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

# DIPLOMA PROJECT

BlobSeer Client Access Control

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

Raluca Andreea Baban

**BUCHAREST**

2011

## Table of Contents

# ABSTRACT

*In recent years large scale distributed systems have enjoyed growing attention from the IT industry as a viable solution for managing large amounts of data. As more applications are being implemented as to sustain or use such systems, an important issue arises, regarding both the application's and the system's security.*

*In this thesis we propose a novel security Layer for the BlobSeer data management system. We discuss what security actually means for a distributed file system as well as have a look at a few good-practice security models and algorithms available for data management services. In particular, we discuss BlobSeer's architecture and mode of operation whilst proposing a number of security enhancements to ensure a practicable and secure client access management.*

# 1. Introduction

It is considered that we live in an informational era. More and more applications are working with huge quantities of data and it is considered that every year approximately 50% more information is generated than the previous year, predictions existing that by the end of 2011 the world will have generated 1.8 zettabytes. Apart from this, more applications are working with huge quantities of data (data-intensive applications). Government and commercial statistics, cosmology, genetics and bio-engineering are just some domains where it is crucial that when working with large amounts of information, effective data management is achieved, free from any security incidents.

Obtaining a scalable and secure data management system is a true goal in itself, more so when not every system user has the best intentions at heart regarding said system. Of course we are referring to viruses, spy-ware, mall-ware, system attacks, and other objects and actions that may harm a system and/or compromise its recourses. It is not enough that only a system's components are secure, but special attention must be given to who is allowed to access the system, and what actions this person is allowed to perform, on what system objects.

Although BlobSeer [1] is a well rounded distributed file system, it is our opinion that it lacks a viable enough security system, or to be more specific, a viable enough client access control. The current implementation of such a feature is based on ip-filtering, insufficient for machines behind NATs (Network Address Translators) and with all the tools on the IT market or open-source tools available for security, it is a pity not to take advantage of them to implement a well rounded security system that complies or even exceeds the security standard for distributed systems. The main objective of this thesis is to propose an efficient client access control strategy for the BlobSeer system.

This main objective may be achieved, by fulfilling the following sub-objectives:
- investigate and discuss already existing technologies regarding client access control in data distribution systems
- analyze a number of security strategies and algorithms.
- understand the BlobSeer architecture, and mode of operation.
- formalize a personalized BlobSeer client access control system.
- evaluate the effectiveness of the proposed system.

A discussion about security systems and distributed systems in general is found in chapter 1, followed by a more detailed look at systems similar to BlobSeer (in chapter 2), and at the true meaning of security in distributed data management systems (in chapter 3). In chapter 4 we familiarize ourselves with BlobSeer, its architecture, main features, primitives and algorithms. Our proposals for a BlobSeer security Layer is thoroughly handled in chapter 5. Chapter 6 presents a series of tests and their conclusions regarding the security Layer implemented; suggestions are made regarding possible future work in chapter 7, as well as a number of final conclusions regarding the thesis.

## 1.1 Distributed Systems

We live in an era where a system's capability of storing and processing large quantities of data is becoming a requisite feature. The chosen solution for this problem is the adoption of a distributed computer paradigm in which a distributed service infrastructure is adopted, as to achieve scalability and performance. Such a system comes in a number of forms, from autonomous processes that run on the same physical computer and interact through message passing to computer networks where individual computers are physically distributed within a random geographical area. When we refer to a distributed system in this thesis, we refer to a group of autonomous computers that communicate with each other through a computer network in order to achieve a common goal.

In recent years, distributed systems have undergone an unprecedented growth. Today, through local and wide area networks, a very large number of computers and micro-computers are connected world-wide. If we also take into account notebooks, mobile telephones and other computing devices, we will find millions of machines that have the potential to communicate with each other by means of cable, infra-red, Bluetooth etc. If we are to take into consideration the fact that each machine has at least one central processing unit (CPU) that is not used 100% of the time, and that each machine has some free storage space, then, as whole, we have an enormous storage capacity and processing power available for use, at a low cost: instead of buying and managing hardware, one may rent virtual machines and storage space.

## 1.2 Distributed data management

Another important aspect of a distributed system is represented by data management, which, according to the DAMA Data Management Body of Knowledge (DAMA-DMBOK), has the following definition: "Data management is the development, execution and supervision of plans, policies, programs and practices that control, protect, deliver and enhance the value of data and information assets"[2].

Distributed data management, not only refers to the way data is managed, but also the way it is distributed over a network. Problems such as fault tolerance, performance, throughput, scalability are critical to the operation of a data management system[3].

There are many advantages in using a distributed data management system [1], such as better data placement (closer to it's source or where it is most needed), higher data availability through data replication, higher fault tolerance through elimination of a single point of failure, potentially more efficient data access (higher throughput and greater potential for parallelism), better scalability, and the list goes on. Because of these advantages, in recent years there has been a growing interest in this specific type of data management and a number of systems have been developed, with the distributed data architecture as their basis, to offer clients the means of handling large quantities of data in an easy, transparent manner.

Out of the many distributed data management systems, we will take a quick look in this thesis at Hadoop Distributed File System (HDFS) - "the primary storage system used by Hadoop applications. HDFS creates multiple replicas of data blocks and distributes them on compute nodes throughout a cluster to enable reliable, extremely rapid computations"; at Amazon S3 – "a simple web service interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, fast, inexpensive data storage infrastructure that Amazon uses to run its own global network of web sites"; we will also take a look at Lustre File System – "a massively parallel distributed file system, generally used for large scale cluster computing" and at NFS (Network File System) – a client-server file system developed by Sun Microsystems, that allows users to access files across a network as if they resided in a local file directory.

## 1.3 Security issues

Independent of the type of infrastructure used, security remains a major issue for any system, more so in a distributed context where we have to deal with decentralized control, remote data access as well as critical data manipulation. Some important security aspects [4] for distributed computing, which will be looked into, later in this thesis, are:

**Data confidentiality** - no one else but the allowed users may have access to data; outsiders may not have access to a conversation between parties using a particular system; no system information may be accessed by unauthorized users.

**Data integrity** - data that has a complete or whole structure on at least one machine that makes up the system; ensures that data is not altered by unauthorized users in a way that is undetectable by authorized users.

**User authentication** - the process of verifying a claim made by a subject, that it should be allowed to access and modify a specified object.

**User authorization** - the process of verifying that an authenticated subject has the permission to perform a certain operation or access certain resources.

**Access control** - the process of modifying a subject's permissions to perform operations and access data – implemented by the above mentioned user authentication and user authorization, amongst others.

**Availability** - the process of ensuring that a system is operational and functional at any given moment, something achieved through redundancy.

It is important for a computer system to have a **trusted computing base (TCB)**, which refers to the whole set of hardware, firmware and software components that are critical to the system's security, in the sense that bugs or vulnerabilities occurring inside the TCB might jeopardize the security properties of the entire system. Apart from this, parts of a computer system outside the TCB must not be able to misbehave in a way that would leak any more privileges that are granted to them in accordance to the system's security policy. The careful design and implementation of a system's trusted computing base is crucial to its overall security.

## 2. Related work

As discussed in chapter 1, there are quite a few applications out there, which implement a distributed data management system. In the coming chapter, we talk about the applications mentioned earlier (in chapter 1.2), showing their architecture, basic data handling principles, as well as underlining their adopted security protocols. This will help us better understand what security features a distributed system should have as well as the effectiveness of said security features.

Management systems similar to BlobSeer, such as the Hadoop Distributed File System, Amazon S3, the Lustre File System and the Network File System, are treated in the following paragraphs, taking a look at their approach to working with distributed data and their approach to assuring a secure system.

### 2.1. Hadoop Distributed File System

The Hadoop Distributed File System[5] is the primary storage system used by hadoop, a popular open-source MapReduce framework that has been widely adopted in the IT industry, by cooporations such as Yahoo! and Facebook. HDFS, a java written application, has a master/slave architecture at its basis. It works with large data sets of gigabytes to terabytes size and provides an interface for applications to move themselves closer to where the data is located. The master server in a HDFS cluster is a NameNode that manages the file system namespace and regulates access to files by clients. DataNodes, one per node in a cluster, manage storage attached to the nodes that they run on. Internally, a file is split into one or more blocks which are stored in the above mentioned DataNodes and replicated, for fault tolerance. These DataNodes are an important component of the system, being responsible for serving read and write requests from the file system's clients as well as being responsible for block creation, deletion and replication, as the NameNode requests (block operations). In Fig. 1, we have an illustration of the HDFS architecture explained above.

HDFS supports a traditional hierarchical file organization, exposing a file system namespace and allowing user data to be stored in files. Apart from the capability of working with quotes and hard or soft links, the system is able to create directories, store files in said created directories, create, remove and move files. The HDFS permissions system is designed to prevent accidental corruption of data within a group of users who share access to a cluster. Thus, HDFS security is based on the POSIX model of users and groups. Security permissions and ownership may be changed by using –chmod, -chown, -chgrp similar to the POSIX/LINUX tools. The Hadoop system is programmed to use the users current login as their Hadoop username, and the users current working group list is considered as the group list in Hadoop. HDFS itself does not verify that this username is genuine. The username that starts the Hadoop process is the super-user. There is also a super group, whose membership is controlled by the configuration parameter dfs.permissions.supergroup. HDFS does not enforce permissions except when permission-related operations take place (such as –chmod), otherwise the permission system may be disbled by setting to false the following option: dfs.permissions.

Although Hadoop is used on a large scale by the cloud community, it has some major security flaws [6],[7] which are deemed to be solved by the end of this year: first of all, Hadoop services do not authenticate users or other services and as a result, users may impersonate other users, malicious network users to impersonate cluster services; secondly, DataNodes do not enforce any access control on access to its data blocks, thus an unauthorized client may read or write data blocks. This being said, some Hadoop deployments use HDFS proxies for server to server bulk data transfer. Hadoop uses the proxy IP addresses, and a database of roles in order to perform a minimal user authentication and authorization.

