Secure Storage in Cloud Computing

Abbas Amini

Kongens Lyngby 2012
IMM-M.Sc.-2012-39

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

Summary

In this Master's thesis a security solution for data storage in cloud computing is examined. The solution encompasses confidentiality and integrity of the stored data, as well as a secure data sharing mechanism in the cloud storage systems. For this purpose, cryptographic access control is used, which is mainly based on cryptography. Based on an analysis of the cryptographic access control mechanism, a design is created for a system, which is intended to demonstrate the security mechanism in practice. Finally, on the basis of the proposed design, a prototype is implemented.

Data confidentiality and integrity are ensured by data encryption and digital signature respectively. For encryption of data, symmetric cryptography, and for digital signature process, asymmetric cryptography is used. The main quality of the system is that all cryptographic operations are performed on the client side, which gives the users more control on the security of their data, and thus the data are not dependent on the security solutions provided by the servers.

The proposed mechanism also supports a secure file sharing mechanism. A user is able to grant other users "read access", or "read and write access" permission to his stored data. The different levels of access permission are granted by exchanging the corresponding keys between users. For granting read access, public key and symmetric key, and for granting read/write access, public, private and symmetric keys have to be exchanged between shared users. The process of exchanging keys is performed by first creating a so called key ring, which contains a list of the necessary keys, and then the key ring is distributed in order to grant access permission to other users.

Resumé

I dette eksamensprojekt er en sikkerhedsløsning til datalagring i "cloud computing" blevet undersøgt. Løsningen omfatter fortrolighed og integritet af oplagrede data, samt en sikret fildelingsmekanisme i "cloud storage" systemer. Til dette formål, kryptografisk adgangskontrol er blevet brugt, som primært er baseret på kryptografi. Baseret på en analyse af kryptografisk adgangskontrolmekanismen, et design er blevet opbygget for et system, som har til hensigt at demonstrere sikkerhedsmekanismen i praksis. På basis af det foreslåede design er en prototype blevet implementeret.

Datafortrolighed og dataintegritet er sikret ved hjælp af henholdsvis datakryptering og digital signatur. Der er blevet anvendt symmetrisk kryptografi til datakryptering, og asymmetrisk kryptografi til digital signatur. Systemets centrale egenskab er, at alle kryptografiske operationer bliver udført på klientsiden, som giver brugerne mere kontrol over sikkerheden af deres data, og dermed er dataene ikke afhængig af sikkerhedsløsningerne, som er blevet stillet til rådighed af serveren.

Den foreslåede mekanisme understøtter også en sikret fildelingsmekanisme. En bruger er i stand til at tildele andre brugere læserettigheder, eller både læse- og skriverettigheder til sine data. De forskellige niveauer af adgangstilladelse kan blive tildelt ved hjælp af udveksling af de tilsvarende nøgler. For at tildele læserettighed, skal den offentlige og symmetriske nøgle, og for at tildele både læse- og skriverettighed, skal den offentlige, den private og den symmetriske nøgle være udvekslet mellem delte brugere. Proceduren af udveksling af nøgler bliver udført ved først at danne en såkaldt nøglering, som indeholder en liste af de nødvendige nøgler, og det er nøgleringen, som så distribueres for at tildele adgang til andre brugere.

Preface

This master's thesis was prepared at the Department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU) in partial fulfilment of the requirements for acquiring the M.Sc. degree in Computer Science and Engineering.

The work was carried out in the period October 24th 2012 to May 4th 2012 and is worth 30 ECTS points. The project was supervised by Associate Professor Christian Damsgaard Jensen, DTU IMM.

The thesis deals with secure storage in cloud computing. In this context, different aspects in cryptographic access control are analysed in order to provide a solution for ensuring confidentiality and integrity of data, and also a secure file sharing mechanism in cloud storages. On this basis a prototype is implemented to demonstrate the proposed security mechanism in practice.

Lyngby, May 2012

Abbas Amini

Acknowledgements

I thank my supervisor Christian D. Jensen for his advice and guidance at the start of the project, and for his constructive feedback throughout this work. I really appreciate his interest and enthusiasm during this project.

I would like to thank my family and friends for their help and support during the project, especially my friend, Ahmad Zafari, for his useful comments and proof-reading of some parts of my work.

Contents

CHAPTER 1

Introduction

The term information technology is not so old, but we can not deny its extremely fast growth, especially in the last decade. There is no doubt about the big progress of the internet, which is the main factor in IT world, especially with regard to speed in data transfer, both in terms of wired and wireless communication. People run their business, do their researches, complete their studies, etc. by using the facilities available via the internet. Al in all the outsourcing of facility management is becoming more and more common.

Since the need for online services is increasing, the extent of services available through the internet, such as online software, platform, storage, etc., is also growing. This leads to formation of a structured provision of services, called cloud computing, which actually provides a huge amount of computing resources as services through the internet. One of the important services in the cloud is the availability of online storage, called cloud storage.

Cloud computing is a result of gradually development of providing services by forming clusters and grids of computers. The main concern is to provide a large amount of services in a virtualised manner in order to reduce the server sprawl, inefficiencies and high costs. So in cloud computing the servers that are used to provide services, among others cloud storage, are fully virtualised. This virtualisation mechanism makes it possible for cloud storage users to get the specific amount of storage that they need, and thus they are only required to pay for the used storage.

Since this huge amount of services is available online, the use of distributed systems is growing, and thus this new technology, namely cloud computing, is becoming more and more popular. People are moving towards using cloud storages in order to make use of the advantages, such as flexibility in accessing data from any where. People do not need to carry a physical storage device, or use the same computer to store and retrieve their data. By using cloud storage services, people can also share their data with each other, and perform their cooperative tasks together without the need of meeting each other so often. Since the speed of data transfer over the internet is increasing, there is no problem in storing and sharing large data in the cloud.

Cloud storage systems vary a lot in terms of functionality and size. Some of the cloud storage systems have a narrow area to focus on, like only storing pictures or e-mail messages. There are others that provide storage for all types of data. According to the amount of services they

provide, they range from being a group of small operations to containing very large amount of services, such that the physical machinery can take up a big warehouse. The facility that houses a cloud storage system is called a data centre. If we just have one data server, and connect it to the internet, it is actually enough to provide a cloud storage system, though it is the most basic level. The common cloud storage systems in the market are based on the same principle, but there are hundreds of data servers that lie at the back end. The computers usually need to be maintained or repaired, so it is important to have copies of the same data on multiple machines. Without this mechanism a cloud storage system cannot ensure data availability to the clients. Most systems store copies of the data to the different servers that are supplied with different power resources. In this way the data would still be available when power failure occurs on one server.

When discussing about all these improvements, we have to remember that there is a very important issue in IT world that must be taken care of, i.e. ensuring security. Users use the cloud storage facility to store and share their data, and especially when these data are secret, the need of security is mandatory. It means that the confidentiality and integrity of data are needed to be ensured. Moreover the stored data must always be available for retrieval, i.e. the system has to provide availability of data. In short, having security in cloud storage is actually ensuring confidentially, integrity and availability of stored data.

Many cloud storage providers claim that they provide a very solid security to their users, but we should know that every broken security system was thought once to be unbreakable. As some examples we can mention Google's Gmail collapse in Europe in February 2009 [36], a phishing attack on Salesforce.com in November 2007 [37] and a serious security glitch on Dropbox in June 2011 [38]. If we look a bit deeper in the structure of cloud computing systems, we may feel even more insecure, because they make use of multi-tenancy. Many cloud computing providers work with third parties, so users lose even more trust, especially when they do not know these third parties well. In such a situation users may not dare use the cloud storage system to store their private data. Apart from this, until now there has not been made any standardisation for the security in the cloud. Any software update could lead to a security breach if care is not taken. The mentioned Dropbox security failure was actually caused by a software update. However there are some "local" security standards within every cloud computing system, and some of the providers claim that for every software update, they review the security requirements for every user in the system. Another remarkable issue is the local government laws, and as a result data can be secure in one country, but not secure in the same level in another country. Because of the nature of cloud computing systems as being virtualised systems, users, in most cases, do not know in which country their data is stored. [39]

If we look closely at the above mentioned security issues, we would know that there is one central cause for the security problem: Users have no other choices than trusting the servers, because all the security operations are applied on the server side. In fact it is the cloud storage that has the responsibility to provide data security. Our goal is to introduce a security solution, which is applied on the client side and thus guarantees confidentiality and integrity of the

stored data in the cloud. This solution moves the security operations away from the servers, and makes it possible to perform these operations solely on the client. In this way users do not need to think about servers anymore, though the cloud storage facility needs to guarantee availability of the stored data. To achieve this security solution we use a mechanism called cryptographic access control [35]. In this project we will describe our solution and its related topics, and in the practical part we will provide a prototype to demonstrate our solution.

The rest of this document contains the following chapters:

State of the Art: This chapter starts with introducing cryptography, in which symmetric and asymmetric cryptography is described with special regard to the security and performance of related algorithms. The next part of this chapter describes the cloud computing system with more emphasis on the security requirements and solutions available for cloud storages. Then we introduce “Infinispan” and its relevance in this project. At the end we introduce our security solution, namely cryptographic access control mechanism.

Analysis: At the start of this chapter it is examined that which kinds of access we can have to stored data. Then it is specified that users can grant three levels of access permission to their data, which is the basis for cryptographic access control. Since there are other similar mechanisms, it is specified why cryptographic access control is preferable. The next part of this chapter discusses, in more details, about how we can achieve a good cryptographic protection of data, and here the importance of key distribution mechanism is emphasised. At the end it is examined why the system uses both symmetric and asymmetric encryption in a hybrid way, and how immune the system is to attacks on data confidentiality and integrity.

Design: This chapter contains the design for the system (prototype) on the basis of previous chapters. Here we mention the decisions that we make regarding the structure of the system. We introduce the main components of the system and the interactions between them, among other things we mention the choice of cryptographic algorithms and how cryptographic access control is applied to the chosen cloud storage.

Implementation: Here we describe the implementation details. The implementation of the prototype is based on the design criteria described in the previous chapter. The first part of the chapter describes the overall structure, and the next part contains a more detailed explanation about the most important functionalities of the system.

Evaluation: In this chapter we evaluate the performance and security of the prototype. In the performance evaluation section, we test the speed of different parts of the system, and compare it with similar systems. In the security evaluation section, we will discuss about how secure the cryptographic access control mechanism is against the well-known attacks on the network systems. At the end we suggest further improvements within the topic.

Conclusion: At the end we will have a conclusion on the overall process of this project.

CHAPTER 2

State of the Art

The use of the internet is growing and the speed of data transfer over the internet is also increasing, especially after the use of Fibernet as a data transfer medium. Gradually people have moved to using online data storage, so that they can access their data from anywhere. Moreover, online backup solutions reduce the risk of losing data when the local machine crashes.

One of the possible solutions is data storage in cloud computing. There are many cloud storage solutions ranging from small and simple online data storage like Dropbox and similar services, which provide a simple file system interface, to large and complex cloud storages like Amazon S3, which provides online storage via web services. It is a requirement that data is stored securely in the cloud, i.e., the cloud storage facility should ensure confidentiality, integrity and availability of the stored data. Secure storage in cloud computing may be achieved through cryptographic access control. Since the main mechanism in cryptographic access control is cryptography, we will first examine cryptographic techniques, and then the application of cryptography in existing cloud storage solutions. At the end of this chapter we will introduce cryptographic access control mechanism.

2.1 Cryptography

Cryptography is the most common technique for ensuring a secure communication between two parts in the presence of a third party. If *A* (Alice) and *B* (Bob) send messages to each other, and they do not want others to read or change the content of their messages, then it means that they want to have a secure communication. In this communication, a transmission medium *T* is used, i.e. *A* sends his message to *B* via *T*. A third party, who wants to interfere this communication by accessing/changing the message, is called an intruder *I*. Whenever a message is on its way towards the destination, it is in danger of being accessed by *I*, who can perform the following actions:

1. He can *block* the message, so it never reaches its destination, and thus the availability is violated.
2. He can *intercept* the message, so it is not secret anymore, and thereby the confidentiality is destroyed.
3. He can *change* the content of the message, and by that the integrity is violated.
4. He can *fake* a message and impersonate the sender *A*, and send the message to *B*. This violates also the integrity of the message.

In communications between two parts the security of messages can be exposed to the mentioned four dangers. In cryptography the encryption techniques are used to handle all these security issues. Encryption is actually the most important method to insure security in communications. [1]

2.1.1 Encryption & Decryption

The techniques used in cryptography are encryption and decryption of data. These are also called encoding and decoding, or enciphering and deciphering. Encryption, encode or encipher is a method by which the original text, often called the plaintext, is changed, such that the meaning of the text is hidden, i.e. the plaintext is transformed into an unintelligible string of text, often called the ciphertext. In order to change the ciphertext back to the plaintext, it has to be decrypted, decoded or deciphered.

Figure 1 shows an overview of encryption/decryption procedure.

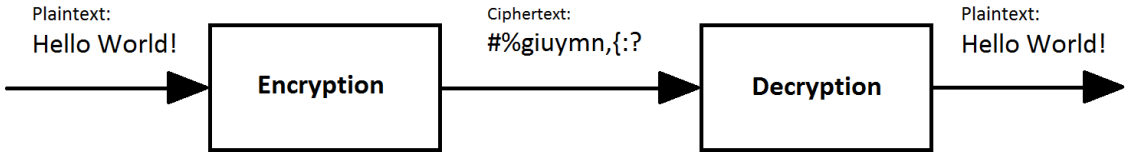

Figure 1: Encryption/Decryption

In the example shown in Figure 1, the plaintext *P* is considered as a sequence of characters *P* = <H,e,l,l,o, ,W,o,r,l,d,!> and in the same way the ciphertext *C* = <#,%,g,i,u,y,m,n,,,{,:,?>. A system that encrypts and decrypts data is called a cryptosystem. If we denote the two processes in a cryptosystem formally, it would be $C = E(P)$ and $P = D(C)$, where *C* is the ciphertext, *P* is the plaintext and *E* and *D* are encryption and decryption algorithms respectively. The cryptosystem is denoted as $P = D(E(P))$, which means that the plaintext *P* is the decryption of encrypted *P*.

In cryptosystems a key *K* is usually used with an algorithm in order to encrypt or decrypt the data. If the same key is used for both encryption and decryption, then the process is called symmetric encryption, and the key is called a symmetric key. In this case the encryption and decryption algorithms are symmetric and they can be considered as reverse operations with regard to each other. The formal notations would be $C = E(K, P)$ and $P = D(K, C)$, and the cryptosystem is denoted as $P = D(K, E(K, P))$.

If the key used for encryption is different from the one used for decryption, then the process is called asymmetric encryption. Here we use two keys, namely an encryption key (often called private key) K_E for encryption and a decryption key (often called public key) K_D for decryption. The formal notations in this case would be $C = E(K_E, P)$ and $P = D(K_D, C)$ and the cryptosystem is accordingly denoted as $P = D(K_D, E(K_E, P))$. Figure 2 and Figure 3 show overviews of the two encryption/decryption methods. [1], [2]

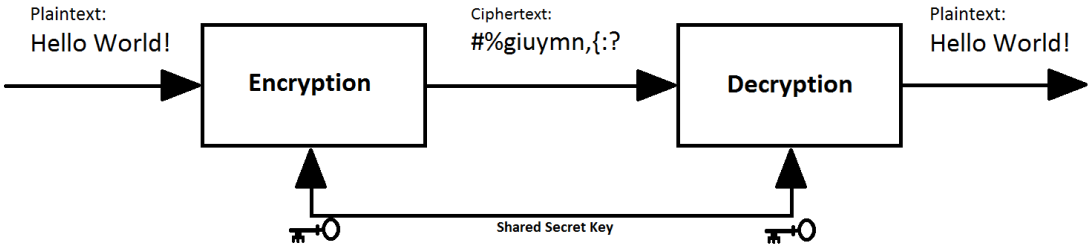

Figure 2: Encryption/decryption using shared secret key (symmetric key)

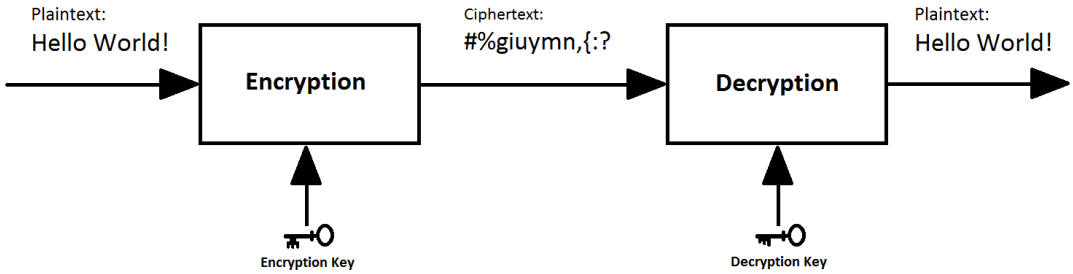

Figure 3: Encryption/decryption using different keys

After introducing the basics of cryptography we will mention some well-known algorithms in the following sections.

2.1.2 Symmetric Algorithms

There are two most famous symmetric algorithms, namely Data Encryption Standard (DES) and Advanced Encryption Standard (AES). Another symmetric algorithm, which is a public domain algorithm, is called Blowfish. We will briefly explain these algorithms and their security issues.

2.1.2.1 Data Encryption Standard (DES)

One of the well-known symmetric algorithms is Data Encryption Standard (DES). It is a block cipher with the block size of 64 bits. It was developed by IBM in early 1970s and in 1976 it was approved as a standard encryption technique in the United States. Firstly it was used in the US, and afterwards it became more and more popular all over the world. DES makes use of substitutions and transpositions on top of each other in 16 cycles in a very complex way. The key length for this algorithm is fixed to 56 bits, which appeared to be too small as the computing recourses became more and more powerful. The main reason for this algorithm to be breakable is its key size. In 1997 a message encrypted with DES was broken for the first time in public by a project called “DESCHALL”, and later on DES keys were broken again and again. With the computer resources and techniques available today it is easier to break a DES key, so DES is considered to be insecure.

However it is worth mentioning that 3DES, also called triple DES, is an approach to make DES more difficult to break. 3DES uses DES three times on each block of data, and in this way the length of the key is increased. It actually uses a “key bunch” containing three DES keys, K_1, K_2 and K_3, which are 56 bits each.

The encryption algorithm works in the following way: *ciphertext* = $E_{K3}(D_{K2}(E_{K1}(plaintext)))$, i.e. encrypt with K_1, then decrypt with K_2, and finally encrypt with K_3.

The decryption process is the reverse of encryption: *plaintext* = $D_{K1}(E_{K2}(D_{K3}(ciphertext)))$, i.e. decrypt with K_3, then encrypt with K_2, and finally decrypt with K_1.

In this way the algorithm would have a good strength, but the drawback of this approach is decrease in performance. [1], [2], [3]

2.1.2.2 Advanced Encryption Standard (AES)

After the weaknesses of DES were accepted, in January 1997 NIST (National Institute of Standards and Technology) announced that they wanted to replace DES, and the new approach would be known as AES (Advanced Encryption Standard). It led to a competition among the open cryptographic community, and during nine months, NIST received fifteen different algorithms from several countries. In 1999, from the received algorithms, five encryption algorithms were chosen. Among these five finalists, NIST choose the algorithm "Rijndael", which was developed by two Dutch cryptographers, Vincent Rijmen and Joan Daemen. This algorithm was approved as a US federal standard in 2001, i.e. it officially became the encryption algorithm for AES. In fact AES is the name of the standard, and Rijndael is the name of the algorithm, but in practice it has been common to refer to the algorithm as AES. [1], [6]

AES is a block cipher with a block size of 128 bits. The key length for AES is not fixed, so it can be 128, 192, 256 and possibly more bits. The encryption techniques such as substitutions and transpositions are mainly used in AES. The same as DES, AES makes use of repeated cycles, which are 10, 12 or 14 cycles (called rounds in AES). In order to achieve perfect confusion and diffusion, every round contains four steps. These steps consist of substitutions, transpositions, shifting the bits and applying exclusive OR to the bits. [1]

AES Modes of Operation

As mentioned before AES is a block cipher, so data is divided in blocks and each block is encrypted. A mode of operation describes the process that is applied to each block of data during encryption/decryption. A mode of operation can be used in any symmetric algorithm. Most of these modes make use of a so called initialization vector (*IV*), which is a block of bits. *IV* is used to randomise the encryption process such that even if the same plaintext is encrypted several times, the corresponding ciphertext would be different each time. As data is divided in blocks, the last block, in most cases, would be shorter in length than the previous blocks, except for the data, which has a length that is a positive multiple of the block size. If the last block is shorter, some of the modes make use of a technique called padding in order to increase the length of the last block to even out the differences in lengths.

There are many modes of operations, but NIST has approved six modes for the confidentiality of data, namely ECB (Electronic Codebook), CBC (Cipher Block Chaining), CFB (Cipher Feedback), OFB (Output Feedback), CTR (Counter), and XTS-AES. [7]

ECB (Electronic Codebook)

ECB is the simplest mode of operation. Every block of data is encrypted separately without the use of any *IV*. This mode does not hide the encryption pattern perfectly, and as a result it does not provide a good distribution and confusion. Because of this weakness, ECB is not recommended to be used. [8]

CBC (Cipher Block Chaining)

In CBC mode the first block of plaintext is exclusive-ORed with an *IV*, and then the result is encrypted to produce the first ciphertext block. Then each of the other blocks of plaintext is exclusive-ORed with each of the preceding ciphertext block, and every result of exclusive-ORing is encrypted to produce ciphertext blocks. In this way, blocks of plaintext is chained with blocks of ciphertext. Every time a plaintext is encrypted, a unique *IV* is used in order to make each ciphertext unique.

In the decryption process the inverse function is used to decrypt the ciphertext blocks, and then the results are exclusive-ORed with the previous ciphertext blocks to produce the corresponding plaintext blocks; except for the result of the first ciphertext block, which is exclusive-ORed with the *IV* to produce the first plaintext block. CBC mode performs the encryption of blocks in a sequential way, and thus it is not possible to perform the operations in parallel. CBC is widely used in cryptosystems, and it is considered to be secure. [8]

CFB (Cipher Feedback)

The operation in CFB mode is similar to CBC. CFB also makes use of an *IV* for encryption/ decryption of the first block. As in CBC, each cipher operation on a block is dependent on the result of the previous cipher operation, i.e. the encryption processes are done sequentially, and thus they cannot be performed in parallel. The differences between these two modes appear when describing the details of each. For instance, one of the differences is whether exclusive-OR operation is used before encrypting a block or after encrypting it. [8]

OFB (Output Feedback)

In OFB mode the *IV* is encrypted to produce “output block 1”, which is then exclusive-ORed with the first plaintext block to get the first ciphertext block. The “output block 1” is then encrypted to produce “output block 2”, which is exclusive-ORed with the second plaintext block to get the second ciphertext block. It continues in this way until we get the whole plaintext encrypted. The decryption process is exactly the same because of the symmetry of the exclusive-OR operation. Similar to the two previously mentioned modes, OFB’s operations on blocks are done sequentially, and consequently they cannot be performed in parallel. [8]

CTR (Counter)

In CTR mode a nonce/*IV* is added with a counter to produce a unique “input block” for encryption. The input block is encrypted and the resulted “output block” is exclusive-ORed with the plaintext block to produce the ciphertext block. In a similar way the decryption process is performed. The difference between CTR mode and OFB mode is that in CTR mode a unique “input block” is used for producing each of the ciphertext blocks, and thus the

operations on blocks are independent of each other. So in contrast to the four previously mentioned modes, in CTR mode operations on blocks can be performed in parallel.

In ECB, CBC and CFB modes a padding scheme must be used for the last block of data if necessary, but in OFB and CTR, padding is not needed at all due to the way exclusive-OR is used in. [8]

XTS-AES

XTS-AES mode is approved by NIST in 2010, and it is developed to be used for confidentiality of data stored on storage devices. This mode is developed for AES algorithm, and it uses a technique called ciphertext stealing. Ciphertext stealing is a method that is used in block cipher modes in order to provide ciphertext without the use of any padding scheme, and thus the ciphertext would not be expanded unnecessarily. This property is important when storing data on a storage device, i.e. the encrypted data must not be larger than the original data. This mode ensures a better confidentiality of stored data than other modes, and it is widely used in encryption software. [9]

Security of AES

Regarding the security of AES, there has not been found any flaws in the algorithm. There has been used two years to analyse this algorithm before it was approved in the US. This is enough to prove the flawlessness of AES. The security of the key is also strong, since the minimum length is twice as long as the key for DES. The key length and the rounds are not limited, so in case we get enough computing resources to come closer to breaking the key, it is possible to increase the key length and also the number of rounds. There is a website [4], where researches and activities about AES are stated. The latest research paper about the AES security is from year 2009 [5]. It is a cryptanalysis of AES with key lengths 192 and 256. Here an attack called “Related Key Boomerang” is used. The paper concludes that the attacks are only of theoretical interest, and is not possible in practice. The data and time complexity are so high that it is unrealistic to be handled in practice with the current technology.

2.1.2.3 Blowfish

Blowfish a public domain encryption algorithm with a block size of 64 bits, and it uses a variable key length. Blowfish was invented by an American cryptographer, Bruce Schneier, and it was introduced in 1993. It is mainly designed for large microprocessors. Its main design criteria are to be fast, compact, simple and having variable security. Its key length can be op to 448 bits long. The same as the two above mentioned algorithms, Blowfish also makes use of cycles/rounds. It consists of 16 rounds, and in each round transpositions and substitutions are used. However it is said to suffer from weak key problem, but with a full 16 rounds implementation, no ways are known to break the security of the algorithm until now. [2], [10]

2.1.2.4 Performance Comparison of Symmetric Algorithms

Figure 4 shows a table, which contains the execution times of popular encryption algorithms on different file sizes. The table is the result of a research stated in a paper [12], which was published in 2005.

