

When requesting the delete of a backup, the application first validates the request by checking the existence of such backup and if the requester is allowed to perform the deletion, that is, if he is the owner or a master of the backup. If valid, then delete requests are sent to all sharing peers available and all the share data that the local peer holds is erased.

When receiving a delete request, a sharing peer first authenticates the request and the peer, and then, if valid, deletes the share and the data. There is no need for sending back a confirmation message because there is no need to know who has in fact deleted the share or not. Any normal peer that failed to delete the share will keep on storing the data, but no sensitive information about it. In a system total storage space point of view, it is preferable that the peer frees the space used by a non-existing backup, but not critical. The share data that the peer holds is considered secure and unbreachable by an attacker. If any other peer type also fails to delete the share, it will be able to perform an integrity operation, which should result in a confirmation that the file was schedule for deletion, and therefore, should be deleted by the peer requesting the operation. This peer will also forward a new request to delete the share to all sharing peers, giving in this way a new chance to peers that missed the first request to erase the data. Peers that have already deleted the share will ignore this request.

Another way to handle this conflict of having master peers with information about a backup that was deleted is by making the peer that requested the delete operation to repeat the request during some limited regular intervals. For example, repeat the request each 10 minutes after the first request during one hour. This represents repeating the same request 6 times within one hour, which for many systems can be considered as enough to reach all peers. Other schedules can be easily implemented in order to easily adapt to different systems, for example, where peers are present at some point during one day, but can be disconnected for periods over than one hour.

If it is used a replication scheme when performing the backup operation, then the number of required peers to perform the deletion is reduced to $n - m + 1$, that is, one more share than the required amount for reconstruction. Even though this fact, the delete requests are still sent to all sharing peers, for the sake of storage space. It should be noticed that split encrypted data shares, that should have been erased, do not represent a threat if the peer is compromised. An attacker holding part of a data share cannot access the information in it due to different factors, being the main one the AON propriety. An attacker not holding the complete data share cannot access the information in it.

4.2.6 The Integrity Check Operation

It is critical for a backup system to ensure, or allow one to verify, the correctness of the backup shares. It can be an automated operation that runs at regular intervals, or an operation initiated by a user. Either way, this operation should find out if a specific backup is integrity valid and fairly available. Depending on the results of the test, it should take proper measures to restore or maintain the integrity and availability of the backup.

The integrity check operation is illustrated in Figure 3.10. It can be performed by any master peer, which sends a request to all available sharing peers. All peers compute the checksum of the file they hold, which should be equal everywhere, and return the value to the requesting peer. This peer will then construct a list with all the hash values and elect a valid one. The reason to elect one instead of using its own value is due to the possibility that the share that the requesting peer holds may have been compromised, in an accidental or malicious manner, and therefore not presenting a correct integrity status.

The election of the valid checksum value follows a simple majority rule. The value that is most common in the list is assumed to be the valid one. The motivation for this rule is the belief that there may exist wrong shares, but a majority should be expected to prevail unaltered.

With one elected valid value, it is possible to sort all other values as correct and incorrect. Based on the list of incorrect shares, a new valid share is sent to all peers holding an incorrect share. The share to be sent is the one that the requesting peer holds, if correct, or from a random peer present in the correct shares list.

Peers not holding a backup share with the same ID as in the integrity request assume that they are part of the sharing peers for that backup file but for some reason did not receive the share during the backup operation. Therefore, they take this opportunity to reply with an obvious wrong hash value so that they will receive a correct share.

Finally, the user is presented with a report that shows how many peers were tested and how many correct, incorrect and missing shares were found. From this report, the user can have a feeling of the status of the backup file in the system. If, for example, only two of the twenty peers belonging to the sharing were tested, then he may assume that many peers are unavailable and the backup file is at risk, and hence he can decide that it is better to reconstruct the backup and store it using other peers.

This protocol assumes that the backup file is fully replicated on all sharing peers. When using a replication model along with the backup operation, a small difference must be taken into consideration. Instead of handling just one hash value for the file, it must handle a hash value for each replicated block of the share data. The process is the same, where the hash values of all blocks are gathered and one value is elected as valid for each block. Using that one, all the invalid or missing blocks are overwritten by correct blocks.

4.3 The Replication protocol

Resilia is not a typical application to apply a replication model off-the-shelf. But it distributes files over a network of peers and is therefore a need to keep all storing peers in the same consistency status. Moreover, it is also important to use a model that allows the repartition of the file itself, in a way that it would reduce significantly the communications overhead. But that would not compromise the availability of the backup.

Based on the theory introduced in Section 2.4, it is possible to define Resilia as a multi-master pessimistic application. Normally, a backup file is stored for some period of time, therefore there are no updates to the backup. A new backup of the same data can take place but it will not overwrite the previous version. Thus, if there are conflicts with the inconsistency of data between peers, it must depart from transmission errors at the distribution stage, or unintended, or malicious, alteration of the data after storage. It is then clear that an optimistic replication model, that would increase the complexity of the application, is unnecessary for this case.

Regarding the other replication properties, the operations are restricted to read and write (or overwrite), which makes it a state transfer definition type. The scheduling of operations is syntactic, since all operations are sorted based on time of creation and queued in a FIFO fashion. Contrarily to pessimistic systems, Resilia does not handle conflicts by blocking or aborting operations. When a data conflict is found, the system should take measures to overwrite a bad share by a good one. The verification of good and bad shares should be provided by hash codes of the data and comparison with other peers' shares.

The two following sections present the design of how Resilia handles the replication of operations and data.

4.3.1 Replication of Operations

Considering the models introduced in Section 2.4.2 and the Jiménez-Peris et al. [17] conclusion on the analysis of those models, the obvious choice for this system is the ROWAA model. The transfer of operations between peers is of so low cost that what is gained by using another model, such as the MQ, does not worth the consequent loss in overall consistency or availability.

Thus, all operations available are performed based on the ROWAA model. That is, read operations, such as obtaining the backup details, are done on one peer, if possible locally, and write operations, such as backup and integrity check, are done in all available peers. In this way it is enforced a primary copy replication.

4.3.2 Replication of Data

Backups can be of a considerable size. That means that if the data is fully replicated to all peers, as it is done when using the SSS or the AON protocol, then the network and peers will be overloaded. Therefore, it is needed a replication model that balances the amount of data distributed to each peer and the backup availability.

The model should not let all peers receive an entire copy of the file, even though that would provide a high level of availability for that backup, nor let the level of redundancy of the file be too low, even though that would reduce the communications cost and storage overhead.

Once the IDA protocol divides the backup data into small shares, this scheme is only designed to be used with the SSS and AON protocols. It is based on the MQ model, in the way that it requires that any majority of the peers are available so that the data reconstruction is possible. For an even number of peers, any subset with half the peers form a majority, that is $m = \lceil \frac{n}{2} \rceil$. The scheme is described in the following section.

4.3.2.1 Model Design

The backup data resulting from the sharing scheme processed is divided into n equally sized blocks, where n is the number of sharing peers. Then $n-(m-1)$ blocks are distributed to each peer. This distribution can be sequential, that is, for example, peer 1 will receive blocks 1, 2 and 3, peer 2 will receive blocks 2, 3, and 4, and so on, although any other sort method can be used with the condition that no blocks combination can be repeated. That is, no other peer can get the blocks 1, 2 and 3 than peer 1. In this example, using the sequential order method, the last peer to receive the data will get blocks n, 1 and 2.

Naturally in this scheme, the redundancy will increase accordingly to the number of sharing peers. To be exact, the backup data is replicated in the system $o = n-(m-1)$ times. This would represent a reduction of approximately 40% in communications and storage costs for cases with a few sharing peers and approximately 50% for cases with several sharing peers, when compared with full replication cases.