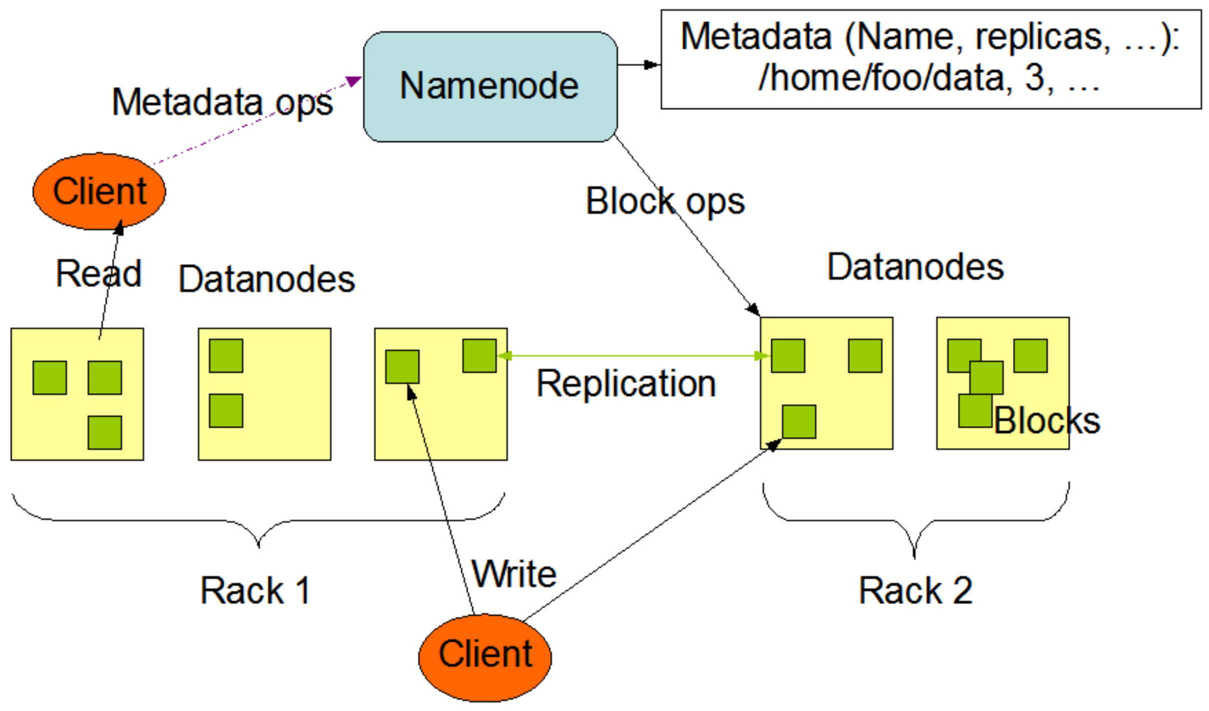

Fig. 1 The HDFS architecture

To improve security, programmers chose to use a Simple Authentication and Security Layer (SASL) with Kerberos instead of SSL, because of Kerberos's symmetric key operations, which are faster than the public key operations SSL provides and because of Kerberos's simple user management interface, for example, revoking a user can be done simply by deleting the user from the centrally managed Kerberos KDS (key distribution center). Communication between the client and the HDFS service is composed of two halves: a RPC connection from the client to the NameNode and block transfer from the client to the DataNodes. The RPC connection can be authenticated by Kerberos or via a delegation token (shared secrets between the NameNode and the client that allow authentication without using the Kerberos Key Servers).

To conclude, main concerns regarding HDFS security revolves around the poor default SASL Quality of Protection, the wide distribution of symmetric cryptographic keys, incomplete pluggable web UI authentication and the use of IP based authentication, elements which hopefully will be improved in the near future.

## 2.2 Amazon S3

Amazon S3 is an online storage service offered by Amazon Web Services [8] that provides storage through web service interfaces such as REST, SOAP and BitTorrent. Details about S3's design and architecture are not made public by Amazon, but it is a known fact that Amazon uses a series of technologies to power parts of the Amazon Web Service (AWS), such as S3. One such technology is Dynamo, internally designed by Amazon to address the need for an incrementally scalable, highly-available key-value storage system. It has properties of both databases and distributed hash tables, and is considered to be an eventually consistent data store (all updates reach all replicas eventually, due to implemented replication techniques).

Dynamo [9] is a completely decentralized system with minimal need for manual administration, storage nodes being able to be added and removed without requiring any manual partitioning or redistribution. Data is partitioned and replicated using consistent hashing, and consistency is obtained by object versioning. Consistency amongst replicas is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs a gossip based distributed failure detection and membership protocol.

S3 [10],[11] stores objects of up to 5 terabytes of data, each accompanied by 2 kilobytes of metadata. The objects are stored into buckets owned by an Amazon Web Service account, and identified by a unique, user-assigned key. Buckets and objects can be created, listed and retrieved using either REST HTTP interface or a SOAP interface. The user is able to control who is able to upload or download data into his Amazon S3 bucket.

Buckets are containers for objects stored in Amazon S3 [8], they 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 (the access control policy associated with a bucket governs the way an object is created, deleted and enumerated) and they also serve as the unit of aggregation for usage reporting. They may be configured to support versioning; also, they are designed in such a way as to enable HTTP URL addressing (http://s3.amazonaws.com/bucket/key) with an option that allows setting a time-bound validity.

Objects are the primary entities of the Amazon S3 architecture [8]. They consist of object data, opaque to Amazon S3, and object metadata, a set of name-value pairs that describe the object (object metadata contains default metadata, such as the date last-modified, standard HTTP metadata such as Content-Type, as well as custom metadata, defined by the developer at the time the Object is stored).Amazon S3 also makes use of keys, to uniquely identify an object within a bucket. Every Object has just one key, but may have different version ID, if versioning is enabled.

Requests are authorized using an access control list (ACL) associated with each bucket [11] and object, each ACL custom implemented by the developer. When a bucket or an Object is created, a default ACL is created, that grants the owner full control over the resource. Permissions to read/write a bucket or an object, or full control permissions can be granted by the element's owner, to an AWS account, or one of the predefined Amazon S3 groups.

Apart form controlling ACLs, the owner of a bucket, may develop bucket policies (in JSON) to define access rights for his resources: allow or deny bucket-level permissions, deny or grant permissions on an object from the bucket.

In 2011, a virtual application has been developed that allows the encryption of S3 objects by an AES-256 algorithm and assures data integrity with the SHA-1 algorithm [10],[11], upon installation into ones AWS account. Other from this application, S3 has an option to encrypt data with Blowfish or AES. Also, S3 protects online storage via password, although, due to the bucket policies, such protection is unnecessary, if the untrusted party doesn't have privileges on that machine.

In conclusion, Amazon S3 is a safe distributed data management system that allows the owner of buckets, and Objects, to implement the security strategy of his or her choice: ACLs, bucket policies, key generation, encryption/decryption etc, but the user level security may be considered insufficient since it is only based on a query-string authentication mechanism (which passes three additional URL parameters, the AWS access key, expiration of the URL and the signature which is actually an encrypted string) which guarantees that a recourse is unavailable to someone who does not have the signature, so misplacing of the URL may easily compromise ones security.

## 2.3 Lustre File System

Lustre [13] is an open-source massively parallel distributed filesystem developed and maintained by Sun Microsystems. Lustre, an object-based filesystem , with data usually in ext4 format, supports tens of thousands of clients, ten of pentabytes of storage and hundreds of gigabytes per second of IO throughput and is very popular in scientific supercomputing. It is composed of metadata servers (MDSs), object storage servers (OSSs) and clients; it has a total data capacity the sum of all individual object storage targets (OSTs).
Fig. 2 illustrates the Lustre architecture [13].

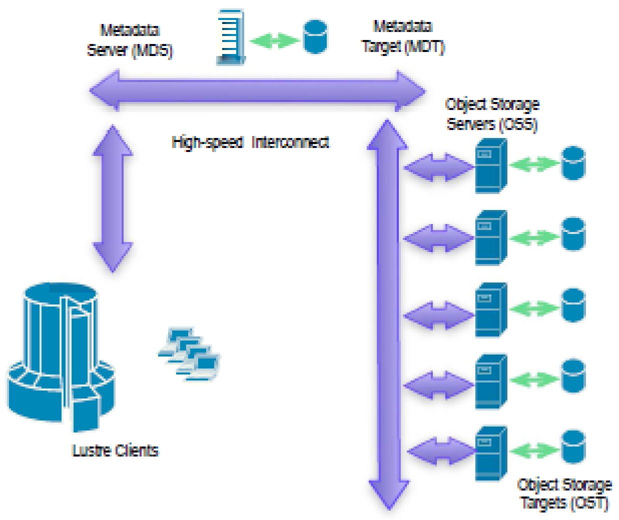

Fig. 2 The Lustre architecture

**Metadata Servers** (MDS) provide metadata services (used by metadata clients) and manages one metadata target (which stores file metadata: file names, directory structures, access permissions, etc).

**Management Servers** (MGS), hold configuration information of the Lustre filesystem.

**The object storage servers** (OSSes) expose block devices and serves data. They are the clients of these services, and the store at least one object storage target (entities which store file data objects).

The MDT, OST [13],[14] and client may be on the same node, but in a typical scenario these functions are on different nodes, communicating over a network. Lustre's network layer (LNET) supports a number of network interconnections, including native Infiniband verbs, TCP/IP on Ethernet, and other network technologies. To improve throughput and reduce CPU usage, Lustre takes advantage of remote direct memory access.

Being a POSIX-compliant filesystem, Lustre presents a **unified filesystem interface** (permitting operation such as open(), read(), write(), etc) to the user, a POSIX characteristic made possible in Linux by a Virtual File System (VFS) layer and also in Lustre due to a layer, hooked to the VFS, called Ilite. A file operation that reaches llite must go through the whole Lustre filesystem.

