2011

POLITECHNICA UNIVERSITY OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

DIPLOMA PROJECT

Improving the security of the BlobSeer data management system

Thesis supervisor:
Prof. Dr. Ing. Valentin Cristea
As. Drd. Ing. Catalin Leordeanu

Student:
Cristian Marius Marinescu

BUCHAREST

2011

Table of Contents

1. Introduction 4
2. Related work 5
   2.1. Lustre File System 5
   2.2. Hadoop Distributed File System 6
   2.3. Amazon S3 8
3. BlobSeer 10
   3.1. Overview 10
   3.2 Architecture 10
   3.3 Blobseer's features 11
   3.4 User Interface 12
4. Security 15
   4.1 Data encryption 15
   4.1.1 Symmetric encryption algorithms 16
   4.1.1.1 DES 17
   4.1.1.2 AES 17
   4.1.2 Public key algorithms 18
   4.1.2.1 RSA 18
   4.1.2.2 Public key certificates 19
   4.2 Access control 19
5. Security enhancements for BlobSeer 20
   5.1 New architecture 20
   5.1.1 Secure client access 20
   5.1.2 Data security 21
   5.1.3 Secure communication 21
   5.2 User interface and algorithms 22
6. Implementation details 25
   6.1 OpenSSL 25
   6.2 Implementation details 25
7. Experimental results 27
8. Conclusions and future work 28

Data management is essential to large scale distributed systems. They are becoming very popular, especially with Cloud Systems becoming more and more widespread and online services such as the Amazon EC2. In this context there are however a number of difficult challenges which must be overcome. One of these challenges is security. A data management system must offer an adequate level of security, thus making sure that the data is stored safely and that only the owners have access to it. In this thesis we propose a novel system to improve the security of distributed data management systems. Our solution offers certificate management, encryption capabilities, as well as credential management. Using these mechanisms we enable authentication, authorization and secure data transfer. The proposed solution was integrated into BlobSeer, which offers high performance for data transfer and efficient data management. We tested our solution, proving that it handles all the security tasks very efficiently, either with the AES or DES encryption algorithm, without adding any significant overhead to the data management system, thus preserving the overall performance of the system.

1. Introduction

In recent years, the number of applications that need to process and store huge amounts of data has grown exponentially. Government and commercial statistics, banking, cosmology, genetics, bio-engineering are just a few examples of domains that used this kind of applications. As ordinary computers could no longer cope with the increasingly volume of data that needs processing, alternative systems must be used, such as large scale distributes systems. These are systems made of multiple autonomous computers, which communicate through a network and interact with each other to achieve a common goal.

Being made up of multiple computers linked together in a network, large scale distributed systems have a series of advantages over regular computers. Firstly they offer a huge processing power, as many of the computers involved in the system work together to complete a common task.

Another feature is the great tolerance to failures. Even if some of the nodes (computers) break down, there still remain a lot others that can complete the current job.

One important aspect of distributed systems is represented by data management, which, according to the Data Management Association [1], represents "the development and execution of architectures, policies, practices and procedures that properly manage the full data life cycle needs of an enterprise." There are many advantages in using data management systems – better data placement (closer to its source or where it is most needed), higher data availability through data replication, higher fault tolerance through elimination of single points of failure. Because of these advantages, in recent years there has been a growing interest in this specific type of data management and a large number of distributes systems have been specifically designed to handle large amount of data. HDFS, Amazon S3, Lustre FS and the NFS are just a few of these applications.

One of the challenges involved in designing and developing this kind of systems is security. In a virtual world filled with viruses, spy-ware, mall-ware, system attacks and other actions intended to harm or compromise the integrity of systems and data, security in the most important aspect of data management systems.

In this thesis we propose a solution to improve security of distributed data management systems. Our strategy addresses some of the common security threats present in any distributed system:

Data confidentiality - only the allowed users have access to data. No system information may be accessed by unauthorized users

Data integrity – the information can be modified only by authorized users or using an authorized manner

Access control – a user can access only the resources for which he has rights

Availability – authorized users can access the data at any time

We apply our solution to the BlobSeer [2] data management systems, a distributed application that offers high performance for data transfer and efficient data management, but lacks any security features.

The thesis is structured as follows. Chapter two presents other project related to our research. We take a look at the architecture and features of HDFS, Lustre FS and Amazon S3 data management systems and how they address the issue of security. The next chapter focuses on BlobSeer. It presents the core algorithms used in this system and the way users can interact with it. Chapter four presents a series of security technologies and algorithms that can be used to improve the security in a data management system. In chapter five, the security strategy for BlobSeer is presented in detail and the next chapter focuses on the implementation details of this strategy. Chapter seven is dedicated to the experimental result obtained after testing the system. The aim is

to maintain the same level of throughput that BlobSeer initially offered. The last chapter presents the conclusions of this thesis and some suggestions regarding future work.

2. Related work

BlobSeer is one of the many systems that implement a distributed data management. In this section we will take a look some similar applications and talk about their architecture, the way they handle data and see how they address the issues of data integrity and security. We will discuss about the most used applications available on the market: Lustre File System, Amazon S3, Hadoop Distributed File System and Network File System.

2.1. Lustre File System

Lustre is an open source distributed parallel object-based file system [3]. It can support tens of thousands of client systems, tens of petabytes (PBs) of storage and hundreds of gigabytes per second (GB/s) of I/O throughput. This is why it is very popular in scientific computing, financial institutions and rich media sectors. The system is composed of three major functional units: metadata servers (MDSs), object storage servers (OSSs) and clients. The way these systems interact can be seen in Fig. 1. Lustre uses block devices for file and metadata storages and each block device can be managed by only one Lustre service. The total capacity is the sum of all individual OST capacities.

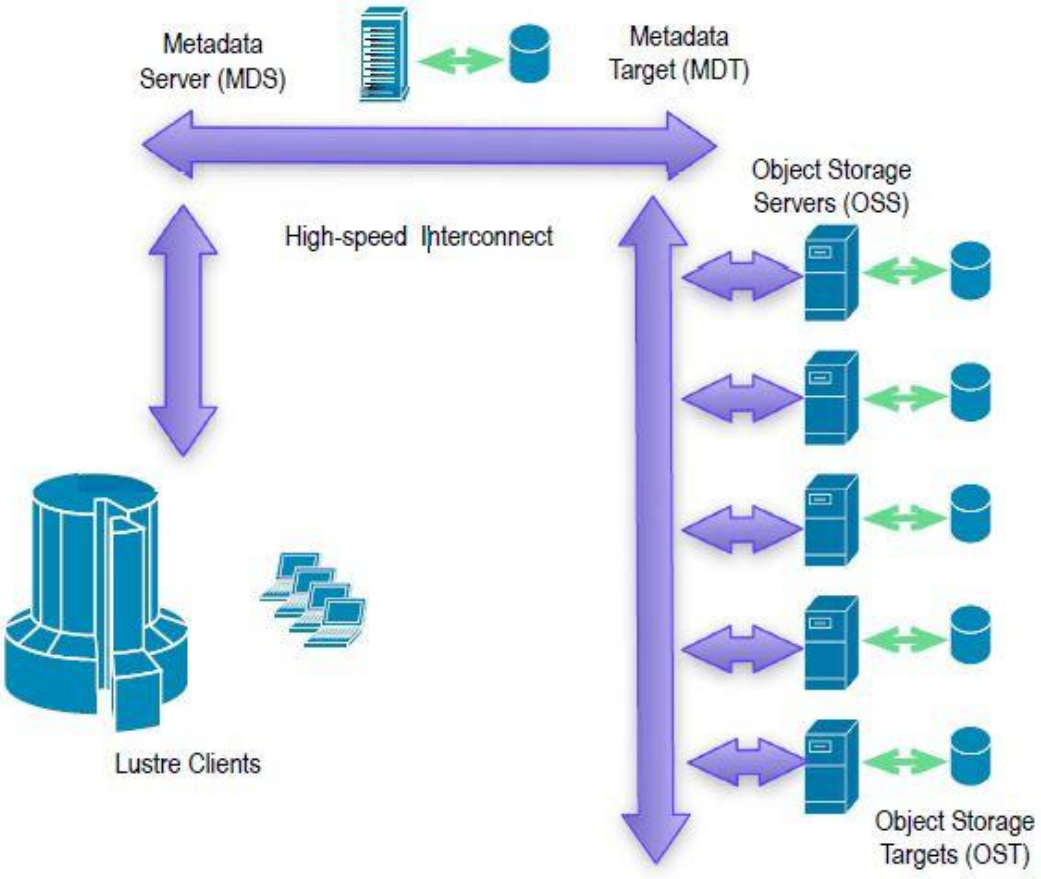

Fig. 1. Lustre File System Architecture

- **Metadata servers (MDS)** – provide metadata services used by metadata clients (MDC) and manage metadata targets (MDT), which store file metadata (file names, directory structure).
- **Management server (MGS)** – serves configuration information of the Lustre file system
- **Object storage server (OSS)** – exposes block devices and server data to object storage clients (OSC). Each OSC manages one or more object storage targets (OSTs), which store file data objects.

In a typical Lustre installation the MDTs, OSTs and clients represent separate modes communicating over a network. The Lustre Network Layer (LNET) supports several network interconnects, including TCP/IP on Ethernet and other network technologies. Also, in order to improve throughput and reduce CPU usage, the system can take advantage of remote direct memory access transfers.

Lustre is designed to be POSIX-compliant so it presents to its clients a unified file system interface which contain standard operations, such as open(), read(), write(). In Linux this interface is achieved through the Virtual File System Layer (VFS). Similar in Lustre there is a layer called llite that is hooked with the VFS to present the interface.

In terms of security, the system solves the issue of maintaining each file's data and metadata integrity by using a distributed lock manager. The metadata locks are managed by the MTDs and are split into multiple bits that protect the lookup of the file (file owner and group, permission and mode, access control list, the state of the inode directory size, contents, timestamps) and layout (file striping).