Taking an example network for the latter case, with 100 sharing peers there would be 51 copies of the backup data in the system. For this and larger networks, the ratio between the number of sharing peers and the number of redundant copies may be excessive, specially in cases where the sharing peers have a low rate of failure or unavailability. Thus, it is also possible to allow the backup owner to specify how many copies of the file should exist in the network by specifying the number of sharing peers needed to restore the backup data. Hence, in the 100 sharing peers case, the backup owner could decide that 90 peers would be required to restore the data. That represents 11 copies of the file distributed in the system, which rises the overhead reduction to almost 90%.

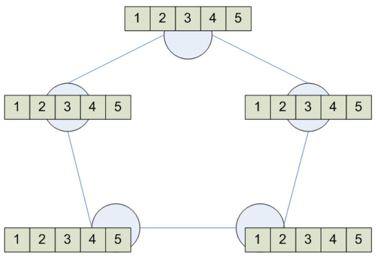


Figure 4.6: Fully replicated backup: $n = 5, m = 1, o = 5$

Figure 4.6 presents a backup example using 5 peers without using the replication model. The backup is fully replicated on all peers, which results in a 5 times blow up of the communication and storage overhead, but just one peer is needed to restore the backup. To note that in this and following figures are not represented all peers connections for the sake of simplicity. The lines connecting two peers only have the symbolic meaning that peers are joined in a group or are sharing peers of a backup.

When using the replication model introduced, as Figure 4.7 illustrates, the backup file is equally dispersed by the sharing peers, reducing the communications and storage overhead to just 3 times the size of the file. However, in this example, are required 3 peers to be able to restore the backup.

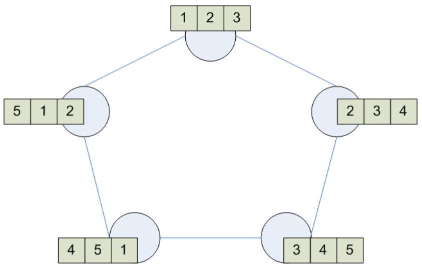


Figure 4.7: Partially replicated backup: $n = 5, m = 3, o = 3$

For the cases of even nodes, it can be argued if the majority should be half $m = \frac{n}{2}$ or half plus one $m = \frac{n}{2} + 1$ nodes. The former choice presents a smaller number of required peers to be available to restore the backup, but a higher value of overhead. This trade off is in fact symmetric, that is,

$$\text{if } m = \frac{n}{2} \qquad \text{then } o = \frac{n}{2} + 1$$

and

$$\text{if } m' = \frac{n}{2} + 1 \qquad \text{then } o' = \frac{n}{2}$$

where the prime notation refers to the latter case.

A network where peer nodes present a high degree of availability should use the former case, and the latter case otherwise.

4.3.2.2 Handling Conflicts

When conflicts arise, such as different data consistency between peers or impossibility to reach one peer, the model should be able to handle it in a way that it maintains the probability of restoring the backup.

In a integrity check operation, the share (or shares) that each sharing peer holds related to one backup are matched to confirm that all are in the same valid integrity status. All sharings hold information about the correct hash code of the data share. If a share is found not in this condition, then a valid share, possibly from the peer that requested the integrity operation, if it has a valid share, will be sent to that peer and overwrite the faulty one.

If a peer is unreachable, then the system is momentarily weaker. The probability of recovering a backup is reduced, which is very undesirable. The system should have a resilience behavior by maintaining this probability, which can be done in two different ways.

The missing peer can be replaced by another peer that did not belong to the sharing peers of the backup. This process requires an extra peer to be available. If one exists, then the subset of shares that the missing peer used to hold are sent to the new peer. This will probably involve some sharing peers, depending on how many shares the peer was storing and how many peers are required to provide those shares. Figure 4.8 illustrates this case.

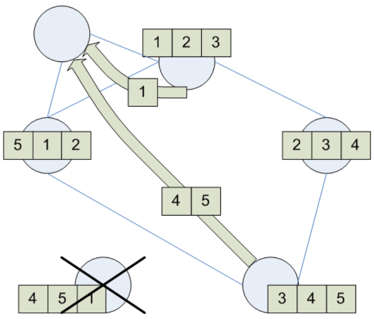


Figure 4.8: Missing 1 peer: using a non-sharing peer to store the "missing" data shares: $n = 5, m = 3, o = 3$

The previous solution restores the original probability of recovering the backup. However, a peer not belonging to the sharing peers of the backup may not be available. In these cases, the current sharing peers need to cooperatively withstand the problem by storing the missing peer shares. Depending on how many sharing peers exist, how the distribution of shares is done and how many shares the missing peer was storing, the redistribution of shares may not reach to all sharing peers. Only sharing peers not holding already a copy of a share will be able to receive it. A peer holding duplicated copies of a share provides no extra redundancy to the system, since if the peer fails for some reason, both shares will be lost. Figure 4.9 illustrates this case.

With this strategy, the probability of recovering the backup will increase, and therefore, the system will be strengthen. This is easily seen by the fact that by reducing the number of sharing peers and increasing the number of shares on each peer results in a higher availability degree of each share. This has however a limitation. When all sharing peers hold all shares, the system cannot rise the probability any more. At that point is reached the highest availability provided by the system, as Figure 4.6 shows. After this point, any missing peer will

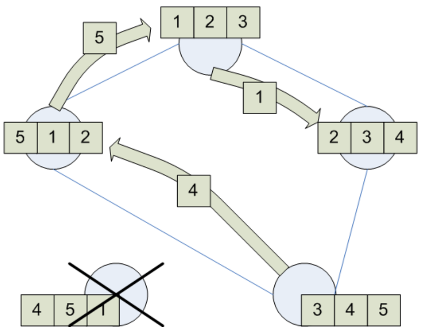


Figure 4.9: Missing 1 peer: using sharing peers to store the "missing" data shares: $n = 4, m = 2, o = 3$

weaken the availability of the backup, since the probability of the availability can be calculated by $(1 - P_f)^n$, where P_f is the probability of failure of one peer [8].

In the example case that has been used, this limit is reached when 2 peers are unavailable at the same time and no non-sharing peers are available. Figure 4.10 illustrates it. At this point, all three remaining peers hold the complete backup data, reducing in this way the number of required peers to only one. The overhead is naturally maintained until this stage.

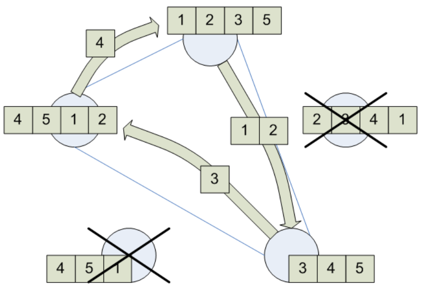


Figure 4.10: Missing 2 peers: using sharing peers to store the "missing" data shares: $n = 3, m = 1, o = 3$

Peers may be unavailable for several different reasons. For some of them, it is expected that they will become available within some period of time. When that happens, these peers may not be at the same degree of replication as the others. Although it is easy to get them up to date with the remaining ones, the cost may not be worth it. Avoiding the update has the consequence of increasing the number of required peers to restore the backup, that is, m. This number can never increase to a higher value than what was before specified,

m_0, by the user or automatically by the system. Moreover, depending on the size of the network and replications performed until that moment, m will most likely have a value lower than m_0, but higher than one. Hence, it is preferable not to update the missing peers. Figure 4.11 illustrates this case, following the example used before.

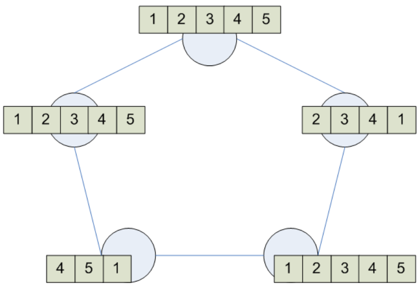


Figure 4.11: The two missing peers reestablish contact with sharing peers: $n = 5, m = 2, o = 4.4$

Another way of handling the reestablishment of missing peers would be by reversing the replication process that was done. The sharing peers could remove the shares that acquired during the replication process, moving the system towards its original settings. This behavior can be very useful in a large network, when several sharing peers have been offline for some period of time, causing all remaining ones to get very overloaded. When all peers, or most of them, reestablish connection to the sharing group, the system will have a unnecessary load. To be able to remove it, all sharing peers need to come aware of which peers are back and what shares do they hold. This process involves a few messages exchange, which represents no cost comparing to what the system may gain in the overall overhead status.