Lustre also has a distributed **lock manager**, to protect the integrity of each file's data and metadata. The metadata locks are managed by the MDT and are split into bits that protect the lookup of the file (file owner and group, permissions and mode, and ACL), the state of the inode (directory information, timstamps, etc) and the layout (file striping). Access and modification of a Lustre file is cache coherent among all the clients.

Security wise, all client/server communications in Lustre are coded as RPC requests and responses, also for improved security and scalability, clients may not modify directly the objects on the OST filesystem, but delegate instead the task to OSSes.

Lustre as the following security features [14]:
- the ability to have TCP connections only from privileged ports.
- server-based group membership handling.
- a security model that follows that of a Unix files system, enhanced with POSIX ACLs (system object permissions, read(r), write(w) and execute(x), that define three classes of user: owner, group and other).
- the inability of clients to see stale data or metadata, due to the implemented atomic operations.
- the recording of changes to feed into an archiving system.
- root squash functionality, which controls super user access rights to a Lustre file system (this consists of re-mapping the user ID (UID) and group ID (GID) to that of the root user UID and GID, specified by the system administrator, via the MGS. This feature allows the administrator to specify the UID/GID set of a client for which the re-mapping does not apply).

Unfortunately Lustre lacks built-in encryption features and for future work Lustre plans to use Kerberos for computer network authentication Furthermore, there is no restriction on client access (a possible work-around being to set up IP firewall rules to block clients), but even so, Lustre is a highly-used file system, integrated with several vendor's kernels and widely used by the IT community.

## 2.4 Network File System

The Network File System (NFS) [15] is a client/server file system, usually associated with UNIX [15], developed by Sun Microsystems that allows users to remotely access files, and treats them as if they resided locally. This is accomplished by the processes of exporting remote filesystems located on the server (the process by which NFS server provides remote clients with access to its files), and mounting locally, on the client and to the client system (the process by which file systems are made available to the operating system and to the user). It is designed not to depend on the computer, operating system, network architecture nor transport protocol. NFS uses a model based on the RPC protocol, for data exchange between client and server, and based on the external data representation (XDR) protocol, with the goal to define a common method for representing common data types, for data translation between heterogeneous systems.

The operation of NFS is defined in the form of three main components (which are more clearly portraid in Table 1) that can be viewed as logically residing at each of the three OSI model layers corresponding to the TCP/IP applications layer [16]. These are: the RPC protocol (a generic session layer for client/server internetworking functionality), the XDR protocol (resides on the presentation layer) and NFS procedures and operations (procedures and operations that usually function at layer sever of the OSI model, and that represent particular tasks to be carried out on the file over the network – the actual exchange of information between an NFS client and server).

Being similar in construction with the UNIX filesystem model, NFS has a similar Access Control Lists (ACL) support [17], for authentication and authorization. The file system takes into consideration the file's access controls, the caller's credentials, and other local system restrictions that might apply. NFS provides support to Role Based Access Control (RBAC), it provides its own ACL protocol, but also supports ACLs proprietary to AIX servers. RBAC support means that a non-root user is able to execute NFS commands once the administrator assigns the RBAC role of the command to the user.

A file can be uniquely specified by using its file name and path name, files and directories are organized as one would see in on a UNIX platform: inodes, dentrys etc. Each file has a set of permissions (read, write and execute) associated to it, to determine who has what permissions. File permissions are based on the type of user attempting to use the file: user, group, other (permissions given to user apply to the owner of the file, permissions given to group apply to a group of users to which the file belongs, and permissions given to other apply to users other than the owner and members of the owner's group).

Security wise, apart from the above discussed ACLs, NFS uses a few more security implementations that will be discussed in the up-coming paragraphs.

**The portmapper** [18] keeps a list of what services are running on what port. The list is used by a connecting machine to see what ports it wants to talk to, as to access certain services. There are two important files: **/etc/hosts.allow** and **/etc/hosts.deny** which the portmapper uses for usage control.

**Root_sqash** is a parameter that helps the server decide not to trust root requests from the client, but the root user on the client may use the **su** command to become any other user, to access and change that user's files. The TCP ports 1-1024 are reserved for root's use, so a non-root user cannot bind these ports.

*Table 1*

**NFS Components**

| Layer Number | OSI Layers | NFS Layers |
|---|---|---|
| 7 | Application | NFS |
| 6 | Presentation | XDR |
| 5 | Session | RPC |
| 4 | Transport | TCP / UDP |
| 3 | Network | IP |
| 2 | Data Link | Ethernet |
| **1** | Physical | Ethernet |

The client can control the amount of trust given to the server by the **nosuid** mount option. Setting this option will forbid **suid** programs to work of the NFS file systsm. Suid programs, such as unix's passwd, set the id of the person running them to whomever is the owner of the file. One can also forbid the execution of files on the mounted file system altogether with the **noexec** option. The **broken_suid** option, is used for protection against older programs that used to rely on the idea that root may write anywhere. Acting on such a presumption, will break new kernels on NFS mounts.

**Full file locking** is supported, meaning that **rpc.statd** and **rpc.lockd** daemons mush be running on the client in order for locks to function correctly. When an application wants to obtain a lock on a local file, it sends its request to the kernel. If an application on a NFS client makes a lock request for a local file, then the Network Lock Manager client generates an RPC call to the server to handle the request. When a client receives an initial remote lock request, it registers interest in the server with the client's **rpc.statd** daemon. The same is true for the network lock manager at the server.

**RPC/DH** is a protocol that uses a combination of Diffie-Hellman key exchange and DES encryption (later updated to AES encryption). User validation is performed by the server, based on information from the RPC request. The first time the client contacts the server, it generates a random session key, and encrypts it with the shared 64-bit DES key. The session key is also a 64-bit DES key. The client also generates a time-to-live value (windows), and a windows value that is one second less than the first windows value. The two window values are then encrypted with the session key. The server knows something is wrong if the second window value is not one less than the firsts, then the client request will be rejected. Also, on every subsequent request to the server, the client will encrypt the current time, using the session key. On the server side, if the timestamp is out of the bounds of the time-to-live window, then the clients request is rejected.

**Public and encrypted keys** are kept in the /etc/publickey file, and the NFS server along with a unique identifier for the machine or user owning the keys. The superuser can add user keys using newkey –u user, which requires a password, to encrypt the private key so that it can be safely added to the publickey maps.

When you log onto a machine running RPC/DH, the login-password supplied is used to attempt to decrypt your encrypted private key. If the login and RPC/DH passwords do not match, then you get an error.

It is recommended that **IPchains and netfilter** (depending on the different kernel distributions) be used to implement a higher level of access security, instead of relying on the NFS daemon. Another good-practice suggestion is to tunnel NFS through SSH (though this practice will affect file locking, disabling it) if one does not trust the local users on the server.

NFS provides **Kerberos authentication and authorization support**, by making use of the RPCSEC-GSSAPI interface defined by Sun. In addition to the three flavors of Kerberos security (authentication, integrity checking and full privacy), the said interface will eventually support other security flavors such as SPKM3. NFS also supports DES and AES encryption.

NFS works with **superuser mapping**. It is considered that root access should be granted on a per-machine basis, thus the superuser is not given normal file access permissions. To enforce this restriction, the root’s UID is mapped to the anonymous user ‘nobody’ in the NFS RPC credential structure. Also, to this structure, NFS requests that do not have the necessary credentials are mapped. Credentials can be anything from authentication information, username and password etc.

NFS has the means of protecting filesystems hosts as well as clients, for example, data may be made available only to a selected set of machines. Also, you may specify whether the filesystem is exported read-only, and to what host, changing the default write access enabled filesystem export.

Over all, NFS is a strong reliable and secure file system that implements a large set of security measures, to assure customer satisfaction. And although a new version has not appeared in some time, the current available versions are versatile enough.

# 3. Security for data management systems

Security is a major issue for distributed systems, more so as the increase of connections result in greater interdependency of the system, less centralized control and more remote data accessing, leaving us with an environment with many challenges regarding data security management. As presented in the previous section, all the current distributed data management system include security mechanisms, which are critical to the usability of the filesystem. When we talk about security, we need to talk about secure communication, secure access and security management. This thesis will discuss all three security aspects, with an emphasis on data management security.

First of all, secure communication refers to the prevention of unauthorized third party interceptors accessing communications in an intelligible form or changing the communications, while still delivering content to the intended recipients.

Secondly, when we refer to secure access we mean that a user or process may use (or may have access to) services or recourses only if the available permissions allow it. Lastly, when we talk about security management, we refer to key management (key creation, distribution) user management, group management, etc.

A secure data management system must implement the above mentioned attributes - secure communication, secure access, and security management – while making sure to cover the security issues raised in paragraph 1.2.

## 3.1 Secure communication

Secure communication [19] includes means by which people can share information without third-party interception. This is possible by adopting one ore more security methods, which will be described in detail in the following paragraphs.

**Encryption** is possibly one of the most frequently used security methods adopted in a distributed system. It consists of encrypting data with a key that only the people that are holding the conversation know of. A key may be generated either by a symmetric-key algorithm or a public-key algorithm.

**Symmetric-key algorithms** use a single shared key, that must be kept secret, in order to keep the data safe and may be used apart for encryption, for non-repudiation purposes. The encryption key is trivially related to the decryption key, meaning that the keys may be identical, or that there is a simple transformation between the two. In practice, the two keys represent a shared secret between the communicating parties. This specific algorithm may be divided into stream ciphers, that encrypt a message bit by bit, and block ciphers, which take a whole block of bits from the message (usually 64 bits, but there are algorithms, such as the Advanced Encryption Standard algorithm or AES, that works with blocks of 128 bits). Apart from the AES algorithm, we have the Data Encryption Standard or DES algorithm, the Blowfish algorithm and many others. These algorithms are usually used mainly for encryption, it is not recommended to use them for authentication.