Lustre implements a multilayered security architecture [4] that consists of:

a trust model – when the system activates network interfaces for the purpose of filesystem request processing or when it accepts connections from clients, the interfaces or connections are assigned a Generic Security Services Application Interface (GSS-API); example: Kerberos 5

Authentication – when a user of the Lustre file system first identifies himself to the system, credentials for the user need to be established. Based on credentials, GSS will establish a security context between the client and server.

Authorization – before the system grants access to the data for consumption or modification, it must do an authorization check based on identify and access control lists.

File encryption – Lustre uses the encryption and sharing model proposed by the StorageTeck secure filesystem, SFS, but a variety of refinements and variants have been proposed.

2.2. Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is a highly fault-tolerant system [5], designed to work on low-cost software. It is written in Java so that it can be run on any platform and provides high throughput access to application data.

The system uses a master-slave architecture that can be seen in Fig. 2. Usually, a HDFS cluster primarily consists of a master server, the NameNode, which manages the file system namespace and regulates client access to the files. Besides the master server there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. The main function of HDFS is to create a file system namespace where clients can store data as files. Internally, the files are split into one or more blocks that are stored in a set of DataNodes. The NameNode's main function is to maintain mappings between DataNodes and file blocks. It also provides an interface that gives access to operations for opening, closing and renaming files and directories, while read and write operations are handled directly by the DataNodes.

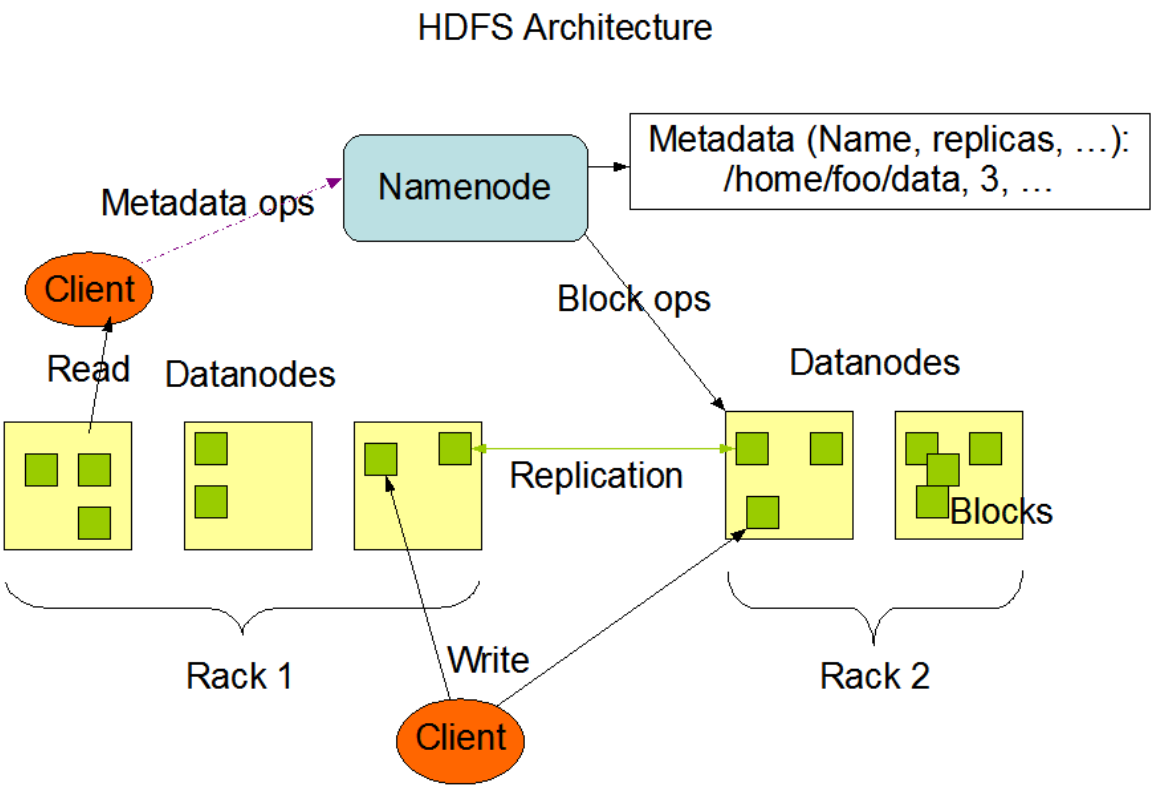

Fig. 2. HDFS Architecture

HDFS supports a traditional hierarchical file organization in which a client can create directories and store files in them. The file system namespace is similar to most existing file systems: files can be created, renamed, relocated and removed. Another feature is the ability to replicate data. Each file is split into block and of fixed size that are replicated for fault tolerance. The replication factor and the block size can be configured per file. The client can specify the replication factor at file creation and he can change it at any time. HDFS also addresses the problem of data integrity. It uses checksum validation on the contents of files by storing computed checksums in separate, hidden files in the same namespace as the actual data. When the file retrieves file data, it can verify that the data received matches the checksum stored in the associated files.

The system uses a three level security model [6], Fig. 3. The first layer is responsible for user authentication. It can work with digital certificates and also handles user permissions. The second layer is responsible for user's data encryption and protecting the privacy of users. The last layer refers to fast user data recovery. Data is stored in DataNodes and there is a possibility of failure. To overcome this HDFS has at least three replicas for each data storage block.

With three-level structure, user authentication is used to ensure that data is not tempered. The authenticated user can manage the data by operations: add, modify, delete and so on. If the user authentication system is deceived by illegal means and malign user enters the system, fyle encryption and privacy protection can provide this level of defence. In this layer user data is encrypted, even if the key was illegally accessed, through privacy protection, malign user will still not be able to obtain effective access to information. Finally the rapid restoration of files layer, through fast recovery algorithms, makes user data be able to get the maximum recovery even in case of damage.

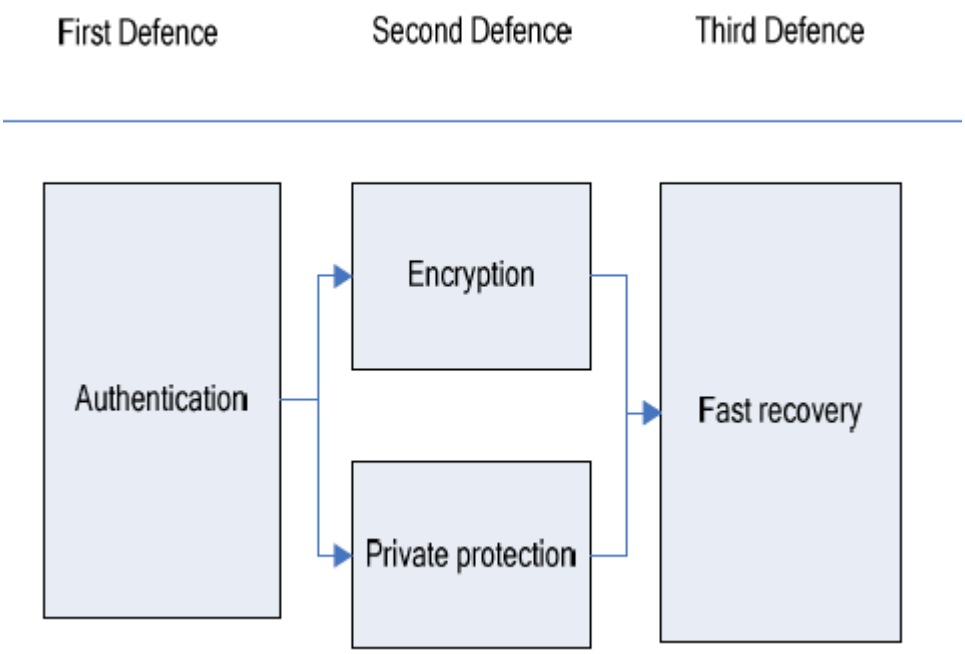

Fig. 3. HDFS Security Model

2.3. Amazon S3

Amazon S3 is an online storage web service offered by Amazon Web Services. It provides storage through web services interfaces: REST,SOAP, BitTorrent. Although the service architecture is not made public, Amazon released some info about how the system works. The main components of S3 are buckets, objects, keys and regions[7].

A **bucket** is a container for objects stored in Amazon S3. Every object is contained in a bucket and can be accessed using a URL. Buckets serve several purposes: they organize the Amazon S3 namespace at the highest level, they identify the account responsible for storage and data transfer charges, they play a role in access control, and they serve as the unit of aggregation for usage reporting. They can be configured to be created only in a specific region.

Objects are the fundamental entities stored in Amazon S3. Objects consist of object data and metadata. The data portion, which can reach sizes of 5 terabytes, is opaque to Amazon S3. The metadata is a set of name-value pairs that describe the object. These include some default metadata such as the date last modified and standard HTTP metadata such as Content-Type. A client can also specify custom metadata at the time the Object is stored. An object is uniquely identified within a bucket by a key (name) and a version ID.

A **key** is the unique identifier for an object within a bucket. Every object in a bucket has exactly one key. Because the combination of a bucket, key, and version ID uniquely identify each object, Amazon S3 can be thought of as a basic data map between "bucket + key + version" and the object itself. Every object in Amazon S3 can be uniquely addressed through the combination of the web service endpoint, bucket name, key, and optionally, a version.

The **regions** refer to the geographical regions where Amazon S3 will store the buckets. A client might choose a Region to optimize latency, minimize costs, or address regulatory requirements. Objects stored in a Region never leave the Region unless the client explicitly transfers them to another Region.