Thus, following the example, the sharing peers should organize themselves and get to initial status presented in Figure 4.7.

4.4 The IDA protocol

One problem resulting from the IDA extension to the application was the poor performance of the algorithm, taking very long time to compute the IDA data shares for all sharing peers. Thus, in an effort to improve this issue and make the IDA backup alternative also usable, a restructure of the protocol has also been designed.

The analysis of the protocol provided in Meira's report [19] pointed out the main problem to the matrix multiplication carried out during the IDA algorithm. Other problem that contributes to the performance issue is the data type used to hold the data, that is the Java BigInteger. This data type facilitates logical and mathematical operations over the data when comparing to other data types, but underperforms them. It has also the advantage of allowing to represent a high number of bytes by a single integer. This feature favors the multiplication of several bytes in a single operation. And that represents that less operations need to be performed when compared to a byte-per-byte computation.

4.4.1 Algorithm re-Design

The organization and structure of the algorithm cannot be changed, but how some operations are performed or the data type that is used can be altered in order to improve the algorithm performance.

In the core of the IDA algorithm there is a matrix multiplication. The current implementation of it uses a standard algorithm which, when multiplying two squared $n \times n$ matrices, it performs $O(n^3)$ operations. In particular, it performs eight recursive matrix multiplication calls and four matrix additions.

However, there are clever ways of multiplying matrices. Strassen's [28] proposes an algorithm that reduces the number of recursive matrix multiplications calls, one less in the same example as above, but at a cost of 18 matrix additions and subtractions. In this way, Strassen's algorithm performs $O(n^{\lg 7})$ operations. Winograd's [13] variant of Strassen's algorithm reduces to 15 the number of matrix additions and subtractions, by reusing common subexpressions, but keeps the number of multiplications. Any of these two algorithms can be implemented in the core of the IDA algorithm, slightly improving in this way the performance of the operation, since additions and subtractions outperform multiplication calls.

Another proposed improvement to the previous implementation is the change of data type that is used. Instead of the BigInteger, use byte arrays. In terms of data representation of a set of bytes, it can be used multidimensional arrays of bytes, but there will still exist the drawback of having to perform byte-per-byte computations. However, the primitive data type byte outperforms the BigInteger, which if implemented in a way that the data can be loaded at once to the computation, there will be a performance gain.

To perform matrix operations, the algorithm uses an auxiliary class `Matrix`, which holds matrix specific methods. This represents a performance problem in a way that for every multiplication of two matrices, they first need to be converted to a matrix structure. Even though this operation may not be that expensive, if the file to disperse is long, then there will be several matrices to use the `Matrix` class, and therefore it represents some computation that could be avoided. To improve it, the matrix multiplication is implemented directly in the algorithm, in a way that it does not require to make a call to the `Matrix` class.

4.4.2 Protocol re-Design

Regarding the IDA protocol, the changes pass by re-structuring it to a similar protocol structure as the one designed for the AON operations, and eliminating the send of shares to all sharing peers phase as soon as they are ready.

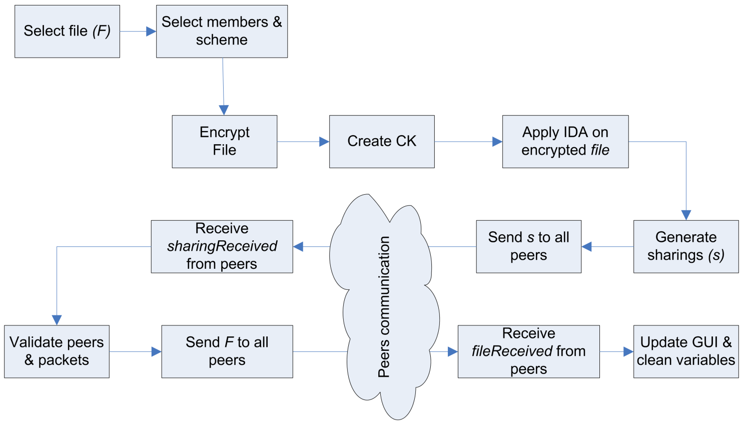


Figure 4.12: The new IDA backup protocol

The overall of the structure is maintained, as it is presented in Figure 4.12, in the way that there exists a component reading the input file, now named `fileReader`, and another to collect the data shares that are processed in the IDA encoder/decoder, named `collector`. These two components, together with the `idaEngine` class replace the previous `idaFileWorker` and `idaWorker`. In this way, the data shares are computed but not sent right away to their destinations. They are gathered, grouped and stored in the hard disk and only sent to all sharing peers at the end of the process.

This change will save time that is being spent to send several small data pieces to all sharing peers, by just sending one final file at the end of the process. And it will still prevent out-of-memory problems while processing long files, since only part of the file is loaded to memory at each moment.

With the introduction of the `share` interface, and in oder to make the IDA sharings agree with the common sharings interface, they need to be re-arranged. Thus, it is introduced a single sharing structure, the `IDAsharing`, which replaces the previous `receipt` and `sharingIDA`. Also, the `IDAshare` is not longer necessary since the share data is sent in one package at the end of the process. The `IDAsharing`, like the other sharing structures in the application, holds the sharing data that is common to all sharing peers, a list of the sharing peers and IDA-specific information, which depends in who is the recipient of the share. Master peers will have a share with a complete information of the backup, that is, including the vector key used in the IDA operation and the key that encrypted the original file. Normal peers will receive a share with no backup-private information.

CHAPTER 5

Implementation

This chapter describes the implementation of the extension to the application prototype. It does not refer to already implemented components in previous versions, for that please review Chapter 3. Here, the main focus is to describe how the new AON algorithm and its data structures are implemented. It is also described the changes made to the IDA protocol and, in a general way, to the structure of the application.

5.1 Application Overview

In the first implementation of the prototype, the application was divided into six packages. They organize classes according to their functions. As the prototype grows towards a more sophisticated and complex application, new classes are added to it that do not fit the previous organization. Thus, in order to accommodate the changes implied by the second extension to the application, two more packages are added: FACTORY and UTIL. Table 5.1 presents an overview of all eight application packages and some of the classes that are part of them.

The MAIN package comprises all classes containing methods to control the application and handle all possible tasks. That includes handling every type of request that can be performed, specifying all protocols, defining sharing structures, cryptographic methods, file input and output operations and organizing information about peers. Specific methods for handling the peers communication, such as sending and receiving packets and files, and operating the JXTA platform are implemented in independent classes, located inside different pack-

Table 5.1: Overview of Resilia's main packages

Main	GUI	Objects	Factory
Control	mainWindow	netData	AONengine
jobMonitor	selectGroupFrame	netPacket	IDAengine
shareManager	backupFileFrame	peerPacket	IDAencoder
peerManager	keyPassFrame	encryptedNetPacket	IDAdecoder
sendManager	okFrame	controlJob	fileReader
cryptoTools	guiStandards	sendJob	Collector
SSSsharing	newGroupFrame	share	
IDAsharing	theMenuBar	...	
AONsharing	...		
...			
Util	Send	Receive	JxtaSys
Constants	fileSender	fileReceiver	jxtaManager
IDAconfigurator	packetSender	packetReceiver	peerGroupTool
Timers	sendJobMonitor		
Matrix			
Log4J			
...			

ages. All data structures that are used for communicating or storing all types of data and information are implemented as objects and grouped inside the Objects package. In the same way, all classes related to the user interface are placed inside the specific GUI package.

The two new packages are added for two different reasons. The Factory package aims to gather the backup algorithms so that they can be used by the application in a modular fashion. In previous versions, the algorithms were implemented directly in the `shareManager`, preventing to combine two algorithms in the same backup operation. In this way, the application can run the algorithms independently of the operation protocol that is being performed. Together with the backup algorithms, named as *engines*, are auxiliar classes that the algorithms require, such as the file reader and the collector, as described in Section 4.2.1.

