UNIVERSITATEA POLITEHNICA BUCUREŞTI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE LICENȚĂ

Servicii de securitate in Cloud folosind BlobSeer

Coordonatori ştiinţifici:
Prof. dr. ing. Valentin Cristea
As. drd. Ing. Cătălin Leordeanu

Absolvent:
Goanță Oana Andreea

BUCUREŞTI

2011

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

DIPLOMA PROJECT

Secure access to Cloud services using BlobSeer

Thesis supervisors:
Prof. dr. ing. Valentin Cristea
As. drd. Ing. Cătălin Leordeanu

Author:
Goanță Oana Andreea

BUCHAREST

2011

Table of Contents

Abstract

Web Services are very widespread in today's Internet and Cloud technologies are becoming more and more popular. Along with the use of these technologies, there are also some important security concerns that arise. Solving the problem of security in this context is not an easy challenge due to the fact that this systems usually have large numbers of resources and users. Users must rely on the Cloud infrastructure to ensure their security and the security of their data. The purpose of this project is to allow secure access to web services in a Cloud environment. Secure access involves encrypted, secure connections, using authentication methods and access rights management for each user. To provide an efficient environment we chose BlobSeer as the security data management system for its distributed nature and performance. In this thesis we will prove that our system is efficient and that it provides an adequate level of security for web service-based applications.

1. Introduction

The purpose of this project is to allow secure access to web services in a Cloud environment. Secured access involves encrypted, secure connections, using authentication methods and access rights management for each user. This thesis introduces BlobSeer, an efficient distributed data management service, as the storage system to be used for security related data. Clouds can offer a large variety of services (from video and audio streams to financial services), each of them requiring a different level of security. Users must only access the services they have been granted access to by the distributed system owners. All of this goals must be achieved without adding much overhead to these web service based applications.

In Cloud computing, users hand off their data and applications to providers. In doing so, they also delegate their responsibility for security. This is why the risks in Cloud computing security are in some ways different from those in conventional IT environments. In the Cloud, security is a blend of new and familiar mechanisms and challenges.

Security mechanisms need to store encryption keys, certificates, the access rights management system needs to store the access control matrix, protection against malicious users needs to store access logs. In a could system with thousands of users, and the constant expansion of the environment, the storage space is an important aspect. Besides effective data, metadata used for security and statistics must also be taken into account. While storage space might not be the problem in a Cloud system with impressive data centers, data management is certainly an issue. This is critical especially for data intensive services, such as video streaming or audio applications. In such cases the storage system has a major impact on the overall performance of the application. Moreover, the throughput of security data reads and writes also directly impacts the performance of the system. Security checks are performed very often, and although the amount of data read/written is small, the access time adds to the overhead of invoking a service located somewhere in the Internet. Therefore, a particular feature needed for security management data is fine-grain access.

In a Cloud environment, it is not always clear which data and applications reside on which servers at a specific moment, so the data management system must offer location transparency. Other key features we are looking for in a data management system for Cloud computing are: high throughput, concurrency, decentralized data management. BlobSeer meets all the needs of Cloud data management. It is a distributed service that has a unique architecture consisting of various entities, each with specific responsibilities. Its design choices (data striping, distributed metadata management and versioning-based concurrency control) are the key to its performance. This thesis introduces a solution for secure access to Cloud web services that is based on the BlobSeer data management service.

Experimental results have shown that BlobSeer is very efficient in allowing transparent fine-grain access to massive data, efficient versioning for large object storage, high throughput under heavy concurrency for Hadoop map/reduce applications, high throughput in desktop grids and improving QoS in large-scale distributed data storage services.

The authentication mechanism of our solution allows single sign-on access control. Cloud platforms can host applications that come from different providers, each with its own security policies. If each of these services would independently manage security, losing control over information security at global level could become an issue in the could, as well as adding other passwords for end users to manage.

A entity called Service Manager will be in charge of the registration and authentication process. In order to authorize access to other services after login into the system, the security manager will grant the user a security token/ticket associated with an expiration date, that needs to be validated by each service the user will invoke during its current session. The sing-on mechanism is based on passwords. The authentication process uses encryption in order to protect the password and the access token.

Because Clouds provide services on a pay-per-use basis, the access control system must allow a fine-grain representation of the user's permission to use these services. Our project uses a representation that is based on clients capabilities to access individual operations exposed by web services for a given amount of time.

In order o allow a flexible way to deploy new services, we have separated the functional layer of a service from the secure access layer. Each web service will be protected by its own instance of a secure access service, which will act as a proxy and will perform all security verifications and then invoke the actual service. This thesis will proof that the overhead caused by the proxy is insignificant compared to the benefits it brings to the system.

The structure of this thesis is as follows. Section 2 offers an overview of the BlobSeer distributed data storage service and provides details on the features needed for our approach. Section 3 provides background information on web services and Cloud Environments in general, while emphasizing the benefits of our project. Section 4 describes similar scientific projects which attempt to offer a degree of security for web service access. Section 5 contains the main contributions of this thesis. It describes the theoretical aspects of our solution which provides secure access to web services in a Cloud Environment. Section 6 focuses on implementation details while Section 7 contains the evaluation and experimental results which prove our solution's functionality and performance. Finally, Section 8 draws conclusions and outlines directions for future work.

2. BlobSeer

2.1 The needs of data-intensive applications storage systems

With the emergence of highly scalable infrastructures, like Cloud computing platforms, the demand for scalable data management has increased. Efficient storing and accessing massive data blocks in a large-scale distributed environment becomes an immediate need. Challenges posed by data-intensive computing have gained an increasing importance, as this applications continuously acquire massive datasets while performing computations over these dynamic datasets. The storage service employed to handle data management for these applications has to address several issues in order to achieve efficient storing and accessing of data [1].

Aggregation of storage space from the participating nodes with minimal overhead is highly needed in distributed systems, the Cloud being the best distributed infrastructure example.

A *scalable storage architecture* is needed because the data management system has to accommodate a large number of data providers, with huge data centers that are continuously being added to the distributed system.

Transparency means that the applications must not be aware of the explicit localization and explicit transfers of data. Most of the existing data management systems rely on data localization, which makes this systems more complex and difficult to work with. BlobSeer handles this aspects in a transparent manner, simplifying user's access to data.

The ability to store *huge data objects* is required by several types of services offered by distributed web systems. Such applications are: multimedia, databases, data mining, etc. They work with *massive unstructured data* that can easily grow to huge sizes (reaching TB), and is continuously modified by the running services. Traditional file systems and databases manage this files with increased difficulty as files expand.

High throughput under heavy access concurrency is an important factor for the system's performance. Traditional parallel and distributed applications use one of the following mechanisms to achieve this: message passing, Map Reduce, Dryad. Massively parallel data access must be handled efficiently by the underlying storage service, as concurrent reads and writes are constantly changing huge data blocks.

Providing *efficient fine-grain access* [2] to massive data blocks, stored in large-scale distributed environments such as grids, is needed because some applications may only need to access small data chunks repeatedly, but which are placed in a huge block. This is also the case of security services, that must read and update security information repeatedly and although the size of the data is small, the access time influences the system's performance.

2.2 Design principles and their advantages

Using BLOBs to store data. The target usage of BlobSeer are applications that process huge amounts of data distributed at large scale. The data is organized in huge objects called BLOBs (Binary Large Objects).The typical size a BLOB can reach is up to 1 TB. The BLOB contains a sequence of bytes representing unstructured data, meaning that it can store any kind of data: pictures, sound, movies, documents, experimental measurements [2]. This approach has two major advantages:

Scalability. Maintaining a small set of huge BLOBs is much more feasible than managing billions of small, KB-sized objects. Even if a data management system would efficiently handle a large number a small data object, the process of mapping application-level objects to file names add much overhead.

Transparency. BLOBs are uniquely identified in the system through global ids. The writing of applications that use BlobSeer's interface will became easier, because the programmer is freed from the task of managing data locations and data transfers explicitly.

Versioning-based concurrency control. Provided a scenario where multiple reads and writes occur simultaneously, BlobSeer's solution for controlling concurrency is based on generating multiple versions of the blob that is being updated. The readers will have access to an older version. For each concurrent write operation a new version will be generated. The versions will receive unique numbers. Concurrent reads and writes will never conflict, as they access different BLOB versions. Users can explicitly use versioning. Even when they don't, versioning is still used internally by the concurrency control system for enabling parallelism [3].