Access to data is controlled using bucket policies and access control lists. Bucket policies provide centralized, access control to buckets and objects based on a variety of conditions, including Amazon S3 operations, requesters, resources, and aspects of the request (e.g., IP address). The policies are expressed in access policy language and enable centralized management of permissions. The permissions attached to a bucket apply to all of the objects in that bucket. An account can grant one application limited read and write access, but allow another to create and delete buckets as well. Unlike ACLs that can only add (grant) permissions on individual objects, policies can either add or deny permissions across all (or a subset) of objects within a bucket. With one request an account can set the permissions of any number of objects in a bucket. Only the bucket owner is allowed to associate a policy with a bucket. Policies, written in the access policy language, allow or deny requests based on: bucket operations and object operations (such as PUT Object, or GET Object), requester or conditions specified in the policy.

Amazon S3 Access Control Lists (ACL) enables the user to manage access to buckets and objects. Each bucket and object has an ACL attached to it as a subresource. It defines which AWS accounts or groups are granted access and the type of access. When a request is received against a resource, Amazon S3 checks the corresponding ACL to verify the requester has the necessary access permissions. When a bucket or an object is created, Amazon S3 creates a default ACL that grants the resource owner full control over the resource.

S3 also has an option to encrypt data [8]. Data will be encrypted using AES 256 or Blowfish and SHA-1 algorithm will ensure data integrity.

3. BlobSeer

3.1. Overview

BlobSeer is a distributed data management service designed to provide storage and efficient access to very large, unstructured data objects. It focuses on heavy access concurrency where data is huge, mutable and accessed by a very large number of concurrent, distributed processes [9].

Data is stored as unstructured sequences of bytes called Binary Large Objects (BLOBs). Blobs typically reach sizes of 1TB. The system offers clients a simple interface through which they can manipulate blobs. They can create a new blob, read/write a sequence of size bytes to it starting at a specified offset or append a new sequence of bytes to the blob. This interface is versioning-orientated. Each time a client writes or appends data a new snapshot of the blob is generated rather than overwriting existing data. This snapshot is labeled with an incremental version and clients can access any past snapshots of the blob by providing its version. Although each write or append generates a new blob version, only the differential patch is actually stored, so that storage space is saved at far as possible. The new snapshot physically shares all unmodified data with the previous versions.

Internally each blob is made up of fixed size blocks of data, called chunks or pages, which are distributed among data providers. Each chunk can be expressed as a range(size, offset) of the blob. Metadata is used to associate such a range with a version of the blob and the location where the chunks is stored. This is stored and managed on metadata providers through a decentralized distributed hash table infrastructure.

3.2 Architecture

BlobSeer consists of several processes that communicate with each other using asynchronous remote procedure calls (RPCs).

- **Clients** – may create new blobs, then read, write or append data to them. There may be multiple concurrent clients and their numbers may dynamically vary in time without notifying the system
- **Data providers** – physically store the pages generated when writing or appending new data to a blob. New data providers are free to join and leave in a dynamic way.
- **Provider manager** – keeps information about the available storage space and schedules the placement of newly generated pages according to a round-robin load balancing strategy.[10]
- **Metadata providers** – physically store the metadata allowing clients to find the chunks corresponding to the blob snapshot version. In order to enhance data throughput through parallel I/O, metadata is organized as a distributed metadata tree. Such a tree exists for each snapshot version of a give blob. This is a binary tree in which each node represents a range of the blob, delimited by an offset and a size. For efficient concurrent access the nodes are stored on metadata providers in a distributed way, using a distributed hash table(DTH).
- **Version manager** – registers update requests (writes, appends), assigns snapshot version numbers and publishes new blob version, while guaranteeing ordering and atomicity.

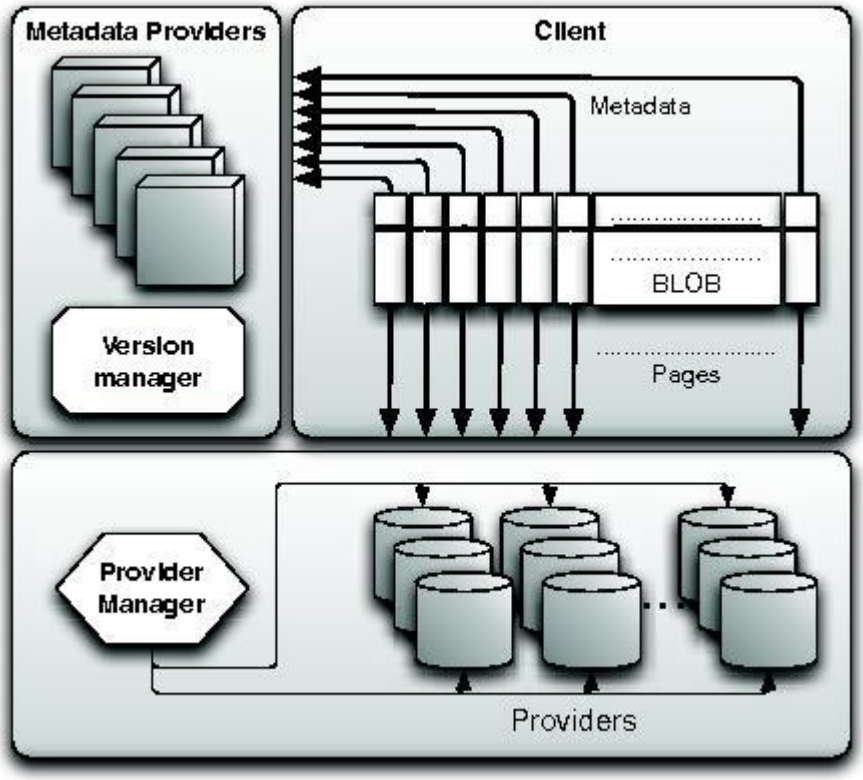

Fig. 4 BlobSeer Architecture

3.3 Blobseer's features

Data striping is a technique to increase the performance of data access. Each BLOB is split into fix size chunks that are distributed all over the machines that provide storage space. Thereby data access requests can be served by multiple machines in parallel, which leads to high performance. An important factor is the chunk distribution strategy. BlobSeer has a configurable chunk distribution strategy. This permits an application to configure the strategy such that it optimizes its own desired objectives. This feature has a major role as a well choosen strategy will lead to a high throughput when concurrent clients access different parts of the BLOB.

Metadata decentralization [11] – as each BLOB is striped over a large number of store space providers, additional metadata is needed to map subsequences of the BLOB defined by offset and size to the corresponding chunks. Although this data is very small compare to the actual BLOB, at large scale it can produce a large overhead. To resolve this issue, BlobSeer maintains the metadata in a decentralized way on the metadata providers, as a distributed hash table(DHT). This leads to high scalability and fine grain access to the BLOBs, as the I/O work associated to the metadata overhead can be distributed among the metadata providers, such that concurrent accesses to metadata on different providers can be served in parallel.

Asynchronous operations – I/O operations can be extremely slow compared to the actual data processing. In the case of data-intensive applications where I/o operations are frequent and involve huge amount of data, an asynchronous access interface to data is crucial, because it enables overlapping the computation with the I/O operations, thus avoiding waste of processing power. BlobSeer offers this kind of operations, in the form of asynchronous RPC calls.

Atomic snapshot generation - it's a key property of the BlobSeer system. Readers are not able to access inconsistent snapshots that are in the process of being generated. The version manager ensures that clients see the new snapshot version only after the writing operations are completed.

Explicit versioning [12] – enables efficient management of highly parallel data workflows. Because every write or append generates a new snapshot of the blob, clients do not need to synchronize concurrent writes on the same blob. Each write/append will be done on its own separate snapshot.

3.4 User Interface

The system's simple versioning-orientated interface provides several primitives to the client. All of these are asynchronous calls and must have a callback function as one of the parameters. When the operation is complete, the callback function is called with the results of the operation as its parameters. In this callback function the calling application can take action based on the result of the operation.

CREATE (callback(id)) - this primitive creates a new empty blob of size 0. The blob will be identified by its id, which is guaranteed to be globally unique. The new id will be passed to the callback function.

READ (id, v, buffer, offset, size, callback(result)) [13] - this primitive reads data from the version v of the blob. It results in replacing the contents of the local buffer with size bytes from the snapshot version v of the blob id, starting at offset, if v has already been generated. The callback function receives a Boolean parameter that indicates whether the read succeeded or not. The way this primitive works can be seen in Algorithm 1 and Fig. 5.

Algorithm 1:
1: **if** *v* is not published **then**
2: **fail**
3: **end if**
4: PD ← READ_METADATA(*v*, *offset*, *size*)
5: **for all** (*pid*, *i*, *provider*) in PD **in parallel do**
6: read *pid* from *provider* into buffer at *i* x *psize*
7: **end for**
8: **return success**

To read data, the client first contacts the version manager, to check whether the supplied version v has been published and fails if it is not the case. He then contacts the metadata provider to find which pages fully cover the requested offset and size and where they are located. Based on this information he generates a set of page descriptors (PD) that holds information about all the pages needed: for each page the unique id pid, its index i in the buffer to be read and the page provider that stores it. Now the client can fetch the pages in parallel and fill the local buffer.