The Util package aims to gather all utility classes that do not belong to any other packages. In this package can be found classes for defining global constants that the application use, for an easy change of settings of the application or algorithms, and for setting up an environment for the IDA operation. The `IDAconfigurator` has the purpose of providing a working environment to the IDA algorithm due to its use of many files during a backup or restore operation. Are also part of this package classes to handle timers, matrix operations and logging tools.

Resilia is implemented in Java programming language and makes use of third-party libraries for handling cryptographic algorithms. The use of Java provides platform independence to the application, making it able to run on different machines and different operating systems without requiring specific implementations for that. This propriety is associated with the JXTA platform, which is also implemented using Java. JXTA advertisements, used for describing and publishing resources, are XML documents that facilitate the sharing of data across different information systems. As a drawback of using Java as programming language, when comparing to other languages such as C or C++, is the lower performance results, mainly when performing low level operations. On the other hand, Java provides an easier and more secure way of programming applications in terms of preventing accidental buffer overflows and such type of flaws.

5.2 New Data-Structures

For the current prototype extension are implemented some data structures. The most important ones are the `AONsharing`, the data-structure holding the information about an AON backup, and the introduction of the `share` interface, which is presented in Listing 5.1. The methods defined in this interface are the basic methods for all sharing structures, which are common to all backup schemes, such as the method to find the owner of the share or to get the sharing ID.

Listing 5.1: The `share` interface

```
public interface share {
  public String findShareOwner();
  public sharingPeer findSharingPeer(int nr);
  public String getOwnerID();
  public BigInteger getSharingID();
  public String getFileName();
  public void setLastUpdateTime(long time);
  public boolean isPartOfSharing(peer p);
  public String getCheckSum();
  public boolean getIsDistributingFile();
  public void setIsDistributingFile(boolean b);
  public void incSharingPeerOK();
  public int getSharingPeerOK();
  public int getK();
  public boolean isRecovering();
  public void isUpdating(boolean b);
  public boolean isUpdating();
  public sharingData getSharingData();
  public Vector<String> getSuperUsers();
  public void setIsRestoring(boolean b);
  public boolean getIsRestoring();
  public boolean isCheckingIntegrity();
  public void setIsCheckingIntegrity(boolean b);
}
```

The `AONsharing`, partially presented in Listing 5.2, implements the `share` and the `Serializable` interfaces, the former for the reasons above mentioned and the latter so that it can be correctly transmitted to other peers. This sharing defines all common variables added with some others that are specific of the AON operation, such as the `aonCK` —hash value of the file after being processed by the AON algorithm— and the key used to encrypt the file. The constructor takes the AON specific data and the common data, in a `sharingData` data structure, which is used commonly by all three schemes.

Listing 5.2: The `AONsharing` class

```java
public class AONsharing implements Serializable , share {

  private static final long serialVersionUID = 1L;
  private sharingData sData;
  private String  aonCK;
  private byte[] encKey;
  private boolean isDistributingFile , isRestoring , isUpdating;
  private boolean isCheckingIntegrity , isRecovering;
  protected int sharingPeersOK , serverNr;
  private long lastUpdateTime;
  protected long fileSize;

  //Status variables: Integrity check
  protected integrityNetData[] integrityPackets;
  protected boolean myFileIsCorrupted ,myShareIsCorrupted;

  public AONsharing(String aonCK, sharingData sData,
       byte[] encKey, int serverNr) {
    this.aonCK = aonCK;
    this.sData = sData;
    this.encKey = encKey;
    this.isDistributingFile = true;
    this.serverNr = serverNr;
    this.sharingPeersOK = 0;
    this.lastUpdateTime = System.currentTimeMillis();
  }
  <...>
}
```

Listing 5.3 presents the new `IDAsharing`. It was converted to a more similar sharing as the AON or SSS sharings. The main differences to its previous version are that it now implements the same interfaces as the AON and SSS sharings and, besides the implied changes of the introduction of the `share` interface, some of the data types used to hold information were changed or removed. For example, replacing the `receipt` is now a HashMap that maps the hash values with each block of the data share. The structure `receipt` had other purposes, such as serve as a ticket of the backup, which was used to prove authenticity of a sharing peer in a backup operation. The information that it stored is now shifted to the `IDAsharing` and it is this new sharing structure that is sent to peers, as it is done in the other schemes.

The use of this data structure improves the performance of computing the election of a valid hash value for each block during an integrity check operation. The Java BigInteger is still used to represent the bytes of the input file by a long integer value.

Listing 5.3: The `IDAsharing` class

```
public class IDAsharing implements Serializable, share {

  private static final long serialVersionUID = 1L;
  private BigInteger[][] vecKey;
  private sharingData sData;
  private HashMap<Integer, String> hm_ck;
  protected int sharingPeersOK, serverNr;
  private byte[] encKey;
  private String shareCK;
  private Vector<IDAsharing> restoredShares;
  private boolean isDistributingFile, isRestoring, isUpdating;
  private boolean isCheckingIntegrity, isRecovering;

  public IDAsharing(int serverNr, byte[] key, BigInteger[][] vecKey,
  sharingData sData, HashMap<Integer, String> hm_ck, String shareCK) {
    this.serverNr = serverNr;
    this.vecKey = vecKey;
    this.encKey = key;
    this.sData = sData;
    this.hm_ck = hm_ck;
    this.shareCK = shareCK;
    this.restoredShares = new Vector<IDAsharing>();
    this.isDistributingFile = true;
  }
  <...>
}
```

5.3 AON Protocol

A backup protocol is processed by different classes in the application. An AON protocol is not different. For an operation like the backup of a file, it starts as a request from the user through the GUI, which deploys the request to the `control` class in its monitor. The `control` then analyses the request and calls the respective method to handle it. It performs some small tasks, mainly preparing the operation for other methods, and delivers the main part of the work to the `shareManager`. This manager takes care of the major part of the protocol but requires the algorithm engine to process the data. After the algorithm finishes the work over the data, the manager creates the shares and passes them to the `control`, which will make use of the `sendManager` to send them to every sharing peer.

In order to explain the details of how the AON protocol has been implemented, follows the example of how the application performs a backup of a file, starting from the point where the `control` class handles the request. It receives information from the GUI about the file to backup, the list of sharing members names and master peers and how many peers are required to restore the backup. The last value is not used in this scheme since the replication model is not yet implemented. Listing 5.4 presents this method.

Listing 5.4: `backupWaon` method

```
private void backupWaon(File f, Vector<String> names,
                        Vector<String> sus, int k) {
  //generate the keys
  byte[] key = cryptoTools.generateAESKey().getEncoded();
  byte[] encKey = cryptoTools.generateAESKey().getEncoded();

  //creating the sharings
  Vector<String> peerOwnerIDs = pm.getOwnerIDs(names);
  Vector<String> susIDs = pm.getOwnerIDs(sus);

  //generate shares
  Vector<share> extSharings = sm.generateAONsharings(f,key,encKey
                      peerOwnerIDs,susIDs,pm.getMyOwnerID(), k);

  //Send the sharings
  for (int i=0;i<extSharings.size();i++)
  sendm.sendSharing(extSharings.elementAt(i),null);
}
```

The method still generates the key that will be used to encrypt the file and the key for the AON algorithm. They are both 128-bit keys, required for the AES algorithm and matching the size of the AON blocks. It also prepares the list of users and delivers all that to the `generateAONsharings` method of the `shareManager`.

As it follows from the protocol, the manager first runs the AON algorithm over the file and then encrypts the result. After that, and not shown in Listing 5.5, the manager constructs the data structures: sharing peers, sharing data and AONsharings. In the end, stores its own share and data share, in case if it belongs to the sharing peers, and returns the remaining ones to `control` for being sent to all other sharing peers. If the owner of the backup does not belong to the sharing peers, it will always keep the share of the backup but naturally not the data share.