**Public-key algorithms** (also knows as asymmetric-key algorithms, due to the was the keys are generated) use two keys, one to encrypt the plain text and another to decrypt, neither key being able to do both functions. These algorithms are much slower than the symmetric-key generation algorithms. The system may be a signature verifier (of data encrypted by the detainee of the private key), or private communication enabled, depending on the key published. The later is true if the encryption key is published whilst the former is true if the decryption key is published. Amongst the better known public-key generation algorithms we have the Diffie-Hellman algorithm, the Rivest, Shamir and Adleman (RSA) algorithm and the Digital Signature Algorithm (DSA). These algorithms used for encryption but they are also used for authentication, due to the fact that we have a private key and a corresponding public key). Also from amongst the widely used protocols which implement asymmetric key algorithms we mention the Secure Shell (SSH) and the Secure Socket Layer (SSL) protocols – both represent a standard security technology that establishes an encrypted link between a client and server, and makes sure the data remains private and integral; this is also a good tool for client authentication.

**Public-key Certificates** are also known as digital certificates or identity certificates. The main purpose of certificates is to ensure that the issuer of the public key contained in the certificate is the same entity to which the certificate was issued. The certificate is conceived and signed by a commonly trusted third-party, the Certificate Authority (CA). The certificate is signed after the CA verifies that the public key matches the given identity (name, e-mail and other information). There are a large number of applications and protocols that implement this security strategy, amongst which probably the most widely used is the SSL protocol.

These certificates can also be used for authentication (a subject discussed in section - 2.2.2, subsection 2.2.2.1), because of the fact that the information contained by the certificate needs to be checked when received.

Another method for assuring secure communication is **Steganography**, the hiding of data inside more innocuous data. Thus, a watermark proving ownership is embedded in the data of a picture for example.

**Identity based networks** represents yet another method that assures a secure environment for communication since the identity of the receiver and sender are known.

**Anonymized networks** have also recently been used to secure communication, making use of the principle the more noise there is, the less you know where the source is. Unfortunately there are applications that monitor communications over entire networks, so anonymized networks alone is not enough to ensure security.

**Secure RPC** [20] is a security protocol built into the RPC software. It is based on DES for encryption, and has an open-ended authentication, meaning that a variety of authentication systems can be plugged into it, such as UNIX (used by a network service; the credentials contain the clients host name, UID, GID and group-access list), Kerberos (a MIT authentication system that will be discussed in section 2.2.2), DH (a combination of DES encryption and Diffie-Hellman public-key cryptography for authentication), etc.

## 3.2 Secure access

When discussing secure access in a distributed system, one automatically thinks authorization and authentication. In the current section we will discuss both of these security aspects (authentication is discussed in subsection 3.2.1 and authorization in 3.2.2), by example of already existing applications, but we will also talk about access control lists and access models.

First of all, it is important that when initiating a conversation, both the client and server authenticate themselves. In a more general manner, it is important that all the entities taking part in the communication be authenticated correctly.

The most rudimentary form of authentication, is that achieved by sending a client's username and password, but this method is inadequate for a distributed system, more so it the data in not encrypted, due to various applications that monitor network packets to find unprotected data to use in a malicious way. There are a number of authentication protocols, some of which will be discussed in the following paragraphs, such as the above-mentioned UNIX authentication, Kerberos authentication.

### 3.2.1 Authentication

**The UNIX authentication protocol** [21]– amongst other applications, it is used by NFS.

- on the client side, the programmer creates a Remote Procedure Call (RPC) client handle and then sets the authentication parameter. Each remote procedure call associated with the client then carries a pre-defined UNIX-style authentication credentials structure, which contains information such as client's UNIX effective UID, client's current group id and client host name.
- on the server side, a request handle is passed to a service dispatch routine. The handle contains information on the service program number, service protocol number, desired procedure number, raw credentials from wire, and credentials (in read only format). Except for the style (or flavor) of the authentication protocol, the raw credentials from wire structure is opaque, containing the following data: style of credentials, address of more authentication information, maximum length of credentials. RPC guarantees that before a request is passed to the service dispatch routine, the raw credentials from wire are in an acceptable form (thus the programmer may inspect what style of authentication was used) and that the read-only credentials field is either null or points to a well-formed structure that corresponds to a supported style of authentication credentials, in this case, an **authunix_params** structure, for UNIX-style authentication.

**SSL** [22] – a certificate based authentication tool. Certificates are digitally signed documents which bind a public key to the identity of the private key owner. With SSL, authorization is performed by an exchange of certificates, which are blocks of data in a format described in ITU-T standard X.509. The X.509 certificates are issues and digitally, signed by an external authority, known as a certificate authority. A certificate contains: two distinguished names (which uniquely identify the user), a digital signature (created by the certificate authority, using the public-key encryption technique), the subject's domain name, and the subject's public key. A free variant of the SSL tool, is the open-source OpenSSL.

**Kerberos** [23]- a computer-network authentication tool, which works on the basis of "tickets" to allow nodes communicating over a non-secure network to prove their identity to one another in a secure manner. Aside from assuring mutual authentication, this useful tool provides authorization and message confidentiality. Kerberos builds on symmetric-key cryptography and requires a trusted third party. It may optionally use asymmetric-key cryptography as well, in some parts of the authentication procedure. It is based on an authentication server that at all times knows the identities off all the clients that are logged-in. The client and authentication server share a secret password and a session key no other entity knows about. In a standard implementation, a Kerberos key is valid for 25 hours and is created using the DES algorithm.

### 3.2.2 Authorization

When we talk about authorization we talk about access rights to resources. There are different types of access models, regarding access and regarding the flux of information. In this thesis we will discuss some of these access models. Access rights can be discretionary (Discretionary Access Control - DAC), meaning that users may delegate their rights to other users, or mandatory (Mandatory Access Control - MAC), meaning that rights cannot be transferred from one user to another.

**Access Control Lists** (ACLs) – when referring to a computer file system, ACLs represent a list of privileges and permissions attached to an object (such as a process, a program or a file), which specify access rights such as whether a subject (a user or a process) may read, write or execute an object. Each accessible object contains an identifier to an ACL entry (or more commonly known, an access control entry - ACE), which specifies a subject and an valid operation. When a subject requests an operation on an ACL-enabled system, the operating system first checks to see if an entry exists for that specific subject and for that specific object. An ACE may even determine if a user or group of users may alter an object's ACL.

**The Graham-Denning model** – presumes that every object has a subject that is its owner (special rights on the former), and that every subject has another subject that represents its controller (a subject with special rights on the former). This is a DAC model, and before any changes can be made to the object, subject or rights, a set of preconditions must be met. The rights are set in an access matrix with a row for each subject and a column for each subject and object.

**The Harrison-Ruzzo-Ullman model** (HRU) [24] – is similar to the Graham-Denning model explained above. It consists of a set of generic rights R and a set of commands C. The subjects are required to be part of the objects, so the access matrix contains one row for each subject and one column for each object and subject. A complex command is treated as a whole, meaning that a failing operation in a sequence will result in the whole sequence failing. A HRU sequence is secure regarding a right "r" if no series of changes will transform the access matrix in such a way that the right will leak (a rule "r" is considered to have leaked if after the execution of a command "c", "r" will exist in the access matrix in a position in which it was not, before the execution of "c").

**The Role Based Access Control model (RBAC)** [25] – a role is a grouping mechanism for subjects, based on a set of the subjects common attributes. In this model, the roles have associated permissions, not the subjects as we have encountered before, and each subject can have one ore more roles. These roles can be easily assigned or revoked, a user may easily go from one role to another, roles can be contained in a hierarchy (for example the chief secretary has the role of a normal secretary, apart from other additional rights). Although it's easier to work with roles, there are a few extra constraints which is wise to implement, such as Static Separation of Duty Relations (two roles cannot be assigned to the same user) and Dynamic Separation of Duty Relations (two roles cannot be active at the same time for the same user). This model is DAC and MAC compliant.

**The Task Based Access Control model (TBAC)** [26] – adds to the traditional models (consisting of subjects, objects and permissions), context information regarding tasks (usage and validity counts, and authorization-steps), and each task having a state of protection (determined by the permissions). Depending on the context, the permissions are activated or deactivated, as the tasks evolve in time.

## 3.3 Security management

Security management refers to the generation and management of keys, secure group management (adding and removing groups, users to groups), and authority delegation.

Authority delegation in this sense is enabled by using a special object (certificates or tokens) by which the holder is granted the same (or fewer) permissions as the person who created the object initially.

Key generation and management, this has already been discussed in the above subsections of 2.2, as for secure group management, it is not the object of this thesis to discuss this issue due to the fact that BlobSeer does not work with user groups but with individual users.

# 4. Introduction to BlobSeer

In this chapter we introduce BlobSeer, a distributed data management service, designed to work with data intensive applications. BlobSeer's architecture is presented, BlobSeer's main features, primitives, and mode of operation likewise.

## 4.1 BlobSeer Architecture

BlobSeer[28] consists of a number of distributed processes that communicate through remote procedure calls. These processes and the way they interact with each other are illustrated in Fig. 3, and are discussed thoroughly in the paragraphs to come.