Input Size (bytes)	DES	3DES	AES	Blowfish
20,527	2	7	4	2
36,002	4	13	6	3
45,911	5	17	8	4
59,852	7	23	11	6
69,545	9	26	13	7
137,325	17	51	26	14
158,959	20	60	30	16
166,364	21	62	31	17
191,383	24	72	36	19
232,398	30	87	44	24
Average Time	14	42	21	11
Bytes/sec	7,988	2,663	5,320	10,167

Figure 4: A comparison of execution times (in seconds) between symmetric algorithms performed on a P-4, 2.4 GHz machine

The results from the table (Figure 4) show that Blowfish is the best in terms of performance. The performance of AES is much better than 3DES and yet AES is more secure. DES is a bit faster than AES, but as DES is not secure, AES would of course be the best choice of the two.

Blowfish is however the fastest and it is also good in terms of security, but as mentioned before AES is thoroughly analysed and its security is approved by NIST, and it is used to encrypt the most sensitive government data in the US. Though the performance of Blowfish is twice as better than of AES, but in order to feel more secure, people would prefer AES instead.

2.1.3 Asymmetric Algorithms

An asymmetric-key cryptosystem was published in 1976 by Whitfield Diffie and Martin Hellman, known as Diffie–Hellman key exchange. This was the first published practical method for exchanging secret keys in a secure way. But in 1997 it was publicly disclosed that asymmetric cryptography was developed by some researchers at the Government Communications Headquarters (GCHQ) in the UK in 1973.

One of the main reasons why asymmetric cryptography was invented is because symmetric cryptography is not suitable for communication in a big network with a large number of users. There is a key distribution problem. Each user has to have/remember the secret key of all other users, with whom he communicates. A network with *n* users would need a total of

$n*(n - 1)/2$

key pairs that has to be exchanged between users through secure channels. That is a lot of key pairs in a large network system. With asymmetric cryptography we do not face such a big problem, because, for instance, any user can encrypt his message with a single public key, and no one can decrypt it except the other user, who has the corresponding private key. Every user has two keys, a public key, which is freely available and a private key, which is kept secret. But on the other hand asymmetric cryptography is very slow compared to symmetric

cryptography. Symmetric algorithms are around 100 to 1000 times faster than asymmetric algorithms, and thus asymmetric cryptography is not suitable for encryption/decryption of large data. It is mostly used in digital signature, and secret key distribution. [1], [6]

In practice most systems use both symmetric and asymmetric cryptography in a hybrid way, and we will also follow this method in the practical part of this project. Details will be mentioned later, in the corresponding sections.

In the following we will introduce some of the most important asymmetric algorithms.

2.1.3.1 The RSA algorithm

The most commonly used asymmetric algorithm is Rivest-Shamir-Adleman (RSA). It was introduced by its three inventors, Ronald Rivest, Adi Shamir and Leonard Adleman in 1977. It is mostly used in key distribution and digital signature processes. RSA is based on a one-way function in number theory, called "integer factorisation". A one-way function is a function, which is "easy" to compute one way, but "hard" to compute the inverse of it. Here easy and hard should be understood with regard to computational complexity, especially in terms of polynomial time problems. For instance, it is easy to compute the function $f(x) = y$, but it is hard or unfeasible to compute the inverse of f, which is $f^{-1}(y) = x$. [1], [6]

The RSA algorithm contains three steps, namely key generation, encryption and decryption. The key generation process is done by first choosing two random prime numbers, p and q. Then the number n should be computed: $n = pq$.

Thereafter a function $\varphi(n)$ is computed: $\varphi(n) = (p - 1)(q - 1)$.

Moreover an integer e is chosen such that $1 < e < \varphi(n)$.

Finally the number d is computed: $d = e^{-1} \bmod \varphi(n)$, such that: $de \bmod \varphi(n) = 1$, and e and $\varphi(n)$ are co-prime.

As a result (n,e) is the public key, and (n,d) is the private key.

Encrypting a message m is done by computing: $c = m^e \bmod n$, and decrypting the message is done by computing: $m = c^d \bmod n$. [11]

The two keys, private key and public key, can be used interchangeably. It means that a user can decrypt, what has been encrypted with the corresponding public key, and the inverse of that: He can use the private key to encrypt a message, which can only be decrypted by the corresponding public key. [1]

2.1.3.2 Security of RSA Algorithm

There are a lot of elementary attacks on RSA, which are not so powerful, because there has been added improvements to RSA, but one of the most famous ones is on use of common modulus for all users, i.e. not to choose different $n = pq$ for each user. This problem can occur in a system, where a trusted central authority generates public and private keys for the users by using a fixed value for n. In this case user A can factor the modulus n by using his own

exponents, *e* and *d*. Then *A* can use *B*'s public key to recover his private key. The solution is simply not to use the same *n* for all users. This attack is not applicable in systems, where every user generates the pair of keys on their own machines. Here the value of *n* would be different for every user.

Sometimes users want to increase the efficiency of RSA by choosing a small value for the private exponent, *d* or for the public exponent, *e*. Unfortunately this leads to an attack that would break the whole cryptosystem. In order to prevent this attack, *d* must be at least 256 bits long when *n* has a length of 1024 bits. For the public exponent, *e*, the smallest possible value can be 3, but to avoid the attack the value recommended for *e* is: $e = 2^{16} + 1 = 65537$. [11]

However the attack on low public exponent is not as serious as the attack on low private exponent, but it is of course a wise decision to choose both exponents large enough.

The attacks mentioned until now are applied on the structure of the RSA algorithm, but there are other types of attacks, which are pointed at the implementation of RSA. One of these attacks is called timing attack. When a user *A* uses RSA algorithm for encryption/decryption or digital signature, an intruder can determine the private key by measuring the exact time it takes to perform decryption or signature. This attack is applicable on the systems that are connected to a network, for instance, the use of a smart card. The intruder cannot read the content of the smart card, because it is resistant to unauthorised access, but by using timing attack he can determine the private key. One of the possibilities to prevent timing attack is adding some delay to the process, such that the process always takes a fixed amount of time. [11]

Generally there has not been performed any successful attack on the algorithm itself. There have also been attacks in the use of RSA, i.e. protocol attacks. Some guidelines have been developed, and if users follow these guidelines, protocol attack would not be successful either. As a result RSA is reasonably secure. [1], [6], [13]

2.1.3.3 Digital Signature Algorithm (DSA)

Another one-way function is called "discrete logarithm", which is a mathematical function in abstract algebra. The asymmetric algorithm, Diffie-Hellman, which was invented in 1977, is based on discrete logarithm. Diffie-Hellman is basically used for key exchange. Later on, in 1985, Taher Elgamal, an Egyptian American cryptographer, invented and introduced the Elgamal algorithm, which is actually based on the Diffie-Hellman key exchange. Digital Signature Algorithm (DSA) is a variant of Elgamal algorithm with some restrictions in order to make it more secure. DSA was proposed by NIST in 1991, and it was approved as a federal US standard in 1994. RSA can be used both for encryption and digital signatures, but DSA can only be used for digital signatures. Both RSA and DSA are widely used as digital signature algorithms, but RSA is the most widely used. [1], [6]

In the next section we will discuss a bit more about signing and verification, and the use of hash functions in this context.

2.1.4 Digital Signatures

Digital signature is used to insure integrity of data, and it has the same principle as the handwritten signature. The difference is that if a digital signature is implemented properly, it is much more difficult to fabricate than the handwritten signature. To apply a digital signature to messages, asymmetric encryption is used. For instance, Alice wants to send a message to Bob. The message can be encrypted or not, but people usually encrypt the message to insure secrecy. Then she generates a pair of keys, i.e. a private key and a public key. She keeps the private key secret, and publishes the public key. She signs her message using the private key, and then she sends the signed message to Bob. When Bob receives the message, he tries to verify the signature by using the corresponding public key. If the signature is verified successfully, then he is sure that the message is untouched and the actual sender of the message is Alice. If the verification is failed, then Bob knows that either the message has been tampered with, or it is not sent by Alice at all.

In practice, usually the message it self is not signed. A hash function (cf. § 2.1.5) is used to hash the message, and by this a short digest is produced, which is then signed and attached to the message as a signature. Figure 5 shows an overview of the process.

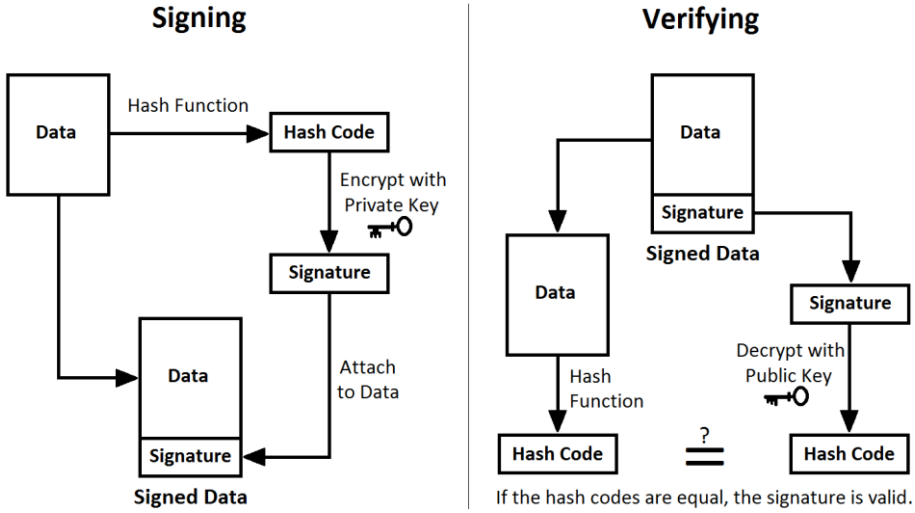

Figure 5: Signing and Verifying data

For the process of signing and verifying we need three algorithms (or steps): A key generation algorithm that, given a security parameter, generates a private key and a public key. Secondly we need a signing algorithm that takes the data and a private key, and outputs a signature. And finally we need a verifying algorithm that takes the data, a public key and a signature, and it outputs either success or failure for the verification. Actually we also need another algorithm, hash function algorithm, in order to generate the hash code.

The most well-known signature schemes that are used with regard to digital signature are RSA signature scheme, DSA, and Elliptic Curve Digital Signature Algorithm (ECDSA). All these three signature schemes contain the necessary algorithms mentioned above, namely key generation, signing and verifying. The DSA and RSA signature schemes are based on DSA and RSA encryption algorithms respectively. We discussed about these two algorithms in the previous

sections. As mentioned about RSA and DSA, ECDSA also makes use of a one-way function, called elliptic curve, which is actually based on the discrete logarithm problem. ECDSA provides the same security as RSA and DSA, though with shorter operands/keys. In order for them to have the same security levels, ECDSA uses operands with lengths of about 160-256 bits, while RSA and DSA use operands that are around 1024-3072 bits long. It means that ECDSA provides shorter signature or ciphertext, and thus it has better performance. It was standardised by American National Standards Institute (ANSI) in the US in 1998. ECDSA is getting more and more popular and in the future it would probably be the most widely used algorithm, but for the time being RSA is number one. [6], [14]

2.1.5 Hash Functions

Hash functions are also widely used, especially in digital signatures. A hash function produces a short and fixed length message digest, which is unique for each message. So it is a great advantage to use a short message digest for digital signature, instead of the whole message, especially if the message is too long.

In contrast to cryptosystems, hash functions are keyless. The main requirements for the security of hash functions are that they must be one-way functions, and they must be collision resistant. A collision occurs when for two different inputs, the hash function gives the same output, for instance, $hash(m_1) = hash(m_2)$.

The oldest hash function algorithm is MD2 (Message Digest Algorithm), which was developed by Ronald Rivest in 1989. MD2 had many security problems, so in 1990 Ronald Rivest developed a new version, called MD4, which appeared not to be secure either. To replace MD4, the inventor developed MD5, which became popular and widely used, but later on, 1994-95, collisions were found for MD5. Another algorithm, Secure Hash Algorithm (SHA-0), was published by NIST in 1993. SHA-0 was appeared to have serious flaws; therefore it was replaced by SHA-1, which is a 160-bit hash function. In order to find collisions for SHA-1, 2^{63} computations have to be performed, which is a really big task for the computing resources we have today, but other security weaknesses have been found in SHA-1. Therefore NIST published a newer version, SHA-2 family, which are SHA-224, SHA-256, SHA-384 and SHA-512. There has not been reported any flaws regarding SHA-2 algorithms, so they are the most secure algorithms until now. For the time being SHA-2 family are the newest and the most widely used hash function algorithms. [6], [15]

2.2 Cloud Computing

Among the biggest commercial providers of cloud computing are Amazon, Google and Microsoft. Many providers, including the three mentioned providers, have their definitions of cloud computing. NIST has also a definition, which sounds: "*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.*" [17]

NIST has introduced a broad definition, which contains most of the definitions given by cloud computing providers. As mentioned in the definition, cloud computing is typically known by five essential characteristics. These characteristics are on-demand self-service, broad network access, customisability, elasticity and per-usage metering and billing.

The four deployment models are private cloud, community cloud, public cloud and hybrid cloud. These deployment models tell about the different usage of cloud computing. For instance a private cloud is obviously “smaller” than a hybrid cloud, because a private cloud would provide fewer services, i.e. only those services that are needed in a single organisation.

The services provided by cloud computing are mainly of three types/layers, namely infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS). These levels can be viewed as a layered architecture, such that services in a higher layer can be composed from the services of the underlying layer. Here IaaS is the lowest layer, and SaaS is the highest.

The following table shows the services and the contents of each service. (The table is best suited to be read from the bottom row upwards.)

Service	Service content
SaaS	This is the top layer of cloud computing systems, and the services provided here can be accessed through user clients, which can be web browsers. Users can use the available software without thinking about where they are installed, and which computing resources they use, and thus it minimises the users’ task with regard to software maintenance. Examples of software services can be: Accounting, customer relationship management (CRM), content management (CM), Office suites, Video processing, etc.
PaaS	This is the programmable layer of cloud computing systems. It contains an environment for developing and deploying software. Users do not need to consider about the computing resources and amount of memory that the software would use. PaaS is the middle layer, and it makes use of the resources provided by IaaS layer. Moreover PaaS layer typically includes operating system, database and web server. In brief, it provides a computing platform for end users. The provided services can be: Programming languages, Application development tools, Database, Web server, etc.
IaaS	IaaS is the bottom layer of cloud computing systems. It provides physical, or more often, virtual resources on demand. These services are mainly computation, storage and communication. Examples of provided services can be: Compute servers, Storage, Networking, Load balancer, etc

In the above table we can see that cloud storage is one of the many facilities cloud computing systems provide, and it belongs to IaaS layer. [16], [17]

As mentioned earlier (Chapter 1), our main concern in this project is providing security for cloud storage. We will, in the following, mention the well-known cloud storage providers, and compare their security solutions.

2.3 Cloud Storage

Cloud storage is an online virtual distributed storage provided by cloud computing vendors. Cloud storage services can be accessed via a web service interface, or a web based user-interface. One of the advantages is its elasticity. Customers get the storage they need, and they only pay for their usage. By using cloud storages, small organisations save the complexity and cost of installing their own storage devices. The same as cloud computing, cloud storage has also the properties of being agile, scalable, elastic and multi-tenant.

2.3.1 Amazon S3

One of the well-known cloud storages is provided by Amazon, called Amazon Simple Storage Service (Amazon S3). It provides data storage and retrieval via web services interfaces, such as REST, SOAP and BitTorrent. Amazon S3 is a key/value store, and it is suitable for storing large files, i.e. up to 5 terabytes of data. For storing smaller data, it is more suitable to use Amazon's other data storage, called SimpleDB.

For managing files in large data stores, like cloud storages, relational database systems are not applicable. It would get very complex and almost impossible to use MySQL, for instance, for managing data. Therefore Amazon S3, SimpleDB and also other cloud storages usually use NoSQL database solutions.

To reduce complexity, Amazon S3 has purposely minimal functionality, so data can only be written, read and deleted. Every object/file is stored in a bucket and retrieved via a unique key. It supports storing 1 byte to 5 terabytes of data, and the number of files to be stored is unlimited. [18], [19], [21]

2.3.1.1 Security of Amazon S3

Amazon S3 provides security mechanisms, by which a user controls who can access his stored data, and how, when and where the data can be accessed. In order to achieve this security, Amazon S3 provides four types of access control mechanisms:

- "Identity and Access Management (IAM) policies" make it possible to create multiple users under a single AWS (Amazon Web Services) account. By using this mechanism, each user can control other user's access to his buckets or files.
- "Access Control Lists (ACLs)" make it possible for a user to grant specific permissions on every file in a selective way.
- "Bucket policies" are used to grant or deny permissions on some or all of objects within a bucket.
- "Query string authentication" is used to share objects through URLs.

Besides these mechanisms, users can store/retrieve data by using SSL encryption via HTTPS protocol. Amazon S3 also provides encryption of data by a mechanism called Server Side Encryption (SSE). By using SSE, data are encrypted during the upload process and decrypted when downloaded. Users can request encrypted storage, and Amazon S3 SSE handles all encryption, decryption and key management processes. When a user *PUT*s a file and request encryption, the server generates a unique key, encrypts the file using the key, and then encrypts the key using a master key. For ensuring more protection, keys are stored in hosts that are distinct from those, where the data are stored. The decryption process is also performed on the server, so when a user *GET*s his encrypted data, the server fetches and decrypts the key, and then uses it to decrypt the data. The encryption is done by using AES-256. [19],[20]

All of the above mentioned access control mechanisms are server centric, and users have no choice other than trusting Amazon S3.

2.3.2 Google Cloud Storage

Google Cloud Storage is a service for developers to write and read data in Google's cloud. Besides data storage, users are provided with direct access to Google's networking infrastructure, authentication and sharing mechanisms. Google Cloud Storage is accessible via its REST API or by using other tools provided by Google.

Google Cloud Storage provides high capacity and scalability, i.e. it supports storing terabytes of files and large number of buckets per account. It also provides strong data consistency, which means that after uploading your data successfully, you can immediately access, delete or get its metadata. For non-developer users, who require fewer services, Google offers another data storage, called Google Docs, which supports storing up to 1 GB of files. [23]

Google Cloud Storage uses ACLs for controlling access to the objects and buckets. Every time a user requests to perform an action on an object, the ACL belonging to that object determines whether the requested action should be allowed or denied. [24]

2.3.3 Dropbox

Dropbox is a file hosting service that allows users to store and share their data across the internet. It makes use of file synchronisation for sharing files and folders between users' devices. It was founded by two MIT students, Drew Houston and Arash Ferdowsi in 2007, and now it has more than 50 million users across the world. Users can get 2GB of free storage, and up to 1TB of paid storage. Dropbox provides user clients for many operating systems on desktop machines, such as Microsoft Windows, Mac OS X and Linux, and also on mobile devices, such as Android, Windows Phone 7, iPhone, iPad, WebOS and BlackBerry. However users can also access their data through a web-based client when no local clients are installed. [25],[26],[27]

Dropbox can be used as data storage, but the main focus is file sharing. If a Dropbox client is installed on users' devices, besides storing the shared data on the server side, these data are also stored on shared users' local devices. Whenever a user modifies the shared data on his

client, the shared data on the server and on all the other shared clients are also updated (when syncing) according to the performed modification. Dropbox supports revision control mechanism, so users can go back and restore old versions of their files. It keeps changes for the last 30 days as default, but they offer a paid option for unlimited version history. In order to economise on bandwidth and time, the version history makes use of delta encoding, i.e. when modifying a file, only the modified parts of the file are uploaded. [28]

Dropbox makes use of Amazon's cloud storage, namely Amazon S3, as their data storage. However the founder of Dropbox, Drew Houston, has mentioned in an interview[29] that they may build their own data centre in the future. They claim that Dropbox has a solid security for users' data, and they use the same security solutions as banks. For synchronisation, Dropbox uses SSL file transfer protocol, and the stored data are encrypted at the server side using AES-256 encryption. [30]

2.3.4 Cloud Storage Security Requirements

In the process of storing data to the cloud, and retrieving data back from the cloud, there are mainly three elements that are involved, namely the client, the server and the communication between them. In order for the data to have the necessary security, all three elements must have a solid security. For the client, it is mostly every user's responsibility to make sure that no unauthorised party can access his machine. When talking about security for cloud storage, it is the security for the two remaining elements that is our main concern. On the server side, data must have confidentiality, integrity and availability. Confidentiality and integrity of data can be ensured both on the server side and on the client side. At the end of this chapter, when introducing the cryptographic access control mechanism, we will discuss about differences between server side and client side security solutions. The availability of data can only be ensured on the server side, so it is the responsibility of the server to make sure that data is always available for retrieval.

Last but not least, the communication between client and server must be performed through a secure channel, i.e. the data must have confidentiality and integrity during its transfer between server and client. One of the ways to achieve secure communication is having a cryptographic protocol, such as SSL.

2.3.5 Cloud Storage Security Solutions

The two mentioned commercial cloud storage providers, Amazon and Google, are large and well-known providers in the market. Dropbox, as being a cloud storage provider and file sharing service, is also getting more and more popular. Moreover there are many other cloud storage providers that use various security mechanisms including cryptography. In the following we will mention some of the security solutions that have been suggested or used, and a comparison between these approaches will also be mentioned.

For some types of data, for instance the data in a digital library, the integrity of data is the main concern, but the confidentiality of data is not relevant. In this case it is important to have a fast mechanism and not so complex communication to verify the integrity of data. For achieving this goal, two approaches are proposed, which are stated in a research work [31].

One is called Proof of Retrievability Schemes (POR), which is a challenge-response protocol used by a cloud storage provider in order to show the client that his data is retrievable without any loss or corruption. The second approach is called Provable Data Possession Schemes (PDP), which is also a challenge-response protocol, but it is weaker than POR, because it does not guarantee the retrievability of data. These two approaches are reasonably fast processes, because the data retrievability is verified without re-downloading the data. [31]

To many other types of users, confidentiality of their data is of much importance. Therefore many of the commercial cloud storage providers give confidentiality solutions to the clients. The table in Figure 6 is taken from a paper [31], which contains security comparisons between popular commercial cloud storage providers. (The last row containing information about Dropbox is not stated in the paper. The information is taken from Dropbox's website, and it has been added to the table).

Cloud Solutions	Own Data Center	Sync	Secure transmission	Data encryption	Multifactor authentication	Free space
Amazon S3	Yes	-	-	No	No	-
Azure	Yes	-	-	No	No	-
Carbonite	Yes	No	Yes	Blowfish-128 Symmetric at client side	Yes	-
Mozy	Yes	Yes*	Yes	AES-256 or Blowfish-448 Symmetric at client side	Yes	2GB
Spideroak	Yes	Yes	Yes	AES-256 Symmetric at client side	No	2GB
SugarSync	Yes	Yes	Yes	AES-128	No	5GB
Ubuntu One	No	Yes	Yes	No	No	2GB
Wuala	Yes	Yes	Yes	AES-128 Symmetric at client side	No	1GB
Dropbox	No	Yes	Yes	AES-256 Symmetric at server side	No	2GB

*At additional cost, per machine.

Figure 6: A comparison between cloud storage solutions

The table (Figure 6) also compares other features of cloud storages, like whether or not they have their own data centres, whether they support syncing between multiple computers or not, and etc., but the column "Data encryption" is relevant here. It shows that six of the mentioned cloud solutions support data confidentiality in form of symmetric data encryption, and four of them support this mechanism on the client side. (However Amazon S3 provides SSE as mentioned before, but since SSE is a new addition to Amazon S3, it is not mentioned in the table.) We can see that ensuring integrity of data is missing in these cloud solutions. We described earlier the two approaches, POR and PDP, for verifying data integrity, but as

mentioned, the two approaches are proofs for showing retrievability of the data without downloading. It is suitable for systems with large data that does not need to be secret. Once the integrity of the whole data is ensured, one can read some amount of the data that he needs.

In the following we will discuss briefly about Infinispan, which is a new and open source approach for building cloud storages.

2.3.6 Infinispan

We know Amazon, Google and other commercial cloud computing providers, who offer cloud storage solutions, but Infinispan is a bit different. It is an open source in-memory data grid platform written in Java. It is quite new and still under development. It can be used to build an online data storage for the cloud. It uses some concepts from Amazon Dynamo for storing and managing data. The same as in Amazon Dynamo, Infinispan makes use of a key-value structured data storage system, and thus it provides high availability of data.

Infinispan is extremely scalable and highly available data grid. It is primarily in-memory data grid, i.e. it uses caches to provide memory. It works in such a way that a number of instances of Infinispan can be created in different machines, and these instances can be connected with each other forming a peer-to-peer network of nodes, which can actually be considered as distributed cache-nodes. Now we can run any application and connect it to this distributed data grid, so that our application uses it as a memory, or we can use the data grid as a data store. The bigger the grid is created, the more the memory would be available. For instance, if we create a grid containing 50 cache-nodes of 2 GB each, we would have a data grid that can provide a total of 100 GB memory. Data would be stored evenly in the grid, because Infinispan divides the data in chunks before storing it. [33]

Infinispan is not only an in-memory data grid, but it can also be configured with cache stores in order to store data in a persistent location on the disk. It is a cloud-ready data store, which means that it can be used to create a big data grid and “install” it in the IaaS layer of a cloud computing system, where it will work as a cloud storage system.