Generating new snapshots implies that only the differences in versions are stored. This eliminates unnecessary duplication of data and metadata, saving storage space and speeding the process of generating a new version. The newly generated version is not a copy of an older version, but the same data with new metadata that points to the location of BLOB stripes in the old version and the new data stripe added by the latest update. The solution follows this design principle: data and metadata is always created and never overwritten. Metadata management offers the client the illusion of an independent, self-contained snapshot.

Blob management interface. BLOB access interface must provide means for users to create, read/write a subsequence of *size* bytes from/to the BLOB starting at *offset* and to append a sequence of *size* bytes to the end of the BLOB. The management operations should be asynchronous. The system must provide a way to access past BLOB versions and should also guarantee atomic generation of new snapshots. This interface can be used directly, or it can lay under higher-level data-management abstractions. The Namespace Manager offers such an abstraction. The benefits of this design principle are:

Explicit versioning. Highly parallel data work-flows are common in many data-intensive applications, where data is acquired while it is processed. This work-flow can be sustained by using versioning. While older versions of BLOBs are used for processing, new versions can be generated at acquisition time,

without disrupting the processing thread. BlobSeer's interface offers versioning-based access to data: the user can specify a particular version of the BLOB to be used in the read/write operation. This frees the programmer from the burden of dealing with synchronization.

Atomic snapshot generation. Synchronization of concurrent writes and reads is achieved by making a write operation atomic. Transiently inconsistent snapshots will never be accessible for reading.

Asynchronous operations. In applications where I/O operations are frequent and the data involved reaches hundreds of GB, the overall system performance is affected, despite high data-processing capabilities. I/O operations should not be blocking. The application should launch a request for writing a huge amount of data and still be able to continue its execution immediately.

Data striping. Each BLOB is split into stripes of data that are distributed all over the data providers' machines. This can facilitate data replication mechanisms and allow fast parallel requests from multiple clients as the data can be served by more than one machine. Two factors need to be taken into account for maximizing the benefits of data striping: chunk distribution and dynamically adjusting chunk sizes.

The *configurable chunk distribution strategy* means specifying where to store the chunks in order to achieve the fastest way to access a specific BLOB. This strategy addresses the issue of load-balancing. High throughput when different parts of the BLOB are simultaneously accessed can only be made possible by a good load-balancing scheme. Another advantage would be minimizing energy consumption, an important aspect of green computing.

Dynamically adjustable chunk sizes. In order to parallelize data processing, data must be divided into chunks. The way the computation is partitioned and scheduled influences the performance of data-intensive applications. Choosing the right chunk size is very important. The dynamically adjustable chunk size feature adapts the chunk size to a particular application.

Distributed metadata management. Because a BLOB is striped over a large number of storage machines, a lot of metadata is needed to map all of these strips to a BLOB id, uniquely defined in the system. A data strip is a subsequence of a BLOB defined by offset and size. At large scales the metadata reaches a considerable size and metadata management brings a large overhead if not well structured. Most traditional distributed file systems have centralized metadata management. BlobSeer offers a distributed metadata management scheme with the following improvements over traditional metadata management [4].

Scalability. The distributed metadata management enables better scaling of the system with both the expansion of metadata and increasing number of concurrent accesses.

Data availability. Metadata can be replicated and distributed to multiple metadata providers, making it possible for multiple metadata providers to serve clients. Redundancy prevents metadata management from becoming a single point of failure. This is very important because a lot of pressure is placed on metadata managers as they have to map every stripe location of a large BLOB to a client's request (specified by a BLOB id, offset and size).

2.3 Architecture

BlobSeer consists of a series of distributed communicating processes [5]. The next figure illustrates the processes and the interactions between them.

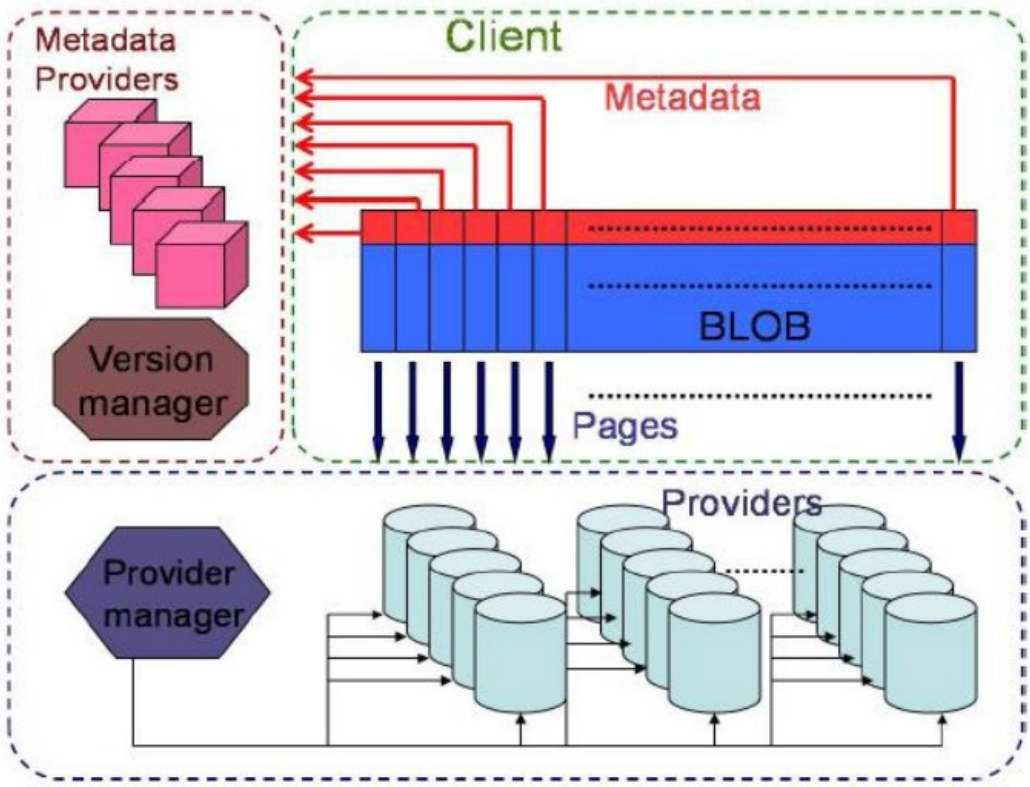


Fig. 1. BlobSeer architecture

Clients may create, read, write and append data to BLOBs. Their number can dynamically vary. A large number of concurrent clients is expected that may all access the same BLOB.

Data (storage) providers physically store the data stripes generated by appends and writes. New data providers can dynamically join and leave the system.

The provider manager stores information about the available storage space and schedules the placement of newly generated stripes. It uses a configurable chunk distribution strategy to maximize the data distribution benefits for a specific application. Each new provider registers with the provider manager. The provider manager choses which providers should be used to store the generated data stripes in such a way that it assures an even distribution of data among providers.

Metadata (storage) providers physically store the metadata that allows identifying the stripes that make up a BLOB version. The metadata management is distributed in order to enhance the performance of concurrent access to metadata.

The version manager assigns new snapshot version numbers and publishes the new snapshots to readers. The version manager offers the illusion of instant snapshot generation, guaranteeing total ordering and atomicity.

This design offers scalability and large-scale distribution. A key design choice is to avoid assigning a static role to nodes: any physical node can play one or multiple roles.

3. Web Services and Cloud computing

3.1 Web Services

A simple definition of this concept is the following: *a web service is a piece of business logic accessible through standard-based Internet protocols [6].* This definition describes a web services as just another web-based application type. What makes this new breed of technology very powerful is the fact that web services offer high interoperability - they are hardware, operating system and programming language independent, since they are based on open standards such as HTTP and XML-based protocols (SOAP and WSDL).

Web services are software components that communicate using standards-based web technologies, designed to be accessed by other applications such as client applications or other web services in a SOA (Service Oriented Architecture) manner. They vary in complexity from simple and discrete operations, such as a currency conversion service or checking a banking account balance, to complex processes running CRM (customer relationship management) or enterprise resource planning (ERP) systems[7].

Distributed systems consist of loosely coupled entities that collaborate in order to complete a common task. Aside from message passing communication, this entities can communicate through another common model – remote procedure call (RPC). This model is very useful because it resembles as close as possible with direct method calls. The programming models that have derived from RPC are CORBA, Java RMI, Microsoft's COM and web services. From all of these technologies, web services are the one that allow for a distributed environment in which any applications (possibly from different organizations) can inter-operate in a standardized, platform independent and language neutral manner. This will create the illusion of heterogeneity in a distributed system.