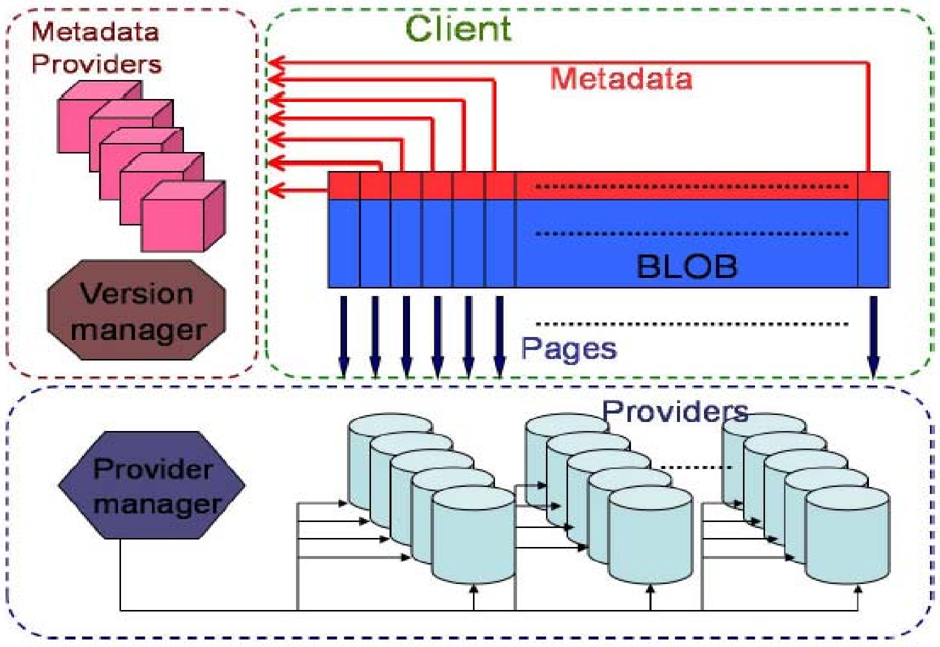

Fig. 3 Global architecture of BlobSeer

**Clients** – create a BLOBs and read, write and append data from or to BLOBs. It is expected that a large number of concurrent clients simultaneously access the same BLOB. The number of clients may vary in time, but the system is not notified of this change.

**Data providers** – physically store the data and manage the pages generated by write and append requests. New providers may dynamically join and leave the system.

**The provider manager** – keeps information about the available data providers, available storage space and schedules the placement of newly generated pages, according to a load balancing strategy.

**Metadata providers** – physically store the metadata, allowing clients to access pages by using corresponding BLOB versions. A distributed management scheme is adopted for the metadata, as to allow efficient concurrent access.

**The version manager** – assigns a BLOB version number to each update request (which is represented by a write or append) and then publishes the new version number, guaranteeing total ordering, atomicity, and that the BLOBS to which the version numbers refer to, have successfully been generated, and that their corresponding data and metadata will never be modified again.

## 4.2 Main features

In the following section, we will discuss some of BlobSeer's main features, which are essential to avoiding data-access synchronization and to distribute the I/O workload [29] at a large scale. These two elements are crucial to having a high aggregated throughput for data-intensive applications.

Data is represented in BlobSeer as **Binary Large Objects** (BLOBs), consisted of long sequences of bytes of unstructured data. A BLOB can reach sizes of up to 1 TB and can be managed by the client through a simple access interface that enables creating a BLOB, writing and reading a sequence of **size** bytes from and to a BLOB starting at **offset**, and appending **size** bytes to a BLOB. This access interface is designed to support versioning explicitly: every time the BLOBs are modified by a write or append a new version (or snapshot) of the BLOB is generated, rather than overwriting any existing data. The client is allowed to read from any previous BLOB version, and even the current BLOB version, after it is made public. The usage of BLOBs offer the system scalability (because of the versioning abilities) and transparency, as each BLOB receives a unique id, freeing the developer from managing data locations and data transfers.

**Atomic snapshot generation** is a key property of the BlobSeer system. Readers should not be able to access inconsistent snapshots that are in the process of being generated.

**Data striping** is a technique used on a large scale basis in distributed applications, to increase the performance of data access. In our case, each BLOB is split into chunks (of a fixed size, specified at the time of the BLOB creation), that are distributed amongst the storage providers of BlobSeer. The system has a configurable chunk distribution strategy with the goal of achieving load-balancing, a strategy that can be implemented by the developer as to reach the desired objectives. A good distribution strategy will result in a high throughput when concurrent clients access different parts of a BLOB.

**Distributed metadata management** is important to the system due to the fact that BLOBs are spread across a large number of storage space providers in a fine-grain manner, forming a Distributed Hash Table (DHT). Thus, BlobSeer needs to maintain metadata that maps the subsequences of a BLOB, defined by **size** and **offset**, to the corresponding chunks. The fact that the metadata itself is distributed cuts down on the overhead generated by the large quantities of metadata generated when working at a large scale.

**Explicit versioning** is used to enhance data access parallelism, as an older version may be accessed whilst a newer one is being created. For every write and append, a new snapshot of the BLOB is taken and past-versions are kept. Reads will never conflict with concurrent writes, as they do not access the same BLOB. Data and metadata are always created, never overwritten, and only the difference with respect to the latest version is stored.

## 4.3 A practical look at BlobSeer

BlobSeer has a special interface [30],[31] that enables the client to manipulate a BLOB. In the following section, we will discuss the available primitives for BLOB manipulation, and take a look at some of BlobSeer's structures and algorithms. We will not go into more detailed primitives and algorithms regarding the reading, writing and appending of metadata and data providers, nor the sharing the metadata over snapshot versions, nor the creation of new snapshot versions, since it is not this thesis's objective to go beyond the client's direct interaction with BlobSeer.

**CREATE(callback(id))**: this primitive will create a new, empty BLOB, and will return the BLOB's id, 0 (which references an empty snapshot). This id is sent by the provider manager and is guaranteed to be unique.

**WRITE(id, buffer, offset, size, callback(v))** and APPEND**(id, buffer, offset, callback(v))**: these primitives update the BLOB (given by the "id") either by writing size bytes at the given offset or by appending size bytes at the end of the BLOB. After a new write or append, a new BLOB version is created (a snapshot), reflecting the changes. The actual version is made public, when the operation completes. The two primitives are very much alike, so we will discuss the former, since it contains the latter.

To write data, the client must first determine the number of pages (n) that covers the range to be written. Then, the provider manager is contacted by the client, with the request to provide a list of n page providers (PP), one provider for each page, that have free disk space for storing the data. For each page in parallel, the client generates a unique page id (pid), contacts the corresponding page provider, to store the data, and creates and updates a corresponding page descriptor (PD) to store this information. Afterwards, the version manager is contacted, to register the update. Upon registration, the version manager creates a new snapshot version and communicates it to the client. The client then creates new metadata and notifies the version manager of success. After this, it is up to the version manager to determine when to publish the new snapshot version. Table 2 represents the WRITE algorithm.

*Table 2*

**BlobSeer WRITE algorithm**

| | |
|---|---|
| 1. | n <- (offset + size) / page_size |
| 2. | PP <- the list of n page providers |
| 3. | PD <- null |
| 4. | ***for all*** i = 0 to n ***in parallel do*** |
| 5. |     pid <- unique page id |
| 6. |     provider <- PP[i] |
| 7. |     store page pid from buffer at i x page_size to provider |
| 8. |     PD <- PD U (pid, i, provider) |
| 9. | ***end for*** |
| 10. | v <- assign snapshot version |
| 11. | BUILD_METADATA(v, offset, size, PD) |
| 12. | notify version manager of success |
| 13. | ***return*** v |

**READ(id, v, buffer, offset, size, callback(result))**: this primitive reads "size" bytes from the specified BLOB's version ("id" representing the BLOB's id and "v" representing that specific BLOB's version to be accessed), starting from "offset". As a result, "size" bytes replace the local "buffer". The callback parameter is a Boolean value which shows if the read was successful or not. The read fails if "v" has not yet been generated, also if the total size of the version "v" is smaller than the value "offset + size".

To read data, first of all the client must contact the version manager to see if the supplied version has been published. If this is true, then the client contacts the metadata providers, to obtain the metadata with information regarding the data providers on which the pages of the corresponding BLOB range are. The information obtained is written into a set of page descriptors (PG), information such as the page's global id ("pid"), the index ("i") in the buffer that will be read, and the provider that contains the page. Finally, the client fetches the pages in parallel from the data providers, with the information from the page descriptors. Table 3 represents the READ algorithm.

*Table 3*

**BlobSeer READ algorithm**

| | |
|---|---|
| 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 page_size |
| 7. | ***end for*** |
| 8. | ***return success*** |

**GET_RECENT(id, callback(v, size))** and **GET_SIZE(id, v, callback(size))**: these primitives help keep track of the most recent snapshot and size of a BLOB. GET_RECENT queries the system for a recent snapshot version of the BLOB given by "id". The result of the query is the snapshot version number and its corresponding size. The GET_SIZE primitive returns the total size of a BLOB snapshot. The value is returned only when and if the operation completes successfully.

**SUBSCRIBE(id, callback(v, size))** and **UNSUBSCRIBE(id)**: the SUBSCRIBE primitive represents another way of learning about a new snapshot version in the system, for a specific BLOB. A notification is sent to the process that calls this primitive, each time a new version is created for the BLOB identified by "id". The UNSUBSCRIBE primitive is enough to terminate the notifications.

**SYNC(id, v)**: this primitive is used for "read your writes" consistency, and consists of the blocking of the caller until snapshot "v" of BLOB "i" is published.

**BRANCH(id, v, callback(bid))**: this primitive allows alternative evolutions of BLOB "i", by the duplication of the original BLOB. The new BLOB is identified by "bid" and has the same version number as the original. The first WRITE or APPEND on bid will result in the creation of a new snapshot, with the version number "v+1" for BLOB "bid".

BlobSeer has a special way of communicating with all its component layers, be means of **asynchronous remote procedure calls (RPC)[30]**. This is possible due to the implemented RPC layer. Each client-side remote procedure call triggers a corresponding server-side event that encapsulates the parameters of the call. Once the server-side is done processing the event, it triggers in return, a completion event on the client-side. This completion event encapsulates the result and is dispatched to the callback function that was registered with the RPC. The RPC layer handles all communication details regarding data transfers, socket management and parameter serialization. BlobSeer implements the RPC layer on top of ASIO, a high-performance asynchronous I/O library that is part of the Boost collection of meta-template libraries.