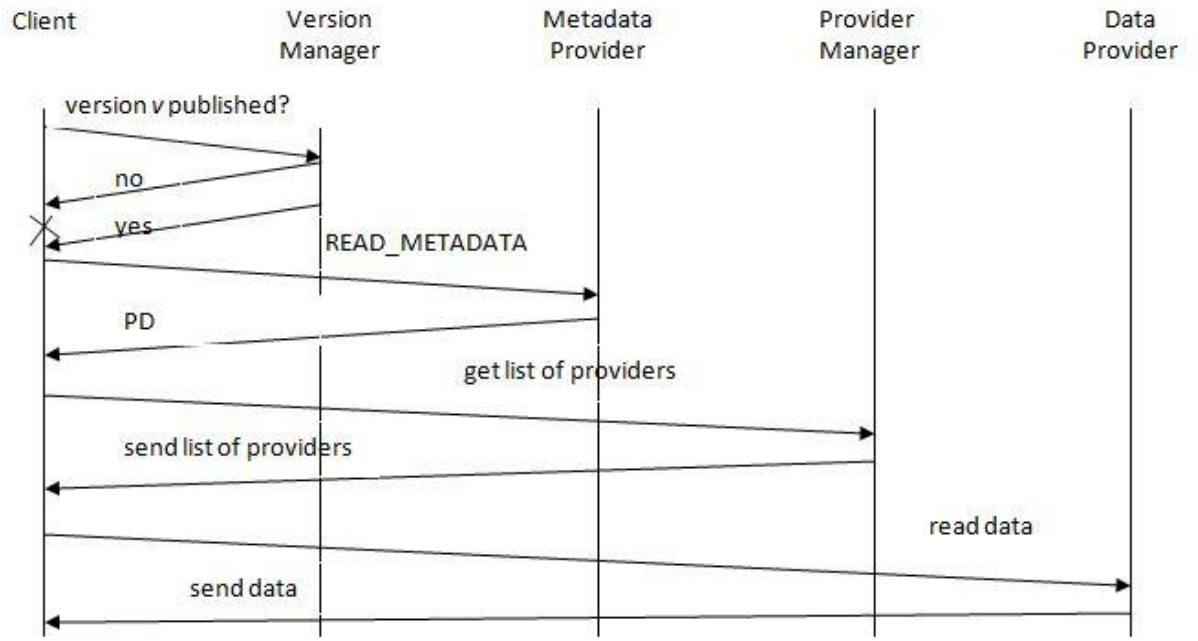

Fig. 5. Read Protocol

WRITE (id, buffer, size, offset, callback(v)) / APPEND(id, buffer, size, callback(v)) - this primitives update the blob. The initiated operation copies size bytes from a local buffer into a blob identified by id, either at the specified offset (in case of write) or at the end of the blob(in case of append). Each time one of these operations is invoked a new snapshot is generated for the given blob. The completion results in the invocation of the callback function with the new assigned version v as parameter. The way this function works can be seen in Algorithm 2 and Fig. 6.

Algorithm 2:
1: $n \leftarrow (offset + size)/psize$
2: PP ← the list of n page providers
3: PD ← Ø
4: **for all** $0 \leq i < n$ **in parallel do**
5: *pid* ← unique page id
6: *provider* ← PP[i]
7: storage page *pid* from *buffer* at i x *psize* to provider
8: PD ← PD U (*pid*, i, *provider*)
9: **end for**
10: *vw* ← assigned snapshot version
11: BUILD_METADATA(*vw*, *offset*, *size*, PD)
12: notify version manager of success
13: **return** *vw*

To write some data, a client first computes the number of pages n that cover the range. Then it contacts the provider manager requesting a list of n page providers PP, one for each page, that are capable of storing the pages. For each page in parallel, the client generates a globally unique page id, pid, contacts the corresponding page provider and stored the contents of the page on it. It then updates the set of page descriptors PD accordingly. This set is later used to build the metadata

associated with this update. After writing all pages, the client contacts the version manager and registers it update. The version manager then assigns to this update a new snapshot version vw and communicates it to the client, which then generates new metadata. After generating the metadata, the client notifies the version manager of success. Finally the version manager can publish the new version vw.

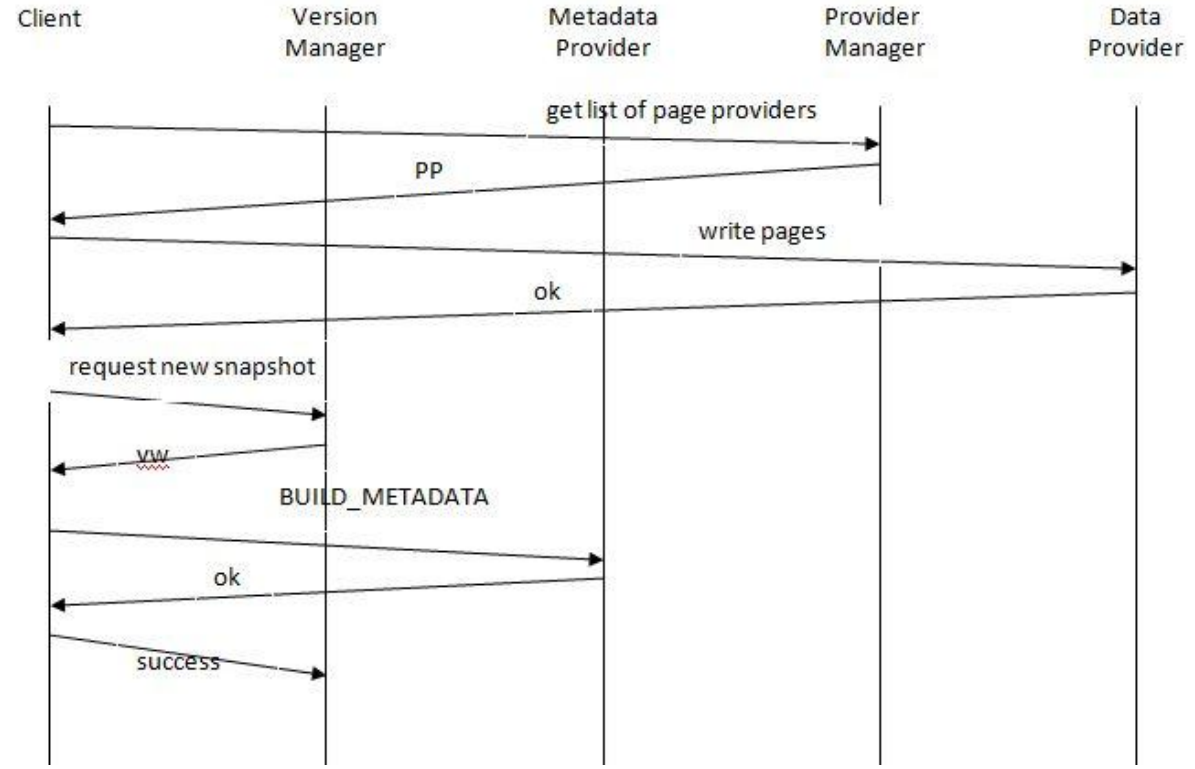

Fig. 6. Write Protocol

GET_RECENT(id, callback(v, size)) - queries the system for a recent snapshot of the blob id. The result is the query is the snapshot number and its size.

GET_SIZE(id, v, callback(size)) - it is used to find the total size of the snapshot version v for blob id. The size is passed to the callback function once the operation has successfully completed.

The general flow of operations inside BlobSeer can be seen in Figure 6.

4. Security

As we have seen in section 2, to address the issue of security, applications do not use just one technology, but rather a combination of different mechanisms. They rely on a mixture of data encryption, secure communications and client access protocols. We will take a look at the main features that these protocols offer and at the algorithms behind them.

4.1 Data encryption

Encryption is the process of transforming information, the plain text, using a cipher (algorithm) to make it unreadable to anyone except those possessing special knowledge, referred to as a key. The result of the process is encrypted information is a ciphertext. Decryption is the opposite of the encryption. It takes a ciphertext and converts it to plain text. An important issue of data encryption is the scheme used – this must be the same for both processes. In order to get the initial plain text, in the decrypting process, the same scheme a in the encryption process must be used. The process can be seen in Figure.

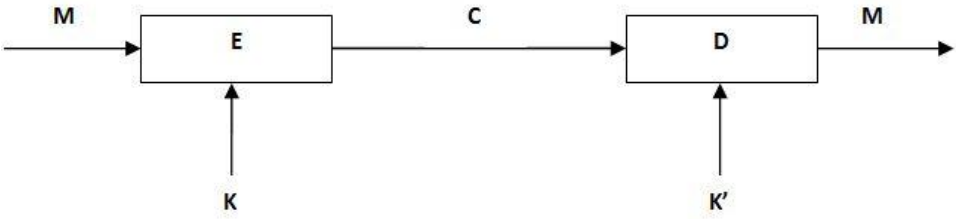

Fig. 7 Basic Encryption Model

Mathematically this can be written:

$$E_K(M) = C \quad (1)$$

$$D_{K'}(C) = M \quad (2)$$

$$D_{K'}\left(E_K(M)\right) = M \quad (3)$$

Where:

M = plain text (message)
C = cipher text
E = encryption method
D = decryption method
K = encryption key
K' = decryption key

The main featured [14] gained through using data encryption in network applications are:

- **Confidentiality** – deals with protecting, detecting and deterring the unauthorized disclosure of information. The main goal of cryptography is to take a plaintext message and garble it in such a way that only the intended recipient can read it and no one else, this being precisely the goal of confidentiality.
- **Data integrity** – deals with preventing, detecting and deterring the unauthorized modification of information. Cryptography addresses this issue by performing verification and validation of data. This is done by computing a digital signature check across the information. If any bit of data changes, the signature will be different

- **Availability** – relates to preventing and detecting the denial of access to critical information. A variety of attacks can result in the loss of or reduction in availability. Some of these attacks can be prevented through the use of encryption and authentication.
- **Authentication** – deals with the problem of verifying that an entity has the said identity.

Cryptographic algorithms are mainly divided in two categories: symmetric encryption algorithms and public key algorithms.

4.1.1 Symmetric encryption algorithms

Symmetric algorithms, also called conventional algorithms or secret-key algorithms are algorithms where the encryption and decryption keys are the same. They require that the sender and the receiver agree on a key before they can communicate securely. The security of these algorithms rests in the key. Divulging the key means that anyone could be able to encrypt or decrypt the messages. As long as the communication needs to remain secret, the key must remain secret.