3.1.1 Web services benefits

Application and data integration. By using XML as the data representation layer for all web services protocols, organizations can easily integrate disparate applications and data formats [7].

Flexibility. Web services can be accessed in a variety of ways: from client applications, web-based interfaces, and even other web services. A client can combine and interpret data from multiple web services.

Code re-usability. Many clients can reuse the function performed by the same web service for different purposes. New applications with different business objectives can be developed by integrating web services that offer specific functions, without the need to rewrite them.

Cost savings. Customized applications for integrating data can be expensive. High interoperability will facilitate the software development process, eliminating the need for "glue" code used for the integration of incompatible software components.

Loosely coupling. In a client-server architecture, a tightly coupled system implies that if the server or client interface changes, the other must also change. Loosely coupled architectures are more easy to maintain. A web service client is not tied to the service directly because the server exposes an XML-based self-descriptive interface (WSDL), which will be discussed later in this paper. The client easily adapts to the service's changes, using the WSDL document to compute a valid request, without the user's acknowledgement [6].

Ability to be asynchronous. Asynchronous capability is a key factor in enabling loosely coupled systems and in accelerating the application response in the case of in-only operations.

Document exchange support. XML has its generic way of representing not only simple data, but also complex documents. Web services support the transparent exchange of documents.

3.1.2 Web Service Architecture

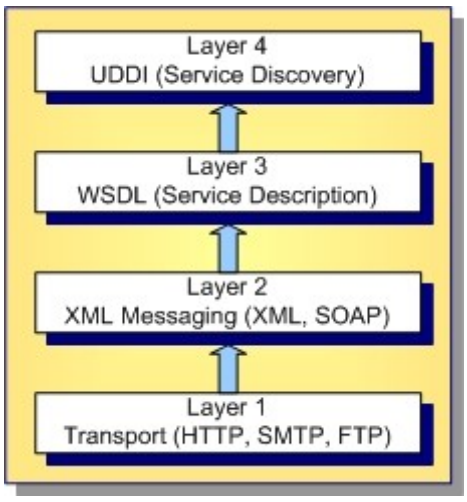


Fig. 2. Protocol stack

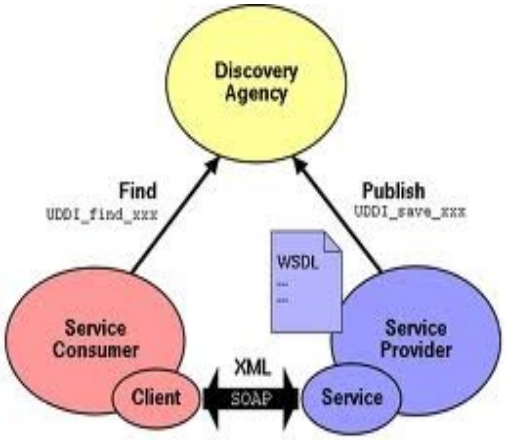


Fig. 3. Web services architecture

SOAP (Simple Object Access Protocol) is the protocol for exchanging data over a variety of standard Internet technologies, including SMTP, HTTP and FTP. Web service technology uses SOAP to send messages between a service and its client. SOAP messages are XML documents that contain some or all of the following elements [7][8]:

Envelope - specifies that the XML document is a SOAP message. It encloses the message itself. All other elements are sub-elements of the envelope. Each can contain namespace declarations as attributes to correctly identify and interpret data. A namespace declaration is used to specify the envelope structure. The envelope may have attributes such as the encoding style and a namespace declaration for it, although a standard encoding type is now used by all SOAP web services.

Header - is optional and contains information relevant to the message such as the date the message was sent, authentication data, etc. The header might also include the “must understand” attribute which implies that the user wants the service not to ignore unknown headers. This is useful when multiple versions of a service exist and the client wants to use a specific version. The “actor” attribute specifies the recipient of a header element. This is important because a request sometimes passes through one or more intermediaries before being processed. SOAP requests can be processed in a distributed manner. This is an advantage SOAP has over other web service messaging protocols which will be discussed later in this chapter.

Body - includes the message payload for a request or response. The message may have an RPC-style structure, specifying the method and its parameters. The data must conform to the WSDL interface, which contains data type definitions.

Fault - this element carries information (the fault code and an appropriate message) about a client or server error within a SOAP message. The SOAP defined faults indicate: version mismatch , the service is unable to interpret a header that has the “must understand” attribute, the client payload did not contain the required information, the server failed to process the request for a different reason that a bad request content.

WSDL (Web Services Description Language) is an XML-based format for describing web services. A client who wants to access a web service must interpret its WSDL file to find the location of the service and its available operations, the communication protocol, and the correct format for sending messages. The main elements are: port type (groups and describes the operations performed by the service), port (specifies an address for a binding, defines a communication port), message (describes the names and format of the messages supported), types (defines the data types, as defined in an XML Schema, referred in the message elements), binding (defines the communication protocols supported) and service (specifies the address - URL for accessing the service) [6].

UDDI (Universal Description Discovery and Integration), often described as the yellow pages of Web services, provides a worldwide registry of web services for advertisement, discovery, and integration purposes. In order to discover a web service that provides a specific functionality, a client must search available web services by names, identifiers, categories, or the specifications implemented by the web service. UDDI provides a uniform way for listing services and communicates via SOAP messaging with the client [6].

These protocols (SOAP, WSDL, and UDDI) interact in the following way: an application that needs a web services has to identify the location of an appropriate web service. This service is located somewhere on the Internet. The client send a request to a UDDI broker to search for the service either by name, category, identifier, or specification. Once identified, the client retrieves the WSDL of that service from the broker. The WSDL document contains information about how to access the web service. The client binds to the service by sending a SOAP message that is valid according to the XML schema found in the WSDL.

Service-Oriented Architecture (SOA) is an architectural approach that promotes building complex application systems by integrating distributed components, called services, that implement a specific function. This components may be provided by different vendors, operating in different domains. Services may be registered in a global catalog in order to be identified. If a large variety of services are available, building a new application simply implies putting together a few already available services [8].

Its main features are: modularity, implementation hided by the Interface, code re-usability, loose coupling, standardization which eases integration, statelessness (any two service invocations are independent). The SOA architecture is often associated with web services, but it has a wider meaning. SOA can be mapped over distributed systems like grids and Clouds.

SOAP versus REST

SOAP is not the only way to build service-based applications. REST is more of an old philosophy(a software architecture for distributed hypermedia systems such as the World Wide Web), but a newer technology for web services.

This thesis focuses on the SOAP approach, because SOAP security poses more challenges than RESTful services. In order to understand this challenges, we must see how the two mechanisms differ, although our intention is not to make an exhaustive comparison between them. While SOAP relies on heavy-weight standards in order to achieve full platform and language independence, in RESTful web services, the emphasis is on simple point-to-point communication over HTTP using plain old XML.

REST is a model for creating services that follows the web model for manipulating resources that are identified by a unique URI. This model is data-centric, not functionality-centric [9]. The action used to manipulate resources is specified by a transport protocol verb(GET, POST, PUT, and DELETE from HTTP 1.1). The nouns are the resources available on the network. REST was initially described in the context of HTTP, but is not completely bound to that protocol(any application layer protocol can be used instead with the right adjustments). The main advantage of RESTful services is the possibility to make full use of the pre-existing, build-in capabilities provided by the chosen underlying protocol and minimize the addition of new application specific features on top of it. REST uses the existing features of the HTTP protocol such as verbs, URIs, response codes, HTTP caching and security.

In SOAP, each application developer must define a new and arbitrary vocabulary of nouns and verbs (the service operations), therefore is more heavy-weight than REST [10]. By doing so, SOAP disregards many of HTTP existing

features (authentication, caching and content type negotiation) and the service designer must re-invent many of these capabilities. SOAP was designed to be extensible, so that other standards could be integrated into it. These standards are collectively referred to as WS-*: WS-Addressing, WS-Policy, WS-Security, WS-Federation, WS-ReliableMessaging.

REST web services are more concise, don't need additional messaging layer and are similar in design to the web architecture. However, they are designed for a point-to-point communication model, therefore less adequate for distributed computing environments where message may go through one or more intermediaries, before completing a task. SOAP can be used in the context of more complex applications, thanks to its additional headers [11].