Listing 5.5: `generateAONsharings` method

```
protected Vector<share> generateAONsharings(File f, byte[] key,
byte[] encKey, Vector<String> peerOwnerIDs, Vector<String> sus,
String myOwnerID, int k) {
  int membersSize = peerOwnerIDs.size();
  byte[][] inFile = fileHandler.readFromFile(f,Constants.AON_BLOCKSIZE);

  //do aon
  aonEng = AONengine.getInstance();
  byte[][] aonedFile = aonEng.doAON(inFile,key,false);

  //save output to file
  String fileCK = cryptoTools.createCheckSum(f);
  File out = new File("Constants.WORKING_FOLDER"+fileCK);
  fileHandler.writeToFile(aonedFile, out);

  <...>

  mySharing.setIsDistributingFile(true);
  mySharings.add(mySharing);
  timers.add(new Timer());

  extSharings.removeElementAt(0);
  return extSharings;
}
```

The call for the AON algorithm provides the file, represented as a two-dimension array of bytes, the 128-bit key and a boolean value specifying if it will be used Rivest's package transform or Desai's alternative, respectively for values true and false. This choice is left to the user to make at the moment of selecting the settings for the backup, through the GUI.

The `AONengine` class implements both Rivest's and Desai's versions of the AON scheme. The class makes use of the singleton design pattern for avoiding to be created more than once. This design pattern has also been applied to many other classes, such as the other algorithm engines and all managers, as a security measure. The `doAON` method runs the package transform algorithm over the input file. Listing 5.6 presents this class and Listing 5.7 the method.

Listing 5.6: The `AONengine` class

```
public class AONengine {

  private static AONengine _instance = null;

  protected AONengine() {}

  public static AONengine getInstance() {
    if (_instance == null)
    _instance = new AONengine();
    return _instance;
  }
  <...>
}
```

Listing 5.7: The `doAON` method

```
public byte[][] doAON(byte[][] in, byte[] key, boolean rivest) {

  int nBlocks = in.length;
  byte[][] out = new byte[nBlocks+1][Constants.AON_BLOCKSIZE];
  byte[][] hashes = new byte[nBlocks][];

  /**Rivest's AONT is done in 3 steps **/
  //step 1 - compute 128-bit blocks as IN[i] XOR E(key,i)
  for(int i=1, j=0; j<nBlocks; i++,j++) {
    byte[] data = intToByteArray(i);
    try{
      byte[] inEncKey = cryptoTools.encryptAON(data,key);
      out[j] = xorByteArray(in[j],inEncKey);
      //step 2 - create hashes of blocks
      if (rivest)
      hashes[j] = cryptoTools.createDigest(xorByteArray(out[j],data));
    }catch(CryptoException ce) { }
    }

  if (rivest) {
    //step 3 - last block
    byte[] bigHash = new byte[Constants.AON_BLOCKSIZE];
    for(int i=0; i<nBlocks; i++)
    bigHash = xorByteArray(bigHash,hashes[i]);
    out[nBlocks] = xorByteArray(bigHash,key);
  } else {
    byte[] bigBlock = new byte[Constants.AON_BLOCKSIZE];
    for(int i=0; i<nBlocks; i++)
    bigBlock = xorByteArray(bigBlock,out[i]);
    out[nBlocks] = xorByteArray(bigBlock,key);
  }
  return out;
}
```

The `AONengine` implements auxiliar methods to perform low level operations that are not available directly from the Java API for the byte data type, but are for the BigInteger type. However, in an effort to not penalize the performance of the algorithm, auxiliar functions are implemented, such as `xorByteArray(byte[] one, byte[] two)`.

When the algorithm finishes to process the file, the data is stored in disk and it is sent to sharing peers using the `fileSender` method from the `sendManager`. This occurs after the backup owner sends out the shares and obtains the replies confirmation. If the file is of a considerable size, the parts of the file will be processed sequentially. Each block will be stored in disk in a concatenated file.

5.4 IDA Protocol

The IDA protocol has suffered some changes in this extension. The implemented algorithm was only changed to perform better the matrix multiplication operation. Instead of using the `Matrix` class to handle this operation, which is very time consuming due to the many times that values need to be converted to the `Matrix` structure, a standard way of performing matrix multiplication is directly implemented in the `IDAencoder` as shown in Listing 5.8. However, the `Matrix` class is still used to invert the vector key matrix. The reason for not implementing the matrix inversion in the `IDAdecoder` class is due to the fact that it only occurs once for each restore operation. Therefore, it does not represent a performance gain.

Listing 5.8: The IDA matrix operation

```
while((chunk = idaFileReader.getChunk()).length > 1) {
  BigInteger[][] computedChunk = new BigInteger[chunk.length+1]
                                               [chunk[1].length];

  //the IDA matrix operation: [ Cik = Ai*B(k-1)m+1 + ... + Aim * Bkm ]
  for (int row=1, vrow=0; row <= n; row++,vrow++) {
    for (int f=0; f<chunk[1].length; f++) {
      BigInteger accValue = BigInteger.ZERO;
      for(int i=1,j=0; j<m; j++,i++) {
        accValue = accValue.add(vecKey[vrow][j].multiply(chunk[i][f]));
      }
      computedChunk[row][f] = accValue.mod(Constants.pIDA);
    }
  }
  <...>
}
```

Each BigInteger represents a block of 512 bytes. This means that replacing the data type BigInteger for byte, would require 512 multiplications. But, a multiplication between two huge BigInteger also requires an expensive computation cycle. Due to time constrains, it was not possible to implement a version of this algorithm using byte as data type, and perform a realistic comparison of the performance achieved by each case.

Regarding the protocol, the main change is in sending the data share to all sharing peers at the end of the operation, just like in the AON and SSS protocols. Thus, the sending process that the previous version implemented, that used the `idaShare` data structure to send each data chunk, is now being stored temporarily in disk. A private folder is setup at the beginning of each backup operation for avoiding conflicts and mistakes with the data shares. The folder uses the hash value of the file as name and it is erased at the end of the operation. Data shares and respective checksums are stored using two HashMap structures, using chunk numbers as keys. These structures are then stored in disk until all chunks are gathered. Then the HashMap related to the checksum of the chunks is taken into a sharing structure and sent to peers, using the `packetSender` class. The other HashMap is sent as a normal file communication, using the `fileSender` class.

The auxiliar classes `fileReader` and `collector` assist the algorithms to work over sequential blocks of the input data, preventing that too much information is loaded to the memory of the system and eventually run out of memory. Both IDA and AON algorithms use them for handling big files. However, the block size used in each algorithm differs. The `fileReader` receives as argument the block size that should be provided to the algorithm. Furthermore, this class provides a way to allow the algorithm to request for a new block, useful for the IDA algorithm that may not be fast enough for processing each block in a similar period of time as the `fileReader` takes for reading and loading the block to the algorithm.

The `collector` class uses a monitor to queue blocks that the AON or IDA engines have finished to process. The blocks are stored in disk, appended to the same file, or files if in the case of an IDA backup.

5.5 Security

The cryptographic security of the application is implemented in just one class, the `cryptoTools`, that is located inside the Main package. It uses the Bouncy Castle libraries as algorithms provider.

This class provides encryption and decryption methods using symmetric and asymmetric algorithms. For encrypting or decrypting packets and files that are sent between files, the symmetric AES algorithm is used with a 128-bit sized key. This key is encrypted or decrypted with the asymmetric algorithm RSA, using a 2048-bit sized key. The RSA key is provided by the user certificate that it must hold to run the application. AES keys can be generated through an existing method in this class. The class also provides methods to compute the hash value, or checksum, of an object or a file. Methods to verify if the checksum of an object or file is correct are also available.

5.6 Extra

There are some features that are useful in any application, such as a proper logging mechanism or an automated way of setting up the working platform. This section discusses the implementation of the former feature.

5.6.1 Log4J

Log4J [14] is a logging system for applications implemented in Java, that allows logging at runtime without modifying the application binary. It provides different levels of logging to be applied depending on the case: info, debug, warning, error and fatal. Developers can use them while implementing an application and then control the logs behavior by editing a configuration file. The output of the log can be a file or the console. This makes logging extremely practical and easy.