There are two usage modes available for Infinispan, namely embedded mode and client-server mode. Figure 7 and Figure 8 show the two modes of interaction. [32], [34]

Infinispan is primarily a peer-to-peer system, which means that instances of Infinispan discover each other, and share data with each other by using peer-to-peer system.

In Figure 7 the embedded mode architecture is shown. In the embedded mode we have our application running in a JVM (Java Virtual Machine). Our application starts an instance of Infinispan within the same JVM. We can actually start a couple of these JVMs. The Infinispan nodes discover each other, and start sharing data. If our application stores some data in one of the Infinispan nodes, it will be available in other nodes too. So if one node dies, the data would still be available in the other.

The embedded mode is a low level type of usage, but a slightly high level and more useful usage is the client/server mode (Figure 8). In this mode, each of the Infinispan instances still runs in a separate JVM, and they discover each other by using peer-to-peer system. Moreover every node opens up a socket and listens to it, and our application can talk to the grid over the network socket. In the client/server mode our data grid can be treated as a remote data store. Our application does not need to be started in a JVM; actually it does not need to be a java application at all, because as long as it speaks one of the supported protocols, it can be connected to the data grid and use the advantages of it.

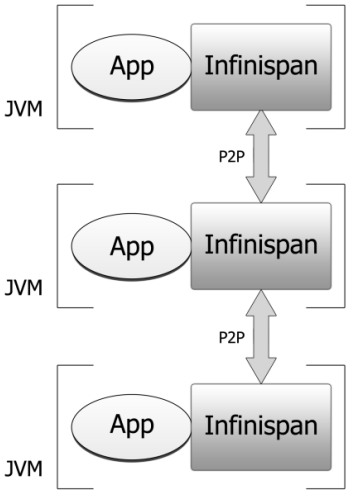

Figure 7: Peer to Peer embedded mode

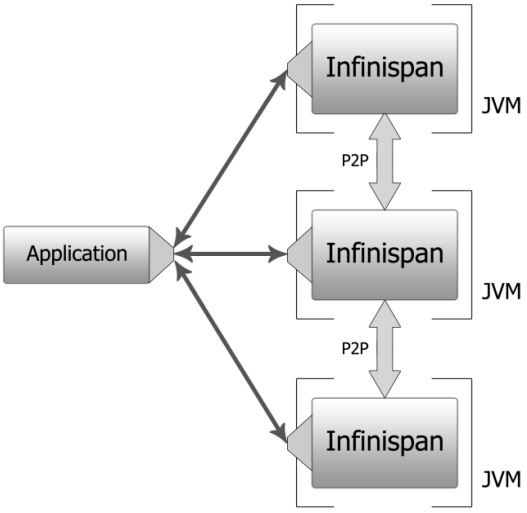

Figure 8: Client/Server mode

The protocols that are supported in the client/server mode are REST, memcached and hot rod. The REST-based protocols are popular in cloud computing systems, and it is easy to manage, but it is a bit slow. Memcached is a protocol, which is both fast and popular, and a lot of client libraries are available in many programming languages. Hot rod is a wire protocol, which is built specifically for Infinispan by the founder of Infinispan. Hot rod is an extension of memcached. One of the extensions is that hot rod is a two way protocol, while memchached is only a one way protocol, i.e. only clients can talk to the servers and get results. [32], [34]

As Infinispan is an open source data grid, and it works the same as cloud storages available in the market, we will use it in this project as our case study. We will base our access control mechanism on this platform.

2.4 Cryptographic Access Control

As mentioned earlier, most of the well-known cloud storages like Amazon S3 and Google Cloud Storage provide a server centric access control mechanism, in which users have to trust the server. Some of the cloud storages provide cryptography at the client side, which ensures confidentiality of the data, but the integrity of the data, which is also of big importance, is missing. In most of these server-centric solutions, users need to configure the access control mechanism on the server to exchange data, which may require every one to have an account. By using cryptographic access control mechanism, the servers are not involved directly in the

access control decision, and thus users do not need to have accounts. So besides providing more security, users would also have more freedom in storing and exchanging data.

In cryptographic access control, the main mechanism used is cryptography. The data is encrypted locally on the client before it is stored in the cloud. A client has to retrieve the encrypted data to his local machine and decrypt it locally before he can get access to the content of the data.

In this process, two kinds of cryptographic mechanisms are used. Firstly, for data encryption and decryption, symmetric cryptography is used. A client encrypts the data locally by using a symmetric algorithm and then stores the encrypted data in the cloud. If this client or any other client, who is authorised, wants to access the data, he has to use the corresponding key to decrypt the data locally.

Secondly, besides encryption and decryption of data, two other important operations are also performed on the data, namely signing and verification. In these operations asymmetric cryptography is used, where two keys are needed, namely private key and public key. After a client encrypts the data, he generates a signature, which is then attached to the data. The public key is given to the server, and it is used by the server to verify the future updates to the data. This means that when an authorised client makes a change to the data and signs it, the server verifies the signature after the upload of data. If the verification succeeds, the server allows the update to the data, otherwise the update is rejected, and the previous version of the data is kept instead. A client can either trust the server's verification, or he can also use the public key to verify the signature locally.

By using cryptographic access control, confidentiality and integrity of the data is ensured. The data is encrypted locally, and then stored in the cloud. So anyone can get the encrypted data, but no one can read the content of the data, except the authorised clients, who are provided with the key(s). This ensures confidentiality of the data.

On the other hand the encrypted data is signed. When an authorised client gets the data, he tries to verify the signature. If the verification of the signature fails, he knows that someone has tampered with the data, but if the signature verifies successfully, he knows that the data is untouched. This ensures the integrity of the data.

In cryptographic access control mechanism, clients can have different access permissions to the stored data. If a client possesses the public key and the symmetric key, he has read access to the stored data, because he can verify the signature of the data, and then decrypt it. If a client possesses the public key, the private key and the symmetric key, then he has both read and write access to the stored data, because in this case he can decrypt the data and modify it, and then he can encrypt the data and sign it.

It is important to mention that cryptographic access control is client-centric, i.e. as mentioned earlier cryptographic operations on data are performed locally on the client's machine. It provides more security and trust for the users than the server-centric solutions, where servers are involved in access control decision. So it is assumed that every authorised client is provided

with the necessary security, such that no one can access the keys that are stored locally. If a malicious intruder accesses an authorised client and gets holds of the keys, then the whole mechanism is violated. [35]

2.5 Summary

In this chapter we started with mentioning how important security is when storing data in the cloud. Since cryptography is a very important factor in ensuring data security, we discussed about symmetric and asymmetric cryptography, as well as the corresponding algorithms. We mentioned the security, performance and usage of these algorithms. As a result we found out that for encryption/decryption of data, AES is the standard algorithm to be used, and for the data signature and verification, RSA scheme with the hash function SHA-2 family would be the best choice.

As our main concern is to apply the cryptographic access control mechanism to cloud storage, we touched on cloud computing with especial regard to cloud storage solutions. We mentioned the most well-known commercial cloud storage providers, and compared their security solutions with each other. Regarding the confidentiality of data, some of the providers have the same mechanism used in cryptographic access control, i.e. symmetric cryptography. For the integrity of data there are other approaches available, which have different usages than the integrity mechanism in cryptographic access control.

In contrast to the mentioned mechanisms used in cloud storages, cryptographic access control is client centric. A user has obviously more control on his local machine, and thus the data would have more confidentiality if they are encrypted locally. However Amazon offers a library called “Amazon S3 Encryption Client”, which makes it possible to encrypt data locally, but every user has to implement the whole mechanism by using the given library. It is a complex and time consuming task for most of the users, who are not familiar with the technology, and besides that, most users prefer to use a ready-made system. As a result Amazon S3 does not provide cryptographic access control.

As mentioned, Google Cloud Storage uses ACLs for access control to the objects. The same as in Amazon S3, Google Cloud Storage does not provide cryptographic access control.

Then we described Infinispan. Since Infinispan is similar to cloud storage solutions, and moreover it is an open source project, we will apply our security solution to it in order to demonstrate how the confidentiality and integrity of data can be ensured in the cloud.

At the end of this chapter, we described our security solution, namely cryptographic access control, where we mentioned that this mechanism is client centric and thus provides more security and control for the users. In contrast to server-centric solutions, here servers are not involved in the access control mechanism. Users do not need to create accounts in order to store and exchange their data.

CHAPTER 3

Analysis

There are various file systems available for different operating systems, but the main principle is almost the same. Generally, the basic operations that can be performed on files in a file system, are reading a file, writing to a file, deleting an existing file and creating a new file. So a user can perform read/write operations on his files through a file system, but should it be possible for other users to run the same actions on the user's files? When this question is raised, the term access control comes to our mind. Some kind of access control mechanism must be available, so that a user can assign access limitations to his data. In such a system, a user would be able to grant trusted users read access permission, or both read and write access permission to his data. Besides this, when a user reads his file, he must be sure that the content of his data has not been modified by some unauthorised users, which means that the integrity of data must be guaranteed. The same requirements are applicable when users store their data to online storages. To put it briefly, we need security for stored data.

The term security for data has always been an important issue, and its degree of importance depends on how secret data are. Also in the old days, when technology did not exist, people had secret data, and they also controlled the access to their data in some degree. In the digital world, when data are stored to servers; according to the degree of secrecy, the term access control to data is defined more clearly. There are many types of access control mechanisms in different systems, but the main idea is controlling read and write access, which fall under confidentiality, and besides that, ensuring data integrity. Finally it is also important that the stored data is always available, but it is solely the task of the server to provide availability for the data. To sum up, we can have three levels of access permission to the stored data:

1. Verifying the integrity of the stored data.
2. Verification and read access to the stored data.
3. Verification, read and write access to the stored data.

To achieve a system that covers the above specifications for access control, we can use cryptography. By using cryptography we can perform the operations locally on the client, which additionally increases the security level. For data confidentiality, symmetric encryption can be used, and for data integrity, asymmetric encryption can be used. In one of the coming sections (section 3.3), we would discuss about why it would be reasonable to use both encryption mechanisms.

By having one or more of the three keys, public, private and symmetric key, a user can have different access permissions to the stored data. The verification of data is actually always possible, because the public key can be freely available. This does not violate the confidentiality or integrity of the data, because the only action one can perform is to verify the signature of the data. To have read access, one must have the symmetric key, and for read/write access, both symmetric and private key must be available. The owner of the stored data has of course all three access permissions, because he has created all three keys, but by distributing one or more of the keys, he can grant different access permission levels to other users. So this mechanism is best suited to be used in constructing the basis for the cryptographic access control system.

As discussed earlier (section 2.4) and specified above, we know that the main difference between the specified mechanism and other available mechanisms is that here the security operations can be performed locally on the client's machine. In other words, the available solutions are server centric, while our solution is client centric. In this way, servers are kept away from the process as much as possible, and therefore users do not need to create accounts. As a result the system would be more trustful, and also users would have more control on their data.

On the basis of the mentioned access permission mechanism, we will discuss about the needs that we have in order to create a system with cryptographic protection of data. Since exchanging keys is one of the most important properties in this system, we will also discuss how key exchange can be performed. At the end we will touch upon robustness of our system.

3.1 Cryptographic Protection of Data

The cloud storage providers claim that they supply the stored data with necessary security solutions. The truth is that users would not feel secure, because they have to trust the servers, while they do not exactly know what is going on inside the servers. Users lose their trusts even more, especially when they hear some news about, or become a victim of a security glitch, such as the security breach that happened in Dropbox in June 19th 2011. It was an authentication error that lasted for almost four hours, which made it possible to access all users' data without password. [38]

The cryptographic access control mechanism must be an approach that should make it possible for users to avoid thinking about server security. The only serious task the server has is to provide data availability. So in order to provide the needed security for the stored data in the cloud, we must ensure that the data is provided with cryptographic protection in terms of client centric solution. If we try to explain, in more details, what we need in our system, the following items are necessary to consider:

- For confidentiality of data, encryption of the data at the client side is needed. The encryption must be performed just before storing/uploading it to the cloud. Also the data needs to be decrypted after retrieving it from the cloud.

- For integrity of the data, using digital signature would be a wise choice on the client side. The signing process is best suited to be performed just after the data is encrypted. Also the signature has to be verified on the client side, just after it is retrieved from the cloud.
- In the case of sharing data between clients, it is necessary for the shared clients to have the needed keys in order to have access to the data. As a result a key exchange mechanism between clients is needed.
- Apart from the public key, all cryptographic keys are as sensitive as the data, because these keys are used to get access to the data. As a result the key exchange mechanism also needs to have the same security as the actual data. (More details about key exchange in the next section.)
- In the case of data sharing, different levels of access permission can be granted. For instance, a client *A* can give another client *B* read access to his data. In this case, *B* needs the symmetric key and the public key belonging to the data. If *A* gives *B* both read and write access to his data, then *B* needs all the three keys belonging to the data.

All the above mentioned properties must be present in the cryptographic access control mechanism in order to provide the stored data with a solid security, and also to make the file sharing mechanism as secure as possible.

3.2 Key Exchange

For every file to be stored to the cloud, a user needs to generate the three keys: public, private and symmetric key. This key set can be called “key files”, such that for every file there are corresponding key files. Besides these key files that the user himself has generated, he can also possess some other key files that he has got from other users in order to have access permission to other users’ files. For example, *B* has given *A* read access to a file *X* by transferring the public- and symmetric key belonging to the file *X*. In this way, file sharing between users can take place.

The presence of a key exchange mechanism is obviously necessary in the system, because in order for the users to share their data, they must exchange their keys with each other. Also it is very important that the mechanism of key distribution must be performed in a secure way.

3.2.1 Granting Access Permission by Exchanging Keys

In a situation, where two users want to share data with each other, i.e. *A* shares his data with *B*, *A* has to send the key files belonging to the shared data *X*, to *B*. The key files are some or all of the keys belonging to the file *X*, depending on which kind of access permission *A* wants to grant to *B*.

We know that the key files must be exchanged in a secure way. One possibility is simply sending them via an encrypted email. Another way is sending them through a secure channel, such as SSL protocol, where communication is done by using an encrypted protocol. And finally RSA key exchange can be used, where *A* uses *B*’s public key to encrypt the key files, and then sends the encrypted key files to *B*, who can decrypt them by using his private key. In this case the encrypted key files can be sent via email, or it can simply be uploaded to the same place as other files are stored, i.e. the cloud storage. Since it is only *B*, who can decrypt the key files,

there is no problem in publishing them. Now if *A* chooses the later solution, i.e. uploading the encrypted key files to the cloud storage, it is necessary to indicate that he has granted access permission to *B*, since there are many other pairs of users that also grant access permissions to each other by publishing their encrypted key files. So in order for *A* to indicate this, he can use the hash value of *B*'s public key and attach it to the encrypted key files. Then *B* can just perform a search on the hash value of his public key in order to discover the access permission that he has got recently. The reason why *A* uses the hash value of the public key, instead of the very public key, is to avoid directly revealing of the access permission he has given to *B*.

The situation, where many users share data with one user, is fundamentally the same as two users sharing data with each other, and the key exchange process is obviously the same as mentioned above.

3.2.2 Access Permission by Using Key Rings

There is another situation, where the user *A* shares his data with many other users than *B*. In this case it would be more efficient to build a structured system for key exchange in order to reduce the complexities. One of the methods is making use of key rings.

A key ring is a file, which contains key files. Key rings can contain key files, which belong to the stored data and/or other key rings. Key rings must be encrypted, signed and stored to the cloud, and they can be treated the same as other stored files in the cloud. The content of a key ring is just a list of keys. In order to read or update a key ring, it is necessary to have the corresponding key files.

When a user *A* wants to share his data with other users, he puts the corresponding key files on a shared key ring. It can be an existing key ring that has been shared between these users, and *A* has read access to, or a new key ring can be created. If *A* needs to create a new key ring, then it is of course necessary to distribute the key files belonging to the key ring to those users, who should have access to the key ring. The key files belonging to the key rings can be exchanged in one of the ways discussed earlier, namely an encrypted email, an encrypted protocol (SSL), or RSA key exchange mechanism.

If a user wants to give read access to a group of users, but right access to a subset of this group, then he has to make use of two different key rings. Before he distributes each of the two corresponding key files to these two groups, such that every group gets its relevant key files, he obviously has to encrypt the key files with different public keys. [40]

Every user can also have his personal key ring, where he puts all his key files, which can belong to the stored data, or other key rings. This personal key ring, as its name indicates, is of course personal, and except its owner, no one has access to it.

The method of using key rings is also used in other systems. In Figure 9 an example of key rings is shown. The ellipses represent the key rings, and the squares represent key files. An arrow, which starts from a key ring, indicates that the key set to the object that is pointed to, is contained in the corresponding key ring.

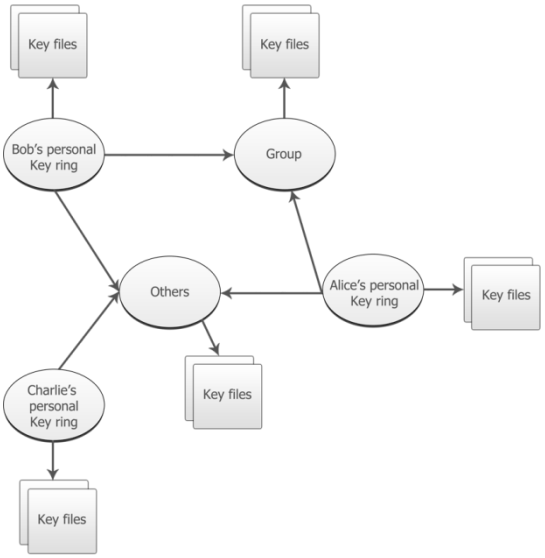

Figure 9: An example of using key rings in a Discretionary Access Control system (DAC) [40]

We can see in Figure 9 that Bob and Alice have access to the "Group" key ring, but Charlie does not have access to this key ring. Bob, Alice and Charlie have all access to the "Others" key ring. This type of granting access control to users is used in Discretionary Access Control system (DAC), and this access control mechanism is also used in most UNIX systems. [40]

3.3 Robustness of Cryptographic Access Control

As mentioned earlier, cryptographic access control insures both confidentiality and integrity of data by using both symmetric and asymmetric cryptography in a hybrid way. In the following sections we will briefly discuss why we use both of the encryption methods. We will also examine strengths and weaknesses of cryptographic access control with regard to confidentiality and integrity of data.

3.3.1 Confidentiality

If we compare the most widely used symmetric and asymmetric algorithms with each other, we know that they are almost equally secure. In terms of key distribution in a large network of users, asymmetric cryptography has less complexity, because it has a pair of keys, and only one of these keys has to be kept secret, while the other is publicly available. So a user only needs to distribute his private key. In symmetric cryptography, key distribution is more complex, because there is one key that has to be exchanged between all users.

The question is, why not simply use asymmetric encryption for confidentiality of data? To answer this question we should mention that asymmetric encryption is very slow, i.e. about 100 to 1000 times slower than symmetric encryption. The data stored by users can often be very large, and thus it would be quite inefficient to use asymmetric encryption. Also when we use asymmetric encryption for digital signature, we do not use the actual data. Instead of that we generate a hash value of the data, which has a fixed and short length for all types of data. So it is obviously more reasonable to use symmetric encryption for data confidentiality. We should not worry about key distribution issues, because cryptographic access control mechanism is mostly suited for private data stored in the cloud by private users, and thus

there will not be a lot of users, who share their data. As a result there would be a rather small network of users, where key distribution would not be such a complex issue.

For ensuring data confidentiality, AES is used. The reason why we use AES, is because its security is very solid. By using a key with a length of 128 bits or more, the algorithm is very powerful and there is not reported any successful attacks against it. The symmetric key is kept secret since it is stored and used on client side. The distribution of the key is also done in a secure way as mentioned in the section “Key Exchange”. As a result there are not any serious flaws for data confidentiality.

3.3.2 Integrity

In order to clarify what integrity of data exactly means in cryptographic access control, we will mention the definition of integrity in this context. It is worth specifying that in cryptographic access control mechanism, ensuring the integrity of data does not mean that data must be protected against being lost, corrupted or changed. The loss or corruption of data could happen accidentally or on purpose. Avoiding this problem is mostly the responsibility of the server, and a part of it belongs to data availability.

Assume *A* sends some data to *B*. Integrity of data means to ensure two things:

1. *B* must know whether or not the data originates from *A*. For instance if an intruder pretends to be *A* and sends some data to *B*, he would know that it is not coming from *A*.
2. *B* must know whether or not the data has been tampered with by an intruder on its way.

In cryptographic access control mechanism we use RSA digital signature scheme for data integrity, which is the most widely used scheme. The reason why we use this algorithm is that it is a well-tested algorithm, and thus its security is strong. As mentioned in Chapter 2, section 2.1.3, through twenty years after invention of RSA algorithm, a lot of attacks, both on the implementation of RSA and on the actual algorithm, have been developed to find weaknesses of RSA. By following the protection guidelines, none of these attacks would be successful. On the other hand some of these attacks are not applicable in cryptographic access control. For instance timing attack (cf. § 2.1.3.2) is not applicable, because the whole process of signing the data is performed on a local client. (The local machine must of course be well protected against unauthorised entrance.) The common modulus attack (cf. § 2.1.3.2) is not applicable either, because every user generates the pair of keys on their own machines, and thus the value of *n* in RSA algorithm would be different for every user. As a result there are not any serious flaws for data integrity.

3.4 Summary

In this chapter we started with specifying which kinds of actions can be performed on the stored data. We clarified that the actions are verification of the data, reading the data and writing to the data. This would result in three levels of access permission, namely verification, read and write access that users can grant to each other. In order to suggest a better security

solution compared with the available solutions in the cloud storage market, we specified that all the operations must be performed on the client side. Then we specified that the cryptographic access control mechanism is based on the mentioned solution. On the basis of the mentioned access permission mechanism, we discussed about the requirements that a system must have in order to protect data in a cryptographic way. We found out that in such a system, granting access permissions can best be done by exchanging keys. Therefore we examined the ways in which key distribution mechanism can be based on, and we found out that the use of key rings would solve many complexities.

In the next part of this chapter we discussed about choices of cryptographic methods for confidentiality and integrity of data. We mentioned that symmetric cryptography is suitable to be used for data confidentiality, because it has much better performance than asymmetric cryptography. Asymmetric cryptography is best suited for data signature. Since hash value of data is used for signing, the low performance is of less importance.

In the last part we examined the strengths and weaknesses of data confidentiality and integrity. We mentioned that attacks on AES algorithm have not been successful. There have also been many serious attacks on RSA algorithm, both on its structure and its implementation. By following the guidelines, RSA is also a powerful algorithm and is resistant to breakage. Therefore it would be natural to make use of AES for data confidentiality, and RSA for data integrity in the cryptographic access control mechanism.

The good thing about cryptographic access control is that it has moved the protection processes away from the network as much as possible, i.e. it is a client centric solution, and as a result it is immune to many attacks. But care must be taken when choosing and implementing the cryptographic algorithms so that other attacks on the structure and implementation of the algorithms do not get success.

CHAPTER 4

Design

This chapter describes the overall design of the system, which is based on the results of the knowledge that we obtained in chapter 2 (State of the Art) and the discussion in chapter 3 (Analysis). The purpose of this system is to put our security solution, namely cryptographic access control mechanism, into practice. As desired, our solution is intended to be applied to a cloud storage system, so we have chosen Infinispan (described in § 2.3.6) as cloud storage facility, mainly because it is an open source approach for building cloud storage systems. Apart from this project, a 3-week course (Appendix B: Introduction to Infinispan Data Grid) is completed, where a primitive prototype is implemented by using the Infinispan API. This prototype consists of Infinispan data grid, and a simple file system. Here we will use this Infinispan data grid as our cloud storage facility, and we will apply our security solution to it.

The main elements in our system would be:

- The infinispan data grid.
- Cryptography, both symmetric and asymmetric encryption.
- File sharing by using key exchange mechanism.
- A graphical user interface (GUI).

4.1 Overview of the System

Here we give an overview of the system by illustrating the most important elements. As we know, encryption and digital signature are the main functionalities of the system. Since the system should also be able to allow users to share their data with each other, access permission management is also of great importance.

In Figure 10 we can see the process of storing and retrieving data, and the components that are involved in the process. The encryption and signing mechanism must be performed on the client before storing the data to Infinispan data grid. The encryption process must generate the symmetric key, and use it for data encryption. In a similar way, the signing process must generate the key pair, namely public and private key, and use the private key to sign the data. Then the encrypted and signed data can be stored to Infinispan data grid by using the file system. The procedure of retrieving data would be the reverse of storing, i.e. after retrieving data from the Infinispan through the file system, its signature would be verified, and finally it would be decrypted. The previously generated public key and symmetric key would be used by verification and decryption processes respectively.

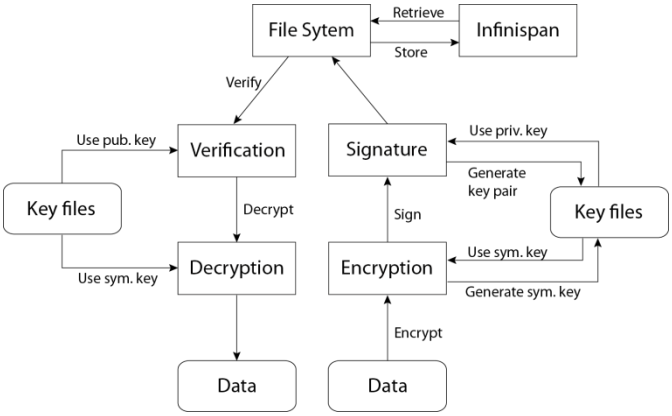

Figure 10 Storing and retrieving data

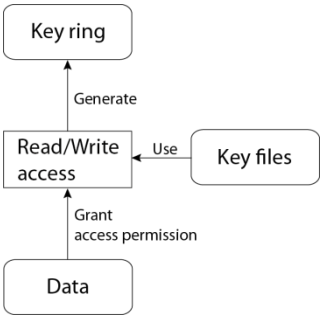

Figure 11 Granting access permission

Figure 11 shows how to assign access permission to data. So by choosing whether to give a user read access or both read and write access to the data, the necessary keys are taken from the key files in order to generate the key ring. The key ring can then be stored to Infinispan in the same way as the ordinary data, after which the authorised user can access it.