3.1.3 APACHE AXIS

There are many software tools for developing SOAP web services: Microsoft .NET, IBM SOAP, Apache SOAP fallowed by Apache Axis, IBM WebSphere, GLUE. Web services can be hosted by a variety of server technologies: servers that support servlets, or can run standalone using their own HTTP server. The most common standards for SOAP-based messaging and RPC APIs for Java web service implementation are JAXM (the Java API for XML Messaging) and JAX-RPC (Java API for XML-based RPC).

For this project we have chosen the Java version of Apache Axis2. Apache Axis2 is an open source web service framework. Axis is basically a SOAP engine: a framework for constructing SOAP processors such as clients, servers, gateways, etc. Axis also includes: a simple stand-alone server for hosting web services (which we have used in this project), a server which plugs into servlet engines such as Tomcat, extensive support for the *Web Service Description Language (WSDL)*, tools that generate Java classes from WSDL or WSDL descriptions from Java classes (this project uses the second development tool), and a tool for monitoring TCP/IP packets.

The current Axis release is SOAP 1.1/1.2 compliant but has also integrated support for the widely popular REST style of Web services. Axis2 comes with many new features and enhancements:

- **Speed** – Axis uses its own object model and StAX (Streaming API for XML) parsing to achieve greater speed.
- **AXIOM** – a light-weight object model for message processing which is extensible, performance optimized, and simplified for developers. This model is also used in our project.
- **Rapid Deployment** – new web services and handlers can be deployed while the system is running, without having to restart the server. Deploying new services means simply dropping the web service archive into the services directory in the repository.
- **Asynchronous Web services** – invocations can be asynchronous by using non-blocking clients and transports.
- Other new features are: SOAP with Attachments support, the possibility to insert extensions into the engine for custom header processing, the absolute abstraction of transport, WSDL version 2.0 support, security add-ons, improved support for extensibility by using modules and phases.

3.2 Cloud computing

Cloud computing describes highly and dynamically scalable computing resources that are provided as an external service over the Internet on a pay-per-use basis [13]. Cloud computing allows both personal and business users to run software (e.g. business applications, e-mail) or use infrastructure (e.g. storage space) on an as-needed basis. The client doesn't need to have the knowledge or control over the technology infrastructure in the Cloud that supports them [14]. The complex infrastructure is completely transparent to the user so that the user doesn't need to install any special software, use a specific operating system or hardware. Users do not download and install applications on their own devices since all processing and storage is maintained by the Cloud server, removing the burden of software maintenance and support for business or personal users. Users can use Cloud services from multiple platforms computers, pads and tablets, smart phones, as long as they have Internet access. This is due to the fact that Cloud services are web-based.

Cloud applications can be implemented with multiple technologies, as long as they are web-based. They range in complexity from very simplistic services that only require the HTTP protocol, to more sophisticated web applications that require databases, web service infrastructure and message queues [15]. A Cloud environment is a distributed system consisting of numerous heterogeneous entities. A good choice for implementing Cloud applications are web services, since they bring homogeneity to the world of distributed computing.

Cloud services are delivered in three ways: Platform as a Service (PaaS), Infrastructure as a Service (IaaS) and Software as a Service (SaaS). This thesis attempts to offer secure access to SaaS. SaaS [16] is based on a “one-to-many” model which means that an application is shared across multiple clients. Next, we will try to bring arguments in favor of the idea that web services represent a suitable technology for the implementation of SaaS.

In order for this infrastructure to be scalable, multi-tenant efficient and configurable to the needs of each user, SaaS applications must be designed to run in a stateless way, which means that any necessary user and session data should be stored either on the client side, or in a distributed store that is accessible to any application. In order to achieve performance the applications should have a design that allows asynchronous I/O and a highly concurrent database. SOAP web services allow asynchronous invocations and function in a stateless fashion. BlobSeer is a data storage service that suites the needs of SaaS applications.

Software as a Service can also take advantage of Service Oriented Architecture to enable software applications to communicate with each other. SaaS applications cannot access internal resources such as databases or internal services unless they use integration protocols and APIs that operate over a wide area network. These are protocols based on HTTP, REST or SOAP.REST and SOAP services enable easy integration between SaaS applications and internal resources or other SaaS applications.

4. Related work

Cloud security is challenging due to the large number of resources and users, the interaction between different security domains each with its own degree of security and different security mechanisms and implementations. A heterogeneous security management system can be achieved by using web services (as discussed in the previous chapter) and web services security standards.

XML Signature and XML Encryption

The two mechanisms were developed for securing business transactions on the Web. Encrypting the communication (e.g. Secure Socket Layer protocol) is not enough to protect such sensitive data; the data needs to be encrypted as well. This mechanisms address the issues of source identity, integrity, confidentiality and non-repudiation. They take advantage of the special nature of XML data and are currently being standardized. Together with XML Canonicalization, they offer a particular way for signing and encrypting XML documents: partial signing/encryption (of individual elements or entire documents). This is highly important for allowing end-to-end security and not only point-to-point security. During a transaction, an XML document can be processed by multiple entities. Each entity might only want to sign the part of the document it is responsible for. When assuring the integrity of data, the XML signature mechanism must specify which elements correspond to each signature, or else the recipient might interpret the data as corrupt if an intermediary has added new data to the document, and the verified signature has been computed for only the data the first entity had attached to the message.

XML Signature and XML Encryption [6] rely on existing security mechanisms such as public key infrastructure and asymmetric cryptography, symmetric encryption, secret key negotiation, hash/digest transformations.

A digital signature ensures the integrity of data by using the mathematical property of irreversible functions that produces a unique outcome for each input, and pair numbers that can be used by encryption algorithms in such a way that the data encrypted using one number can only e decrypted using the pair number (the numbers represent public and private keys). Data that has been encrypted with the intended receiver's public key can only be interpreted by the original receiver (using its private key) and the other way around. In order to correctly identify the sender, public keys and sender identity need to be recognized by a third part. This is called a Certificate Authority and the mapping between identity and public key represents a certificate. The recipient verifies the signature by applying the same algorithm on the message and comparing it with the signature. Only the digest of the secured data will be signed, due to the complexity of the used algorithms .

XML Signature requires that the data that was signed is accessible either by specifying the URI inside the signature, embedding the data in the signature or referencing the data as another element inside the same XML document as

the signature. An XML signature element contains sub-elements that specify the following: signed data reference, digest method and digest value for each reference, the signature method, a “key info” element that can embed or reference the sender's X.509 Certificate (the standard for certificates) and the value of the signature. The XML EncryptedData element has a similar structure, containing the encryption algorithm, key info, cipher data which contains the cipher value (the encrypted data). This two mechanisms are very flexible because they allow both symmetric key and asymmetric key algorithms, and allow the user to choose from a variety of mechanisms such as md5/sha, des, triple-des, aes, rsa, etc. [18].

Kerberos

Kerberos is a security mechanism that addressed mutual authentication. Although it was originally used in operating systems security, it is a suitable authentication protocol for network security and Cloud access to services ass well. It addressed the issue of single sing-on on systems with many independent resources. We describe this protocol in detail because our solution shares some concepts of Kerberos [19]. The core of this protocol is using tickets for mutual authentications.

The are four entities involved: the client machine, the authentication server (AS), the ticket granting server (TGS), and the service server (SS). The steps are:

- the user enters the username and password only once; the client application computes the hashed value of the password, which will be used as the user's secret key
- the client send a simple request to the AS; the AS computes the user's secret key from the password that had already been stored in the AS database; AS sends back a key shared by the client and the TGS encrypted with the client's secret key and ticket-granting-ticket encrypted with TGS's private key; the client obtains a session key for client-TGS communication from the received message
- when requesting access to a service, the client sends the ticket granting ticket, the service name, the client id and time stamp and receives a ticket for accessing the service and a client-SS session key.
- the client can communicate with the service server using the shared session key that the service server will obtain from the client's request after decrypting it with its private key, and will be authorized to use the service by the ticket (which will eventually expire).

The features that we keep from this authorization protocol are: single sign-on and the ticket granting service. Instead of using private and public keys to encrypt the communication between client and TGS, we will implement a shared key negotiation protocol. We will eliminate the need for AS. A single entity will play both roles: authentication service and security token granting service.

WS-Security

WS–Security is the standard for web service security. This standard does not define new security solutions, but relies on existing standards and specification. It describes the way to integrate this mechanisms with SOAP messaging protocol [6].