In this case equation (3) becomes:

$$D_K\big(E_K(M)\big) = M \qquad (4)$$

There are two basic types of symmetric algorithms: block ciphers and stream ciphers. Block ciphers operate on blocks of plain text and cipher text. Stream ciphers operate on streams of plaintext and cipher text one bit or byte at a time. With a block cipher, the same plaintext block will always encrypt to the same ciphertext block, using the same key. With a stream cipher, the same plaintext bit or byte will encrypt to a different bit or byte every time it is encrypted.

A cryptographic mode [15] usually combines the basic cipher, some sort of feedback, and some simple operations. The operations are simple because the security is a function of the underlying cipher and not the mode. Even more strongly, the cipher mode should not compromise the security of the underlying algorithm. The common modes used in encryption rely on block ciphers:

- **Electronic Codebook Mode (ECB)** – a block of plain text encrypts into a block of ciphertext. Data streams are broken into blocks that are individually processed. Usually this mode is padded to accommodate messages that aren't a multiple of the cipher's block size length. The problem with ECB mode is that if a cryptanalyst has the plain text and the cipher text for several messages, he can start to compile a code book without knowing the key.
- **Cipher Block Chaining Mode (CBC)** – introduces a solution for the code book problem of ECB by making each cipher text block dependent not only on the plain text that generated it, but also on the previous plaintext blocks. In this mode the current plain text block is XOR-ed with the previous ciphertext block before it is encrypted. After a plaintext block is encrypted, the resulting ciphertext is also store in a feedback registers. Before the next plaintext block is encrypted, it is XOR-ed with the feedback register to become the next input for the encryption function. Decryption is the reverse of this process. Dictionary attacks are still possible if the data streams have common beginning sequences. For this reason it is possible set a block of data, called initialization vector IV, to be XOR-ed with the first block of plain text before it is encrypted. The IV doesn't have to be secret, but must be random. Also it must be available in order to decrypt the data.
- **Cipher Feedback Mode (CFB)** is a way of turning a block cipher into a stream cipher. Like CBC, CFB can use an IV. Here the IV is more important because if two data streams are encrypted with the same key, and have the same IV, then both streams can be recovered.

- **Output-Feedback Mode (OFB)** is another way of turning a block cipher into a stream cipher.

The most common used symmetric encryption algorithms are: the Data Encryption Standard (DES) and the Advanced Encryption Standard (AES).

4.1.1.1 DES

Is a symmetric block cipher [16], operating on 64-bit blocks using a 56-bit key. It encrypts data in blocks of 64 bits. The input of the algorithm is a 64-bit block of plaintext and the output from the algorithm is a 64-bit block of ciphertext after 16 rounds of identical operations. The key length is 56 bits by stripping off the 8 parity bits, ignoring every eight bit from the 64-bit key.

The basic building block of DES is a suitable combination of permutation and substitution of the plaintext block. After an initial permutation, it splits the block into a right half and a left half, each 32 bits long. Then there are 16 rounds of identical operations, called ***Function f***, in which the data are combined with the key. After the sixteenth round, the right and left halves are joined, and a final permutation, the inverse of the initial one, finishes the algorithm. The function f consists of four operations. First the key bits are shifted and then 48 of them are selected. Then the right half of data is expanded to 48 bits via an expansion permutation, combined with the 48 bits from the key using an XOR and the result is sent through 8 S-boxes (these are lookup tables that take a sequence of m bits as input and transforms it in a sequence of n bits) producing 32 new bits that are permuted again. The output of the function is then XOR-ed with the left half. This result becomes the new right half and the old half becomes the new left half.

The decryption process follows the same steps, but uses the key in reverse order.

DES supports all for modes of operation: ECB, CBC, OFB, CFB. ECB and CBC should be used for encryption, while OFB and CFB are recommended for authentication.

DES is now considered insecure due to the availability of increasing computational power, which makes a brute force attack possible.

To solve this issue **Triple DES (3DES)** was designed. The algorithm uses three independent keys which are used in three steps, generating an effective key of 168 bits. First it encrypts the plain text block using the first key, then it decrypts the result using the second key and finally it encrypts this result with the third key.

4.1.1.2 AES

Is a block cipher that uses 128bit blocks and variable key size [17]. The possible key sizes are 128, 192 and 256 bits. It was designed to have the following characteristics: resistance against all known attacks, speed on a wide range of platforms and design simplicity.

Like DES, the AES algorithm is involves number of repetitions called rounds that transform the input plain text into a cipher text. AES operates on a 4×4 matrix of bytes, termed the state. The cipher is specified as a number of repetitions of transformation rounds that convert the input plaintext into the final output of ciphertext. Each round consists of several processing steps, including one that depends on the encryption key. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key. A round (other than the final round) is composed of four different transformations:

Round(state, roundKey) {

```
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, roundKey);
    }
```

The meaning of each transformation is:

- SubBytes – a non-linear substitution step where each byte is replaced with another according to a lookup table
- ShiftRows – a transposition step where each row of the state is shifted cyclically a certain number of steps
- MixColumns – a mixing operation which operates on the columns of the state, combining the four bytes in each column
- AddRoundKey – each byte of the state is combined with the round key using bitwise xor

4.1.2 Public key algorithms

These algorithms are designed so that the keys used for encryption and decryption are different from each other. Furthermore the decryption key cannot be calculated from the encryption key in a reasonable amount of time. The term public key comes from the fact that the encryption key (often called public key) can be made public. Anyone can use it to encrypt a message, but only a specific person with the corresponding decryption key (private/secret key) can decrypt the message. Compared to symmetric key algorithms, public key algorithms are much slower at encrypting and decrypting data. They are mostly used in:

- Digital signatures – the sender signs a message with its private key. The receiver can determine the validity of the message by verifying the signature using the senders public key.
- Key exchange algorithms – two sides cooperate to exchange a session key. The private and public keys are used in the process of generating the session key.

The most popular public key algorithm is the Diffie-Hellman Algorithm, the Digital Signature Algorithm (DSA) and the Rivest, Shamir and Adleman Algorithm (RSA). They are used in a variety of applications and communication protocols, including the Secure Shell (SSH) and Secure socket Layer (SSL).

4.1.2.1 RSA

RSA is an algorithm for public-key cryptography [18]. It is the first algorithm known to be suitable for signing as well as encryption, and is widely used in electronic commerce protocols, and is believed to be sufficiently secure given sufficiently long keys and the use of up-to-date implementations. The algorithm involves three stages: key generation, encryption and decryption.

To generate the keys, two random large prime numbers of equal length, p and q, are chosen. The next step is to compute the product:

$$n = p * q \tag{5}$$

Then randomly choose the encryption key, e, such that $(p-1)(q-1)$ are relatively prime. Finally, the Euclidean algorithm is used to comput the decryption key, d.

$$ed \equiv 1 \; mod \; (p-1)(q-1) \tag{6}$$

$$d = e^{-1} \; mod \; ((p-1)(q-1)) \tag{7}$$

The numbers *e* and *n* are the public key; the number *d* is the private key. The two primes, *p* and *q*, are no longer needed and should be discarded and never revealed.

The next stage of the algorithm is the encryption of the message. The first step is to divide the message, *m*, into numerical blocks smaller then *n*. The encrypted message, *c*, will be made up of similarly sized message blocks, *ci*, of about the same length. The encryption formula is:

$$c_i = m_i^e \ mod \ n \quad (8)$$

To decrypt a message, the algorithm takes each encrypted block *ci* and computes:

$$m_i = c_i^e \ mod \ n \quad (9)$$

4.1.2.2 Public key certificates

A public key certificate is an electronic document whose role is to bind a public key with a distinguished name. The distinguished name refers to the name of the person or entity that owns the public key to which it's bound. The certificate can also contain additional information about the owner, like e-mail, location, etc.

Certificates are issued by a commonly known third party called Certificate Authority. The CA first checks that the public key from the certificate matches the owner. After this, the certificate is signed with the CA's private key, thus proving the validity of the information contained in the certificate. An entity can check the integrity of a certificate by verifying the CA signature using the CA's public key.

The certificates have several uses in application, the most important being the possibility to be used in an authentication process.

4.2 Access control

Another important part of distributed systems security is the client access control. This actually refers to user authentication and authorization. There are several authentication protocols, the most used being SSL which takes advantage of public key encryption and digital certificates (discussed in the previous section).

Authorization is the function of specifying access right to resources. The models used for authorization are split into two categories: Discretionary Access Control (DAC) – these let any user to delegate its right to another user, and Mandatory Access Control (MAC) – which don't give users the possibility to delegate their right to others.

Some of the protocols used for user authorization are:

- **Access Control Lists (ACLs)** - are lists of permissions attached to an object. Their role is to limit the actions an entity can perform on an object. When a subject requests an operation on an object in an ACL-based security model the operating system first checks the ACL for an applicable entry to decide whether the requested operation is authorized. ACL models may be applied to collections of objects as well as to individual entities within the system's hierarchy. [19]
- **Graham – Denning model** – this model assumes that each object has a special subject, the owner, with special rights and that each subject has another subject with special right, controller. The model defines a series of actions that a subject can execute on an object or on another subject. These actions are set in an access table in which each row represents a subject and each column represents an object or a subject. The intersection of a row with a column specifies what actions a subject can perform on an object or a subject.

5. Security enhancements for BlobSeer

In this chapter we present our proposed security solution for the BlobSeer data management system. We take a look at the new feature this solution offers, what technologies and algorithms are used to implement this module and how it addresses the issues identified in the first chapter.

5.1 New architecture

The security layer we propose adds a new entity to the BlobSeer architecture, the Security Manager (SM). This represents a separate process capable of interacting with all the other entities in the system. The Security Manager makes uses of much of the technologies and algorithms presented in the previous chapter. It offers confidentiality, data integrity, secure connections between entities in the system, client access control, data encryption and key exchange algorithms.