In the following sections we will explain the overall structure of the different parts of the system in a bit more details.

4.2 Infinispan Data Grid

Here we briefly describe the overall structure of the Infinispan data grid. This description is taken from the report provided in the 3-week course (cf. Appendix B).

The infinispan cache is mainly used to form a cluster of cache nodes, such that every instance of the cache is running on a separate machine. They form a peer to peer network and share data with each other. The cache is configured to have data persistency, so it stores the data to a predefined location in hard disk. Figure 12 shows an overall structure of the system.

In Figure 12 an example is shown, where the three instances of Infinispan and the corresponding grid file system are running separately on three machines. The Infinispan cache can be accessed via the file system. Data can be stored to cache, retrieved and removed from the cache through the file system. Whenever some data is stored to the cache in one of the machines, it is instantly available on other machines too. Since the caches are configured to be persistent, the data is also stored to the disk. If all cache nodes are shut down and started again, the data would be loaded back to the caches when started. Actually there are two caches running on every machine, one is for storing data and the other is for storing metadata, so every of the Infinispan in the figure contains a pair of caches.

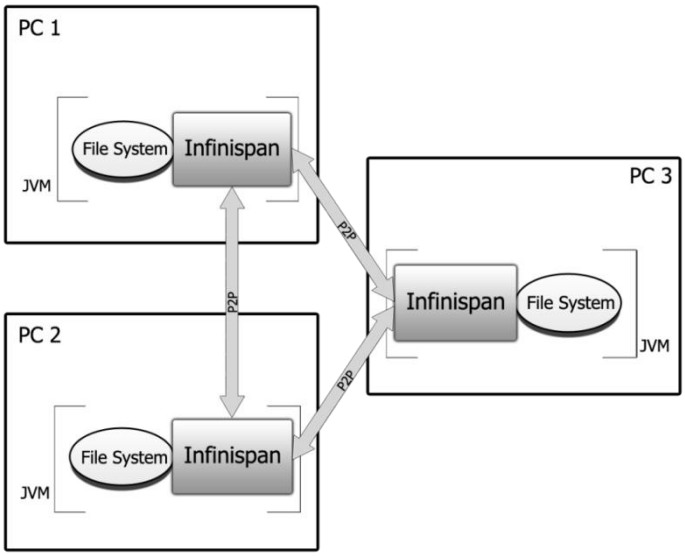

Figure 12 An overview of Infinispan data grid

As shown in the figure, this usage of Infinispan is a peer-to-peer embedded mode usage, i.e. every instance of Infinispan and the file system run in separate JVMs, and the Infinispan caches discover each other via a peer-to-peer connection.

It is worth mentioning that the data grid can be expanded by adding more nodes, and in the same way as shown above, they will discover each other and form a bigger data grid.

4.3 Cryptography

As discussed in chapter 3 (Analysis) we must use both symmetric and asymmetric cryptography in our system. The symmetric encryption is needed for data encryption/decryption and the asymmetric encryption is necessary for digital signature. So for every data to be stored to Infinispan, three key files, namely symmetric, public and private keys, are necessary to be generated and stored locally in the client. These key files are used when the corresponding data is retrieved from Infinispan, or when the stored data is modified by an authorised client. Moreover the key files are used to generate a key ring when the user wants to assign access permission to his data.

4.3.1 Symmetric Encryption

According to our achieved knowledge in chapter 2 (State of the Art), AES gives us the most secure algorithm for data encryption/decryption, so we will use AES in our system. It is necessary to use a key length, which must at least be 128 bits long to achieve the necessary security. A key length of more than 128 bits would decrease the performance of encryption.

We show an overview of encryption process in a sequence diagram.

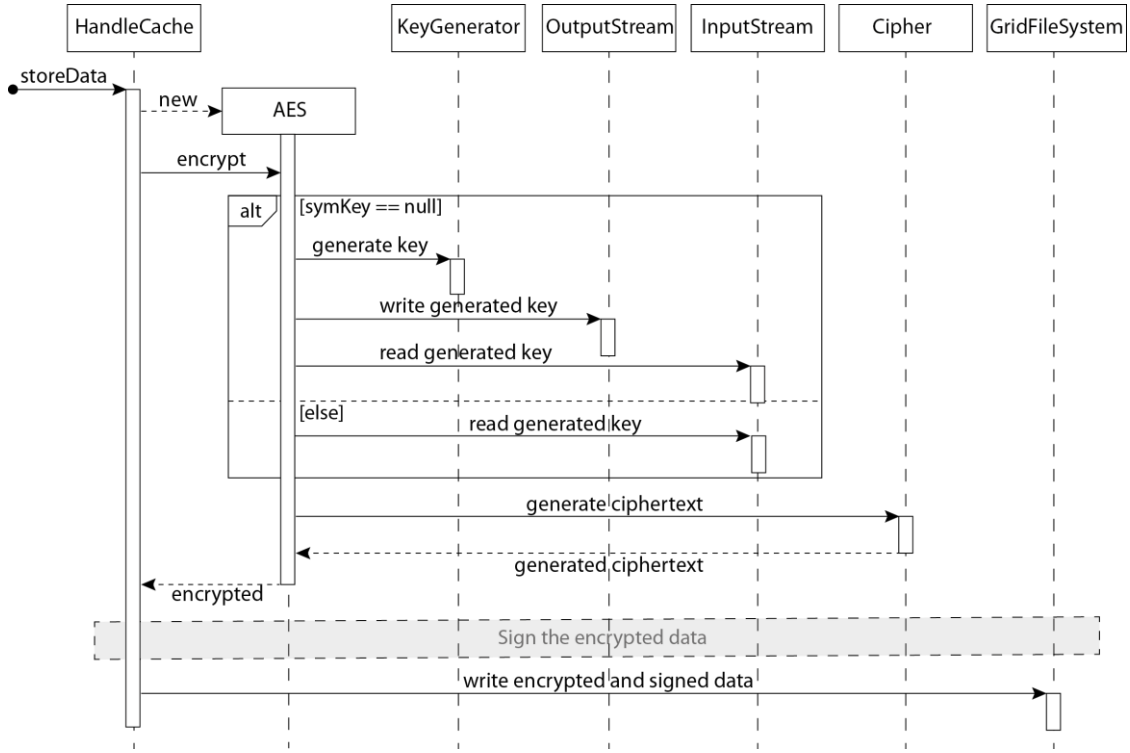

Figure 13 Sequence diagram for showing the encryption process

The sequence diagram in Figure 13 shows the process of storing data, which also contains the encryption procedure. It shows how the steps in encrypting data are intended to be performed by using AES. We can see in the figure that when the function `storeData` is called from a class that can be called `HandleCache`, an instance of `AES` is created, and then the function, `encrypt`, is invoked. Then it is checked whether or not the symmetric key exists on the local client. If the key does not exist, it is generated and stored locally, and then it is read to be used in ciphertext generation; otherwise the previously stored key is read and used to generate the ciphertext. Hereafter the data has to be signed. The signing process is omitted in this sequence diagram to avoid complexity. It will be illustrated in another diagram in the next section (§ 4.3.2). Finally the encrypted and signed data is stored to Infinispan data grid through the file system.

During the encryption process it is necessary to use a mode of operation for AES. According to section 2.1.2.2, any mode of operation rather than *ECB* (Electronic Codebook) can be used, because *ECB* is very simple, and thus it does not hide the encryption pattern perfectly. The other modes that have a good security, require an initialisation vector (*IV*), so after generating the symmetric key, it is also necessary to generate the *IV*. The sequence diagram does not show the generation of *IV* to avoid complexity, but we intend to generate an *IV* and append it to the symmetric key file, and whenever the key file is read to encrypt or decrypt data, both symmetric key and *IV* are read, and used for encryption/decryption.

In a similar way the decryption process would be performed when retrieving data from the Infinispan data grid. After the data is retrieved, and its signature is verified, the function,

`decrypt`, should be called from the class `AES`. Hereafter the file key, which contains the symmetric key and the *IV*, should be read and used to generate the plaintext.

4.3.2 Asymmetric Encryption

Besides symmetric encryption, the asymmetric encryption would also be used in our system, which is for the purpose of signing and verifying data. As discussed in chapter 2 (§ 2.1.3.1), the researches show that the most widely used asymmetric algorithm is RSA. The algorithm is thoroughly tested with regard to its security. The result has shown that if the key length and other parameters that are used in the implementation of RSA, are chosen correctly, no security breach would occur. However the Elliptic Curve Digital Signature Algorithm (ECDSA) has a better performance, and provides almost the same security as RSA, but ECDSA is newer than RSA, and thus it has not been tested in the same amount as RSA. For the time being, RSA is the most popular algorithm, and we also intend to use RSA signature scheme in our system.

As we know, the data it self is not signed, but it is hashed to produce a short digest, which is then signed and attached to the data. The purpose of this procedure is to increase the performance. So we need to use a hash function in our system. We found out (in § 2.1.5) that the SHA-2 family provides the most secure algorithms for generating hash codes. Among these algorithms, SHA-512 is the most secure one, so we will use SHA-512 with RSA signature scheme.

In order to show the overall structure of the signing process, a sequence diagram is constructed, which is shown in Figure 14. It starts with calling the function `storeData`. Hereafter the encryption of data is performed, which was shown in Figure 13, and therefore it is omitted here. So after encryption of data, it is checked whether the private and the public key exist on the local client. If the key pair do not exist, they are generated and stored locally, then the newly stored private key is read to be used for signature generation; otherwise the previously stored private key is read to be used to generate the signature.

After key generation process the function `genSig` is called from a class that can be called `GenSig`, which should be responsible for signature generation. By using an RSA signature scheme class, the SHA-512 hash algorithm is used to generate a hash for the data, and then the hash code is signed. Then the generated signature is attached to the encrypted data, and finally the encrypted and signed data is stored to Infinispan data grid through the file system.

The verifying process is performed when data is retrieved from the Infinispan data grid. After data retrieval, the previously generated and stored public key would be read to be used for verification. Then a verifier class that can be called `VerSig`, would be responsible for verifying the signature. If the signature verifies true, then the data can be decrypted.

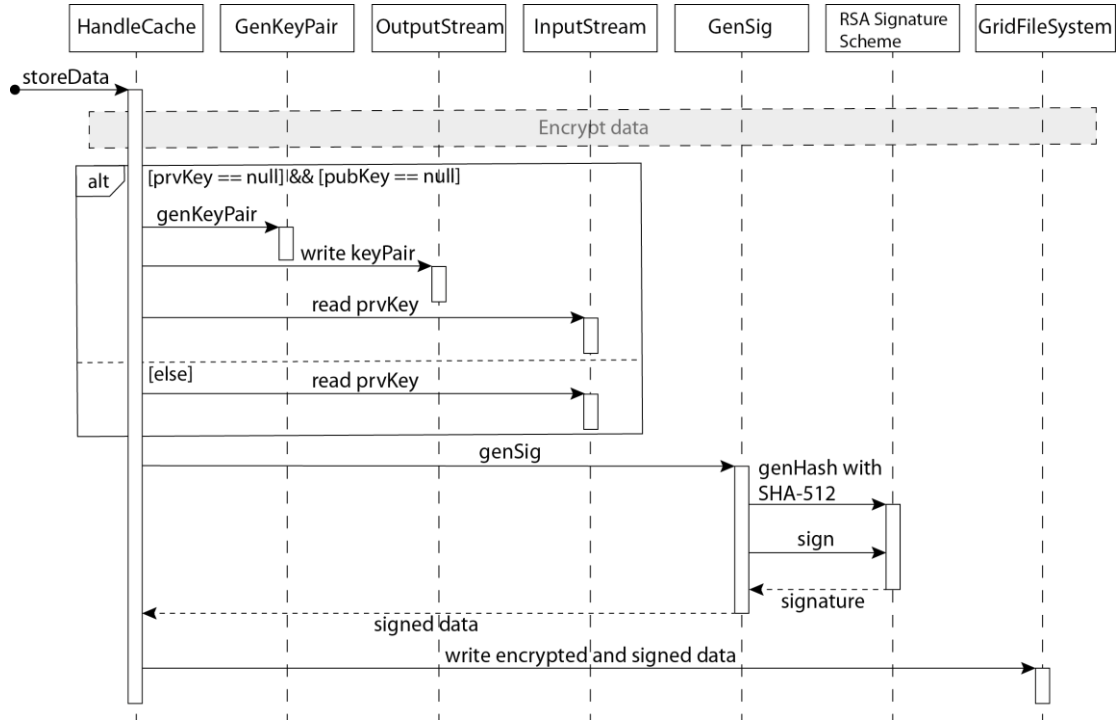

Figure 14 Sequence diagram for showing the signing process

4.4 Granting Access Permission

As file sharing must also be supported by our system, a key exchange mechanism is needed in order to grant a shared client read or write access to the shared data. When two users share their data with each other, the keys belonging to the data can be exchanged between them in any possible secure way, like via encrypted email, encrypted channel or one of the key exchange algorithms, such as Diffie-Hellman key exchange, RSA key exchange, etc. The key exchange in the mentioned situation would not be supported by our system. What is relevant here is when a user wants to share his data with many users. In such a situation, the system should have the possibility to create a key ring, which would contain all the necessary keys belonging to the user's data.

4.4.1 Key Ring

An overview of the process of creating a key ring is shown in a sequence diagram in Figure 15.

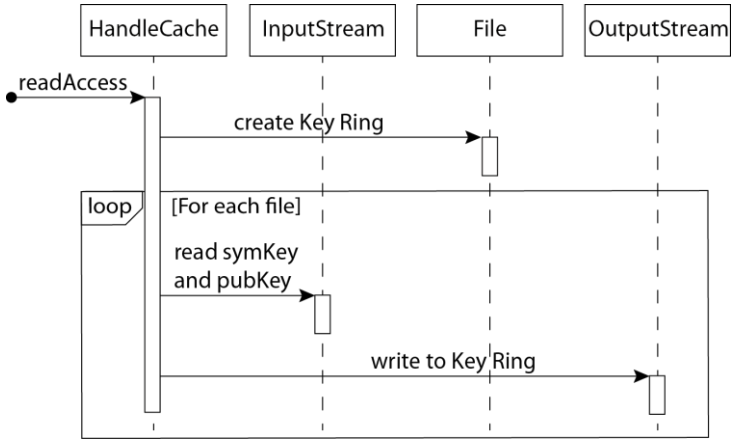

Figure 15 Sequence diagram for granting read access permission

The sequence diagram in Figure 15 shows the key ring creation when a user assigns read access permission to his data. When the user wants to grant read access permission, a key ring file is created. Then for each of the files that the user selects, the corresponding symmetric key and public key are read, and these two keys are written/appended to the key ring. In the same way the process of assigning read/write access permission is performed, but with this difference that instead of appending only two keys, all the three keys, symmetric, private and public keys, are appended to the key ring. The generated key ring can be treated as ordinary files, and can be stored to Infinispan data grid in the same way as storing other files, i.e. the key ring would be encrypted and signed and then stored. The key files belonging to the key ring must of course be sent to the shared clients by using one of the mentioned key exchange methods.

We also intend to provide a method for retrieving the key ring from the Infinispan data grid, such that all the keys listed in the key ring should be extracted and saved as separate key files. These key files are then ready to be used for accessing the corresponding data. However it would be possible to retrieve the key ring in the same way as other ordinary data are retrieved, but in this way the user would get the key ring as a single file that contains a list of keys. Then he manually has to extract the keys from the key ring.

4.5 Graphical User Interface (GUI)

In order to make the system user-friendly, we will provide a GUI for the system.

Besides the provided features (in the GUI provided in the 3-week course), the GUI should have the following functionalities:

* When a user wants to store a file, he should be able to browse his file and store it in encrypted state. Just before the file is stored, the process of key generation, encryption and signing would automatically be performed.
* When a user wants to retrieve a file, if he has the corresponding keys, he should be able to select the file and store it in decrypted state to anywhere he wants in his hard disk. The process of verifying and decryption would automatically be performed.

* A user should be able to assign read or write access to one or more of his stored files by creating a key ring. The key ring should be created and stored locally on user's hard disk, in a predefined location.
* A user should be able to retrieve a key ring, to which he has read or write access. When retrieving the key ring, the extraction of key files and saving them in separate files should be done automatically.

4.6 Summary

In this chapter we started with introducing the main components of the system, and continued with describing some details about the interactions between them.

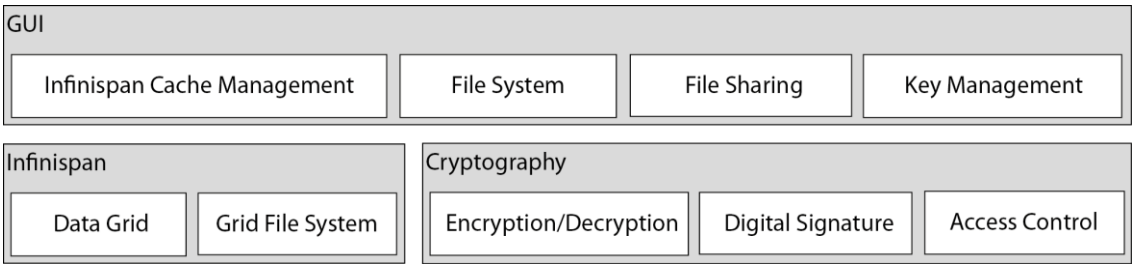

Figure 16 The main components of the system

Figure 16 shows the intended main components that the system should have.

The Infinispan is provided by the 3-week course. It contains data grid and a simple file system created by using the Infinispan API. The data grid has elasticity, i.e. it can be expanded by starting more Infinispan cache instances in different machines, and then they will discover each other via a peer-to-peer communication and form a bigger data grid.

Cryptography would be used for encryption, decryption, signing and verifying data. For encryption and decryption, symmetric cryptography would be used with the AES algorithm and a key length of 128 bits. For signing and verifying, the asymmetric encryption would be used with the RSA signature scheme and the hash algorithm SHA-512. The encryption, decryption, signing and verifying processes would be performed on the local client.

The GUI is based on the provided GUI in the 3-week course. The existing features are Infinispan cache management and a simple file system. Moreover a key management mechanism should be added in order to make it possible for the users to share files with each other.

CHAPTER 5

Implementation

The implementation of the prototype is based on the decisions made in chapter 4 (Design). In this chapter, first of all, we will show the overall structure of the system using a class diagram, followed by a short description about how classes are categorised and how they interact with each other. In the second part of this chapter, we will discuss about the most important implementation details.

5.1 Structure of the System

The prototype is implemented in Java. The classes are categorised in three packages, which are `cac`, `infinispan` and `gridFileSystem`. The package `cac` contains the classes responsible for encryption/decryption and digital signature. The tasks of the classes in `infinispan` are creating the GUI and providing the main functions of the system. The package `gridFileSystem` is imported from the Infinispan library because of a little modification in one of the classes.

In Figure 17 we can see a class diagram, which shows the classes and the three packages. In order to reduce complexity, only the important and relevant fields and methods are shown.

In the package `cac` there are five classes. The class `AES` contains methods for encryption and decryption of data. The class `GenKeyPair` generates the public key and the private key. The singing and verifying process are performed by the classes `GenSig` and `VerSig` respectively. The class `HexByteArrayConverter` contains methods for conversion between byte array and hex string. It is used to convert the keys to hex string before saving, and convert them back to byte array after reading them. As mentioned before, the four classes in the package `gridFileSystem` are imported from the Infinispan library. These classes are used to create a file system for the grid. By default, the file system is designed to be compatible with Unix based systems. In order to make it compatible with Windows OS, we have changed the representation for file separator in the class `GridFile`, and for that it was necessary to import these four classes in the project.

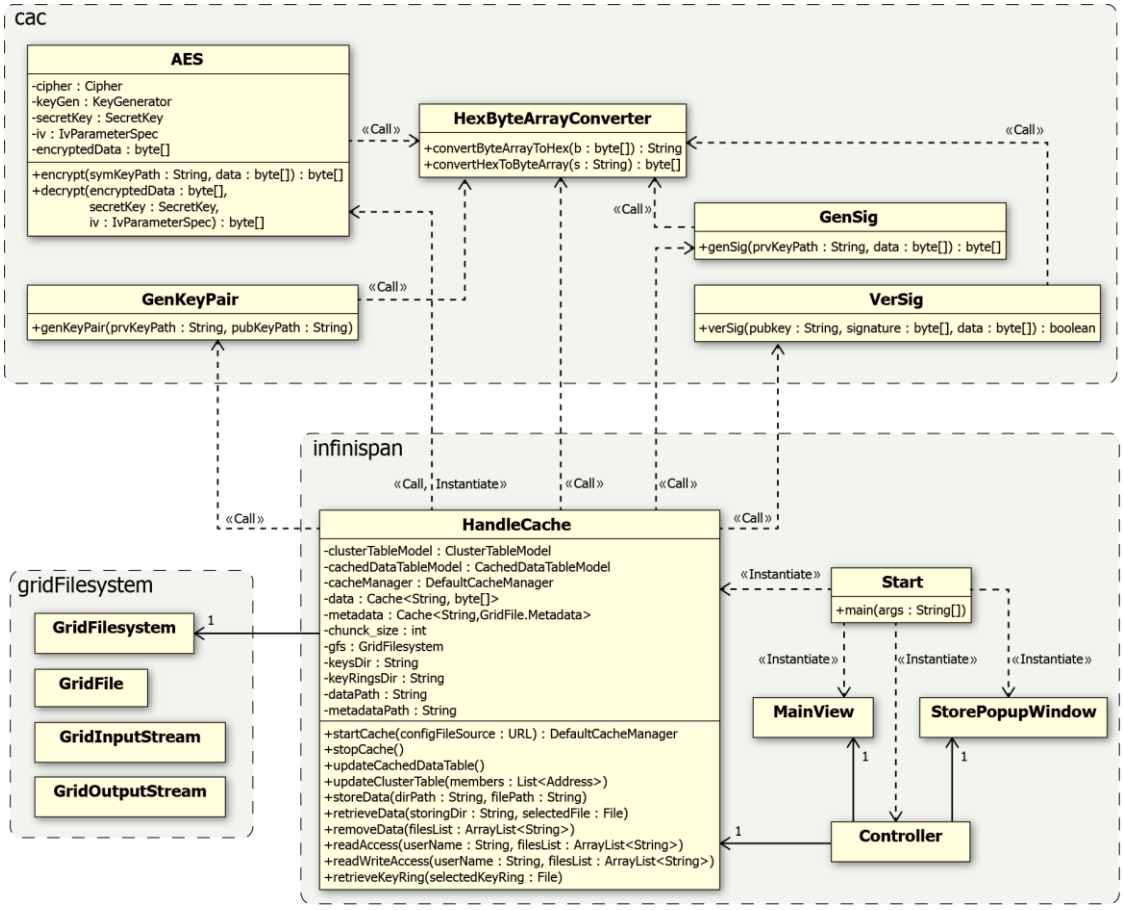

Figure 17 Class diagram

In the package infinispan the classes `MainView` and `StorePopupWindow` contain the GUI, and the class `controller` is responsible for adding listeners to the components of the GUI. The class `HandleCache` has a crucial role in the system. It contains the methods for starting and stopping the infinispan cache, storing and retrieving of data, creating key rings and updating some parts of the GUI related to lists of cache nodes and data. The methods for encryption and decryption and digital signature are also called in this class while storing and retrieving files. Finally we have the class `Start` that contains a `main` method for starting the program.

We would like to specify again that the major part of the GUI and the grid file system in the infinispan package were provided in the 3-week course (Appendix B). In this project we have added the key management functionality to the GUI, and we have also modified the class `HandleCache` in order to apply the cryptographic access control mechanism to the data grid.

5.2 Cryptography

Cryptography is the core of our system since both symmetric and asymmetric encryption is used to form the basis for cryptographic access control. In the following we will mention the important implementation details about how it is used to perform encryption and digital signature.

5.2.1 Symmetric Encryption

For the encryption process, we have used AES implementation from the Java library. A key length of 128 bits is used, which provides a good performance for the encryption process, and at the same time the security is also guaranteed. A longer key length can however be used to increase the security even more, but it would of course affect the performance.

The class `AES` contains two methods, one for encryption and the other for decryption. The encryption method, `encrypt(String symKeyPath, byte[] data)`, takes the two parameters, namely the path for saving or reading the symmetric key and the data to be encrypted. The method returns the encrypted data in byte array type. In the body of the method, first of all it is checked whether or not the symmetric key exists at the given path. If it exists, then it is read and used for encryption, otherwise a new key is generated and saved to disk using the specified path.

Among the modes of operation provided by the Java library, we have used the counter (*CTR*) mode. *CTR* mode is one the most secure methods in symmetric encryption, which do not use padding technique, and as a result it provides a ciphertext that would not be longer than the plaintext. Since our system is used for data storage purpose, this mode of operation is best suited, because the encrypted data would not be expanded unnecessarily, which would result in occupying more disk space than the original data.