# 5. Adding access control to BlobSeer

In this chapter we propose a client access control strategy for BlobSeer, based on adding a new component to the system, generically named the Smanager (Security Manager - SM). We talk about REGISTRATION, LOG-IN features, BlobSeer client verification, and other issues and also discuss what security technologies and strategies we base our proposal on as well as our motivations to do so.

## 5.1 Adopted security structures and models

In the following subsections we justify our made choices, regarding security algorithms and structures, regarding the implementation of the security Layer, a client access control module in BlobSeer. We will talk about X.509 certificates and our option to use RSA keys (5.1.1), as well as client access models (5.1.2). For more information on the X.509 certificate and RSA key generation, please consult with subsection 3.1 of this thesis, as for information on authorization and authentication techniques, please see subsection 3.2.

An important structure of the BlobSeer security Layer is the data structure the SM uses to register clients, by writing their information to a BLOB. This structure is that described in table 3.

*Table 3*

**Client info structure**

| | |
|---|---|
| Keypass | Serialized client public-key |
| Certificate | Serialized client certificate |
| Log-in | Variable that says if the client is logged in or not |
| Access | Monitors how many erroneous accesses the client has had |
| Permissions | 3-bit value that show what permissions the client has |
| Write_perm | List of blob Ids the client has access to |
| App_perm | List of blob Ids the client has access to |
| Read_perm | List of blob Ids the client has access to |
| Prev | Offset of other entry for the same client with more permissions |
| Next | Offset of other entry for same client with more permissions |

### 5.1.1 OpenSSL: X.509 and RSA

We propose that for implementing a secure client access control system for BlobSeer we use OpenSSL[32], a light-weight tool for authentication and authorization. OpenSSL is based on and very similar to the widely-used SSL tool, some of the differences worth mentioning being the fact that OpenSSL is open-source, has encrypting and decrypting capabilities, and implements digital certification, digital signatures and random numbers. This tool will be used over BlobSeer's RPC layer, to send keys and certificates, from the clients to some of the entities they communicate with, for BLOB manipulation. The tool may also be used to encrypt the said communications, but it is not in the scope of this thesis to discuss such procedures. We are only interesting in BlobSeer client management. We consider that OpenSSL meets our needs in expanding BlobSeer.

OpenSSL is able to produce DSA and RSA keys and even though DSA keys are the official standard, we opt for using the RSA key-generation protocol, due to the fact that it is faster than the former and may also be used for encrypting (such a feature may prove useful in BlobSeer's future).

For client authorization and authentication we propose to work with certificates. OpenSSL has libraries to implement X.509 certificates, and certificate authorities. After generating a private key, a Certificate Signing Request (CSR) is generated, leading to the generation of a self-signed certificate (to be used by the Certificate Authority - CA). The CA afterwards, generates certificates based on client requests. These certificates are used at log-in, for authentication, and whenever a client wants access to a BLOB, for authorization. Working with a certificate authority, and RSA encrypted-keys is much faster and easier to implement in OpenSSL (than with other tools, such as Kerberos), offering robust security techniques (more robust than let us say the UNIX authentication protocol). Also, using this tool, we leave the door open for future BlobSeer enhancements, such as encryption and decryption possibilities.

### 5.1.2 RBAC, TBAC and ACLs

There is a single data object in BlobSeer, to which clients have access, and that is the BLOB. In the current BlobSeer system, any client may access any BLOB, if the BLOB id is known.

We propose a RBAC model, where each client may have a full-user role (create, read, and write permissions regarding a BLOB), and partial-user roles (subsets of the initial full-user role), both adjustable depending on the client's actions. For example, If a malicious client creates blobs, without writing in them, or with writing spam-data, then it's create and write permissions may be revoked, first time as a warning. Eventually the permissions may be revoked permanently.

We consider the TBAC model inappropriate due to the fact that it is unnecessarily more complex in comparison with BlobSeer's system and mode of operation (no complex tasks take place, other than the accessing of BLOBS).

ACLs are useful in this client access system proposal, in the sense that these lists will be client based: ach client will have a list entry, appended to a BLOB maintained by the SM, specifying what permission is available for which BLOB (for now we work with the essential read, write, append and create permissions).

Both ACLs and user-roles will be attended to by the Security Manager, updated for a client when the system is accessed. More information on how security will be managed by the SM can be found in the following paragraphs.

## 5.2 BlobSeer Security Layer

As stated at the beginning of this thesis (chapter 1), BlobSeer has no security Layer. To solve this problem we propose a new module, a Security Manager, be added to the existing BlobSeer architecture (section 4.1) that handles client access control. This section presents the module, as well as dependencies between the existing BlobSeer entities and the Security Manager.

The Security Manager (SM) allows clients to register and log-in keeping client information in BLOBs or log-in files (LOGS). For every client access request received by BlobSeer, a client-verification request is issued to the Security Manager, so that it can determine if the client has the claimed access rights. If the claim is valid, then BLOB manipulation takes place as normal. If not, then the BlobSeer entity is notified and the client is warned, whilst the Security Manager keeps track of the number of unauthorized accesses. Depending on the number of unauthorized accesses, a client's set of permissions can be altered by the Security Manager. Apart from the SM's BLOB (in which information regarding each registered client is held), the SM also has a LOG, which is kept up-to-date with information regarding a client's log-in and log-out from the system. These two data storage objects are protected, in such a way that only the SM has access – the BLOB is protected by the SM itself whilst the LOG is protected by file rights.

We consider the SM not only a client access manager but also a Certificate Authority, generating certificates based on data received from the client. Also, the SM has its own pair of asynchronously generated keys, and its own self-generated certificate, for authenticating it's self to entities when accessing a BLOB.

Upon entering the system for the first time, a client must register with the Security Manager to enable BLOB access. If it is not the first time the client enters the system, then a log-in sequence must take place, in which the client presents a pass to the Security Manager. This pass is actually a security certificate issued by the Security Manager.

Whenever the client wants to create, read, write or append a BLOB, credentials must be sent to the version manager (used for BLOB creation and reading) and to the provider manager (used in BLOB writing and appending). These managers are implemented in such a way that they check with the Security Manager that both, the credentials are valid and that the client has specific permission enabled. To implement this, the READ and WRITE algorithms stated in section 4.3 will be modified.

## 5.3 Algorithms and schemas

**REGISTRATION**: When a new client wants to use BlobSeer, we propose that a REGISTRATION method be implemented in the way that the client may not access BlobSeer if this primary stage has not finished successfully. Normally, a client that wants to be able to work in BlobSeer, initializes is a Object_handler class object, which then can access commands amongst which create, read write BLOBs (of normal data, or metadata). We have implemented these functions in such a way that they cannot work if the client has not registered, and logged-in (LOG-IN is to be discussed in a later paragraph). On registration, the client is not prompted for data but the information is retrieved from a configuration file, specified at initialization time.

On the client side an asymmetric RSA key-set is generated and the public key is sent to the Security Manager, who in turn generates a special certificate for the client, seting the client's public key in the certificate, and then signing the certificate with it's own public-key. Apart from this, the server writes in a BLOB, information regarding the client, such as public-key and an initial default role (the "full-user" role, meaning that the client may create BLOBs as well as read, write and append data from/to BLOBS). Back on the client-side, the new X.509 certificate is in a special folder with limited access, along with the private and public keys.

From this moment on, the client is considered to be registered, and will be allowed to access BLOBs according to the attributed permissions.

In table 4 is the algorithm for the server side of the REGISTRATION. More information may be found in section 5.4.2.

*Table 4*

NEW_REGISTRATION algorithm (SM)

```
1. key <- receive_client_public_key()
2. If (!(cert<- Find_client_in_blob(key))) do
3.     param <- get_other_client_param()
4.     cert <- generate_certificate(param)
5.     p <- set_client_permissions(full_user)
6.     client_data <- set_client_data(blob_page_size,p,cert,key)
7.     return cert
8. return cert
```

**UNREGISTER**: For symmetry, this function is offered so that the client may undo its registration. This will mean that on the client-side, the folder created when the client was being registered, is deleted; on the server-side, the certificate for that specific client will be deleted, but the entry will still exist, because of the BlobSeer is created, to allow only BLOB-growth (the addition of data, data is not deleted, eventually re-written), and will be re-written when another client registers. In table 5 is the algorithm for the server side of the UNREGISTER. More information may be found in section 5.4.2.

**LOG-IN**: If registration has already taken place, then the client must log-into the system, to be able to manipulate the BLOB data. This means that the client must present the Security Manager with the certificate obtained at registry time. The Security Manager will respond in accordance to the data it has collected regarding client registration, if the client is registered or not. It is recommended that the LOG-IN procedure be implemented separately than with the other BLOB manipulation functions, for the same reasons that were evoked for the REGISTRATION procedure.

In table 6 is the algorithm for the server side of the LOG-IN. More information may be found in section 5.4.2.

*Table 5*

UNREGISTER_OUT algorithm (SM)

| | |
|---|---|
| 1. | cert <- get_client_cert() |
| 2. | If (! check(cert_signed_with_SM_key)) |
| 3. |     **return** error |
| 4. | key <- get_client_public_from_cert() |
| 5. | **If** (!(client_offset<- Find_client_in_blob( cert))) **do** |
| 6. |     cert <- set_cert_to_zero( client_offset+sizeof(key), sizeof(cert)) |
| 7. |     **return** success |
| 8. | **return** error |

*Table 6*

LOG-IN algorithm (SM)

| | |
|---|---|
| 1. | cert <- get_client_cert() |
| 2. | If (! check(cert_signed_with_SM_key(cert))) |
| 3. |     **return** error |
| 4. | **If** ( !( cert<- Find_client_in_blob( key ))) **do** |
| 5. |     **return** error |
| 6. | cliend_id <- get_client_id ( cert ) |
| 7. | **If** write_to_log_file( client_id, log_in ) == successful **do** |
| 8. |     **return** success |
| 7. | **return** error |

**LOG-OUT**: For symmetry, we propose a LOG-OUT procedure be implemented as well, so that a client can be more easily monitored by the System Manager. This LOG-OUT is implemented as a notification from the client to the Security Manager, that the client has currently finished working with the system. The LOG-OUT algorithm for the server side, as well as the client side the same with that of the LOG-IN algorithm. More information may be found in section 5.4.2.