The Security Manager can actually be seen as three security layers in one. The first layer refers to secure client access, through the use of authentication and authorization. Each client has assigned by the SM a unique id with which it identifies in the system and has a set of access rights to different blobs. The user is considered to be the owner of the blobs it created and can grant or revoke access to them to other users. This is done through the use of X509 certificates that can authorize and authenticate a user and access control lists for each blob.

Another important feature of the SM is data integrity and confidentiality. It offers a user the possibility to encrypt data using symmetric key algorithms before it is stored in the system. This way nobody but the users that know the encryption key can have access to the user's data. Also, all the encryption operations are done on the user's side so no overhead is added in the system. Because the system permits rights delegation, the secret key used for the encryption will be stored on the SM for each user, encrypted with the user's public key. This way we assure that even if an unauthorized user can gain access to a blob, he will not be able to decrypt it.

The third layer of security refers to secure communication. Entities that communicate must first prove their identity to each other. Also all messages exchanged are sent encrypted through the network so that even if they are intercepted by another party they cannot be read.

5.1.1 Secure client access

This represents the most important feature of the security manager as it regulates access to the blobs. Firstly, this refers to user authentication. The SM requires each user to have a public key certificate signed by a CA, containing information that can be used to identify the user. Also each user must have an id with which he is identified uniquely in the system. When a user first enters the system, he must call the REGISTER method on the SM. Through this, the user and the SM exchange their public key certificates to verify each others identity. After verifying the client's identity, the SM first checks his list of registered users. If the user already exists in the list, the SM gets the id from the list and returns it to the user. Otherwise, a new id will be generated and the user will be added to the list.

The other aspect of secure client access is represented by authorization. The SM gives each user a set of rights. With these users have the ability to perform different actions on the blobs. A user can have one or more of the following rights:

- **read** – gives the user access to the blob, but just to read from it
- **write** – gives the user access to read and write to a blob

- **grant_read** – the user can grant another user the right to read a blob
- **grant_write** – the user can grant another user the right to read and write to a blob
- **own** – the user created the blob and has all the above rights; for a blob only one user can have this right

The SM uses these rights to determine if a user can perform a certain action. For each blob it stores an Access Control List (ACL) that contains a list of users and their access rights to that blob. When a user wants to access a blob, the SM will first search the ACL of that blob to see if the user has the right to perform the action.

5.1.2 Data security

The main aspects of data security refer to data confidentiality and integrity. Only authorized users can have access to the blobs and only in an authorized manner. This is achieved not only through client access control, but also through the use of data encryption. Using encryption a user can ensure that the data cannot be accessed by the system or by a user that even though it doesn't have rights, it gained access to it.

The SM offers users the possibility to store the blobs in an encrypted format using symmetric encryption. For this a user can specify which encryption algorithm and what key to use or he can let the SM choose them. The key must be generated when the blob is created. It will then be stored on the SM, encrypted with the user's public key.

When a user A chooses to give access to another user B, he first takes the encryption key from the SM and decrypts it using his private key. Then he gets from the SM the other B's public key (from the user certificate stored on the SM), encrypts the key with B's public key and stores it on the SM. After this B just has to take the key from SM, decrypt it using his private key and he can have access to the blob.

The system has a couple of symmetric ciphers from which the user can select. This must be done when the blob is created. Also a user can select a custom key or initialization vector (IV) or it can let the system generate some random. The available ciphers are:

- AES – using keys of 128, 192 or 256 bits and CBC or EBC cipher modes
- 3DES - in CBC or EBC cipher mode

5.1.3 Secure communication

This ensures that a message can be read only by the entity it was addressed to. The SM accomplishes this through the use of messages encrypted with the public and private keys. When a user sends a message to another entity (except to another user) in the system, he signs the message, which must include the user's id, and then encrypts it using the SM's public key.

When the entity receives the message it first decrypts it using the SM's private key. It then checks the signature using the public key of the user. If the check is successful, then the message was received from the respective user and also it was not modified during network transfer. The same process is used when a user receives a message from an entity of BlobSeer. He first decrypts the message with its private key and then checks the signature using the SM's public key.

Another important aspect of the secure communication is the method used to send messages from a user to the page provider. Because the individual pages are encrypted, the user doesn't have to encrypt them one more time. The SM requests that only the header of the message be signed and encrypted using the algorithm presented earlier. This header will also contain a hash of the encrypted pages to make sure that they were not modified during the transfer.

5.2 User interface and algorithms

The new security layer modifies some of the BlobSeer's primitives and also adds new ones:

CREATE(enc_type, key, iv, callback(id)) – now, this primitive takes 3 more parameters: the type of symmetric cipher used for blob encryption, the key for the encryption and the initialization vector.

REGISTER(user_cert, callback(sm_cert, user_id)) – is a new primitive that the user must call when it enters the system. The first parameter, use_cert, represents the user's certificate that contains its public key. The other parameter is a callback function which receives as parameters the SM's certificate and the id assigned to this user. The way this primitive works on the SM side can be seen in Algorithm 3:

Algorithm 3:

1: **if not** valid (*user_cert*)
2: **return** fail
3: **endif**
4: **if** *user_cert* **in** *reg_users_list*
5: **get** *user_id*
6: **return** *user_id*
7: **endif**
8: generate *user_id*
9: add (*user_id*, *user_cert*) to *reg_users_list*
10: **return** *user_id*

When the SM receives a new register request, it takes the client certificate and tries to validate it. A certificate is valid only if it contains the public key of the users and if it was signed by a Certificate Authority. If it's not valid, then the process stops and the SM sends to the users only its certificate. Otherwise the SM searches for the user in its list of registered users. If the user is found, the id will be taken from the matching entry in the list, if not a new id will be generated and added to the list. Finally the SM sends to the user its certificate and the user id.

GET_CERTIFICATE(user_id, req_user, callback(user_cert)) – this primitive can be used to retrieve from the SM a registered user's certificate, an thus its public key. The parameters represent the id of the user that sends the request, *user_id*, the id of the user who's certificate is needed, *req_user*, and a callback function that is called once the primitive ends. When the callback is called, its parameter will hold the requested certificate, if the user is registered. Although the function takes as parameters two user ids, they are not sent to the SM as plain text. First they are signed with the user's private key and then they are encrypted using the SM's public key. This way the message cannot be read by anyone expect the intended receiver and the receiver can now for sure that the message was sent by the user. When receiving the message the SM first decrypts the message using its private key. The decrypted message is made up of the two user ids and the digital signature. At this point the SM will take the sender's user id and check if the user exists in the registered users list. If the user is not found then the message is discarded. Otherwise, using the user's public key, the SM will check the digital signatures to confirm the identity of the sender. Only if the signature is valid, the SM will try to find the requested certificate. As the certificate does not hold critical information, the response message doesn't need to be encrypted. It will just be signed by the SM, so that the user can verify the identity of the sender (SM).

GET_KEY(user_id, blob_id, callback(key, iv, cipher_type)) – a user must call this primitive in order to be able to successfully decrypt the data stored in the blob. The parameters are encrypted before they are sent to the SM, exactly like in the case of GET_CERTIFICATE method. The SM uses the same method to validate the user's identity and then searches in the user's list of keys for the blob id. It takes from here the cipher type, the key and the iv used to encrypt the respective blob. As this information is stored encrypted with the user's public key, the SM will just sign the response message that contains them.

CHECK_PERMISSION(user_id, blob_id, permission, callback(result)) – using this primitive the different entities that make up BlobSeer can check whether a user has the right to access the blob in the specified manner. Upon receiving the message, the SM checks the access control list of blob_id and tries to find the pair (user_id, permission). If the pair exists SM send true, otherwise it sends false. The response will be found in the parameter of the callback function.

DELEGATE (user_id, blob_id, permission, key, callback(result)) – a user can use this primitive to delegate some of its access permissions to another user. The possible values for permission have been presented in 5.1.1. A user can delegate a read or write permission to another user only if he has grant_read or grant_write for that blob. Also only the owner of a blob can delegate a grant_read or grant_write permission to another user. The SM checks these conditions and if they are met, the it make the necessary changes to the blob's ACL. The key parameter represents the key, iv and cipher type used to encrypt the blob. They are sent encrypted with the user's public key and will be stored on the SM in this format.

REVOKE(user_id, blob_id, permission, callback(result)) – is used by the owner of a blob to revoke permissions granted to another user. Only the owner of the specified blob is allowed to use this function. He can revoke any permission of any user that exists in the ACL of the respective blob, even if that user gained the permission from another user, other then the owner.

READ (id, v, buffer, offset, size, callback(result)) – the primitive does that same thing as before, the algorithm is the same, but the protocol used to interact with other entities has changed and can be seen in Fig.7.

WRITE (id, buffer, size, offset, callback(v)) / APPEND(id, buffer, size, callback(v)) – this primitive also uses a new communication protocol. It can be seen in Fig. 8.