Moreover, *CTR* mode encrypts every block of data independently, and thus blocks can be encrypted in parallel. This quality in *CTR* mode can be used to increase the performance of encryption in multi-processor machines.

However the most suited mode of operation for storage devices is the *XTS-AES* mode, which is approved by NIST as being a secure method. It does not use padding technique either. Many well-known software, among others TrueCrypt, also uses *XTS-AES* mode, but the Java library does not support this mode of operation, so we used *CTR* mode alternatively, which is close to *XTS-AES* in terms of quality and performance.

After generating the symmetric key, an initialization vector, *IV*, is also generated, and whenever the data is decrypted, the same *IV* must be used for decryption, so we need to save the *IV* too. In the method `encrypt(…)`, the *IV* and the symmetric key are both converted to hex string using the necessary method from the class `HexByteArrayConverter`. They are separated by the character ":", and saved in a text file. The specified separator character is then used to separate them when we read the text file. Then they are converted back to byte arrays and used to decrypt the corresponding encrypted data.

The method for decryption has this signature:

```
decrypt(byte[] encryptedData, SecretKey secretKey, IvParameterSpec iv).
```

As we can see it takes three parameters, namely the encrypted data, the symmetric key and the *IV*. It returns the original text in byte array type.

5.2.2 Digital Signature

The digital signature process has three steps, namely key pair generation, signing and verifying. These are performed using three classes, `GenKeyPair`, `GenSig` and `VerSig`. The class `GenKeyPair` contains the method:

```
genKeyPair(String prvKeyPath, String pubKeyPath).
```

This method takes two strings as parameters. These two parameters specify the paths, to where the private and the public key must be saved. For generating the key pair, we use RSA algorithm by calling the method, `getInstance("RSA")`, from the Java class called `KeyPairGenerator`. Then we initialize the keys with the key size 1024 bits, and by using a random number generator class from the Java library:

```
initialize(1024, new SecureRandom()).
```

Finally the keys are converted to hex string and saved to the given path.

The other class, `GenSig`, contains a method called:

```
genSig(String prvKeyPath, byte[] data).
```

This method takes the path for the private key and the data as parameters, and it returns the signed data in byte array type. First of all the private key is read from the specified path. After that a signature object is created by using a specific hash algorithm. Here we use the algorithm SHA-512: `Signature.getInstance("SHA512withRSA").`

The signature object is initialised by using the private key. Then the signature for the data is generated as byte array and attached to the data. Finally the signed data is returned in byte array type.

The last but not least procedure is verifying the signature of a data. In the class `VerSig`, the method:

```
verSig(String pubKeyPath, byte[] signature, byte[] data)
```

is responsible for performing the verifying process. This method takes three parameters, namely the path for the public key, the generated signature and the data. In this method after reading the public key, a signature is generated for the data by using the public key. Then the newly generated signature is verified by using the given signature:

```
newSig.verify(signature),
```

where `newSig` is the newly generated signature. If the verification succeeds, the method would return true, otherwise false.

5.3 Data Handling

Storing, retrieving and removing data are performed by methods in the class `HandleCache`. In the following we will briefly describe how these three actions are performed.

5.3.1 Storing Data

The method `storeData(String dirPath, String filePath)` is responsible for storing data to the data grid. The method takes two parameters: directory path and file path. The directory path has to be given by the user via the GUI, to where the file should be stored. The parameter `filePath` is the path of the file on the user's client, from where the file would be read. Moreover, this parameter is also used to extract the file name, which is used while storing the file, such that, the stored file would have the same name as the local file.

In the method `storeData(…)`, first of all the paths for the three keys are defined as strings. Then the process of key pair generation is performed, and the generated private and public keys are saved to the specified path. Then the encryption process is done by calling the method, `encrypt()`, with the specified path for the symmetric key. (As mentioned before, in the method, `encrypt()`, the symmetric key would be read from the path if it already exists, otherwise the key would be generated and saved to the specified path.) The encryption method returns the data in encrypted form, after which it is signed and stored to the data grid. At the same time the encrypted file is also saved on the local disk, to a predefined path, if data persistency is enabled.

The paths for the three keys are defined in a systematic way, so that the user does not need to think about it. We would like to give an example in order to make it clear how the keys are saved for a file. For example, a user wants to store a file, called "mytext.txt", to the data grid. He browses the file via the GUI. Let us say the location of the file is:

"C:\files\myfiles\mytext.txt".

Then he gives a directory path, to where the data should be stored, for instance: "\user1".

After storing, the stored path would be visible via the GUI: "\user1\mytext.txt".

Now, the directory path for the three keys belonging to the stored file would be defined by merging *the predefined keys directory* with *the stored file path*, where the extension is removed. So the directory for the keys belonging to the file "\user1\mytext.txt" would be:

"..\keys\user1\mytext ".

The last child directory would always have the same name as the file, but without the extension.

So the three keys belonging to this file would be saved in the client's machine with the following paths:

"..\keys\user1\mytext\symKey.txt"
"..\keys\user1\mytext\prvKey.txt"
"..\keys\user1\mytext \pubKey.txt"

The parent directory for "keys" is the program folder.

In this way, the keys for every stored file would have a unique directory path at the client's machine, and whenever the user wants to retrieve the file, the keys belonging to that file would be read from the client's machine, which then would be used for decryption and verification.

5.3.2 Retrieving Data

In the class `HandleCache`, the method used for retrieving data is called:
`retrieveData(File selectedFile, String storingDir).`

The method takes two parameters, namely `selectedFile` and `storingDir`.

`selectedFile` is the file selected from the data grid via the GUI, and `storingDir` is the path for the directory, where the selected file should be saved on the local machine.

At the beginning of the method, two paths are defined, one for the symmetric key and the other for the public key. We mentioned in the previous section about how the paths of the keys are mapped with the path of the corresponding file. So here, when we define the paths for the keys, we simply use the selected file path without the file extension, and then the keys are read from the specified paths. In the next step, the selected file is read, after which its signature is verified, and finally, if the verification succeeds, the file is decrypted and saved to the disk. The location to where the file should be saved is selected by a file chooser via the GUI.

5.3.3 Removing Data

The class `HandleCache` also contains a method for removing data from the grid, which is called `removeData(ArrayList<String> filesList)`. Its parameter is an array list of file paths that are selected from the data grid, which means that it is possible to remove more than one file at once. The data and the metadata are removed from the data grid by using nested for loops to iterate through the `filesList`. The methods, `data.remove()`, and `metadata.remove()` are called from the Infinispan API, which deletes the data and metadata both from the data grid, and from the local disk in case of data persistency. Then the corresponding keys are also removed by using the array list since the selected files are mapped with the corresponding keys. At the start of this method it is checked whether or not the user has all three keys available on his local client. If he has the keys, then he is able to remove the selected file(s), otherwise he is considered to be an unauthorised client, and thus cannot perform the remove action.

5.4 Key Ring Management

The creation of a key ring is also performed in the class `HandleCache` by two methods:

```
readAccess(String userName, ArrayList<String> filesList)
```

```
readWriteAccess(String userName, ArrayList<String> filesList)
```

Depending on whether a user wants to assign read access, or both read and write access to his file(s), these methods are used to create a key ring. The practical process is done via the GUI,

which will be mentioned in the next section (see § 5.5), but here we will explain the implementation details. The parameters for these two methods are `userName` and `filesList`. `userName` is used as prefix for the name of the key ring file. `filesList` is an array list containing the selected files, to which a certain access permission must be assigned.

As mentioned before, every file is mapped with the corresponding keys. So in the process of creating key rings, when a user chooses one or more files from the data grid, an array list, containing the paths of the files, is created, which is given to the above two methods as parameter. The method contains nested for loops for iterating through the list, and for every file it iterates through the corresponding keys. These keys are read, and appended to the key ring file. If user chooses to assign read access to one or more files, the symmetric and the public keys belonging to the file(s) are listed in the key ring file. In contrast to that, if user chooses to assign both read and write access to his files, then all three keys for every file would be listed in the key ring file.

For example user1 wants to grant access permission to user2 by assigning read and write access to his file, “mytext.txt”, but only read access to his another file, “mypicture.jpg”. After the key generation process, the key ring file would be generated with the name:

“`user2-KeyRing.keyring`“.

The key ring will be saved under the directory “`keyrings`” under program folder.

The content of the key ring would be:

```
ReadWriteAccess
\user1\mytext\prvKey.txt|<private key>
\user1\mytext\pubKey.txt|<public key>
\user1\mytext\symKey.txt|<symmetric key>
ReadAccess
\user1\mypicture\pubKey.txt|<public key>
\user1\mypicture\symKey.txt|<symmetric key>
```

The strings “`ReadWriteAccess`” and “`ReadAccess`” are used to detect the corresponding access permission, and the character “|” is used to separate the key path and the actual key. These signs are used when a user wants to retrieve the key ring.

This key ring file can now be uploaded to the data grid in the same way as other ordinary files. For retrieving the key ring, the method `retrieveKeyRing(File selectedKeyRing)` in the class `HandleCache` is implemented. The function of this method is in fact the inverse of key ring creation. It extracts the three keys for “mytext.txt”, and saves them in separate files in the correct path (so keys and the corresponding file are mapped in the earlier mentioned way), such that whenever user2 wants to retrieve “mytext.txt”, the keys are automatically read, and the file is encrypted, verified and saved to his disk. The same applies to the file “mypicture.jpg”. The only difference is that user2 would only be able to read the file, but since he does not have the private key, he cannot sign it.

5.5 Structure of the GUI

In the following we will give an overview of the structure of the GUI by showing a screenshot of the prototype. The GUI contains three tabbed panes. The screenshot in Figure 18 shows only the “Manipulate Data” tab, which is mostly relevant to be described here. (In order to have an overview of all parts of the GUI, see Appendix B)

The “Control Panel” tab contains a button for starting and stopping the cache. Moreover, when the cache is running, the cache configuration file would be shown in a text field area. The “Cluster View” tab contains a table showing the list of cache nodes connected to the data gird. The “Manipulate Data” tab contains the major part of the GUI. As we can see in Figure 18, this tab contains a primitive file system. The table to the left contains a list of data that has been stored to the grid. The first column shows in which directory the files are stored, and the second column shows information about the data, such as the size of the data, the chunk size and the last modification time. When we store data to the grid, they are divided in chunks, so “chunk_size” is simply the size of every chunk of data in bytes.

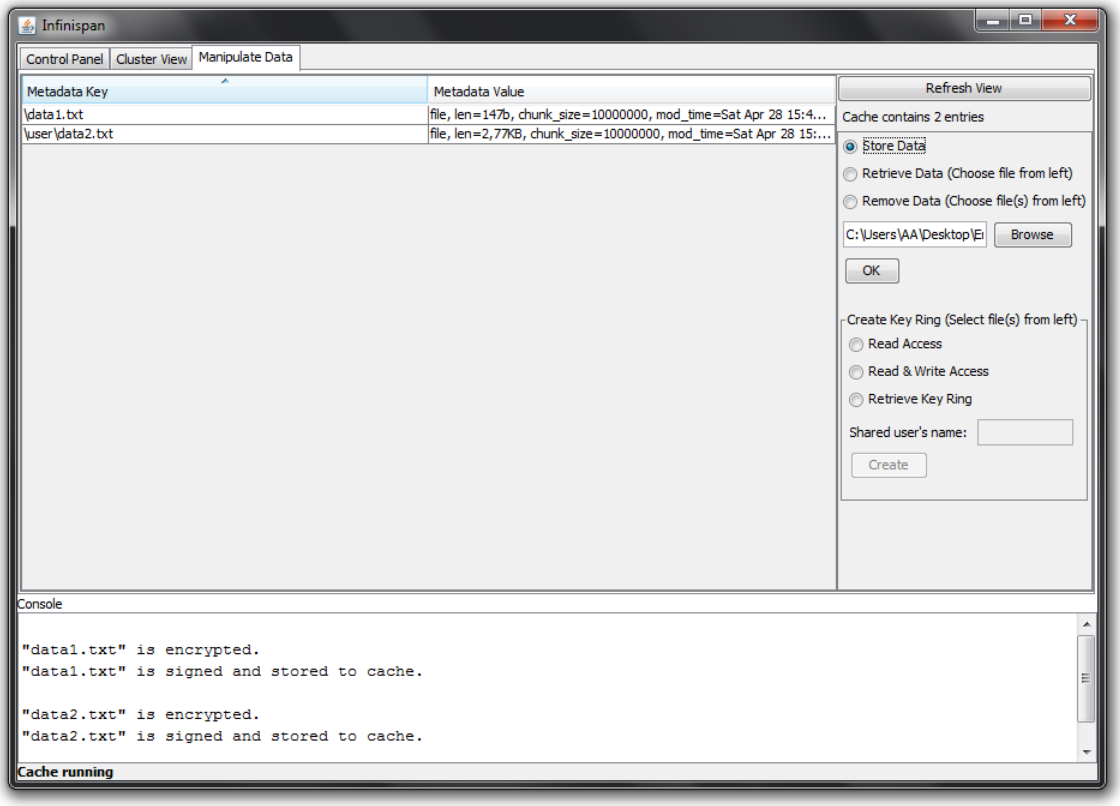

Figure 18 Manipulate data tab

At the right side of this tab the “Refresh View” button refreshes the data table. This button is used when another cache node joins the grid. When we click on this button, the files stored to the data grid would instantly be available in the table of the newly joined cache node. If the radio button “Store Data” is selected, it is possible to browse a file from the local machine, and by clicking the “OK” button a popup window appears, where it is possible to choose an existing folder or create a new folder for the data to be stored to the grid. If the radio button “Retrieve

Data" is selected, then we have to click on a file in the table. Then by clicking on the "OK" button a file chooser window will be opened in order to specify where the data must be saved. If the radio button "Remove Data" is selected, then it is possible to select one or more files from the data table to be removed from the grid. It is worth mentioning that whenever we manipulate data in the grid, the data stored to the disk is also simultaneously manipulated, because the caches are configured to have data persistency.

Then we have the functionality to create the key ring. When one of the radio buttons "Read Access" or "Read & Write Access" is selected, the user has to select one or more data from the table, and then he has to specify the name of the user, to whom he grants access permission. Then by clicking on the button "Create" the key ring file is created and saved to disk. By selecting more files, the user can assign more access permissions to them, and if he does not change the "shared user's name" field, then the previously created key ring would just be updated. The key ring can then be uploaded to the data grid. After selecting the key ring file from the table, the authorised user can select the radio button "Retrieve Key Ring". Then the name of "Create" button would be changed to "Retrieve". By clicking on it, the user would retrieve the key ring.

5.6 Summary

In this chapter we described how the developed design has been implemented. The implementation can be divided in the following major parts:

Cryptography: Cryptography is the most important part of the system. It is the basis for the cryptographic access control mechanism. The system makes use of both symmetric and asymmetric encryption in a hybrid way. For the encryption/decryption of data, AES is used with the key length of 128 bits, and the CTR mode as mode of operation. The digital signature process is implemented by using RSA signature scheme, where SHA-512 is used as the hash algorithm. The process of signature has three steps, namely key pair generation, signing, and verifying.

Infinispan: The Infinispan data grid with its corresponding grid file system is provided by a 3-week course. Since it is similar to cloud storage systems, and it can actually be used as a cloud storage system, we have used it to apply the cryptographic access control mechanism to it.

Key Management: The system also supports file sharing, where a key exchange mechanism is used to control the access to users' data. The system enables users to create a key ring containing the keys for his files. Depending on which level of access permission a user wants to assign to his file(s), the key ring can contain only two keys, namely symmetric and public keys, or all three keys. The key ring can then be distributed between shared clients, who can access the shared data.

GUI: The GUI contains three tabbed panes. In the Control Panel tab, users can start or stop the Infinispan cache. In the Cluster View tab, users can see a list of connected Infinispan cache nodes. In the Manipulate Data tab users can manipulate data, like storing, retrieving and

removing data. Moreover users are able to create key rings, which can be uploaded to the data grid. They can also retrieve existing key rings from the data grid.

CHAPTER 6

Evaluation

Here in this chapter, we evaluate the implemented system with regard to its performance and security. In the first part of this chapter, we evaluate the performance of different functionalities, namely storing data, retrieving data, encryption/decryption, digital signature and key management. In the second part, we evaluate the security of the system by discussing about whether the system is immune against the various possible attacks on the security of the system. AT the end of the chapter, on the basis of the results achieved from the evaluation, we discuss about further improvements that can be applied to the system.

All the tests are run on a laptop with the following specifications:
OS name: Microsoft Windows 7 Professional
System type: 32 bit
Processor: Intel(R) Core(TM)2 Duo CPU, 2,10 GHz
Physical Memory (RAM): 2,00 GB
Hard disk drive: 160 GB, 5400 rpm, ATA interface

6.1 Performance Evaluation

Since our security solution is applied on the Infinispan data grid, we will test the performance of storing data to and retrieving data from the Infinispan. We will test it both with and without cryptographic access control.

As we know, the system is implemented in Java, so every reading and writing that is applied to a file, is performed by using Java's `FileInputStream` and `FileOutputStream` respectively. The two mentioned Java classes are the basis for all readings and writings in our system. So here at the beginning, we will test the speed of copying a file from one place to another place on the hard disk by using both Windows' "copy and paste" function and Java, in order to have a comparison between them.

6.1.1 A Comparison between Windows and Java file copying

The following table shows the average time that took to copy and paste files with different sizes using Windows' "copy and paste".

Time/seconds	Size/MB
0,012	0,01
0,014	0,1
0,02	1
0,05	10
0,095	30
0,155	60
0,21	80
0,25	100
0,4	150
1	200
1,3	250
2,8	300
4,1	350
5,4	400
7,6	500
10	600

In order to have an overview of the speed of copy and paste function, we draw graphs for the above table, where the average speed is determined.

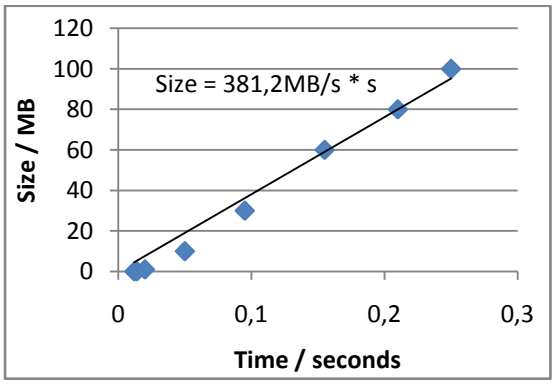

Figure 19 Copying files up to 100 MB

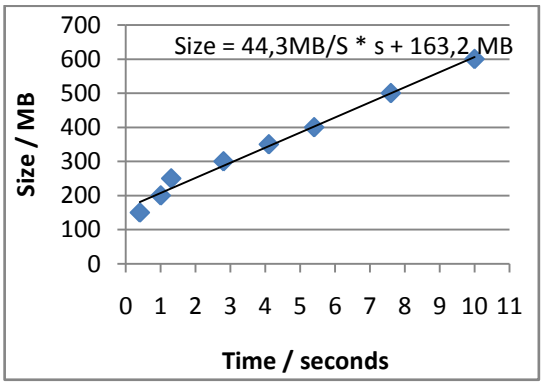

Figure 20 Copying files, 100 – 600 MB

The above figures, Figure 19 and Figure 20, show two graphs belonging to the data in the above table.

Figure 19 contains the graph for files up to 100 MB, and we can see that the average speed of copying is approx 381 MB/s. For the files above 100 MB (until 600 MB) the average speed of

copying is approx 44 MB/s, so it is much faster to copy small files. The overall average speed is shown in Figure 21.

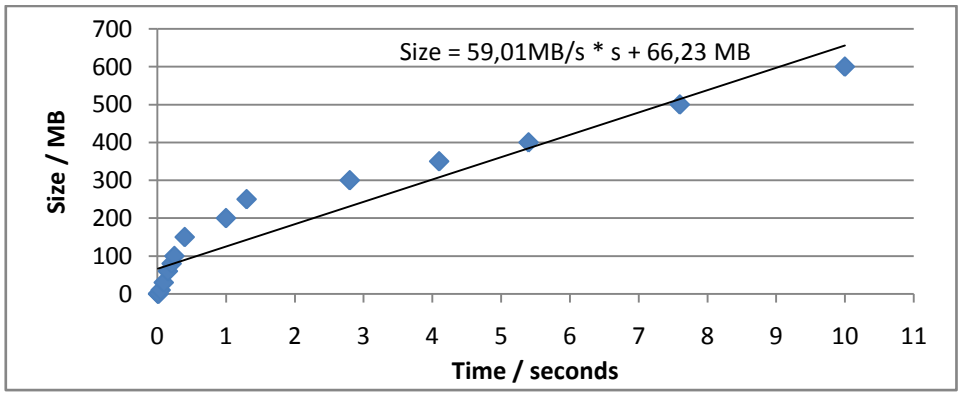

Figure 21 Copying files 0,01 MB – 600 MB

As shown in Figure 21, the overall average speed of copying files in Windows is approx 59 MB/s.

Copying files using Java

The following table shows the average time that took for copying files with different sizes.

Time/seconds	Size/MB
0,00597	0,01
0,0243	0,1
0,0088	1
0,074	10
0,52	30
1,352	60
1,922	80
2,392	100
4,011	150
5,333	200
7,201	250
10,462	300

In the current pc, where the tests were run, Java cannot copy files greater than 300 MB at once. During the copying process, Java keeps the whole file in memory, so it cannot copy a file larger than the available memory. However it is possible to copy larger files by reading them in small pieces.

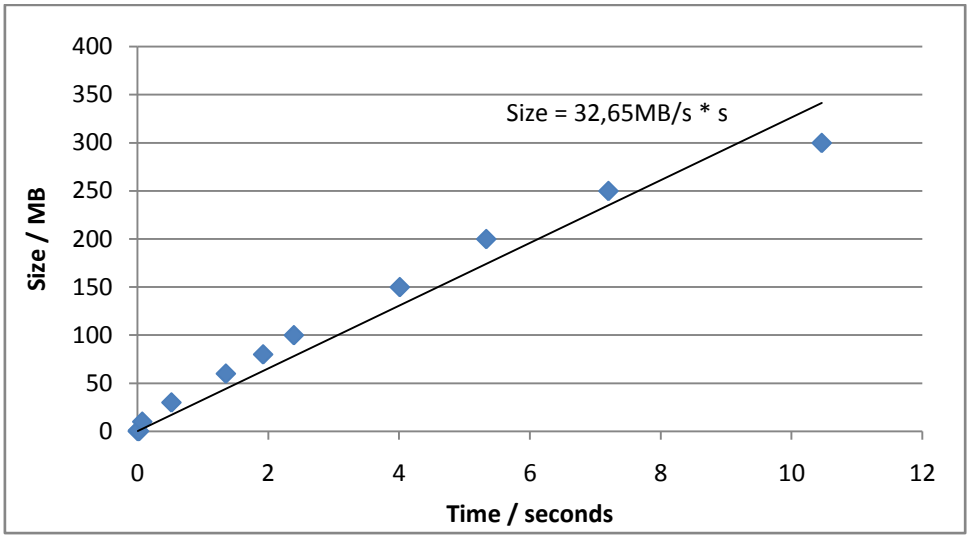

Figure 22 Copying files using Java, files: 0,01 MB – 600 MB

Figure 22 contains a graph showing the speed of copying files up to 300 MB. We can see that the average speed is approx 32 MB/s. It appears that file copying in Windows is twice as fast as copying files using Java. As a result we should keep in mind that the file transferring mechanism in our system is of course affected by this low performance.

6.1.2 Storing and Retrieving files using Infinispan as Local Cache

In this section we test the functions, storing and retrieving files, by using Infinispan as local cache. It means that one instance of Infinispan is running on the pc, where we perform the actions, storing and retrieving.

We run multiple tests both with and without cryptography and data persistency. Since data is always stored to cache, we have a limitation on the size of data to be stored and retrieved.

Storing files "with" persistency, without encryption and signing

Chunk size: 100 KB		**Chunk size: 1 MB**		**Chunk size: 10 MB**		**Chunk size: 100 MB**	
Time/sec	Size/MB	Time/sec	Size/MB	Time/sec	Size/MB	Time/sec	Size/MB
0,145	0,01	0,157	0,01	0,151	0,01	0,34	0,01
0,161	0,1	0,154	0,1	0,179	0,1	0,36	0,1
1,41	1	0,165	1	0,253	1	0,346	1
30	10	4,06	10	0,846	10	0,903	10
111	30	19,46	30	3,555	30	2,19	30
-	60	90,1	60	13,79	60	3,58	60
-	80	-	80	24,6	80	4,79	80

The above table contains data showing speeds of storing files with different sizes to Infinispan using data persistency. Here the encryption and signing are switched off. We have tested storing of files using chunk sizes from 100 KB to 100 MB. Because of the current available memory, storing of data more than 80 MB was not possible.

If we use small chunk sizes, like 100 KB or 1 MB, it takes very long time to store files, compared with the larger chunk sizes, like 10 MB and 100 MB. When choosing small chunk sizes, the large files have to be sliced in small pieces, and every piece must be written to the disk, and that is why it takes such long time. We can see that the optimal speed can be achieved by using a chunk size of 100 MB for files up to 80 MB.