The LOG-OUT as well as the LOG-IN procedure will help BlobSeer keep track of how many clients are using the system, and help the system have a relative notion on who everybody is. In the SM's BLOB, each client will have a special field that keeps track of the client's status (logged-in or not). This field not only will improve the SM's searching speed when verifying if an already logged-in client has a certain permission (this issue will be discussed in the following paragraph) but will also help to analize a client's intentions. This will prove useful for when we want to modify a client's set of permissions, based on their actions. Such a strategy will help BlobSeer protect itself from a number of attacks, amongst which we can mention the DoS attack.

**BLOB ACCESS**: Every time a client tries to manipulate a BLOB, it will send its certificate (issued by the Security Manager) as an access request, to the entities it is necessary to communicate with. These BlobSeer entities will then query the SM to see if the certificate is valid, and if the client has the claimed permissions (check_permission(client_cert)). At this step, the SM will first take the certificate and see if it is signed with the correct public-key (the SM public-key), and if this is true, then go on an try see if the possible-client's key is valid (if the key is registered in the BLOB). If these two conditions are met, then the SM will search for the client entry in the SM BLOB to check the client's permissions. If the permissions are those required, then the SM will respond to the query in a positive manner, otherwise it will return false, allowing the entities to deny the client access. The SM will also make a note in its LOG of the client's erroneous behavior. The schema for this line of communication is in Fig 4.

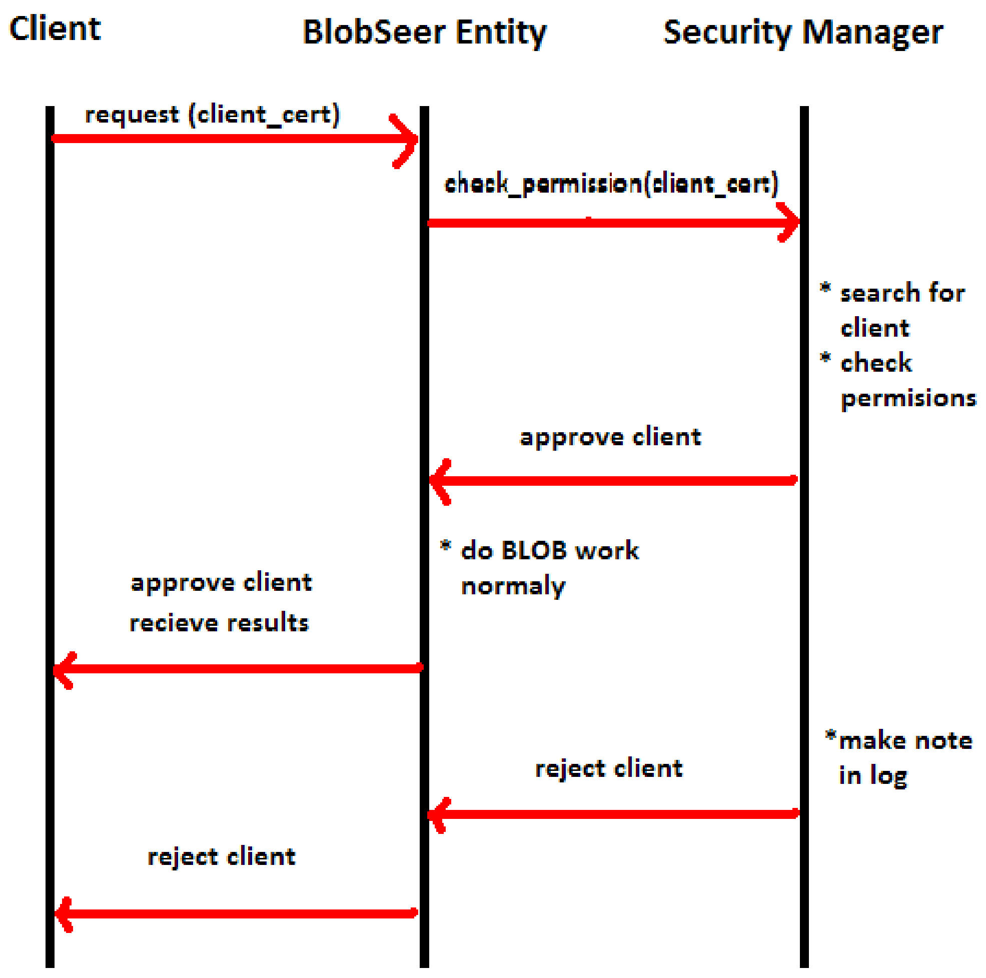

Fig. 4 New Client BLOB access

### 5.4 Implementation details

The most commonly used libraries are those of OpenSSL, since they are the ones that offer the tools needed for creating keys and certificates, as well as key testing, certificate verifying and so on. In this section we first take a look at some of the above mentioned libraries, and then discuss some of the implemented classes.

#### 5.4.1 Useful libraries

The following is a list of important libraries that are used repeatedly in the security Layer implementation:

**openssl/evp.h** – provides a high-level interface to cryptographic functions.
Out of the many implemented functions we work with the EVP_PKY functions, which provide a high-level interface for asymmetric algorithms. These functions mostly work with a PEM format (a base64 encoded data surrounded by header lines).

**openssl/rsa.h** – implements RSA public-key encryption, and signature functions.

**openssl/pem.h** – allows the reading and writing of PEM structures from a BIO (an object OpenSSL based on an I/O abstraction – more information can be found at the openssl/bio.h entry) or from a file. These PEM structures may represent keys or certificates.

**openssl/bio.h** – is an I/O abstraction that hides many of the underlying I/O details from the application, enabling transparent handling of SSL connections, unencrypted network connections and file I/O.

**openssl/x509.h** – is a library used for certificate handling

#### 5.4.2 Written classes, handles and methods

The Security Manager module is made up of the following files (Fig. 5 illustrates the class diagram), which are at the basis of the security Layer implementation:

**smanager.cpp** – implements the initialization of the SM. Here, the configuration parameters for the manager are read from a configuration file defined at BlobSeer initialization, handles (implemented in mysecurity.cpp) are registered with the RPC server, a self-signed certificate is generated and a new BLOB is created. A LOG file is also created, to register client log-ins and log-outs. The keys, certificate and log-file are kept in a specially created folder, to which only the SM has access.

**smanager.hpp** – contains the registration ids for the handles.

**mysecurity.cpp** – implements the handles defined in mysecurity.hpp and auxiliary functions to aid the handles; this class inherits the object_handler class, for BLOB manipulation.

These handles are:

- **new_register()**: First creates a new certificate for the client, based on the information received. If the information is incorrect, an appropriate error message is printed. Secondly, writes to the BLOB created at SM initialization, client information.

- **new_log()**: checks if the client is registered (has an entry in the BLOB); writes in the LOG file, information regarding the client.

- **unlog_out()**: writes in the LOG that the client has logged out. Returns success if the write is successful.

- **uregister_out()**: if the client is registered, it set's the certificate to 0. It returns success if the operation succeeds.

Some of the other methods implemented in this class are:

- **find_client()**: searches for the client (by key) in the BLOB, reading keylen (the size of a client's key) bytes of data, in each page of BLOB. If the client if found, the offset in the blob is given.

- **del_client()**: deletes the client by writing 0s in the client's certificate.

- set_client_data(): prepares the client data for writing. This data must in multiples of blob_page_size.

**mysecurity.hpp** – contains the handle definitions, auxiliary functions, and useful parameters.

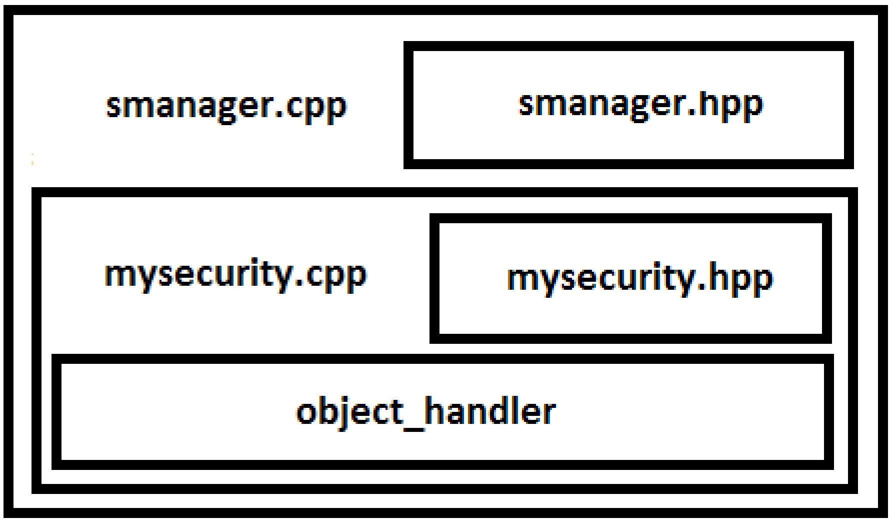

Fig. 5 Security Manager class diagram

As stated in earlier sections, before a user can access the BlobSeer system (more specifically, access BLOBs), registration and log-in must first take place (unless the user is the Security Manager). To implement this, we added the registration and log-in interface, registration.hpp. The main methods defined here and implemented in object_handler.cpp, are:

- **get_register()**: generates the client's RSA private and public keys, prepares the Security Manager request for registering, by preparing the following parameters: the public key (serialized), and authenticating information for creating a certificate (Country Name, Address etc). When the SM send's its answer, the response is sent to the solve_register handle, for processing.