Using HTTP security for caller identification, message signing and content encryption is a solution that only applies to point-to-point communications. Web service requests might be processed by multiple intermediaries before reaching destination, in a SOA fashion. Section 3.1.3 introduces the SOAP message format and the “actor” attribute that identifies the recipient of a specific header. Once the header is processed, its contents are dropped and the rest of the message is send to the next actor. In order to allow secure end-to-end communication, new security protocols need to be developed. Another reason why relying solely on the transport protocol security mechanisms is insufficient is the fact that SOAP aims to be transport independent and the message can be carried out over different transport protocols from one recipient to another.

The existing solutions for authentication, message signing and encryption that WS-Security uses are: Kerberos and X.509, XML Signature, XML Encryption and XML Canonicalization (this is used for preparing the XML message for signing and encryption). All of these elements will be embedded into the SOAP message.

WS-Security defines a new SOAP header that will contain all security elements. Any number of *EncryptedData* and *Signature* elements can be inserted in this header. WS-Security also introduces a mechanism for transferring authentication credentials. The *UsernameToken* element can be used for simple authentication credentials (a custom authentication based on username and password), while *BinarySecurityToken* will manage more complex credentials (Kerberos tickets or X.509 certificates) [20].

A typical authentication message flow is the following: the web service client sends a request to a security token service and obtains an access token with a time-stamp. The client will include this token in his message, sign the message and send it to the web service. The security token service might not be implemented as a web service. When using Kerberos, the client should sign the message with a secret shared only by the service and client, such as the client's hashed password. If X.509 certificates are used, the client can sign the message with its private key and the server will decrypt it using the client's public key.

WS-Security uses additional utility elements contained in the *Wsu* namespace: id and timestamp. Timestamps are used for preventing reply attacks, while the id elements are used to simplify processing for the intermediary actors. The SOAP security header has a different format for each type of authentication method: basic and digest password authentication, X.509, Kerberos. Any other methods can be used as well, because the WS-Security standard doesn't require a specific format. The syntax for signing and encryption will be the same as described for XML Encryption and XML Signature in the previous section.

5. Secure access to services using BlobSeer

5.1 Addressed security concerns

Our security solution is based on a central authority referred to as *Security Manager*, that is responsible for users, services and access control management as well as for offering authorization credentials to legitimate users**.**

5.1.1 Client identity

The first important step in securing a Cloud environment, as with any other Internet accessible system, is preventing access from unwanted users. To authorize access to resources, applications first need to authenticate the source of the request (referred to as the *subject*), which can be any entity, such as a person (the end user's client program) or a service (a service the user has accessed that invokes another service).

The *authentication* typically involves the subject demonstrating some form of evidence to prove its identity. In our implementation the evidence is the password. In order to to avoid password attacks, we have chosen to encrypt the password with a secret key shared by the user and Security Manager, addressing the issue of confidentiality. Once authenticated, the subject owns a set of security attributes, referred to as *credentials*. The credentials are used to authorize access to subsequent service invocations. In order to protect the system against reply attacks or minimizing the impact of such attacks, credentials must be unique, arbitrary generated and have a short life span. This are the characteristics of a session identifier. In our application, the *session id* is generated by the Security Manager as fallows:

$$SID = hash\ function(username) + random\ number + login\ number$$

5.1.2 ACL management

A protection system groups all the conditions under which a system is secure. An access control matrix[17] is a way to describe a protection system. It must describe relations between all resources (objects) and users (subjects). The access control matrix model has its origins in operating systems design and database access, but can be successfully applied to Cloud resources access control too. There are two ways for implementing the two-dimensional array (access control matrix): access control lists (ACLs) and capabilities lists. In the access control list approach authorities are bound to the objects, in the capabilities list authorities are bound to the subjects. An ACL is a list of permissions (subject, operation) associated with various objects. A capabilities list is a list of objects and their corresponding permissions attached to a subject. Both implementations have advantages and disadvantages, but choosing one of them depends on the system that needs to be secured. Capabilities lists allow a finer grain of protection and allow a safer way for managing rights transfers. In a Cloud environment, users outnumber services and are more dynamic, so a

capabilities-based control access is more suitable. Although we choose to implement access control as capabilities list, we will refer to them as ACLs, because of the wider meaning of this term.

Cloud access management is based on concepts like on-demand resources and pay-per-use services. Thus the access control scheme must take into consideration the fact that a certain user can only request access to some of the available services, to only some of the operations this services provide, for an indicated period of time.

We have decided to implement the access control scheme with *Access Control Lists*. Scalability has been taken into consideration when choosing a “per user” list, instead of a “per service” list, because the number of users is usually larger than the number of service providers and services. Besides this, users tend to leave or join the system in a more dynamic way. The allowed access levels are: 1 – owner, 2 – user.

Security Manager must provide the means to update a user's ACL in the following situations: a user asks for permission to *invoke* a service, a user asks for permission to *host* his services on the Cloud, a user *delegates* some of his rights to another user.

5.1.3 Single sign-on

A Cloud platform can host applications (in this case web services) that come from different providers, each with its own security policies. If each of these services would independently manage security, losing control over information security at global level could become an issue in the could, as well as adding another password for end users to manage. This would pose a risk to security and increase the complexity of security management both in the Cloud and for the client.

Our solution offers single sign-on to SaaS web services without requiring integration of the authentication infrastructure with the web service owner, in order to accelerate the deployment of new services. In order to achieve this, all security information (single password, single encryption key) is logical *centralized* using the BlobSeer storage system, still ensuring high throughput data access because of the distributed nature of this system. The second measure is to separate the functional layer and the security layer in the services. The security layer is represented by a universal security service that will perform all necessary access control verifications. The functional layer is the unaltered service a owner hosts on the Cloud. The Service Oriented Architecture pattern that has inspired this approach will be discussed in section 5.3 along with the issues that arise from this approach.

5.1.4 Access logs

Access logs are the starting point for security audit in this scenario. By storing information every time a user logs in the system or invokes a service, the application ensures full trail of access for all users that may come in hand in the event of a security breach.

5.2 Project requirements

This section lists the main operations Security Manager must offer in order to make full use of the proposed access scheme capabilities. The steps for invoking a service to which a user has gained access by using any of the Security Manager operations are also mentioned.

1)Registration. The client must provide a password that will be encrypted using a secret key that Security Manager and the client exchange during the registration process.

2)Authentication. Service Manager will provide the client with a session id, considering that web services use a stateless protocol such as HTTP.

3)Available services list. The client needs to be aware of the existent services, so that it won't unnecessarily ask for permission to a service that is no longer available or he already has access to, try to invoke it, or try to deploy a web service under the same name with a pre-existing web service.

4)Grant access. The client can ask for permission to invoke an operation of a web service for a given period of time. The access level this operation sets is “user”.

5)Grant owner rights. A client can ask for permission to host a service on the distributed system. This operation sets the “owner” access level.

6)Delegate rights. A client can offer access (“user” level) to another client for a service he owns or has access to.

Once a client has gained access to a service (using one of the operations 3, 4 or 5), the client can invoke the service in a secure manner. The following operations will not be performed by the Security Manager. This operations are done by an individual entity for each deployed service in order to ensure scalability with the number of services, users, geographical scalability, and independence from Security Manager.

7)Check for permission to invoke the service.

8)Log the client's access to the service (web service name, operation name, access time).

9)Invoke the service. The invocation process is independent from the Security Manager.

5.3 Decoupling the security layer and functional service layer

Section 5.1.3 has highlighted the benefits of single sign-on in a distributed system with many independent applications. In our implementation the client must present the credentials obtained from the Security Manager in the authentication process (using a single password) each time he invokes a service. The credentials and the permission to invoke the service verifications are performed in the same way before invoking any service.

We have established that a service invocation consists of the secure access layer and functional layer. The question that remains to be answered is: where in this infrastructure should the secure access layer of the invocation be placed?

Before a web service answers a request, the system must check if the user's session is still valid and if the service is listed in the user's ACL. If the service performs this verification itself, then the original code of the web service whose functionality has been clearly specified in its WSDL description document must be altered. In this approach the web-service is security-aware. The disadvantages of this implementation are obvious: the security layer in the Cloud and the functional layer of the web-service are tightly coupled.

A much more flexible solution is placing the secure access layer in a different web service. This will ease the integration of the authentication infrastructure in the Cloud with the owner service code, without the need to alter the original service code or re-deploying the service each time the security mechanisms change.

This solution is compatible with *logic centralization*, a Service Oriented Architecture design pattern that aims to increase the re-usability of logic by ensuring that services do not contain redundant logic and that any reusable logic is placed in the most suitable service functional context [23]. The security logic will be contained in a new service having the role of a proxy for each service.