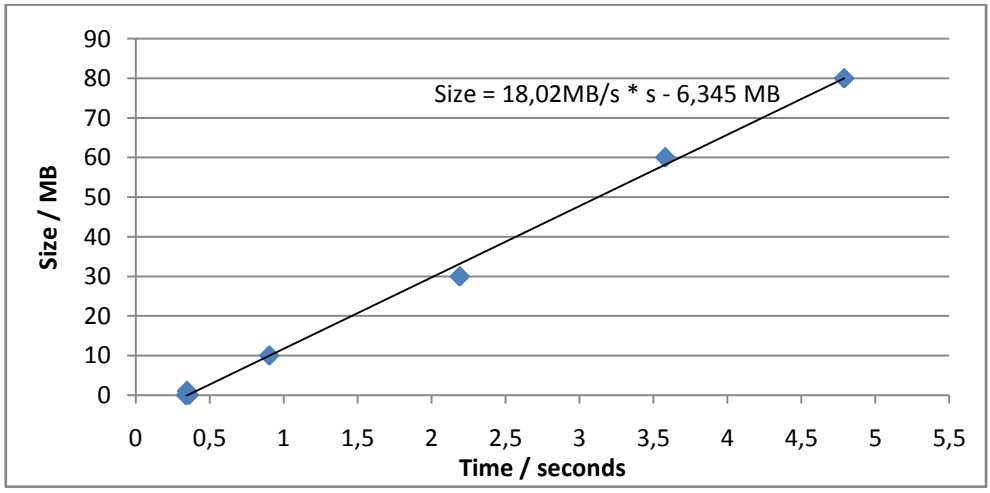

Figure 23 Storing files to Infinispan with persistency, without encryption, files: 0,01 MB – 80 MB, chunk size: 100 MB

We can see in Figure 23 that the average speed for storing files is approx 18 MB/s when a chunk size of 100 MB is chosen. It means that every file is less than the specified chunk size, so files would be stored as a whole without dividing them in chunks. In this case chunking would be useless here, but in order to have the optimal performance, this would be the best choice.

Storing files "without" persistency, and without encryption and signing

Chunk size: 100 KB		Chunk size: 1 MB		Chunk size: 10 MB		Chunk size: 100 MB	
Time/sec	Size/MB	Time/sec	Size/MB	Time/sec	Size/MB	Time/sec	Size/MB
0,002	0,01	0,006	0,01	0,016	0,01	0,173	0,01
0,005	0,1	0,006	0,1	0,024	0,1	0,185	0,1
0,029	1	0,021	1	0,042	1	0,194	1
0,137	10	0,124	10	0,164	10	0,25	10
0,270	30	0,237	30	0,305	30	0,358	30
0,546	60	0,400	60	0,457	60	0,697	60
0,639	80	0,515	80	0,591	80	0,741	80
0,756	100	0,610	100	0,683	100	0,841	100

The above table shows the times that took for storing files to Infinispan without data persistency, and also without encryption and signing. Also here we tested file storing with different chunk sizes, and we can see that there is not a big variation in the results compared to the previous results "with" data persistency. However we can see that the optimal speed can be achieved by choosing a small chunk size, i.e. 100 KB or 1 MB, which is the opposite of the previous situation ("with" data persistency).

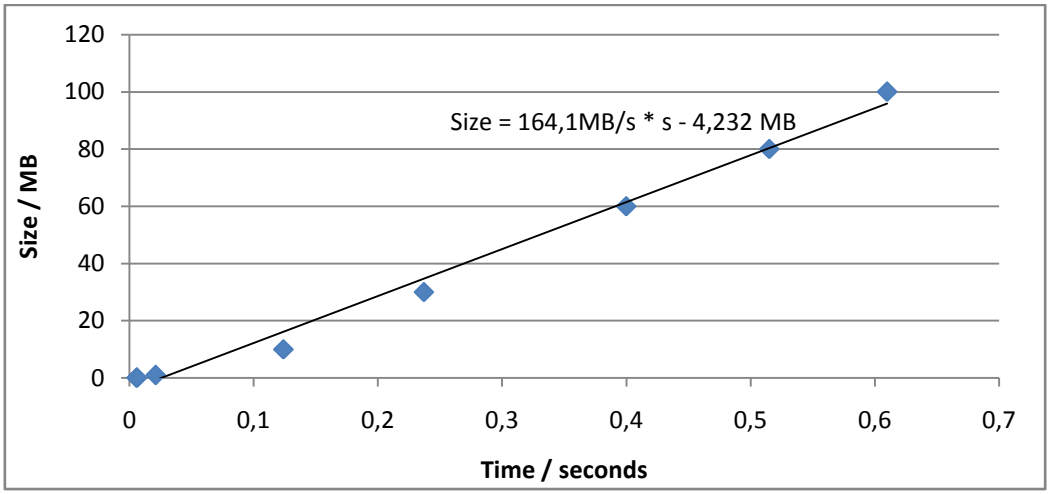

Figure 24 Storing files to Infinispan without persistency without encryption, files: 0,01 MB – 100 MB, chunk size: 1 MB

Figure 24 contains the graph for data taken from the above table, where chunk size is 1 MB. By choosing this chunk size, we achieve the most optimal speed for files up to 100 MB. We can see that the average speed is approx 164 MB/s, which is almost nine times faster than file storing with data persistency.

Retrieving data without persistency, and without decryption and verifying

Since files are always stored to memory, they will be read directly from memory when retrieving, so it would take the same time to retrieve a file both with and without data persistency. Here we only test retrieving of data without persistency.

Chunk size: 100 KB		**Chunk size: 1 MB**		**Chunk size: 10 MB**		**Chunk size: 100 MB**	
Time/sec	Size/MB	Time/sec	Size/MB	Time/sec	Size/MB	Time/sec	Size/MB
0,008	0,01	0,006	0,01	0,006	0,01	0,008	0,01
0,033	0,1	0,040	0,1	0,022	0,1	0,024	0,1
0,017	1	0,010	1	0,009	1	0,010	1
0,068	10	0,051	10	0,057	10	0,050	10
0,447	30	0,450	30	0,470	30	0,448	30
1,251	60	1,213	60	1,188	60	1,240	60
1,800	80	1,716	80	1,700	80	1,704	80
2,250	100	2,268	100	2,370	100	2,250	100

The above table shows the times that it took for retrieving data from the Infinispan, without decryption and verification, and without persistency. (Retrieving means reading data from infinispan and storing them locally.) We can see that the times for retrieving data are quite close to each other for different chunk sizes. So it means that the chunk size does not have any considerable affect on data retrieval.

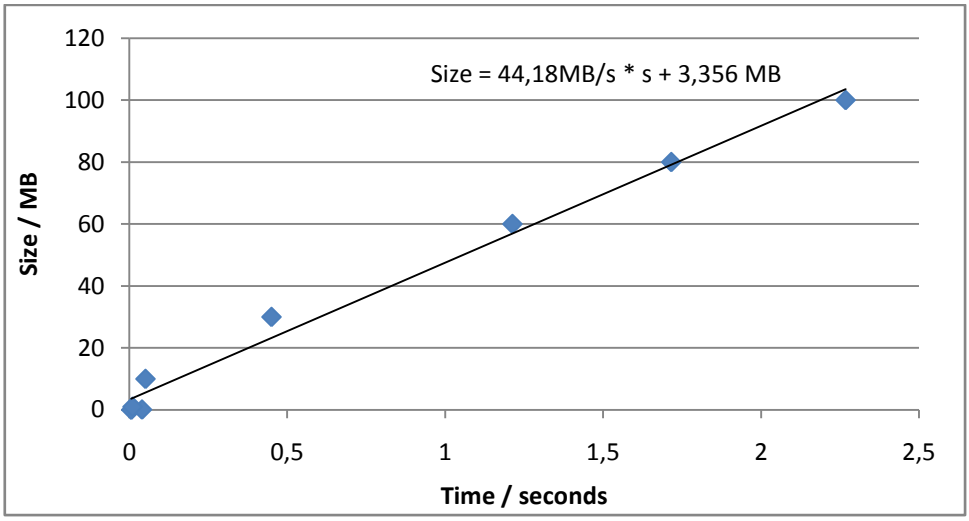

Figure 25 Retrieving files from Infinispan, without data persistency, without decryption, files: 0,01 MB – 100 MB, chunk size: 1 MB

Figure 25 shows a graph for the data from the table with the chunk size of 1 MB. We can see that the average speed is approx 44 MB/s. As the file is directly read from the memory, it would take very little time to read the file, but the most taken time is when the file is written to disk. As expected, data retrieval takes less time than storing data with persistency, because with data persistency, data has to be read from the disk, and then it has to be written both to the memory and the disk.

Encryption using AES-128

Here we test the performance of encryption, which is performed by using AES-128.

With key generation & saving the key		Using an existing key	
Time/sec	Size/MB	Time/sec	Size/MB
0,084	0,01	0,004	0,01
0,080	0,1	0,015	0,1
0,110	1	0,048	1
0,400	10	0,330	10
1,100	30	1,075	30
2,065	60	2,034	60
2,670	80	2,650	80
3,295	100	3,250	100

The above table shows two tests, where the first one is with generating the symmetric key and saving it to disk, and the other test is performed by using an existing symmetric key. As expected, the encryption is faster when using an existing key, but the difference is not too much, because a symmetric key of length 128 bits is rather small, and thus its generation and saving would not take too much time.

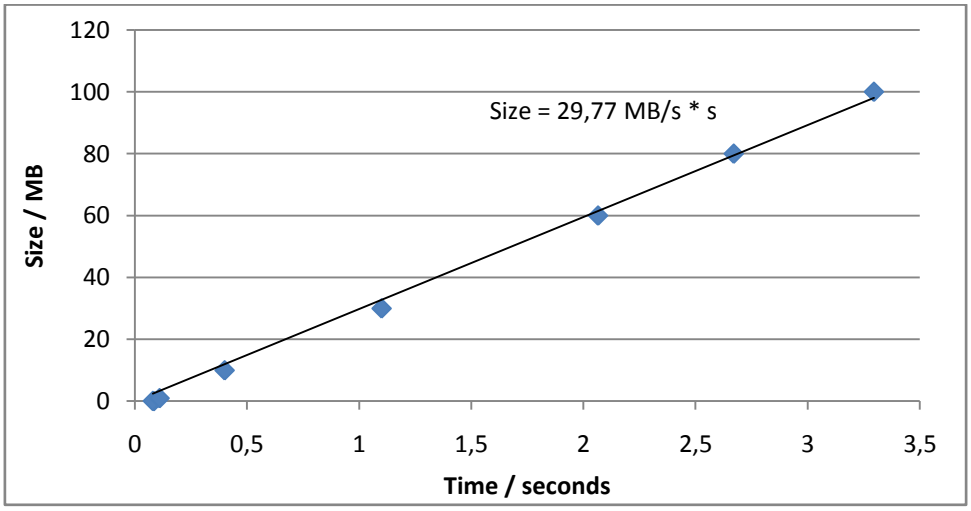

Figure 26 Encryption using AES-128, with key generation, files: 0,01 MB – 100 MB

We can see in Figure 26 that the average speed of encrypting files with key generation is approx 30 MB/s.

Decryption using AES-128

Time/sec	Size/MB
0,001	0,01
0,005	0,1
0,038	1
0,365	10
1,066	30
2,135	60
2,907	80

A graph for the above table for determining the average speed of decryption is shown in Figure 27.

As we can see in Figure 27, the average speed for decryption is approx 28 MB/s, which is almost the same as encryption speed. As AES uses a symmetric encryption algorithm, so the encryption and decryption processes are the inverse of each other, and as a result their speeds would also be the same.

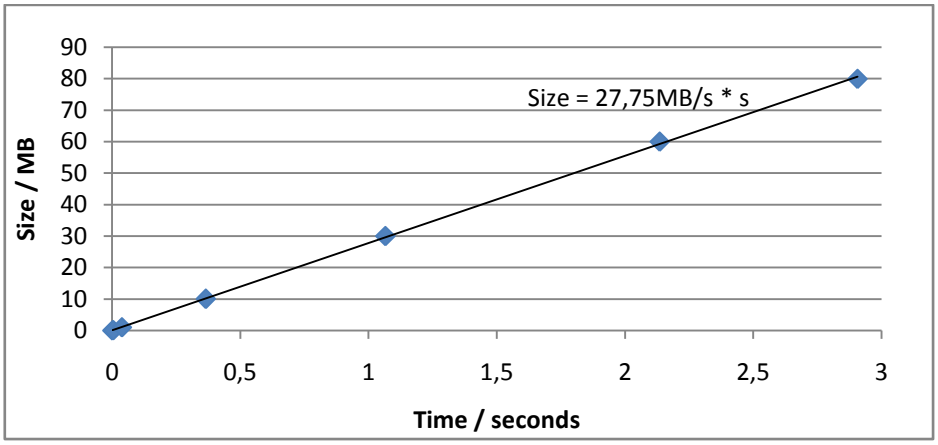

Figure 27 Decryption using AES-128, files: 0,01 MB – 80 MB

As we know AES is widely used in the world, and there are a lot of software available in the market that provide data encryption using AES. One the most well-known software is TrueCrypt, which is a freeware. TrueCrypt also contains a benchmark for testing the performance of the supported encryption algorithms. It would be a good idea to compare out implementation of AES, with the TrueCrypt implementation.

TrueCrypt Encryption and Decryption

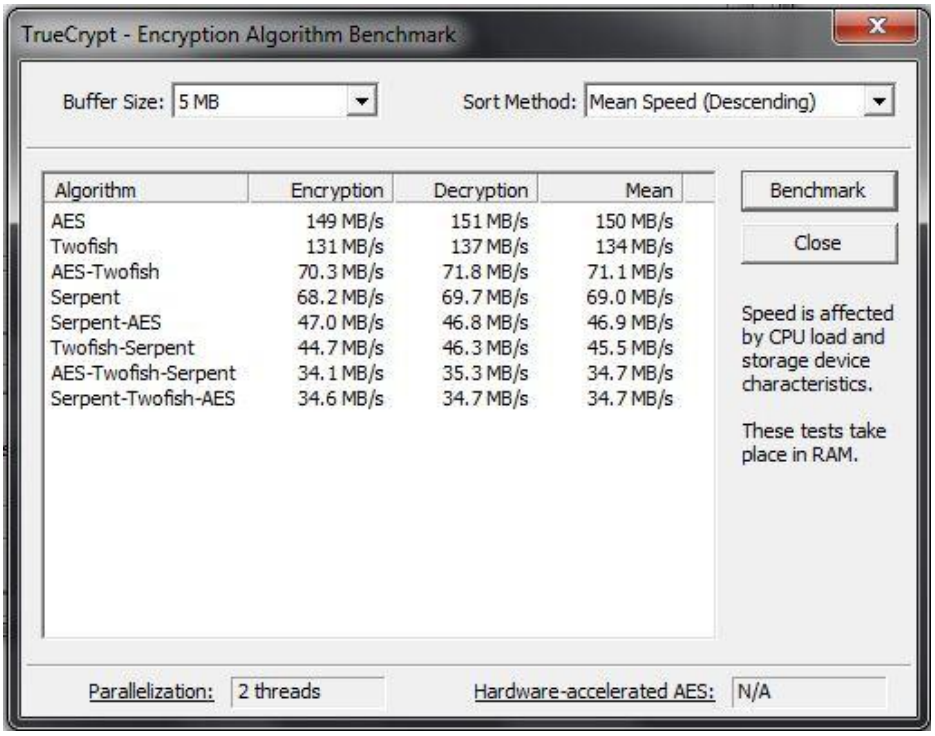

Figure 28 TrueCrypt encryption algorithm benchmark

TrueCrypt uses AES-256 as one of the encryption algorithms. Figure 28 shows a benchmark, which is one the features provided in TrueCrypt. After running the Benchmark, it gives an

average for the speeds of encryption and decryption for the supported algorithms. We can see that the average encryption speed for AES is 149 MB/s, and the decryption speed is 150 MB/s. It is 5 times faster than our implementation of AES. As mentioned in TrueCrypt's website [41], a file container can be created, where encrypted files are stored. When reading/writing the files from the file container, they are encrypted/decrypted on the fly, i.e. portions of data are encrypted/decrypted in RAM, which results in very good encryption performance.

Signing data using RSA Signature Scheme with SHA-512

With key pair generation & saving		Using an existing private key	
Time/sec	Size/MB	Time/sec	Size/MB
0,400	0,01	0,027	0,01
0,290	0,1	0,036	0,1
0,372	1	0,080	1
0,800	10	0,540	10
1,860	30	1,515	30
3,215	60	2,977	60
4,163	80	3,950	80
5,177	100	4,810	100

The above table shows two different tests for signing data, one is with key pair generation and saving, and the other one is with using an existing private key. We can see that, as expected, when using an existing private key, the signing process is performed a little faster.

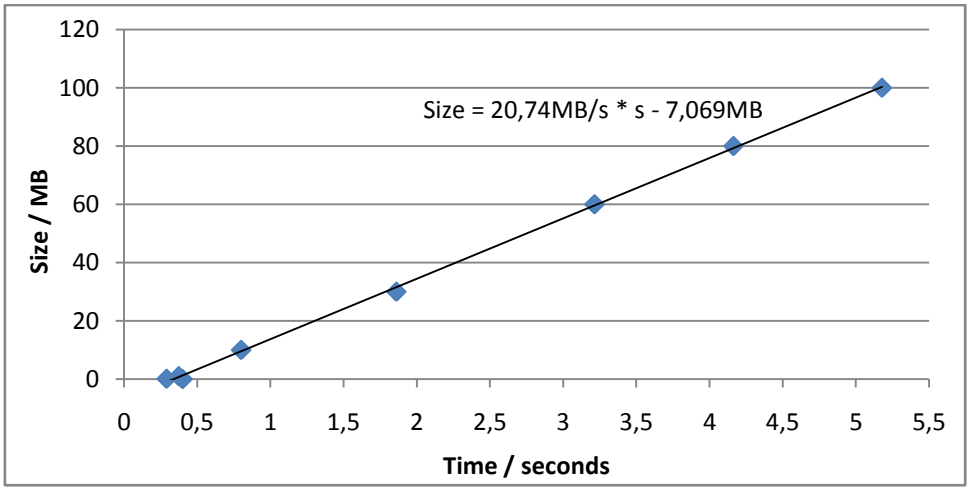

Figure 29 Signing data using RSA with key pair generation, files: 0,01 MB – 100 MB

Figure 29 shows that the average speed of signing data is approx 21 MB/s, which is actually slower than the encryption/decryption process being approx 30 MB/s. We know that it is not the actual data that is signed, but a short digest of data is signed. Since the asymmetric encryption is always much slower than symmetric encryption, it would have taken much more time to sign large files without using a hash function.

Verifying data using RSA

Time/sec	Size/MB
0,083	0,01
0,080	0,1
0,130	1
0,568	10
1,528	30
2,990	60
3,950	80
4,908	100

The above table shows the time it took for various file sizes to be verified using RSA.

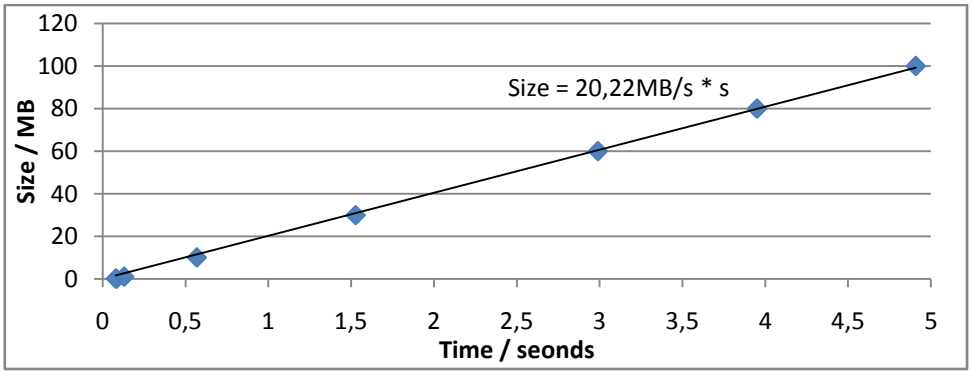

Figure 30 Verifying data using RSA, files: 0,01 MB – 100 MB

We can see in Figure 30 that the average speed for verifying data is approx 20 MB/s. It is almost the same as the speed of signing data.

Until now we have tested storing and retrieving data and found out the chunk size for optimal speed, and moreover we have tested encryption and signature for different file sizes. In the following we will test storing and retrieving data with encryption, signature and data persistency, in order to have an overall average speed of the system.

Storing data to Infinispan with Encryption, Signing and Data Persistency

We found out that when we have data persistency, the optimal speed is achieved by using a chunk size of 100 MB for storing files up to 80 MB.

Chunk size: 100 MB (Optimal for files up to 80 MB)	
Time/sec	**Size/MB**
0,698	0,01
0,750	0,1
0,875	1
1,910	10
4,480	30
8,850	60

The above table shows the taken times for storing data with encryption, signing and data persistency for different file sizes.

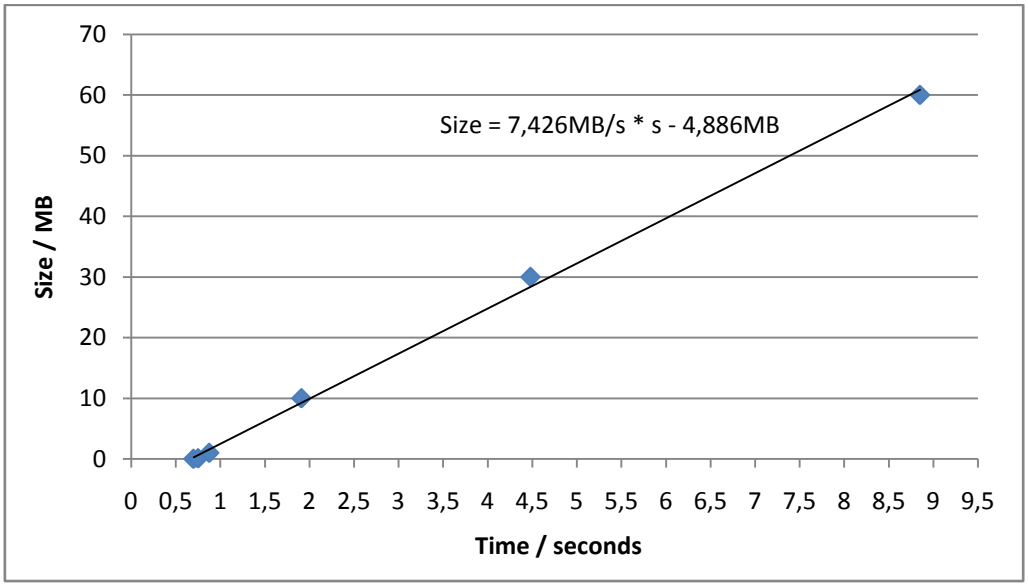

Figure 31 Storing data to Infinispan with Encryption, Signing and data persistency

The overall average speed for storing data to Infinispan with encryption, signing and data persistency is shown in Figure 31, which is approx 7,4 MB/s.

Retrieving data from Infinispan with Decryption, Verifying and Data Persistency

Chunk size: 100 KB (Optimal for files up to 100 MB)	
Time/sec	**Size/MB**
0,103	0,01
0,119	0,1
0,173	1
0,974	10
3,009	30
6,490	60

We found out that when we have data persistency, the optimal speed is achieved by using a chunk size of 100 KB for retrieving files up to 100 MB. The above table shows the taken times for retrieving data with decryption, verification and data persistency for different file sizes.

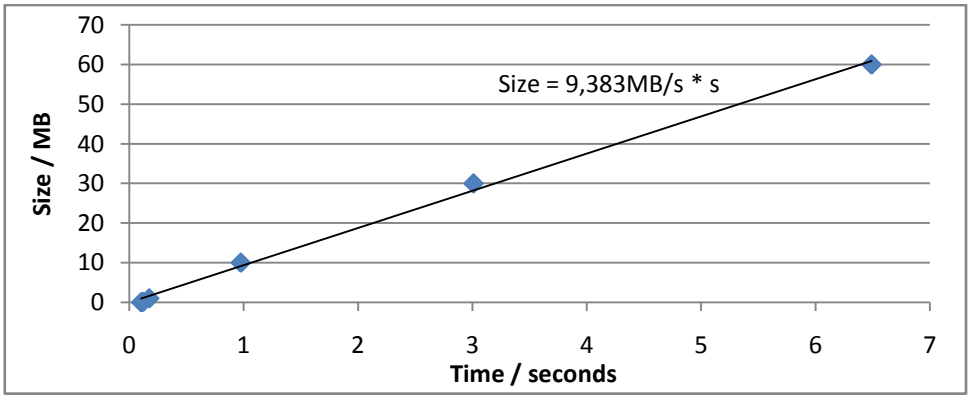

Figure 32 Retrieving data with Decryption, Verification and Data Persistency

The overall average speed for retrieving data from Infinispan with decryption, verification and data persistency is shown in Figure 32, which is approx 9,4 MB/s. All in all the speed of data retrieval is a little faster than the speed of storing data; though they are very close to each other.

6.1.3 Storing and Retrieving data using Infinispan in Distributed Mode

In the previous section, all the results were achieved by running one instance of Infinispan on a local PC. Here we will run Infinispan in distributed mode, i.e. multiple instances of Infinispan running on multiple PCs. Since the process of encryption and digital signature is always performed on the local client, we do not need to test them in this context. So we only test data transfer between Infinispan instances.

In the configuration file, it can be specified whether or not communications should be synchronous. In case of synchronous communication, a timeout can be assigned, which is used to wait for an acknowledgement when making a remote call. After the timeout the call is aborted and an exception is thrown.