- **get_unregister()**: sends data to the SM to request the user be unregistered. If the SM succeeds in fulfilling the request then the certificate and passwords created initially, are deleted.

- **get_logged()**: sends information received from the SM when the user was registered, back to the SM, to announce that the user is now online. This function awaits a response from the SM, to confirm if the request was satisfied. An error message will appear if the user is not registered, or if an unknown error has occurred on the SM side (for information on the SM-side error, the messages printed must be checked on the console where SM is running).

- **get_unlogged()**: sends information received from the SM to announce that the user wishes to log-out. This function awaits confirmation from the SM that the process has finished successfully.

Most of the already existing methods in object_handler.cpp have been modified, to send the client certificate to the required entity, and await a response regarding the authorization of the client.

# 6. Experimental results

In this chapter we demonstrate that apart from being useful, the solution we supply carries a low enough overhead.

We tested our model on creating a BLOB, writing a BLOB and reading a BLOB, without modifications to the client, without registering to the server, and with the modifications to the client and with registering to the Service Manager (the phase when the server manager checks the key, and generates a certificate to send back to the client). We create a blob of 65536 bytes, with storage managers that can store up to 512 MB

We ran our tests three times for each type of BLOB access:

1) Without modifications:

| | Create blob | write blob | read_blob |
|---|---|---|---|
| Real | 0m00,012 | 0m30,322 | 0m08,097 |
| User | 0m00,008 | 0m01,804 | 0m03,060 |
| Sys | 0m00,000 | 0m01.548 | 0m01.036 |
| | | | |
| Real | 0m00,016 | 0m19,588 | 0m07,928 |
| User | 0m00,008 | 0m01,868 | 0m03,080 |
| Sys | 0m00,004 | 0m01.300 | 0m01.124 |
| | | | |
| Real | 0m00,013 | 0m05,555 | 0m07,765 |
| User | 0m00,008 | 0m00,264 | 0m03,140 |
| Sys | 0m00,001 | 0m00.236 | 0m01.124 |

As we can se, there is a small overhead in the BlobSeer system when having to work with BLOBs. As an observation, the third time we ran the write_blob test, an internal error occurred, and the overhead there is the smallest.

2) With modifications:

| | Create blob | write blob | read_blob |
|---|---|---|---|
| Real | 0m24,012 | 0m23,322 | 0m09,128 |
| User | 0m24,808 | 0m24,804 | 0m03,180 |
| Sys | 0m00,000 | 0m01.548 | 0m01.114 |
| | | | |
| Real | 0m24,012 | 0m00,770 | 0m07,847 |
| User | 0m24,808 | 0m00,296 | 0m03,056 |
| Sys | 0m00,000 | 0m00.236 | 0m01.116 |

As we can se, there exists an overhead in the BlobSeer system when having to work with BLOBs, and register with the SM as well. As an observation, the second time we ran the write_blob test, an internal error occurred, and the overhead there is the smallest.

# 7. Conclusions and future work

It is important for a distributed data management system to have a well developed security strategy, to assure, amongst other security features, data protection, data integrity, client confidentiality, and system accessibility. All these may be compromised if an unauthorized user is allowed to access the system and rage havoc.

We consider that the work presented in this thesis has led to a safer BlobSeer system with a client authentication and authorization service that addresses issues raised in the paragraph above, and we will propose at the end of this chapter a number of ideas for improving even more BlobSeer's security. Even though there is a rise in the overhead generated, the largest being at registration and log-in, it is not exaggeratedly large, and can be overlooked in comparison with the benefits of a secure system.

BlobSeer's no longer ignores the clients that are accessing the system, but maintains information regarding each and every user in a distributed fashion (the BLOB). At the moment, the Security Manager may prove to be a bottle neck for the system, when various clients try to register or when BlobSeer entities try to authenticate their user, but with data replication, this issue may be solved. Although request are received and resolved by the Security Manager in a sequential manner, this may be resolved by using threads.

As future work we propose that an extra permission be added to a client's role, that of delegation. By this permission, a client should be able to transfer to another client a set of rights for one or more BLOBs. This will offer better client access control, by restricting even more a client's access to a BLOB. Initially all clients may access all BLOBs, if the id is known. By adding the improvements proposed by this thesis, a client may access BLOBs specified in their permissions (at the moment a client has rights to access a BLOB it has created). In the future we hope that by delegating, client A may transfer its rights on its created BLOB to client B, so that both client A and B may access the same BLOB. This method we consider to be most appealing, since another option is that the permissions for each client be set manually by an administrator. We realize that since this delegation option means that client A must know about client B, and that client A must have a way of communicating this knowledge to the SM, so we propose that a method be implemented so that client A may ask the System Manager for an updated list of registered clients.

We also propose that the Security Manager have a special thread to deal with client analysis, meaning that this thread would track the System Manager LOG file, eventually have a log of its own, and dynamically assess which BlobSeer registered client must have its permissions reevaluated.

In conclusion, we consider that after taking a look at BlobSeer, and having in mind the ideas proposed in the above chapters, this thesis has succeeded in its goal of making BlobSeer a secure distributed file system, by implementing a simple module that keeps track of accessing clients. These improvements not only offer BlobSeer a client access manager, but also, it offers BlobSeer a way to protect itself against malicious attacks from the network.

## BIBLIOGRAPHY

[1] B. Nicolae, G. Antoniu, L. Bouge, 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.

[2] 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

[3] “Security for Distributed Systems,” H. Abie http://www.nr.no/~abie/security.htm fetched on 11.05.2011

[4] “Data Management”, Tech-FAQ, http://www.tech-faq.com/data-management.html 2011

[5] S. T. Ross ”Unix System Security Tools”, The McGraw-Hill Companies, 1999, pp.1-3 http://www.albion.com/security

[6] “Hadoop DFS User Guid 2007”, the Apache Software Foundation 2.0 http://hadoop.apache.org/ 2007

[7] O. O’Mally, K. Zhang, S. Radia, R. Marti, C.Harrell, “Hadoop Security Design”, web, 2009, pp. 1-19 http://bit.ly/75011o

[8] “Hadoop Security Design, Just Add Kerberos? Really?”, A. Becherer , iSEC Partners, Inc, 2010 pp. 1-8. https://media.blackhat.com/bh-us-10/whitepapers/Becherer/BlackHat-USA-2010-Becherer-Andrew-Hadoop Security-wp.pdf feched on 20.03.2011

[9] “Amazon Simple Storage Service (Amazon S3)”,Amazon Inc. http://aws.amazon.com/s3/ fetched on 11.05.2011

[10] “All things Distributed”, W.Vogel, http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html October 2, 2007

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

[12] R.Mason, The Masuni Blog, http://www.nasuni.com/news/nasuni-blog/amazon-s3s-enhanced-cloud-storage-security-is-nice-but-not-good-enough/ 06.07.2010

[13] “Amazon S3” Amazon Inc. http://docs.amazonwebservices.com/AmazonS3/latest/dev/ fetched on 25.05.2011

[14] 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

[15] “Managing Lustre Security” Oracle, http://wiki.lustre.org/manual/LustreManual20_HTML/ManagingSecurity.html#50438221_pgfId-5529 fetched at 15.05.2011

[16] “The TCP/IP Guide” C. M. Kozierok, http://www.tcpipguide.com/free/t_NFSOverviewHistoryVersionsandStandards.htm , 2005

[17] “File Sharing with NFS” Microsoft TechNet http://technet.microsoft.com/en-us/library/cc976863.aspx, fetched at 06.2011

[18] “NFS Access Control Lists support” IBM http://publib.boulder.ibm.com/infocenter/aix/v6r1/index.jsp?topic=/com.ibm.aix.commadmn/doc/commadmndita/nfs_aclsupport.htm

[19] H. Stern, O. Reilly, M. Eisler and R. Labiaga, “Managing NFS and NIS”, 2nd edition 2002 http://docstore.mik.ua/orelly/networking_2ndEd/nfs/ch12_05.htm

[20] “Secure communication” http://en.wikipedia.org/wiki/Secure_communication fetched at 17.05.2011 Fetched at 27.04.2011

[21] “Secure RPC” Oracle http://download.oracle.com/docs/cd/E19082-01/819-1634/rfsrefer-59/index.html fetched at 29.05.2011

[22] “Unix Authentication Example” IBM http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.progcomm/doc/progccpc_unixauth_ex.htm fetched at 29.05.2011

[23] ”SSL Authentication Example” IBM http://publib.boulder.ibm.com/infocenter/cicsts/v3r1/index.jsp?topic=%2Fcom.ibm.cics.ts31.doc%2Fdfht5%2Ftopics%2Fdfht50g.htm fetched at 29.05.2011

[24] “Kerberos(protocol)” http://en.wikipedia.org/wiki/Kerberos_(protocol) fetched at 30.05.2011

[25] ”HRU (security)” http://en.wikipedia.org/wiki/HRU_(security) fetched at 30.05.20011

[26] “Role Based Access Control” http://en.wikipedia.org/wiki/Role-based_access_control Fetched at 30.05.2011

[27] “Task Based Access Control Model” http://en.cnki.com.cn/Article_en/CJFDTOTAL-RJXB200301011.htm D. Ji-Bo, China, 2003

[28] B. Nicolae, G. Antoniu, and L. Bouge, “Enabling High Data Thtoughput 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.

[29] B. Nicolae, “High Throughput Data-Compression for Cloud Storage,” Proc. Globe’10:3rd International Conference on Data Management in Grid and P2P Systems, Bilbao, Spain, 2010, pp.1-12.

[30] B. Nicolae, G. Antoniu, L. Bouge, “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.

[31] B. Nicolae, “BlobSeer: Towards effiecient data storage management for large-scale, distributed systems,” PhD Thesis, 2010

[32] “OpenSSL” http://www.openssl.org/ fetched at