There are multiple ways to map the proxy web services to the actual services: as a single secure access service that acts like a proxy for all services, multiple replicas of the secure access service, or each service with its own secure access service. We have chosen the third approach because it is more scalable. This also allows anonymous access to a service. The client will know the endpoint of the secured service, but not the endpoint of the actual service. The client will invoke the proxy service, which offers the same interface as the actual service, and the proxy service operation will invoke the actual service after checking weather the client has access to the operation or not.

The disadvantage is the overhead added by the secure access service invoking the functional service, which results in two service invocations instead of one. The experimental results will analyze the overhead.

5.4 Invocation steps

This section describes the full functionality of the operations listed in section 5.2 presenting the preconditions needed for each operation to succeed and the changes each operation causes to the system's security information.

1)Register. This operation is divided in two phases: requesting a username and exchanging keys (stage 1) and sending the encrypted password (stage 2).The first phase can fail if the username is unavailable or if an error occurred when exchanging keys, otherwise a new entry will be created in the users database and the secret key will be stored in the corresponding file. After stage 2 the password will also be stored.

2)Authentication. The preconditions for this operation are a valid username, a valid encrypted password and no other open session. This operation results into writing the generated credentials (session id, authentication time and expiration date) to the corresponding file.

3)Grant access. The client must be authenticated, the requested service and operation must be available. Service Manager must thus consult the user's credentials file, files containing information about existing services and their operations. This operation will alter the user's ACL file.

4)Grant owner access. The user must be authenticated, the requested service name must be available. This operation will alter the user's ACL file and a new entry will be created in the available services directory.

5)Delegate access. The user must be authenticated, the user who is being granted access must be registered, the service and operation must exist, the user delegating access must have access to the service himself. This operations will alter the second user's ACL file.

6)Service invocation. The user must be authenticated, the secured service must consult the user's ACL. Each invocation will log the name of the operation, service and time, after which it will invoke the original service.

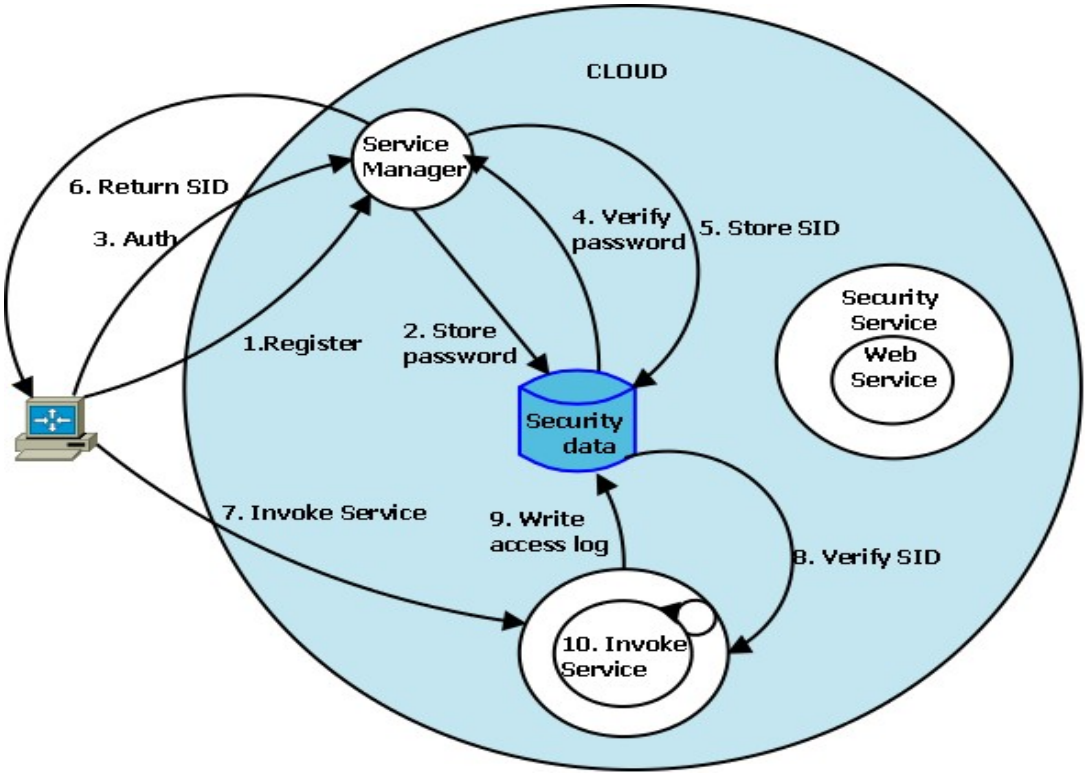


Fig. 4. Usage diagram

5.5 Security data storage

The previous section highlights the basic information every security management system needs to store. Additional information like logs may also be needed in order to address other security issues such as Denial of Service attacks from malicious users.

The proposed security scheme stores data individually for each user and service. The *services* directory will contain an individual file named after each service containing the following information. Likewise, the *users* directory will contain a subdirectory for each user.

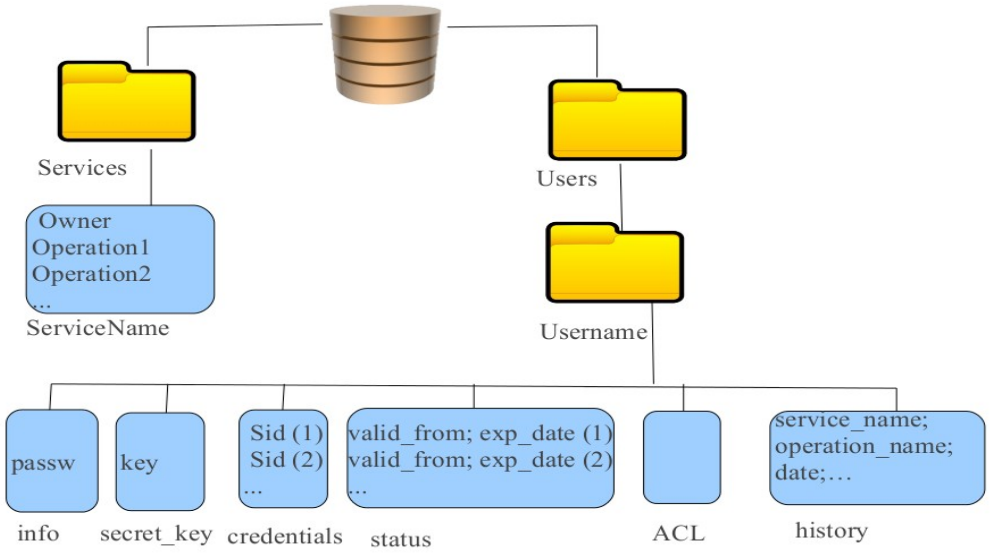


Fig. 5. Security storage

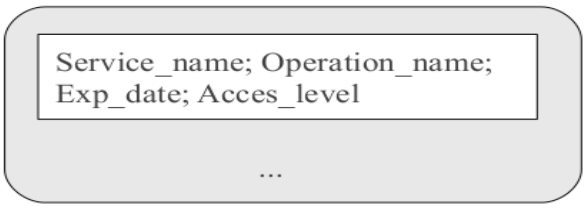


Fig. 6. ACL file

The amount of data stored for each user for security purposes may not be significant, but the overall amount of data can easily reach TB when storing access logs for long term users in systems with numerous users. The scalability of the storage system is an important factor because Service Manager must create a new directory each time a user registers.

The necessity of high throughput under heavy access concurrency raises from the fact that more than one user can login, access a service, ask for permission to use/deploy a new service or invoke multiple services simultaneously. Most of a user's files can be altered or read simultaneously by Service Manager and other services the user invokes. The application makes use of the decentralized data feature of BlobSeer, but more important than this is the decentralized metadata management.

5.6 Key exchange

This project uses symmetric key cryptography for authentication. The shared keys are generated using the Diffie-Hellman protocol. The Diffie–Hellman key negotiation method allows the two parties (Security Manager and new client) that have no prior knowledge of each other to establish a shared secret key over an insecure communication channel. HTTP, SMTP, FTP are valid application layer protocols used as transport for SOAP (the most common messaging Protocol for web services), but HTTP has gained wider acceptance as it works well with today's Internet infrastructure. The transport protocol used in this project is HTTP, therefore an insecure communication channel. This key will be used to encrypt subsequent communications using the AES symmetric key cipher.