For the tests we used three PCs, where an instance of Infinispan was running on each of the PCs, so we had a data grid with three cache nodes. First we tested the data grid in synchronous mode with 15 seconds for the timeout. We tried to store small files, like 10 – 100 KB, to the Infinispan data grid. The time that took for transferring data between cache nodes varied a lot, from 0,5 seconds to 10 seconds, and very rarely it exceeded the timeout limit. In the case of timeout, the transfer was aborted and a “timeout-exception” was thrown. The exception was always thrown by one of the connected clients, and thus the data was not fully transferred to that client. When we tried to store files larger than 100 KB, up to 10 MB, then the timeout limit was often exceeded, and the transfer was failed. We increased the timeout to 150 seconds, but it still occurred once in a while. The Infinispan is designed to have concurrency, i.e.

multiple threads are created to write the data to cache nodes. For example, transaction 1 (T1) performs the operation sequence: *write key1*, *write key2*, and transaction 2 (T2) performs the operation sequence: *write key2*, *write key1*. If T1 and T2 happen at the same time, and they perform their first operation, then they will wait for each other for ever to release own locks on keys. So in practice it is necessary to assign a timeout, and after the timeout one of the transactions will rollback allowing the other to continue. The provider of Infinispan has used the same mechanism, which results in throwing the mentioned exception by one client, and continuation of the other. [42]

As a second scenario, we tested the data grid in asynchronous mode. In this case, the communication is done asynchronously, i.e. whenever a thread sends a message, it does not wait for an acknowledgement to return. When we tried to store files of 10 – 100 KB, they were stored to the data grid, and were available in all three clients. The speed of data transfer varied as in synchronous mode, but as a whole it was better. When we tried to store larger files, 1 MB – 30 MB, the data transfer succeeded more often than before, but about 10% of the time the stored file was not available in one of the clients. So in the asynchronous mode there would be more concurrency, and thus the performance would be increased, because you do not need to wait for a transaction to be completed, but the drawback would be that you would not know whether or not a file is fully transferred to a client.

As a result we found out that the current version of Infinispan is not useful to be utilised as a cloud storage system. The cryptographic access control mechanism has after all achieved better performance results, and thus it is on its way to be used in real-world cloud storage systems. If TrueCrypt has achieved such a high speed in encryption, then it seems that it is possible to improve our system to achieve even better results, and finally make it realistic to be used as a security solution in the market.

6.1.4 Key Management

As we know, granting access permission is done by creating a key ring, which can be uploaded to the Infinispan data grid in order to make it available to the authorised client. Here we test the process of creating a key ring, which is performed solely on the client side.

Read access		Read & Write access	
Number of files	**Time/sec**	**Number of files**	**Time/sec**
1	0,026	1	0,025
2	0,043	2	0,041
5	0,068	5	0,072
8	0,088	8	0,111
12	0,12	12	0,15
20	0,184	20	0,235
25	0,213	25	0,287
30	0,27	30	0,333
35	0,284	35	0,388
40	0,334	40	0,45
50	0,4	50	0,54
60	0,46	60	0,66

For granting read & write access, three keys must be read for every file, after which they must be appended to the key ring, whereas for granting read access, two keys are involved for every file, so it will take a little longer to create a key ring with all three keys. Since each key has the same size for all types of data, the size of data does not have any affect in this context. The two first columns in the above table shows the times it took to create key ring for assigning read access to different number of files. The third and fourth columns contain the same data regarding read & write access. As expected, we can see that the times in the column related to read access are a little less.

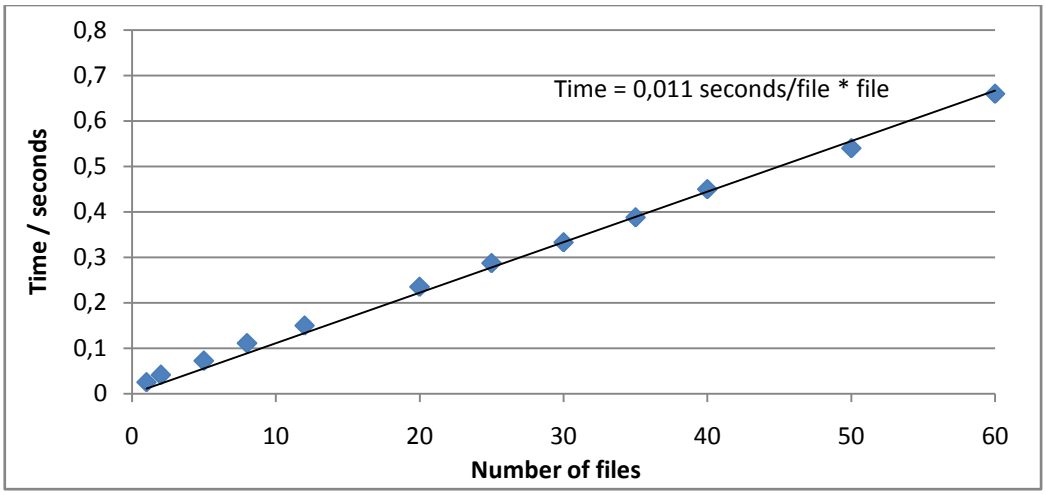

Figure 33 Assigning Read & Write access to files

Figure 33 shows a graph for creating key ring when assigning read & write access to a file. We can see that in average it takes about 11 milliseconds per file to create a key ring. For example, if we select 100 files from the data grid, and assign read & write access to them by creating a key ring, then it will approximately take 11*100 = 1100 milliseconds, or 1,1 second to create it. So as a result the process of key ring creation is reasonably fast.

6.2 Security Evaluation

As discussed in chapter 2, the available security solutions for the well-known cloud storage systems are server centric. There are many providers, who offer cryptography in their cloud storage systems, but the encryption and decryption processes are performed on the server. Moreover they do not support any kind of trustworthy data integrity mechanisms. As we have mentioned before, in cryptographic access control mechanism, all the security operations are performed on the client side, and therefore it gives more security and control to user's data. The data can freely be available on the cloud, and anyone can download it. The only action an unauthorised user can perform is to verify the integrity of the data, because the public key can also be freely available. Since the data is encrypted with the most powerful encryption algorithm, it is practically impossible to decrypt it without the symmetric key. If a key length of 128 bits or longer is used, a brute force attack would not be successful with the current computer technology.

Another action that a malicious user can perform is to modify the data without knowing the content of it, but because the system ensures integrity of data, the authorised users would know that the data has been modified by an intruder whenever they retrieve the data. Here in our system we use Infinispan data grid, which does not support verification of data on the server side, and therefore the data can be updated by everyone, but the authorised users would always detect an unauthorised modification when he downloads the data. By using a cloud storage system that supports verification on the server, an intruder would not be able to update the stored data without the use of the correct private key. In such a system, whenever the modified data is uploaded, the server would use the corresponding public key to verify the signature, and if the data is not signed with the proper private key, the verification fails, and thus the data is not updated.

To sum up, we can say that the attacks on the actual data without the presence of corresponding keys would not be successful. If we assume that the keys are kept secret and are not accessible in any way, then the confidentiality and integrity of data are guaranteed.

However other kinds of attacks that can be performed on network systems are also applicable in cloud storage systems, even with the presence of cryptographic access control. In the following we mention some of the well-known attacks that threaten almost all network systems.

6.2.1 DoS Attack

Denial-of-service (DoS) attack has always been a threat to the network systems. The main principle in DoS attack is to send a lot of requests to a service provider occupying most of its resources, such that it cannot respond to legitimate traffic. A more advanced type of this attack is called distributed DoS (DDoS) attack, which uses many systems to attack a single system. In this case there would be more than one victim, namely the target, and all the other systems that are used for this purpose. The attack is mostly targeted towards the web servers, but as the use of cloud computing systems increases, this attack also threatens the cloud services. Because of the elasticity of the cloud computing systems, the attack mostly affects the users. If a communication between a user and the service is a victim of the attack, and as a result the service is not available for the user, then the cloud can just provide more resources to make the service available. In this case the user has to pay for both the resources he has used and for the resources that was used by the attack.

As other services in the cloud can be victims of this attack, the cloud storage system would not be an exception. An attacker in this case can transfer a lot of files to a server node using DDoS, which would cause the node to become busy, and as a result data transfer by legitimate users would not be possible, or become very slow on that node. In the case of data retrieval, if the cloud storage system supports replicated data, like the Infinispan data grid, then there would not be a problem in accessing data, but the choice of replication would have a high cost for a user, because he will use much more space than the size of his data. In practice users would like to avoid high costs, so they would only require the space needed to store their data. In such a situation their data would be unavailable when the server node is attacked. However those servers, who support some kind of backup for the stored files as default, makes it

possible for the user to access his data anyway, but if the attack is widely spread in the cloud storage system, then it can significantly decrease the data availability. All in all the affect on availability of data depends on the extent of a DDoS attack, and the power of services provided by the cloud to avoid the attack.

In the case of storing data to the cloud, there will not be such a big problem for a user when a server node is attacked. As we know, the cloud computing systems are built in a way that they provide services in a virtual manner. When a user buys a specific storage space in the cloud, he gets a virtual storage space, which means that his data are not necessarily stored to one server node. So whenever a server node is not available, which can be because of a DDoS attack, it will still be possible for a user to store his data, because the cloud storage facility would just provide another server node for the user, where he can store his data.

It is worth mentioning that users usually use applications as services provided by the cloud when storing and retrieving data, and if these services are targeted by DDoS attack, then the cloud computing system would provide more resources to make the services available. This leads to more cost for the user as we mentioned in the beginning of this section. So as a result, whenever DDoS attack occurs in the cloud, the user interactions in the attacked part would often be affected, which leads to typical problems like high cost, data unavailability, etc.

6.2.2 Man-in-the-middle Attack

We discussed about DDoS attack, which is mostly related to data availability in the cloud storage. It is not a threat against data confidentiality and integrity, and therefore the attack is not applicable on the cryptographic access control mechanism. Another type of attack, which is related to cryptographic systems, is man-in-the-middle attack. We shortly discuss about whether or not this attack can be successful on our system.

When storing and retrieving files are performed, there are two endpoints involved in the process, namely the client and the server. The client gives his public key to the server, which is used to verify the future updates to the file. Except the public key, no other keys are provided, so the only possible malicious action that the server can do, is to allow unauthorised modification of data. In such a situation the user would still know that the data is modified when he retrieves it back from the server. The man-in-the-middle attack is only applicable when two users want to exchange their symmetric and private keys with each other in order to grant access permission to a file or a key ring. Here an intruder can interfere in the communication in order to get the keys. The key exchange is performed outside the whole system, so the server is not involved in it. Users can use a secure channel that protects against man-in-the-middle attack to exchange keys, or a 100% secure way would be to exchange the keys by meeting each other in person. The system does not support symmetric and private key exchange, so it is the user's responsibility to distribute his keys in a secure way. Apart from that, man-in-the-middle attack is not applicable in the system.

6.2.3 Traffic Analysis

Traffic analysis is an attack that is applicable in the systems, even with the presence of cryptography, because one of the purposes of this attack is to watch how the sizes of files are

changed, and by that the attacker can estimate the number of updates that have been performed on the files. By using traffic analysis one can also know which users have shared their data with each other. This knowledge can be achieved by watching which users make updates to a single file.

By only looking at the stored file and downloading it, one cannot know from where the file is uploaded, but by watching the traffic, an intruder may find the location of the client. It enables the intruder to perform his attack on the client in order to access the symmetric and private keys. To prevent this, users can use “the onion routing” (Tor) mechanism, which makes it possible to have anonymity when interacting with network systems. Moreover users can install firewall on the client side, which also hinders unauthorised access.

6.3 Further Improvements

During the implementation and testing, we have noticed a number of improvements that can be added to the prototype and the Infinispan. The Infinispan data grid only supports replicated clustering, which means that the stored data would be available on all the connected nodes. The provider claims that it also supports distributed clustering. In distributed clustering a limited number of cache nodes can be chosen for every data, and thus there would be more space available for storage. When we tried to use it in our system, the Infinispan could not load data from local disk, but an exception was thrown instead. One of the most important improvements in Infinispan would be debugging this error, and fully enabling the support of distributed clustering. Moreover the use of concurrency in the data grid should be reconsidered in order to prevent deadlocks. Adding these improvements would move the Infinispan a step closer to be used in real world cloud storage systems.

For the prototype, the file system can be improved to become more user-friendly, and also more actions can be added to the GUI like creating, renaming and copying directories and files. In short, its usability can be improved by adding the functionalities of a commonly used file system, such as in Windows or Linux OS.

Besides improvements in the file system, the encryption and digital signature processes can be improved in terms of performance. In our system we have chosen to use the Java implementation of the algorithms, but the comparison between our system and TrueCrypt showed that TrueCrypt has used a very efficient way of implementation for the encryption and decryption algorithms. So it would be worth one’s while to improve the system by implementing the symmetric and asymmetric algorithms in more efficient way, and if necessary, the use of another programming language can be considered.

Some users would like to have freedom in terms of choosing between different algorithms for encryption and digital signature, and also the length of the keys. It would be useful to provide a number of most secure algorithms, and the different key lengths for the users to choose from.

Chapter 7

Conclusion

In this project a solution for data confidentiality and integrity in cloud storage systems is examined. The available solutions in the market are studied, whereupon the possibilities for a solution based on cryptography is analysed, and as a result the cryptographic access control mechanism is proposed. In this security solution, cryptography is used to ensure confidentiality and integrity of the stored data. The main quality in this solution is that the security operations are performed at the client side, and thus the users do not need to trust the cloud servers. The only elements that make it possible to access the stored data are the corresponding keys, and thus file sharing between users can only happen by exchanging keys.

On the basis of the analysis, a design is developed for a system that should have the mentioned properties. According to this design, a prototype is implemented to demonstrate the security solution in practice.

This document starts with a description of the core element in the system, namely cryptography, followed by a discussion and comparison of the available security solutions for well-known cloud storage systems. This exploration shows that there is not any standard or a common agreement regarding the security solutions in the cloud. Various solutions are used by different providers, usually in a hybrid way. For instance, Amazon uses ACLs (Access Control Lists) as an access control mechanism, and SSL channel for data transfer, and besides they also use cryptography for data confidentiality. Most well-known cloud computing providers like Amazon, Google and Microsoft use security solutions at the server side. However there are some providers, such as Carbonite, Mozy, etc. that offers symmetric encryption at the client side, but what is common for most of the providers is that the presence of a trustworthy data integrity mechanism is missing. In contrast to the available security solutions, cryptographic access control ensures confidentiality and integrity of data at the client side.

In order to demonstrate the cryptographic access control mechanism, we needed a cloud storage system to apply our solution to it. For this purpose, we used Infinispan, which is an open source in-memory data grid platform that can be used to build a cloud storage system using multiple Infinispan cache nodes connected to each other. Infinispan provides primarily an in-memory storage, but it can also be configured to support data persistency. The prototype therefore contains two main parts, namely the cryptographic access control mechanism and its integration with the Infinispan data grid.

We have accomplished an evaluation of the system, which gave us an idea about whether or not the system is usable in real-world. The evaluation process can be divided in three main parts: Performance evaluation of the Infinispan data grid, performance evaluation of the cryptographic access control mechanism and security evaluation of the whole system.

First of all, the Infinispan data grid with enabled data persistency was run in local mode, i.e. running one instance of Infinispan on a local machine. The results showed that the performance of file transfer could be considered as being acceptable compared to the actions performed locally in the file system of Windows 7 OS. However Infinispan was about 3 times slower than Windows file system operations, but since it is implemented in Java, the result was not surprising. A more interesting situation was when we run Infinispan in distributed mode, i.e. running multiple instances of Infinispan cache in different machines. When we tested it in distributed mode, the performance was decreased remarkably, and sometimes the data transfer did not succeed in some of the cache nodes. We found out that this is because of the concurrency, which is used in writing data to the cache nodes. In this case a deadlock occurs, and after a predefined timeout, the operation is interrupted. As a result, in the current situation of Infinispan, it is not suitable to be used in practice.

The result of the test regarding the performance of cryptographic access control mechanism showed that actually this solution is on its way to be usable in practice. We compared the encryption process with a well-known and widely used software, called TrueCrypt. However the implementation of TrueCrypt was about 5 times faster than ours, but after some further revisions and developments, it can surely be used as a practical security solution for the cloud storage systems.

Regarding the security of cryptographic access control, we should mention that the only possible way to read or write the data is to have the corresponding keys, and the keys are only available on the authorised client's machine. (It is of course assumed that the client's machine is immune against unauthorised access.) On the other hand, the algorithms used for encryption and digital signature are fully tested and approved over many years, and as a result they are practically unbreakable with the current technology. Ergo we have to admit that the solution primarily guarantees data confidentiality and integrity.

There are many other threats against the security of network systems, and the cloud computing system is not an exception in this context. The well-known attacks are discussed in this document, which are DoS (Daniel-of-service) attack, man-in-the-middle attack and traffic analysis. Many of these kinds of attacks are mostly targeted towards the data availability and communication, but they do not threaten the cryptographic access control mechanism directly. However traffic analysis can be used to trace the client, and thus get access to the keys, but if clients use firewalls and other mechanisms that enable them to have anonymous interactions with the cloud storage, this threaten would be diminished.

All in all the system works as expected according to the results achieved by functional tests stated in appendix A. However storing data to Infinispan data grid in distributed mode is not

fully consistent with what is expected, but it is a shortcoming in the Infinispan implementation. Considering and applying a solution to this issue would be out the scope of this project.

APPENDIX A

Functional Test

The following table contains the list of tests that have been performed on the system. Every row contains a number for the test, a description of the test, the expected outcome, and a comparison between the test and its result. If the result of a test is as expected, it is indicated with an “OK” in the last column. If a place in the last column does not contain “OK”, it indicates that the test outcome is not as expected.

#	Test description	Expected outcome	Result
1	Starting Infinispan.	Infinispan starts without any exceptions.	OK
2	Closing Infinispan.	Infinispan closes without any exceptions.	OK
3	Starting Infinispan cache.	The configuration file is loaded, the cache starts, and the configuration file is shown in the configuration text area at the starting screen of the GUI. All tabbed panes of the GUI are enabled.	OK
4	Closing Infinispan cache.	The cache closes, and the two tabbed panes, “Cluster View” and “Manipulate Data” are disabled.	OK
5	Starting two or more instances of Infinispan caches in machines connected to the same local network.	The started Infinispan caches connect with each other. A list of connected caches is shown in the Cluster View tab in every instance of the running caches.	OK
6	Storing a file to one instance of Infinispan running in local mode, without data persistency.	The file is read from a specified path in the local disk. The read file is encrypted, signed and stored to a specified path in the cache.	OK
7	Storing a file to one instance of Infinispan running in local mode, with data persistency.	The file is read from a specified path in the local disk. The read file is encrypted, signed and stored to a specified path in the cache. The file is also stored to the specified path in the local disk in encrypted and signed state.	OK
8	Retrieving a file from one instance of Infinispan running in local mode, without data persistency.	If the client has read access, the file is read from the cache, decrypted, verified and saved to the specified path in the disk.	OK

9	Retrieving a file from one instance of Infinispan running in local mode, with data persistency.	If the client has read access, the file is read from the cache, decrypted, verified and saved to the specified path in the disk.	OK
10	Removing file(s) from one instance of Infinispan running in local mode, without data persistency.	The selected file(s) are removed from the cache.	OK
11	Removing file(s) from one instance of Infinispan running in local mode, with data persistency.	The selected file(s) are removed from the cache. The corresponding file(s) on local disk are also removed.	OK
12	Storing a file to the Infinispan data grid running in distributed mode, without data persistency.	The file is read from a specified path in the local disk. The read file is encrypted, signed and stored to a specified path in the data grid.	(OK)*
13	Storing a file to the Infinispan data grid running in distributed mode, with data persistency.	The file is read from a specified path in the local disk. The read file is encrypted, signed and stored to a specified path in the data grid. The file is also stored to the disk of every machine, where instances of Infinispan caches run.	(OK)*
14	Retrieving a file from the Infinispan data grid running in distributed mode, without data persistency.	If the client has read access, the file is read from the data grid, decrypted, verified and saved to the specified path in the disk.	OK
15	Retrieving a file from the Infinispan data grid running in distributed mode, with data persistency.	If the client has read access, the file is read from the data grid, decrypted, verified and saved to the specified path in the disk.	OK
16	Removing file(s) from the Infinispan data grid running in distributed mode, without data persistency.	If the client has all the three key files belonging to the selected file(s), the selected file(s) are removed from the Infinispan data grid.	OK
17	Removing file(s) from the Infinispan data grid running in distributed mode, with data persistency.	If the client has all the three key files belonging to the selected file(s), the selected file(s) are removed from the Infinispan data grid and from the local disks, where the instances of Infinispan run.	OK
18	Key generation.	During the process of storing data to cache, the three keys, symmetric, private and public keys are created, and saved to a predefined location in the disk.	OK
19	Creating key rings.	If “Read Access” is chosen, a key ring is created containing the symmetric and public keys belonging to the chosen file(s). If “Read & Write Access” is	OK

		chosen, a key ring is created containing the symmetric, private and public keys for the chosen file(s).	
20	Updating a existing key ring.	If “Read Access” is chosen, the symmetric and public keys belonging to the chosen file(s) are appended to the key ring. If “Read & Write Access” is chosen, the symmetric, private and public keys belonging to the chosen file(s) are appended to the key ring.	OK
21	Retrieving a key ring from the Infinispan data grid.	If the client has read access to the key ring, the keys in the key ring are extracted and saved to the predefined location in the local disk.	OK

* Because Infinispan uses concurrency in distributed mode, deadlock can happen. Depending on the duration of the given timeout, after this timeout, a timeout exception is thrown, and the process of writing is interrupted. It happens more often for files larger than 1 MB.

APPENDIX B

Introduction to Infinispan Data Grid

The report for the 3-week course, which was completed alongside this project, is stated here. The prototype implemented in this project is based on the outcome of this 3-week course.

Introduction to Infinispan Data Grid

Technical University of Denmark
Department of Informatics and Mathematical Modelling
24-02-2012

Author:

s062422 Abbas Amini

Introduction

Infinispan is an open source in-memory data grid platform written in Java. It is quite new and still under development. Infinispan makes use of a key-value structured data storage system, and thus it provides high availability of data.

Infinispan is primarily in-memory data grid, i.e. it uses caches to provide memory, but it also supports data persistency, i.e. it can be configured with cache stores in order to store data in a persistent location on the disk. It works in such a way that a number of instances of Infinispan can be created in different machines, and these instances can be connected with each other forming a peer-to-peer network of nodes. Then we can use the data grid as a data store for storing our data.

In this project a data grid is created using Infinispan library. Moreover a file system is created for the data grid, which is used for storing, retrieving and removing the data. Also a GUI is provided to make it possible for users to interact with the system. The goal of the project is to introduce Infinispan, and to show that it provides a data grid with high availability of data.

In the first part of this report the overall structure of the program is described. Here an overview of the Infinispan data grid is created. Moreover the structure of the GUI is also described. In the next part some implementation details about the file system and the data grid is briefly explained. Finally a conclusion is provided, where the result of the project is evaluated.

Overall Structure

The project consists of two packages, `gridFileSystem` and `infinispan`. The package `gridFileSystem` contains four classes from the Infinispan library. These four classes are used to create a file system for the cache. A little modification was necessary in one of these four classes, namely `GridFile`, therefore they are imported in the project folder instead of just having reference to the library. The modification is regarding the path separator used in the grid file system. It is defined according to Linux-systems, which is '/'. So in order to make the file system compatible for Windows, the path separator is changed to '\' in the class `GridFile`.

The main part of the project is the content of the package `infinispan`. This package contains five classes. The classes `MainView`, `StoreToCachePopupWindow` and `Controller` are responsible for creating the GUI and adding different functionality to it, i.e. adding listeners to the relevant parts of the GUI. The class `HandleCache` is responsible for handling the infinispan cache, such as starting/stopping the cache, storing data to the cache, retrieving data from the cache, removing data from the cache and clearing the cache. Finally the class `Start` contains the `main` method, from where the program starts to run.

The infinispan cache is designed to be configurable. The configuration of the cache can be done in an xml file. In this file several properties of the cache are defined, among other things type of the cache, the persistency of data, how cache nodes must discover each other, etc. In

this project the file `config-file.xml` contains the relevant configurations for the cache. More details in the “Implementation” section.

Infinispan Data Grid

As mentioned earlier the infinispan cache is mainly used to form a cluster of cache nodes, such that every instance of the cache is running on a separate machine. They form a peer to peer network and share data with each other. The cache is configured to have data persistency, so it stores the data to a predefined location in hard disk. Figure 34 shows an overall structure of the system.

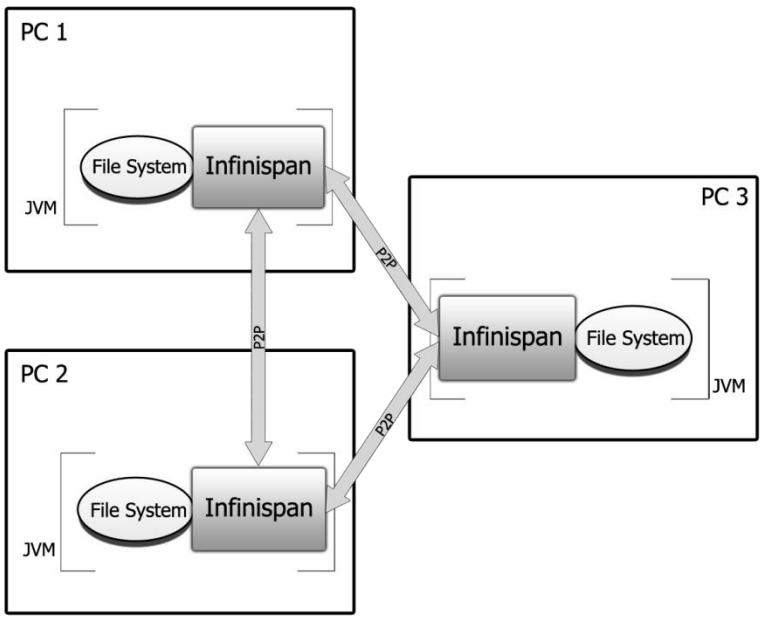

Figure 34: An overview of Infinispan data grid

In Figure 34 an example is shown, where the three instances of Infinispan and the corresponding grid file system are running separately on three machines. The Infinispan cache can be accessed via the file system. Data can be stored to cache, retrieved and removed from the cache through the file system. Whenever some data is stored to the cache in one of the machines, it is instantly available on other machines too. Since the caches are configured to be persistent, the data is also stored to the disk. If all cache nodes are shut down and started again, the data would be loaded back to the caches when started. Actually there are two caches running on every machine, one is for storing the data and the other is for storing the metadata, so every of the Infinispan in the figure contains a pair of caches.