Resilia logging mechanism outputs messages the console, but any debugging log has to be activated in the application source code. This is not practical for using logs for debugging error scenarios that occur to users. A more convenient way is to place debug logs in a special file so that it can be easily found by the user. Also, allowing users to easily access the logging configuration file and change the logging level, will give them the power to control the log system, for cases where logging everything would be too expensive in terms of space. JXTA makes use of the Log4J package, so it is already available with Resilia.

Listing 5.9 presents the implementation of the Log4J system in the `control` class. All other classes can instantiate this logger or create a new one. There is no limit for how many loggers can one system be running. The example in Listing 5.9 logs all levels available and outputs them into a file call `LOG` under the working folder of the application, which is `tmp/`. Commented is an alternative way for outputting the log to the console.

Listing 5.9: Logging setup in `control`

```
private static final Logger LOG = Logger.getLogger(Control.class);

<..>

LOG.setLevel(Level.ALL);
myLayout = new SimpleLayout();
myAppender = new FileAppender(myLayout, Constants.WORKING_FOLDER+
                        +"LOG"); //new ConsoleAppender(myLayout);
LOG.addAppender(myAppender);
```

CHAPTER 6

Evaluation

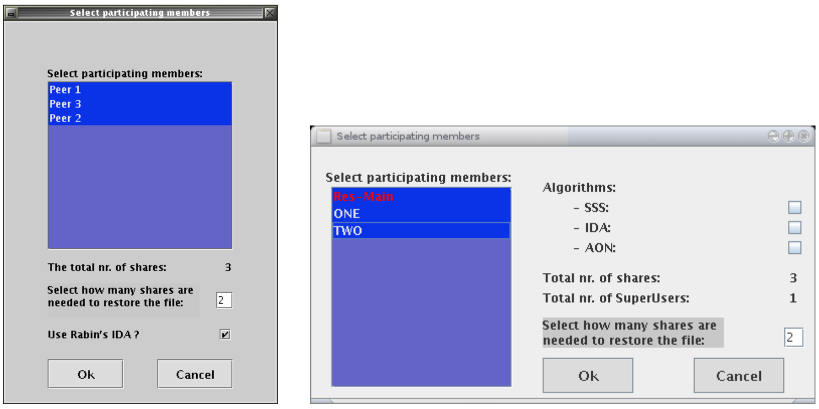
This chapter presents an evaluation of the implemented AON algorithm and the improvements made to the IDA algorithm. This evaluation is mainly based on the results of running operations in a small network of peers.

6.1 Prototype Overall Evaluation

The prototype has been heavily tested during the evaluations performed in [21, 19]. Thus, it is not necessary to re-run some of the evaluations since the changes in the application would not interfere with them.

As a backup application to SMEs, Resilia aims to be simple to setup and to use. For peers directly connected to the Internet, this setup is easy. However, more attention is required to setup a peer that is behind a firewall or inside a private network that uses NAT. This is a typical network scenario for a SME. The configuration of peers under these circumstances requires users to provide valid IP addresses or URLs of rendezvous and relay peers outside of the private network.

After a correct setup of the P2P platform, Resilia is a very easy to use application. Throughout the graphical interface, users access all operations available and obtain direct information about the peers present in the same peer group and two lists of backups: one for backups the user owns or is a master of and one for backups that he is currently storing in his machine. Performing any of the available operations is done in a few simple steps. The most complex part is the choice of the settings for performing a backup of a file and even this one does not require must expertise. The user must choose the algorithm to be used, the number of peers that will store the backup and how many shares will be required to restore it. The last value is not yet used with the AON protocol since the replication system designed is not yet implemented. The graphical frame that interacts with the user for setting the algorithm proprieties has been improved during this implementation, as can be seen in Figure 6.1.


Select participating members

Select participating members:
Peer 1
Peer 3
Peer 2

The total nr. of shares: 3
Select how many shares are needed to restore the file: 2
Use Rabin's IDA ? ✓

Select participating members

Select participating members:
Res-Main
ONE
TWO

Algorithms:
- SSS: ☐
- IDA: ☐
- AON: ☐

Total nr. of shares: 3
Total nr. of SuperUsers: 1
Select how many shares are needed to restore the file: 2

a) b)

Figure 6.1: Backup settings frame: a) previous version b) current version

The application is platform independent due to the XML advertisements used by JXTA, which is also implemented in Java programming language, as the prototype. This means that users do not have to worry about what operating systems are they and the other peers running. The same goes for what type of network architecture they are in. To prove this independence, tests were performed using Windows and Linux operating systems.

Regarding the functionality of the application, it can be evaluated by confirming that the implemented protocols and operations perform the tasks they were designed to. For example, users cannot launch the application without holding a valid certificate and inputing the correct password for it.

6.2 Operations Evaluation

For evaluating the new implemented operations, a simple 3-node network was setup on a private network. For this configuration, there was no need to setup any rendezvous or relay peer, however, when peers enter a new peer group they look for an existing rendezvous peer for the group. If one is not present, one peer will become rendezvous for that peer group. Machines with 2.4 GHz Pentium 4 processor and 512 Mb of memory were used to run the tests.

6.2.1 Backup and Restore

The backup and restore operations are the main ones of the application. They comprise the purpose of any backup system.

6.2.1.1 Using AON

In order to evaluate how the AON protocol performs, different setup tests were made. Since there is not yet available a replication scheme to be used with the AON backup protocol, files are always fully replicated to all peers. For the AON algorithm, the number of sharing peers does not change any of the computation that is performed. This means that the only possible setting that can be altered in a backup operation is the size of the backup file. Therefore, in order to better analyse how costly is to perform an AON backup, three files of sizes 2, 5 and 10 Mb were distributed to all 3 sharing peers. The results are presented in Table 6.1. Desai's scheme was used in the experiment.

Table 6.1: AON Backup and Restore operation results

File Size	**Backup**		**Restore**	
	AON	Total	AON	Total
2 Mb	1.2 s	9.6 s	1.1 s	1.9 s
5 Mb	2.8 s	13.7 s	3.2 s	4 s
10 Mb	7.3 s	25.7 s	5.1 s	5.8 s

From the results it is possible to conclude that the AON operation is performed quickly, representing an interval from 15% to 30% of the total time taken by the operation to conclude. The time for the AON algorithm is measured from the point when the `AONengine` is instantiated until it returns the computed file. The total time is measured since the point where the `control` class receives the request, until it receives the file received confirmations. Thus, it includes the sending of the files, which the bigger the file, the longer it takes to send, and the reception of the confirmation message of the majority of the sharing peers, which in this case is just one (excluding the peer performing the backup operation). Is then acceptable the difference between the time that the AON algorithm takes to run over the different sized files and the time the backup operations take to complete.

Regarding the times obtained by the restore operation, they represent very similar measurements for the AON process as in the backup operation, which was expected, since the AON algorithm and its inverse require the same number and type of operations. However, these measurements for both backup and restore runs of the AON algorithm not always present proportional values, which can be explained by some extra CPU activity at the moment of the test. The relatively low times for the total restore operations are due to the fact that the peer restoring the backup stores the backup file. Hence, it does not need to request for the backup to other peers, avoiding communication costs and performing faster. Restoring a backup when the owner peer does not belong to the sharing peers takes a longer period of time to complete because of the implied request and transfer of the backup file from other sharing peer. This scenario increments a few seconds to the total time, depending on the size of the file and the speed of communication. For example, using the same 5 Mb file as in the previous test, the restore operation in this case scenario takes approximately 8 seconds to finish.

6.2.1.2 AON schemes: Rivest vs Desai

To compare both AON implemented schemes, the original packet transform from Rivest [24] and a variant from Desai [12], a simple test was carried out just using the algorithms and not the entire backup and restore protocols. The results of running the methods `doAON` and `undoAON` over an approximately 700 Kb file are presented in Table 6.2.