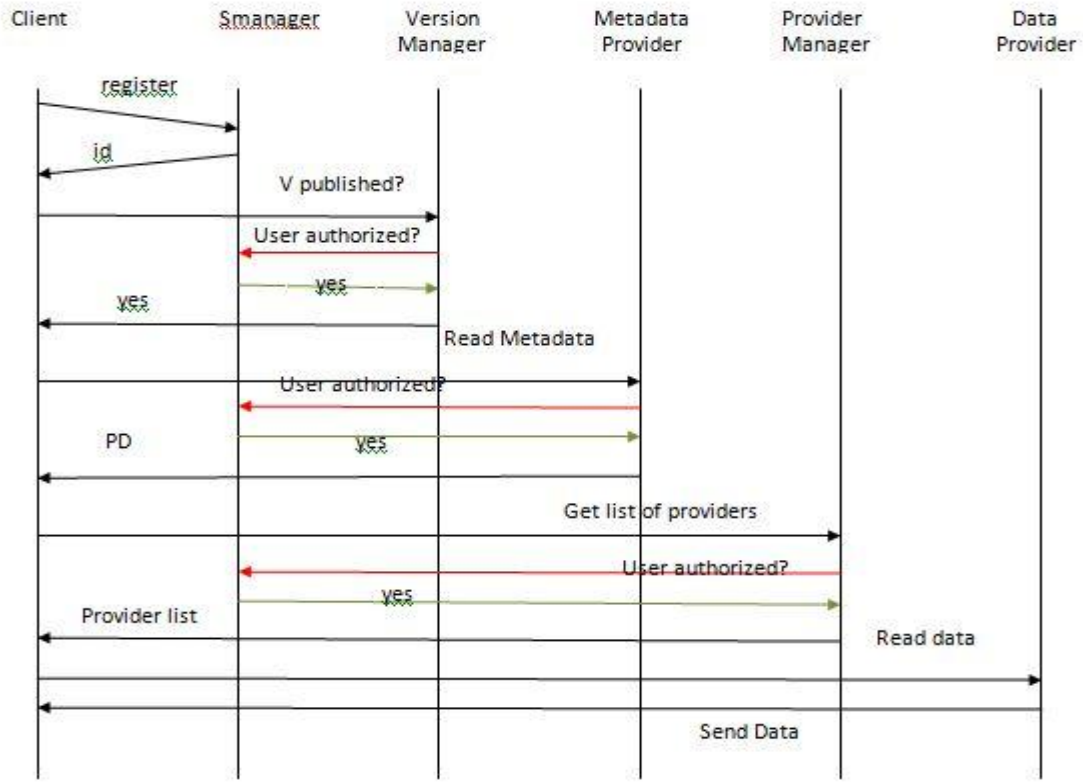

Fig. 7. Read Protocol

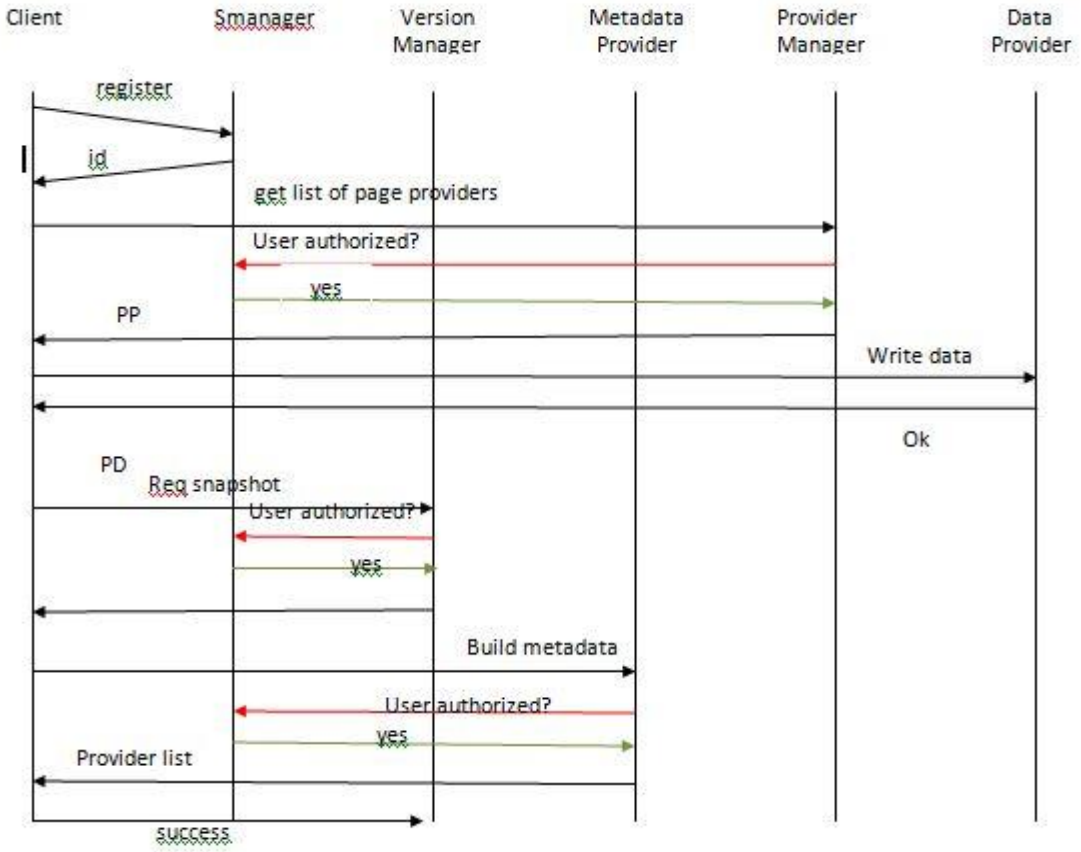

Fig. 8. Write Protocol

6. Implementation details

In the previous chapter we described in detail the new features introduces by the new security layer. In this chapter we will talk about how they are implemented and also take a look at the OpenSSL library, the tool we used to implement the security module.

6.1 OpenSSL

OpenSSL is a general purpose cryptographic library [20] that provides implementation of the industry's best regarded algorithms, including encryption algorithms such as AES, 3DES and RSA, as well as message digest algorithms and message authentication codes. It also provides pseudorandom number generation and support for manipulating common certificate formats and managing key material. It is the only free, full featured SSL implementation currently available fot use with C and C++ programming languages. It works across every major platform, including Unix, Linux and Microsoft Windows.

Here is a series of libraries that we used in the implementation of the security layer:

* **openssl/evp.h** – the EVP API – provides an interface to every cipher OpenSSL exports. Amongst many, it provides functions for symmetric encryption, such as EVP_Encrypt/Decrypt, for public key encryption, using EVP_PKEY objects, functions for message digest computing and digital signatures.

- **openssl/rsa.h** - it's a library used for generating RSA keys, for public key encryption using the RSA algorithm and for data signing and verifying.
- **openssl/pem.h** – allows for reading and writing PEM (Privacy Enhanced Mail) objects. PEM data is base64-encoded and provides the ability to encrypt data before encoding it.
- **openssl/x509.h** – provides functions for manipulating X509 certificates. This is one of the most important features of OpenSSL as X509 is the most widely used format for digital certificates. The most recent version of this format is the X509v3, for which all modern applications offer support.

6.2 Implementation details

The security manager is made up of several classes of which the most important are:

- **SymCipher** – is an abstract base class for the symmetric encryption algorithms implemented in BlobSeer. It implements the basic algorithm of any private key encryption algorithm. It has several pure virtual functions such as: getKey(), getIV(), getKeyLength(), getIVLength(), enc(), dec(). The only methods implemented by this class are encrypt() and decrypt() which implement the actual algorithms for encrypting and decrypting data. These methods internally call enc() and dec() which must be implemented by derived classes.
- **AesCipher** – is a class that implements the AES algorithm and also extends the abstract class SymCipher. The class supports keys of 128, 192 and 256 bits and can work in ECB or CBC modes.
- **3DesCipher** – is a class that implements the 3DES algorithm and, like, AesCipher, extends the abstract class SymCipher. The cipher modes supported by this class are ECB and CBC.

- **HashCipher** – is another class that extends SymCipher but it does not implement an encryption algorithm. This class is used to compute the hash of some data in the same way the data can be encrypted using a symmetric cipher
- **ClientInfo** – is a class that keeps information about a registered user. It stores the user's name, the digital certificate and public key, the unique user id and a list of pairs (blob_id, key) that keep the secret key and iv for every blob it has access to. The key and iv are encrypted with the user's public key.
- **SManager** – is the main class of our security layer. It implements the methods presented in 5.2. The class implements the client-server architecture, being implemented as a RPC server. Each method implemented is identified through a unique id and can be called by users using this id. The class store several lists: a list for each blob in the system – the blob's access control list which keeps track of which users has access to the respective blob. Also it keeps a list of ClientInfo classes representing the list of registered users.

7. Experimental results

The testing environment used is Intel Core 2 Duo 2.2 GHz with 4 GB of RAM. The operating system that runs on this system is an Ubuntu 11.04 with 2.6.38 Linux kernel version. The system was build using g++ 4.5.2 compiler and Boost 1.46 and OpenSSL 1.0.0d libraries.

We tested out solution on the basic operations available in BlobSeer: writing to a BLOB and reading from a BLOB. We first run a set of reads and writes using the initial version of BlobSeer and then another set using the modified BlobSeer. The tests consisted of writing and reading 4 sets of random data of different sizes: 100MB, 256MB, 512 MB and 768 MB. In the second set of test we run the tests using some of the implemented encryption algorithms: AES with keys of 128 and 256 bits in ECB and CBC modes. Also the system used secure communication between entity through the use of public key encryption and digital signatures.

The BlobSeer configuration used to run these tests used a version manager, a metadata provider, a data provider, a provider manager and a security manager. The system used pages of 64KB with no data replication or compression.

The results of the test can be seen in the following figures. Fig. 10 shows the time it took to write the data, while Fig. 11 presents the result of the read operations.

As we can see, there is an overhead in the system, but we consider that it can be overlooked in comparison to the benefits the security module offers.