As shown in the figure, this usage of Infinispan is a peer-to-peer embedded mode usage, i.e. every instance of Infinispan and the file system run in separate JVMs, and the Infinispan caches discover each other via a peer-to-peer connection. In contrast to that there is the client/server mode, where only Infinispan instances run in their JVMs, and each of them open a socket and listens to it. Then our application can talk to these running Infinispan caches over the sockets.

It is worth mentioning that the data grid can be expanded by adding more nodes, and in the same way as shown above, they will discover each other and form a bigger data grid.

GUI Structure

This section contains screen shots of the program followed by descriptions of them. Here the goal is to get an overview of the system, but the implementation details would be discussed later.

When we download the Infinispan library package, a demo program, called “Infinispan GUI Demo”, is also included in the package. Our GUI is based on this demo program, but many changes have been made both on the actual GUI and on the way the cache is handled. Moreover the file system for the cache is also added.

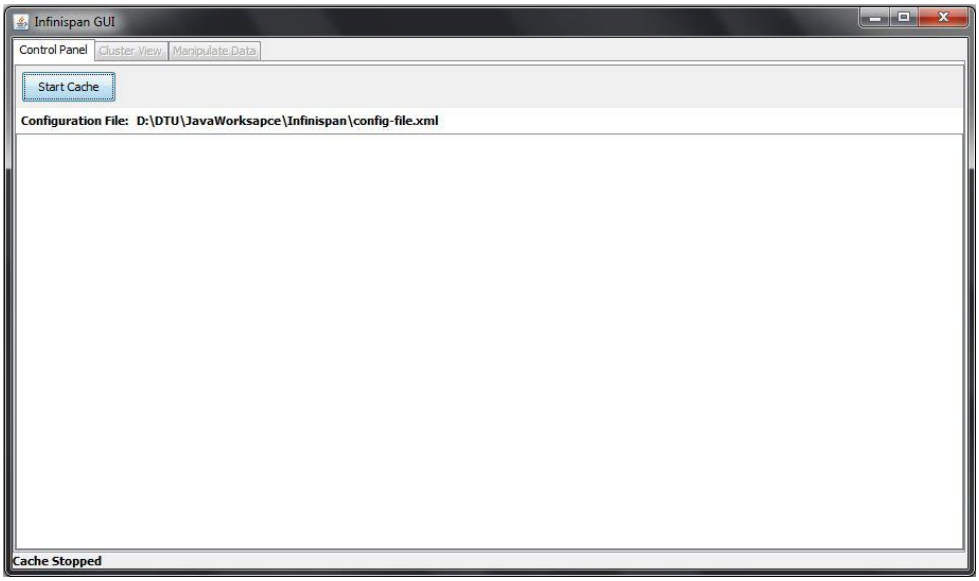

Figure 35: Control panel tab, before starting the cache

Figure 35 shows the first screen just after starting the program. We can see that in the control panel tab we can start the cache by clicking the “Start Cache” button. When the cache is not running the two other tabs “Cluster View” and “Manipulate Data” are deactivated.

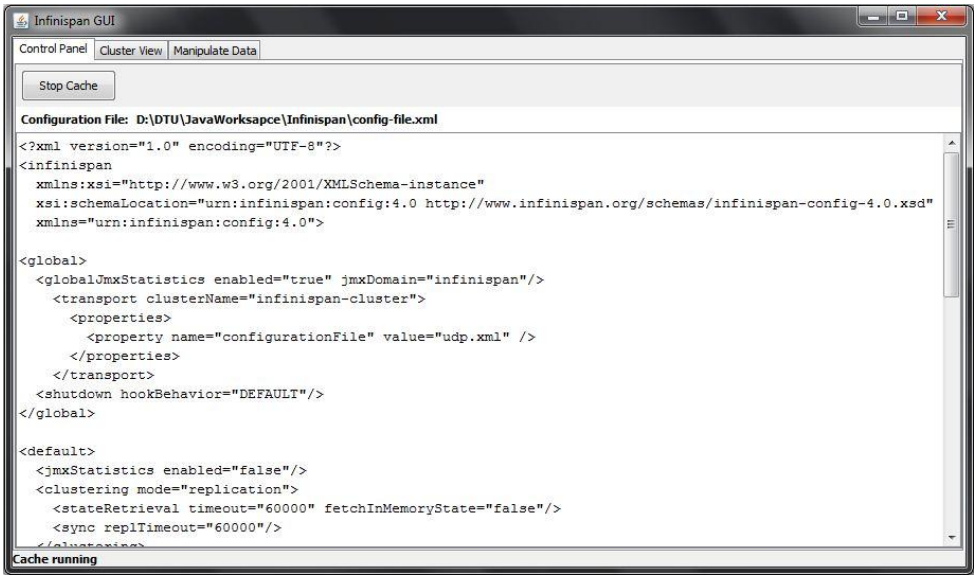

Figure 36: Control panel tab, after starting the cache

Figure 36 shows that the cache is running, and the cache configuration file is shown in the text field area. When we start the cache, it means that there are actually two caches that are running, one for the data, and the other for metadata.

We can also see that the two other tabs are activated. The cache can be stopped by clicking on “Stop Cache” button.

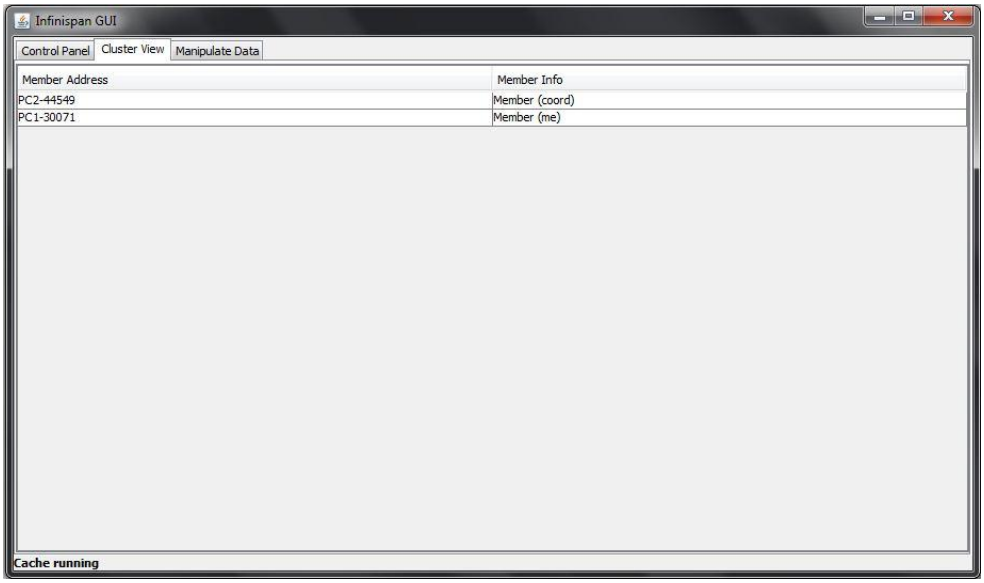

Figure 37: Cluster View tab

Figure 37 shows the Cluster View tab, where we can see the list of Infinispan cache members. The first column shows the member address. The caches are running on two different machines. The second column shows the member info. “PC2” is the coordinator, i.e. it has started an instance of the cache firstly. “PC1” has started another instance of the cache afterwards, and they have discovered each other and formed a data grid.

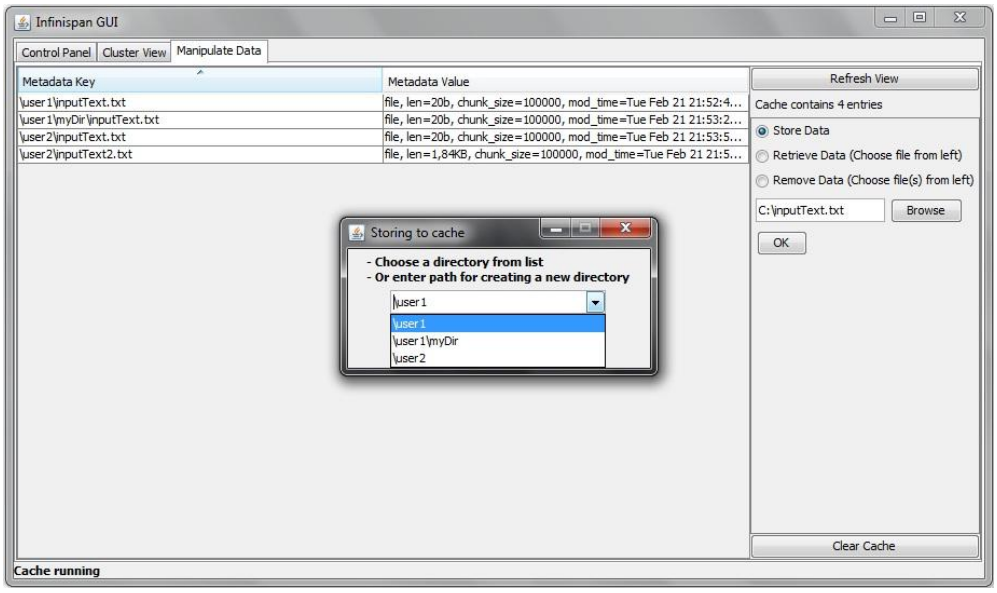

Figure 38: Manipulate Data tab

Figure 38 shows the Manipulate Data tab. This tab contains a primitive file system. The table to the left contains a list of data that has been stored to the grid. This list is actually the content of metadata cache. The metadata cache stores the path of the stored data as key and the metadata information about the stored data as value. So the first column shows in which directory the files are stored, and the second column shows information about the data, such as the size of the data, the chunk size and the last modification time. When we store data to the grid, they are divided in chunks, so “chunk_size” is simply the size of every chunk of data.

At the right side of this tab the “Refresh View” button refreshes the data table. This button is used when another cache node joins the grid. When we click on this button, the files stored to the newly joined cache node are instantly available in the table. If the radio button “Store Data” is selected, it is possible to browse a file from the local machine, and by clicking the “OK” button a popup window appears, where it is possible to choose an existing folder or create a new folder for the data to be stored to the grid. If the radio button “Retrieve Data” is selected, then we have to click on a file in the table. Then by clicking on the “OK” button a file chooser window will be opened in order to specify where the data must be saved. If the radio button “Remove Data” is selected, then it is possible to select one or more files from the data table to be removed from the grid. Finally the button “Clear Cache” is used to remove all the data from the grid at once. It is worth mentioning that whenever we manipulate data in the grid, the data stored to the disk is also simultaneously manipulated, because as mentioned the caches are configured to have data persistency.

Implementation

Until now we have got an overview of the system. In this section we mention some of the important implementation details briefly.

Infinispan Grid File System

Here we describe the process of using the grid file system to store data to the grid and retrieve and remove data from the grid.

Grid File System is a new API added to Infinispan in version 4.1.0, and is available from this version onwards. This API exposes an Infinispan-backed data grid as a file system. It consists of four classes. Three of them, `GridFile`, `GridInputStream` and `GridOutputStream` are extensions of JDK's three classes, `File`, `InputStream` and `OutPutStream` respectively. The fourth class is a helper class, called `GridFilesystem`.

`GridFilesystem` includes two caches, one for actual data and the other for metadata information. In order to make use of the memory evenly, data is stored in chunks with a defined size. For example, we have 4 cashes, which are 2 GB each, and as a result we have 8 GB of memory. If we store some few large files, say 500 MB, without dividing them in chunks, then some caches would be filled and others would be free. Another problem is that we will not be able to store, for example, a 2.5 GB file, even though we have a total of 8 GB memory, because 2.5 GB is greater than the total memory of each cache. So in order to have an even distribution of data, they are stored in chunks of small sizes.

`GridFilesystem` makes use of two created and running Infinispan caches. The following code excerpt is from the method `statCache(…)` in the class `HandleCache`:

```
Cache<String,byte[]> data;
Cache<String,GridFile.Metadata> metadata;
GridFilesystem gfs;

data = cacheManager.getCache("CacheStoreReplData");
metadata = cacheManager.getCache("CacheStoreReplMetadata");
gfs = new GridFilesystem(data, metadata, 100000);
```

We can see in the above code excerpt that one cache is used for data, and the other is used for metadata. The actual data is stored in chunks as byte arrays. The chunk size can be given as a parameter for the `GridFilesystem` constructor, but if not given, the default size would be used. The names of the two caches are given as parameters for the method `getCache()`. We will discuss about the details regarding the names of the caches later.

Two major functionalities of the grid file system are copying data to the grid, and reading data from the grid. The following is a code excerpt from the method `storeData(…)` in the class `HandleCache` showing how to write data to the grid:

```
public void storeData(String dirPath, String filePath) {
   …
   FileInputStream in;
   OutputStream out;
```

```
    …
    in = new FileInputStream(filePath);
    out = gfs.getOutput(dirPath+"\\"+fileName);
    byte[] buffer = new byte[20000];
    int len;
    while((len = in.read(buffer, 0, buffer.length)) != -1)
      out.write(buffer, 0, len);
    in.close();
    out.close();
    …
}
```

Here we can see that a file is read from the local file system, with the specified path, `filePath`. The method `getOutput()` creates an instance of `GridOutputStream` and it can be used to write into the grid to a specified directory, `dirPath`.

In a similar way the data can be retrieved from the grid by calling the method `getInput()`, which is defined in the class `GridFileSystem`. The method `getInput()` creates an instance of `GridInputStream`. The file is then stored locally after specifying the directory path.

The main methods in `GridFilesystem` are `getOutput()`, `getInput()` and `getFile()`. The method `getFile()` is used to create a new file, list files, or create a new directory. Due to this functionality of `getFile()`, it is called in the methods `getOutput()` and `getInput()`.

The class `GridFile` overrides most, but not all, of the methods in the class `File`. So it contains methods for creating a new file, creating a directory, getting a path, deleting a file or directory, listing files, etc. [I],[II]

Removing data from the grid is done by calling the method `remove()` on the running caches, i.e. `data.remove(<key>)` and `metadata.remove(<key>)`. By giving the key as parameter for this method, we specify which data and its corresponding metadata must be removed. For removing the whole content of caches, the method `clear()` is used. Since this method clears all files at once, it does not take any parameter.

Cache Configuration

The configuration of a cache is defined in an xml file, called `config-file.xml`. Whenever the cache starts, its configuration is set using the xml file. The file can obviously be manipulated if anyone wants to change the configuration of the cache.

Here we mention the important parts of the configuration file.

The configuration file has the following overall structure:

```
<infinispa>
  <global>
  </global>
```

```
   <default>
   </default>

   <namedCache>
   </namedCache>
</infinispan>
```

In the element `<global>` many properties for the data grid can be set. Here we mention the property that is relevant for our data grid. The following is the content of `<global>` from the file `config-file.xml`:

```
<global>
  <globalJmxStatistics enabled="true" jmxDomain="infinispan"/>
    <transport clusterName="infinispan-cluster">
      <properties>
        <property name="configurationFile" value="udp.xml"/>
      </properties>
    </transport>
  <shutdown hookBehavior="DEFAULT"/>
</global>
```

Here in the element `<transport>` the name of the cluster is set, so it is important that every cache node in the grid must have the same cluster name in order to be able to discover each other. The element `<transport>` also configures the communication that should be performed across the cluster, i.e. between cache nodes. In the element `<properties>` the transport property is set to be taken from another xml file "`udp.xml`". This xml file is included in the Infinispan library, and it configures the communication to be performed using UDP protocol both for transport of data and for the discovery of new joined cache nodes. Other configurations for the communication are also available in the library, for instance using TCP protocol instead of UDP. Here we use UDP, because it is more suitable for the "replicated" caches [III]. More about replication and other types of caches will be discussed a bit later.

In the configuration file another element is `<default>`. Here the cache settings can be configured, and if the element `<namedCache>` is not set or used by the cache, then these default settings are used instead.

The element `<namedCache>` configures all the settings for the cache. There are many settings that can be configured in this element, but we will mention the relevant items here. One of the important settings for the cache is whether its clustering mode should be "replicated" or "distributed":

```
<namedCache name="CacheStoreReplData">
   <clustering mode="repl">
   …
   </clustering>
   …
</namedCache>
```

In the above excerpt we can see that the clustering mode is set to “replicated”. The difference between these two modes is that in replicated mode whenever we store data to one cache node, it will be available in all the other cache nodes in the data grid, such that every cache node would have a copy of the file. This mode provides high availability of data, and is mostly suited for small files. In contrast to that, if distributed clustering mode is enabled, then the data would be distributed in the gird, such that every file would only be available in two cache nodes. This is the default setting, but the number of caches can be set to more than two. For large amount of data, it is more suitable to use distributed mode. If we use distributed mode, the cache fails to load the data from the hard disk when restarted. This is due to some problems in the Infinispan library, which has not been resolved yet.

Another important configuration is the data persistency, which is defined in the element `<loader>` inside the `<namedCache>` element. Here the location for storing data is also given. The following shows the content of `<loader>` element:

```
<loader fetchPersistentState="true" … >
  <properties>
    <property name="location" value="C:\store"/>
  </properties>
</loader>
```

The above configuration is for the data cache, and in the same way configurations are set for the metadata cache. A complete documentation on the configuration mechanism is available in the Infinispan website [IV].

The method `startCache(URL configFileSource)` in the class `HandleCache` starts the cache. It takes the URL for the configuration file as its parameter. The following code excerpt from this method shows how to start the cache:

```
  cacheManager = new
DefaultCacheManager(configFileSource.openStream());
  data = cacheManager.getCache("CacheStoreReplData");
  metadata = cacheManager.getCache("CacheStoreReplMetadata");
  gfs = new GridFilesystem(data, metadata, 100000);

  data.start();
  metadata.start();
```

We can see that the configuration file is read when we initialise the cache manager, and then the two caches are initialised. The method `getCache(…)` takes the name of the cache as its parameter. The name is specified in the configuration file in the element `<namedCache>`. In this way every cache gets its property from the configuration file.

Conclusion

The program has been tested functionally by trying the different functions manually. The tests show that the program works as expected. However there is room for more improvements. Currently the data grid does not support storage for large data, because of the replicated clustering mode. The cache gets out of memory when the data is larger than the size of the cache. This issue can be resolved by using the distributed mode, but since distributed mode has problems, the first step would be to fix this bug in the infinispan library, which would be a time consuming task.

References

[I] Infinispan's GridFileSystem - An In-Memory Grid File System, http://www.infoq.com/articles/infinispan-gridfs

[II] Grid File System, https://docs.jboss.org/author/display/ISPN/Grid+File+System

[III] Clustered Configuration QuickStart, https://docs.jboss.org/author/display/ISPN/Clustered+Configuration+QuickStart

[IV] Infinispan configuration options, http://docs.jboss.org/infinispan/4.0/apidocs/config.html#ce_default_clustering

Bibliography

[1] Charles P. Pfleeger, *Security in Computing,* Fourth Edition, Pfleeger Consulting Group, Shari Lawrence Pfleeger - RAND Corporation, Prentice Hall, 2006

[2] Schneier, B., *Applied Cryptography*, 2nd Edition, Wiley, 1996

[3] Curtin, Matt, *Brute Force: Cracking the data encryption standard*, Copernicus Books Springer, 2005

[4] IAIK - TU Graz :: AES Lounge, http://www.iaik.tugraz.at/content/research/krypto/aes/#security

[5] Alex Biryukov and Dmitry Khovratovich, *Related-key Cryptanalysis of the Full AES-192 and AES-256*, University of Luxembourg, ePrint Archive: Report 2009/317

[6] Christof Paar and Jan Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*, Springer 2010

[7] NIST.gov - Computer Security Division - Computer Security Resource Center, *Block Cipher Modes*, http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html

[8] NIST, *Recommendation for Block Cipher Modes of Operation*, NIST Special Publication 800-38A, 2001 Edition, http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

[9] NIST, *Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices*, NIST Special Publication 800-38E, January, 2010, http://csrc.nist.gov/publications/nistpubs/800-38E/nist-sp-800-38E.pdf

[10] Performance Analysis of Data Encryption Algorithms, http://www.cs.wustl.edu/~jain/cse567-06/ftp/encryption_perf/index.html#2_4_1

[11] Dan Boneh, *Twenty Years of Attacks on the RSA Cryptosystem*, Notices of the American Mathematical Society (AMS), Vol. 46, No. 2, pp. 203-213, 1999, http://crypto.stanford.edu/~dabo/pubs/abstracts/RSAattack-survey.html and http://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf

[12] Aamer Nadeem et al, *A Performance Comparison of Data Encryption Algorithms*, IEEE 2005

[13] Public key algorithm - Public key encryption algorithms, http://www.encryptionanddecryption.com/algorithms/public_key_algorithms.html

[14] Jonathan Katz, Yehuda Lindell, *Introduction to Modern Cryptography*, Chapman & Hall/CRC, 2008

[15] NIST's Policy on Hash Functions, http://csrc.nist.gov/groups/ST/hash/policy.html

[16] Rajkumar Buyya, James Broberg, Andrzej Goscinski, *CLOUD COMPUTING: Principles and Paradigms*, John Wiley & Sons, Inc., 2011

[17] NIST, *The NIST Definition of Cloud Computing*, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[18] Christof Strauch, *NoSQL Databases*, http://www.christof-strauch.de/nosqldbs.pdf

[19] Amazon Simple Storage Service (Amazon S3), http://aws.amazon.com/s3/

[20] New - Amazon S3 Server Side Encryption for Data at Rest, http://aws.typepad.com/aws/2011/10/new-amazon-s3-server-side-encryption.html

[21] Amazon SimpleDB (beta), http://aws.amazon.com/simpledb/#sdb-vs-s3

[22] Class AmazonS3EncryptionClient, http://docs.amazonwebservices.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/s3/AmazonS3EncryptionClient.html

[23] Introduction - Google Cloud Storage - Google Code, http://code.google.com/apis/storage/docs/getting-started.html

[24] Access Control - Google Cloud Storage - Google Code, http://code.google.com/apis/storage/docs/accesscontrol.html

[25] About Dropbox, https://www.dropbox.com/about

[26] Dropbox, https://www.dropbox.com/pricing

[27] What are the system requirements to run Dropbox?, https://www.dropbox.com/help/3

[28] Does Dropbox always upload/download the entire file any time a change is made?, https://www.dropbox.com/help/8

[29] DropBox : Review, Invites, and 7 Questions with the Founder, http://www.makeuseof.com/tag/dropbox-review-invites-and-7-questions-with-the-founder/

[30] How secure is Dropbox?, https://www.dropbox.com/help/27

[31] Nikos Virvilis, Stelios Dritsas, Dimitris Gritzalis, *Secure Cloud Storage: Available Infrastructures and Architectures Review and Evaluation*, TrustBus'11 Proceedings of the

8th international conference on Trust, privacy and security in digital business, Springer-Verlag Berlin, Heidelberg ©2011

[32] Data as a service with infinispanfull (A talk given by Manik Surtani, founder of Infinispan, on youtube.com): http://www.youtube.com/watch?v=K8qx0QdqfMI

[33] Infinispan - Open Source Data Grids - Jboss community: http://www.jboss.org/infinispan

[34] Podcast Infinispan, Data Grids and Cloud Storage (A talk given by Manik Surtani, founder of Infinispan): http://skillsmatter.com/podcast/home/infinspan-data-grids-cloud/js-2047

[35] A. Harrington and C. D. Jensen. Cryptographic Access Control in a Distributed File System. In *SACMAT'03*, pages 158-165, Como, Italy, June 2-3, 2003.

[36] Google blames Gmail outage on data centre collapse, http://www.theregister.co.uk/2009/02/25/google_gmail_data_centre_fail/

[37] Salesforce.com, customers hit with phishing attack, http://searchcrm.techtarget.com/news/1281107/Salesforce-com-customers-hit-with-phishing-attack

[38] Dropbox confirms security glitch--no password required, http://news.cnet.com/8301-31921_3-20072755-281/dropbox-confirms-security-glitch-no-password-required/

[39] Top five cloud computing security issues, http://www.computerweekly.com/news/2240089111/Top-five-cloud-computing-security-issues

[40] Søren Hjarlvig og Jesper Kampfeldt, *Kryptografisk adgangskontrol i peer-to-peer netværk*, Msc Thesis, IMM DTU, 2003

[41] TrueCrypt - Free Open-Source On-The-Fly Disk Encryption Software for Windows 7/Vista/XP, Mac OS X and Linux – FAQ, http://www.truecrypt.org/faq

[42] Increase transactional throughput with deadlock detection, http://infinispan.blogspot.com/2009/07/increase-transactional-throughput-with.html