The two parts establish a secret key during the registration process. In order to protect the password at registration time, the registration process is divided in two phases. During phase 1, the client generates the Diffie-Hellman parameters (the *P* and *G* numbers), generates random number *Xa (secret),* computes *Ya = G ^ Xa mod P(public)* and sends a request containing *Ya* and the username to Security Manager. The manager can reject the request or accept it, in which case it generates the secret number *Xb*, computes shared secret key, stores it and sends back to the client a message containing *Yb = G ^ Xb mod N.* Now the client can compute the secret key also. This protocol consists of two service invocations.

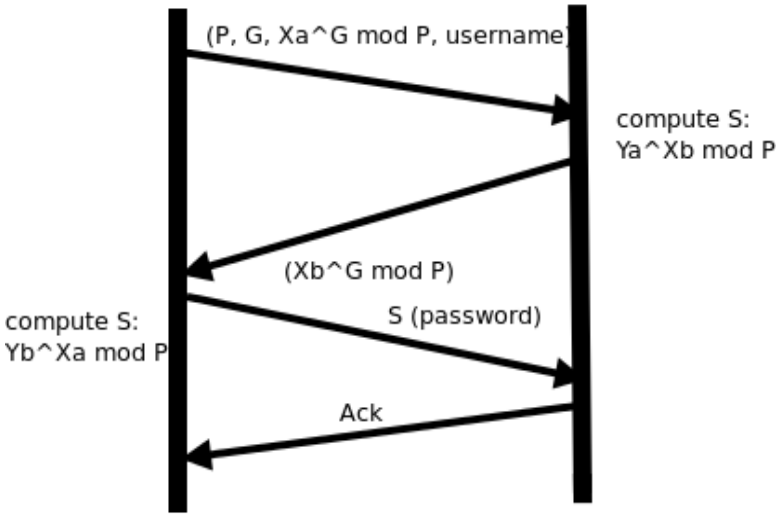


Fig. 7. Registration protocol

The next diagram describes the authentication protocol. This is a challenge-response protocol that attempts to prevent sniffing or reply attacks. In order to prove its identity, the client must provide the correct password and posses the previously negotiated secret key. The secret key generated in the registration step and used for authentication, could be the starting point for the encryption of all subsequent communication. This could be achieved by using session keys that will be transferred to the client in a secure way, encrypted with the secret shared key.

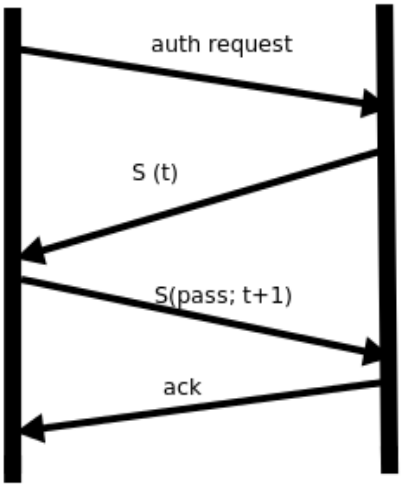


Fig. 8. Authentication protocol

5.7 Delegation

Delegation is a mechanism of assigning access rights to a user. Security Manager already offers grant-access and owner permission operations. Delegation is not a redundant functionality for our system, because this mechanism is top-down oriented (from higher access rights to lower rights) as opposed to the other two access control managing operations.

Delegation comes in many forms: administrative delegation, user delegation, delegation at authentication/identity level, delegation at authorization/access control level, grant or transfer delegation [21]. Delegation at authentication level, while very popular in operating system, is not suitable for Cloud platforms. Therefore, our application will only address access control level delegation.

Administrative delegation allows an administrative user to assign access rights to a user but does not necessarily require that the administrative user has the ability to use the access right. Therefore, we will only address user delegation. This allows a user to assign some of his rights (requires that this user can use the right himself) to another user. Only individual permissions can be delegated. Delegating all permissions would almost be the same as delegating the identity. Cloud clients sign up for particular application and every client has specific needs at some point, so there is no need to transfer all permissions to another user. Only the *use*r access level can be granted by this mechanism. The first user must have *owner* rights or the right to access a service for at least the amount of time it grants to the second user.

We have chosen a grant delegation model instead of a transfer model, so that the service is still available for the first user. The motivation is the fact that our system offers time limited access to a service, instead of a pay-per-use model where the client is billed according to the number of invocations.

6. Implementation details

The project implementation consists of demo web services, their corresponding proxy services, the Security Manager service and the client application.

All web services and the client application were developed using Apache Axis2 Axiom API for the Java programming language. Axiom stands for AXis Object Model (an XML data model conceptually similar to DOM and JDOM) and it is an API for manipulating the XML data in the SOAP request and receive messages.

This project has a modular design divided in several packages. The Security Manager service implementation class and the client implementation class are included in packages from the following package diagram. All other proxy services for each demo service import the “common” package, which contains security primitives that rely on BlobSeer. The “common” package imports “fblobseer”, the Java bindings for Namespace Manager's client. Namespace Manager enables a file-oriented access to BlobSeer by mapping BLOBs and files.

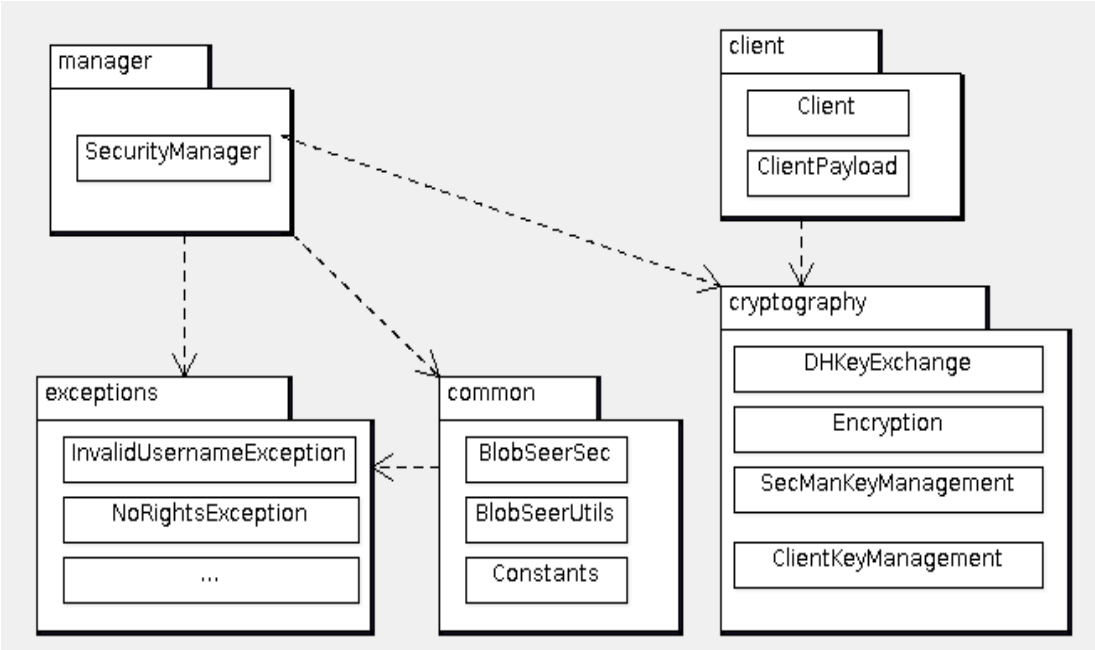


Fig. 9. Package diagram

All packages in the above figure will be detailed in the following section.

- *manager* package

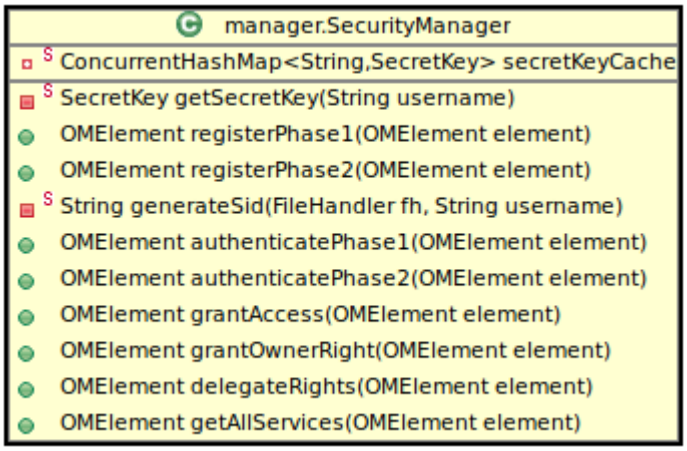


Fig. 10. manager package

- The *SecurityManager* class implements all operations exposed by the corresponding service. The two private functions are used for generating a unique session id that authorizes operations subsequent to authentication and for retrieving a user's secret key.
- Each operation exposed by this service interacts with BlobSeer for testing the preconditions of each operation and updating ACL files, status files, credential files etc. according to the specification describes in section 5.4(Invocation steps).