Table 6.2: Comparison between Rivest and Desai algorithms

Method	**Rivest's scheme**	**Desai's scheme**
`doAON`	925 ms	725 ms
`undoAON`	698 ms	422 ms

The extra encryption operation used by Rivest's scheme makes it 20–40% slower than Desai's simplified variant. This difference can be significative when running the algorithm over larger files. To note as well the unexpected difference between the measured times to run the algorithm and its inverse. The `undoAON` method performs 25% faster than the `doAON` when using the Rivest's scheme, and over than 40% when using Desai's scheme.

6.2.1.3 Using IDA

The changes to the IDA algorithm and protocol also need to be evaluated. For the IDA, one more setting can be tested than for the AON protocol, which is the number of required peers to restore the backup. Thus, using different file sizes and different settings, the same type of test is carried out. Table 6.3 presents the results obtained.

It follows from the algorithm that when the ratio $\frac{n}{m} \approx 1$, the result is space efficient, producing smaller shares. Hence, faster computation of the shares. This is the reason behind the measured time for the setup with $m = 3$ achieving better performance results than the setup using $m = 2$. This also means that

Table 6.3: IDA Backup results

File Size	$n = 3, m = 2$		$n = 3, m = 3$	
	IDA	Total	IDA	Total
1 Mb	4.6 s	11 s	-	-
5 Mb	58.3 s	85.3 s	30.6 s	48.4 s
10 Mb	150.6 s	174.1 s	113.2 s	161.5 s

the relation between the total time for completing the operation and the time for running the IDA algorithm should be expected to be smaller than the verified for the case of the 10 Mb file. This result can be explained due to some delay for receiving the sharings received confirmation from the sharing peers.

Comparing these results to the results obtained by the first implementation of the IDA algorithm, there is an obvious improvement. For the $n = 3, m = 2$ setup, the previous implementation obtained the times: 19.6 s, 141 s and 203 s for the IDA operation and 40.8 s, 185 s and 221 s for the total backup protocol, for the respective files of sizes 1, 5 and 10 Mb. These results represent an improvement ranging from 30% to 75%.

Although the achieved improvements, the IDA is still an expensive algorithm to run. However, it brings advantages in relation to the space that is saved for storing smaller chunks of data, with the added hability for recover a file even in the presence of a limited amount of data errors. When the other designed improvements are implemented to the algorithm and the protocol, it is expected another performance improvement.

6.2.1.4 Comparison Between Algorithms

Table 6.4 uses the backup of a 10 Mb file for establishing a comparing term for all three schemes in relation to performance and communication overhead.

Table 6.4: Comparison of beckup schemes for a 10 Mb file

Algorithm	**Time for Backup**	**Data Communicated**
SSS	23.7 s	30 Mb
IDA	174.1 s	15 Mb
AON	25.7 s	30 Mb

It is possible to conclude that both SSS and AON algorithm achieve similar results. They send the entire backup file to all sharing peers, obtaining a fully replicated backup, and performing in a reasonable short period of time. However, the AON works over the entire file length, which can be several times bigger than the fixed size 2048-bit (256 bytes) secret that the SSS works with. Therefore, for larger files it is expected that the time taken by the SSS algorithm increases much less than the time taken by the AON. Both protocols need to distribute the same amount of data to peers, but the AON algorithm has to compute the entire file.

On the other hand, the IDA protocol achieves a poor performance, when compared with the other two, but reduces by half the data that is communicated and stored in all peers. For this example, the size of each share could still be shortened to a minimum of 10 Mb by setting $m = 3$. Thus, there is a trade off at users disposal, between the time to perform the backup and the space to store it.

6.2.2 Integrity Check

To evaluate the integrity check operation, a backup of a file was executed and then, manually, the share data on one of the peers was altered. The data alteration only consisted in editing the file, randomly removing and adding parts of data, so that it could be considered as incorrect and would require to be overwritten by a correct backup during the integrity check operation.

This operation requests to all sharing peers a hash value of the backup and compares them to find incorrect shares. If an incorrect, or inexistent, share is found, a valid one is sent to the peer. Using a file of 11 Mb, the integrity check took 3.1 seconds to make the request, receive and analyse the replies. It correctly discovered the wrong share and sent a valid one, in this case its own share, to overwrite the corrupted one to the peer. The time taken for sending the correct share is not accounted in the measured time for the operation.

6.3 Security Evaluation

It is of great importance that a backup system is able to enforce the security proprieties that has been set forth in the first implementation of the application. The implemented algorithms and data structures follow the same security standards as the ones already implemented, therefore it can be assumed that the security level of the application is maintained in the current prototype.

6.3.1 Security of AON Backups

Backups that use the AON algorithm are protected by the AON propriety that slows down a brute-force attack on the encrypted blocks of the file. Moreover, after the AON transform, the file is encrypted with a 128-bit key and the AES algorithm, which is still assumed to be resistant to practical attacks, although theorical attacks, such as [9], have been presented that might be possible to break it.

With the introduction of a replication scheme, each peer will only store part of the backup, which provides another difficulty to attackers, since they cannot obtain the entire backup data by just breaking in one peer. Thus, the implementation of the replication scheme will provide another level of security to the application when using AON and SSS algorithms. The IDA protocol already provides this dispersion of the data.

The communication of the backup to other peers takes place when the file is already encrypted, thus the above mentioned beliefs in the used cryptographic algorithms assure that an attacker will not access the data even in the case of obtaining the backup data during its communication to a peer.

The communication of the share is also done in a secure way. The share contains sensitive information, such as the keys used to encrypt the file and for the AON algorithm. The share is protected by the communication packet, which encrypts again the sharing data with a new symmetric key and then this key with the recipient's RSA public-key.

6.3.2 Threat Evaluation

An application evaluation of threats has been made during previous implementations. Thus, here is presented a brief overview for the sake of completeness. For a more detailed evaluation, please refer to the [21, 19].

Attacks to communication protocols are successfully performed if the attacker gets any information of it, or if he changes the normal behavior of the system. They can be of different types:

- Injection of data from the outside;
- Modification of a backup share on a trusted node;
- Man-in-the-middle attack;
- Replay attack.

By injecting a packet in the system, an attacker tries to cheat one or more nodes. In order to a peer accept a data packet, it must be encrypted and signed by a valid key, which can be verified through the peer's certificate. Assuming that the attacker is not a user of the system, then he does not have a valid certificate and therefore is unable to fake a data packet.

Modifications of data shares are easily detected. Computing and comparing checksums of shares or blocks of shares provide a correct analysis of their integrity. Damaged or tampered shares are rejected by the application. The integrity check operation provides a way to frequently verify the integrity of all distributed shares and replace incorrect by correct shares.

A man-in-the-middle attack takes place whenever an attacker stands in between two communicating peers, stealing exchanged packets and replacing them by new ones or altering part of the data packets. A man-in-the-middle attack cannot be performed because the keys for uncovering the desired data travel encrypted by public-key cryptography. This invalidates any action performed by a man-in-the-middle, since packet modifications would be noticed, as mentioned above. The creation of new packets fails due to signature verification, performed by each node for each packet, as mentioned for the packets injection attack[‡].

Replay attacks are avoided by timestamping all packets during their creation. Resilia keeps control over the time of the last packet received. Thus, older packets are ignored. Although this security issue is implemented, it assumes that all peers run a clock synchronism mechanism. The application does not enforces the use of one[‡].

6.4 Jxta Evaluation

JXTA is the platform providing peer-to-peer functionality to Resilia. It specifies how all communication takes place. The current prototype of Resilia is using version 2.4 of JXTA, which depends on the following packages and APIs:

- Java Standard Edition 5.0 Compatible VM (`http://java.sun.com`)
- Log4J 1.2.13 (`http://logging.apache.org`)
- BouncyCastle 1.33 (`http://www.bouncyCastle.org`)
- Jetty 4.2.25 (`http://jetty.mortbay.com`)

To note that the version of each package or API mentioned above is not the exact requirement of JXTA, but the version that is being used in the current implementation of the prototype.