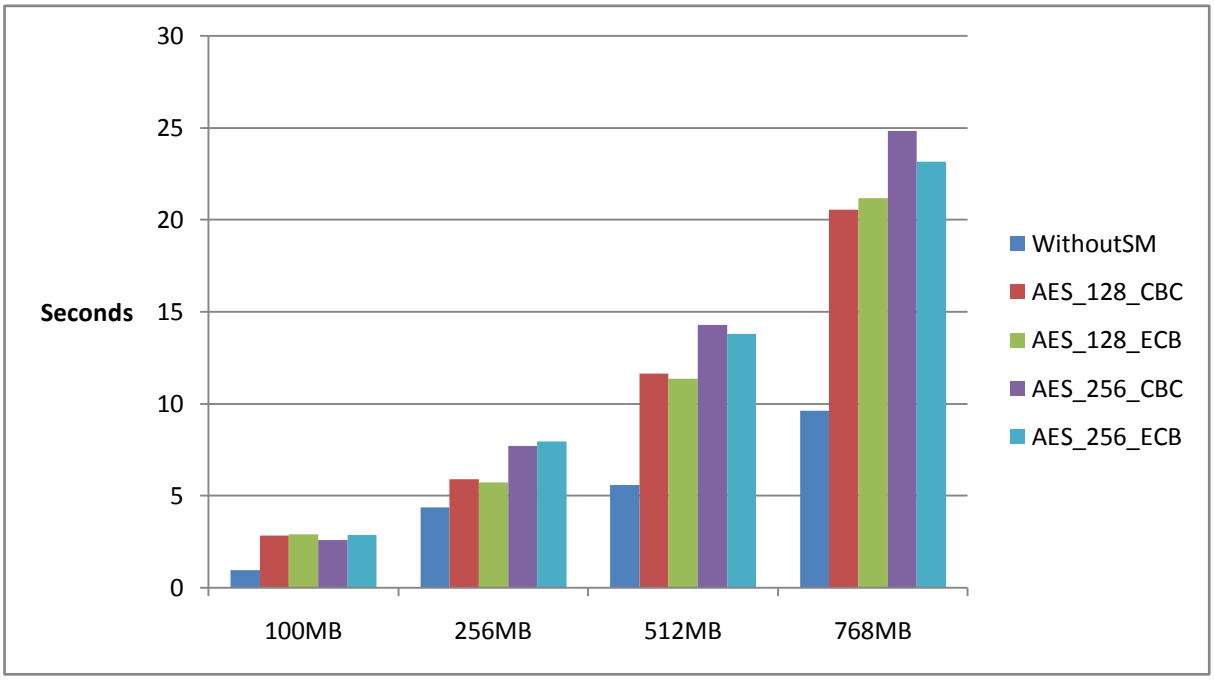

Fig. 10. Speed of writing data

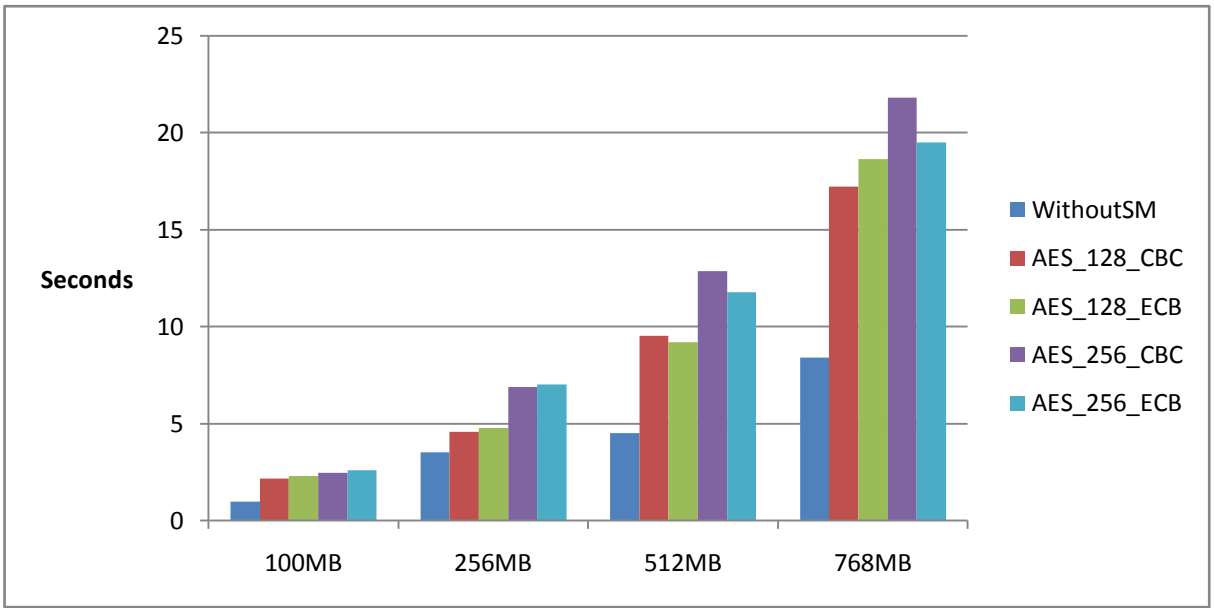

Fig. 11. Speed of reading data

8. Conclusions and future work

In this thesis we have presented our approach in implementing a basic security module that has made BlobSeer a safer distributed data management system. We consider that we have succeeded in implementing a security strategy that addresses the issues of data security and integrity, secure client access and secure communications.

Even though testing showed that the module generates some overhead, it is not very large and can be overlooked in comparison to the newly gained benefits of a secure system.

As future work we propose to add some improvements to the Security Manager. At this moment it generates a single point of failure that can be fatal for the system. Also, when the number of concurrent users greatly increases, the SM can be flooded with client requests that he cannot process fast enough and, so, reduce the high throughput of the system. To solve this issues we propose that the SM be replicated on multiple machines and even have multiple threads that can serve client requests in parallel.

We must also improve the way user information is kept in the system. Now the SM stores everything in memory and in the event of a crash all data will be lost. We propose that the data be stored on the disk and to have in memory just a cache with a limited amount of entries. The algorithm for this cache will be based on the least recently used strategy. Also, a similar cache should be used in the other entities (version manager, provider manager, etc.) so that the security manager won't get so many requests.

Another optimization should be done regarding user access. The system should support a history of what rights a user has granted and to whom.

BIBLIOGRAPHY

[1] M. Mosley "DAMA-DMBOK Guide (Data Management Body of Knowledge) Introduction & Project Status",
DAMA International, http://www.dama.org/files/public/DI_DAMA_DMBOK_Guide_Presentation_2007.pdf 2007

[2] B. Nicolae, G. Antoniu, L.Bougé, D.Moisem, A. Sánchez, M. S. Pérez, "Using Global Behavior Modeling to Improve QoS in Large-scale Distributed Data Storage Systems"

[3] F. Wang, S. Oral, G. Shipman, O.Drokin, T. Wang, I. Huag, "Understanding Lustre Filesystem Internals", UT-BATTELLE, 2009
http://wiki.lustre.org/images/d/da/Understanding_Lustre_Filesystem_Internals.pdf
fetched at 03.04.2011

[4] "Lustre Architecture – Security", Oracle, http://wiki.lustre.org/index.php/Architecture_-_Security
Fetched at 03.04.2011

[5] "Hadoop DFS User Guid 2007", the Apache Software Foundation 2.0 http://hadoop.apache.org/ 2007
Fetched at 03.04.2011

[6] Dai Yuefa, Wu Bo, Gu Yaqiang, Zhang Quan, Tang Chaojing, "Data Security Model for Cloud Computing", International Workshop on Information Security and Application
21 Nov. 2009, http://www.academypublisher.com/proc/iwisa09/papers/iwisa09p141.pdf

[7] "Amazon Simple Storage Service (Amazon S3) Developer Guide",
http://docs.amazonwebservices.com/AmazonS3/latest/dev/
fetched at 16.04.2011

[8] "S3 Backup FAQ", maluke.com, http://www.maluke.com/software/s3-backup/faq fetched on 24.05.2011

[9] B. Nicolae, G. Antoniu, and L. Bougé, "Enabling High Data Throughput in Desktop Grids Through Decentralized Data and Metadata Management: The BlobSeer Approach," in Proc.Euro-Par '09: 15th International Euro-Par Conference on Parallel Processing, Delft, The Netherlands, 2009, pp.404-416.

[10] V.-T. Tran, G. Antoniu, B. Nicolae and L.Bougé, "Towards a Grid File System Based on a Large-Scale Blob Management System", in "CoreGRID ERCIM Working Group Workshop on Grids, P2P and Service Computing (2009)"

[11] B. Nicolae, G. Antoniu, L. Bougé, D. Moise and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," J.Parallel Distrib. Comput., vol. 71, pp.169-184, 2011.

[12] B. Nicolae, G. Antoniu and L. Bougé, "BlobSeer: Efficient Data Management for Data-Intensive Applications Distributed at Large-Scale", in "24th IEEE International Symposium on Parallel & Distributed Processing", 2010

[13] B. Nicolae, G. Antoniu, L. Bougé, "BlobSeer: How to Enable Efficient Versioning for Large Object Storage under Heavy Access Concurrency," in Proc. EDBT/ICDT '09 Workshops, Saint-Petersburg, Russia, 2009, pp. 18-25.

[14] E. Cole, R. D. Krutz, "Hiding in Plain Sight: Steganography and the Art of Covert Communication", Wiley Publishing, Inc., Indianapolis, 2003

[15] W. Stallings, “Cryptography and Network Security Principles and Practices, Fourth Edition”, Prentice Hall, 2005

[16] H. C.A. van Tilborg, “Fundamentals of Cryptology – A Professional Reference and Interactive Tutorial”, Kluwer Academic Publishers, New York, 2000

[17] T. St Denis, S. Johnson, “Cryptography for Developers”, Syngress Publishing, Inc., Rockland, MA, 2007

[18] W. Mao, “Modern Cryptography: Theory and Practice”, Prentice Hall PTR, 2003

[19] R. S. Sandhu and Pierangela Samarati, “Access Control: Principles and Practices” in “IEEE Comunications Magazine, September 1994”, pp. 40-48

[20] P. Chandra, M. Messier, J. Viega, “Network Security with OpenSSL”, O’Reilly, 2002