Throughout the experiments carried out, the JXTA platform performed reasonable well, connecting all peers involved, setting up correctly all the resources, such as peer groups, and handling well the publishing and discovery of advertisements. The communication, through the use of JXTA sockets, is reliable and fast. However, a few timeout exceptions occurred when two peers tried to communicate.

The JXTA configurator that is initiated the first time Resilia runs, has the purpose of setting up the platform. Although it is not hard to configure it, an easier to use interface could be implemented by the application so that an automatic setup of the platform could be done. No tests have been done to confirm the configuration of a scenario involving peers behind NAT or a firewall that try to communicate with the outer world. However, it is assumed that it is only a question of configuration, and therefore assumed to work.

Regarding the security aspects, the JXTA platform provides a cryptographic toolkit with a basic set of security algorithms that can be used by the applications. However, Resilia does not take advantage of this toolkit and implements its own security methods using Bouncy Castle as a provider. Besides the cryptographic toolkit, the JXTA platform provides Secure Socket Layer (SSL) as way of securing the communications. Secure pipes can be specified, which internally use SSL to guarantee safety against attacks. Furthermore, this layer requires the use of certificates. Resilia makes use of peer certificates which are distributed along with the peer advertisement and serves as prove of its identity. Also, the peer certificate is used as a personal security tool as a first defense against physical attacks, by requiring that the user inputs the correct password to decrypt the private key of the certificate in order to start the application.

Chapter 7

Conclusion

This project undertook the task of designing and implementing the All-or-Nothing package transform proposed by Ronald Rivest [24]. This mode of encryption provides an interesting propriety that forces the decryption of the entire ciphertext in order to gain access to any piece of the plaintext. The package transform integrates with an existing peer-to-peer backup system that already provides two alternative ways of ensuring the confidentiality, integrity and availability of backups.

Also part of the goals set forth in Section 1.1, this project investigates the applicability of a replication system that can be combined with the implemented package transform, in order to provide fault tolerance to backups and the possibility of reducing the communication and storage load.

As a consequency of building an extension to existing projects, a re-organization of the application structure is required for a cleaner and better functioning. Furthermore, a revision of the already implemented IDA scheme is taken into consideration, in an effort for making it usable and perform better.

7.1 Achievements

The outlined objectives have been achieved to a large extent as now discussed.

Backups are cumbersome. Resilia has been designed to be efficient, easy to use and secure. With a new alternative way for performing backups, the application allows users to choose between a quick backup or a reduction of the required storage space. All the schemes implemented are believed to provide similar levels of security to stored backup files.

The developed AON scheme implements the original package transform from Rivest and a simplified version from Desai [12], which performs better by skipping one security feature of Rivest scheme. The experiments carried out during the evaluation of the changes in the prototype revealed good results while running all AON operations. Not only they performed their task correctly as they also ran in acceptable time. The time to backup a 10 Mb file was approximately the same as using the SSS alternative, whereas the SSS only computes an 256-byte piece of information while the AON computes the entire 10 Mb of the file.

When used together with the designed replication scheme, the good performance results will be maintained but it will be space efficient. In this scenario, this third alternative will represent a half-way point in the trade off between the fast backup achieved with a SSS backup and the space efficient backup achieved with an IDA backup.

Regarding the alterations to the IDA protocol and algorithm, the results achieved are also very positive. The improvement of an average of 40% in the performance makes this alternate way more usable. It can be expected to obtain further improvement when the other designed changes proposed in Section 4.4 are implemented.

The overall result is a reliable, robust and easy to manage backup system, which provides safe and secure backups.

7.2 Future Work

As have been mentioned throughout this report, there are some components or protocols that were designed but not implemented, due to time constrains. All these are part of future work, along with some other details, in order to make Resilia a more usable, robust and reliable backup tool.

The designed replication scheme is probably the most important feature to be added next to the application. It will complete the AON backup protocol in such a way that will make this scheme a fast, reliable and space efficient method for backing up files. And can also be extended to be used with the SSS backups, providing space efficiency to this backup protocol. Another mentioned feature is the JXTA configurator, to make the life of users easier. This is however a not so important feature to add.

Other important suggestions for future work are the schemes combination in one backup. Associating the advantages of the SSS scheme with the IDA or AON protocols, would allow users to increase the security of the backups, by protecting the data shares and the share secrets.

An interesting feature to the application is the introduction of a compressing algorithm to be applied before the backup operation. This would make the backup files smaller, improving in that way the time that all algorithms would need to perform the backup operation and reducing the storage and communication load.

The current prototype allows the backup of a selected file. Users can be interested in backing up an entire folder, with sub-folders and files. Although users can work around this by creating beforehand an archive of the desired folder, and possibly compressing it too, allowing the application to do those tasks for the user would make their life easier.

By last, an improved and more appealing user interface could make the application more user-friendly.

Bibliography

[1] 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.

[2] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. *Lecture Notes in Computer Science*, 950:92–111, 1995.

[3] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[4] Victor Boyko. On the security properties of oaep as an all-or-nothing transform. In *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 503–518, London, UK, 1999. Springer-Verlag.

[5] Valér Canda and Tran van Trun. A new mode of using all-or-nothing transforms. In *Proceedings of the 2002 IEEE International Symposium on Information Theory*, page 296, 2002.

[6] Ran Canetti, Yevgeniy Dodis, Shai Halevi, Eyal Kushilevitz, and Amit Sahai. Exposure-resilient functions and all-or-nothing transforms. *Lecture Notes in Computer Science*, 1807:453–469, 2000.

[7] CleverSafe. Cleversafe dispersed storage project. http://www.cleversafe.org/wiki/Cleversafe_Dispersed_Storage.

[8] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, Concepts and Design*. Addison-Wesley, third edition, 2001.

[9] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *Lecture Notes in Computer Science*, volume 2501, pages 267–287, Queenstown, New Zealand, December 2002. Asiacrypt 2002, 8th International Conference on the Theory and Applications of Cryptology and Information Security, Springer.

[10] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36(SI):285–298, 2002.

[11] Landon P. Cox and Brian D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 120–132, New York, NY, USA, 2003. ACM Press.

[12] Anand Desai. The security of all-or-nothing encryption: Protecting against exhaustive key search. In *CRYPTO '00: Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, pages 359–375, London, UK, 2000. Springer-Verlag.

[13] Patrick C. Fischer and Robert L. Probert. Efficient procedures for using matrix algorithms. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, pages 413–427, London, UK, 1974. Springer-Verlag.

[14] Apache Software Foundation. Log4j. `http://logging.apache.org/log4j/docs/`.

[15] Joseph D. Gradecki. *Mastering JXTA: building Java peer-to-peer applications.* Wiley Publishing, 2002.

[16] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Proc. Crypto'95 (LNCS 963)*, pages 339–352, NY, 1995.

[17] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and B. Kemme. How to Select a Replication Protocol According to Scalability, Availability, and Communication Overhead. In *IEEE Int. Conf. on Reliable Distributed Systems (SRDS'01)*, New Orleans, Louisiana, October 2001. IEEE CS Press.

[18] Emin Martinian. Distributed internet backup system (dibs). `http://www.csua.berkeley.edu/~emin/source_code/dibs/`.

[19] Fernando Meira. Resilia: A safe & secure backup-system. Final year project, Engineerng Faculty of the University of Porto, May 2005.

[20] Sun Microsystems. Project jxta. `http://www.jxta.org`.

[21] Jacob Nittegaard-Nielsen. Sikkert og plideligt peer-to-peer filsystem. Master's thesis, Technical University of Denmark, 2004.

[22] The Legion of the Bouncy Castle. The java bouncy castle crypto apis. http://www.bouncycastle.org/.

[23] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, 1989.

[24] Ronald L. Rivest. All-or-nothing encryption and the package transform. *Lecture Notes in Computer Science*, 1267:210–218, 1997.

[25] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[26] Claude Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, 1949.

[27] D. R. Stinson. Something about all or nothing (transforms). June 1999. Avaliable from http://cacr.math.uwaterloo.ca/~dstinson/.

[28] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969.

[29] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.