- *common* package

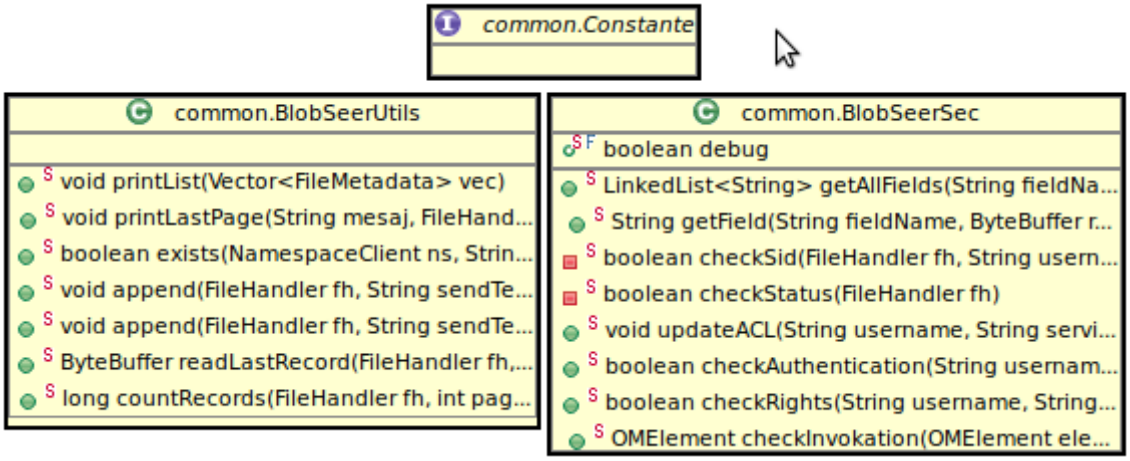


Fig. 11. common package

- The *Constants* interface defines configuration parameters for managing BlobSeer files such as the BlobSeer configuration filename, BLOBs page size, the name of each file that stores user or service information, the entry size for each type of file, constants that describe the data structure

for each file. The encryption algorithm is also defined as a constant. The purpose of this interface is to group all configuration parameters the user of the application might adjust to its needs.

- *BlobSeerUtils* is a class that provides higher-level functions for working with BLOBs. The current BlobSeer implementation allows writing only multiple of page size bytes to a BLOB. The wrapping *append* function allows an arbitrary data size. An arbitrary size write operation implies reading the last page, updating it and writing it back. A page size closer to the average record size advantages the write operation. Since a record size is much smaller (256B) than the usual page size (64KB) this could result in significant metadata overhead. Choosing the right page size is a trade-off between the performance of a write operation and metadata management. The other functions are used for exploring the content of files and directories.

- *BlobSeerSec* implements security primitives. The *getField* and *getAllFields* functions are used for conveniently exploring the structure of security data files.Most of the functions in this class are used by the Security Manager class for managing the access control list, while *checkInvocation* contains the entire logic behind the proxy services.

■ *exceptions* package

Fig. 12. exceptions package

- This package defines exception classes for every reason a request to Security Manager service or service invocation might fail. They are caused by user errors, while internal errors are reported by any other type of exception. All Security Manager operations are in/out operations returning a success message or a message corresponding to the thrown exception. Proxy services offer in/out wrapper operations regardless of the invocation type of the original operations.

- *cryptography* package

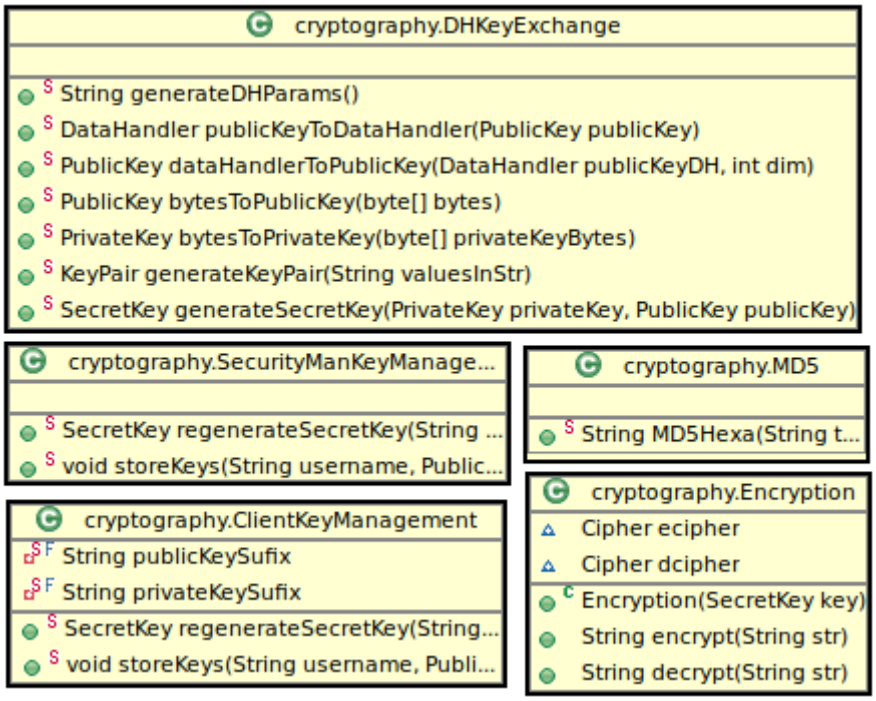


Fig. 13. cryptography package

- This package is shared by *manager* and *client packages. The* message digest function if used for generating the session id. The *Encryption* class instantiates a chipper for the encryption algorithm defined as a constant.
- *DHKeyExchange* functions are used in the key establishment process.
- The *SecurityManKeyManagement* and *ClientKeyManagement* classes expose methods used for saving secret keys to blobs/binary files and restoring them.

- *client* package

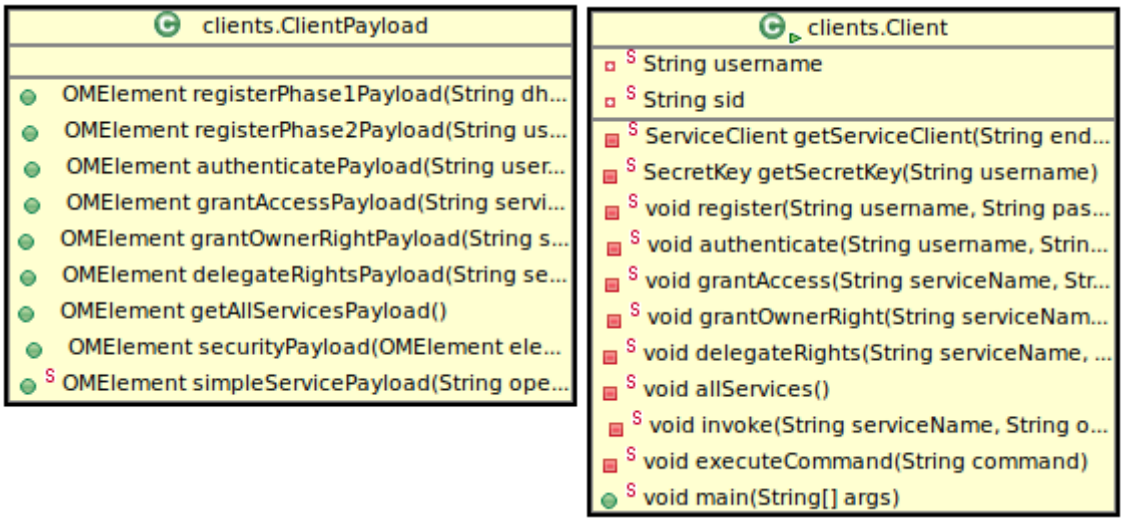


Fig. 14. client package

- The *ClientPayload* class generates the request payload for each type of service operation. This separates the web service call specific code making the *Client* class less verbose.
- The client accepts input from files, standard input and command line parameters. The *executeCommand* method interprets commands and invokes the correspond web service operation.

SOAP attachments

During the registration, the client application and the web service must exchange public keys. The keys are represented by the PublicKey class. Despite the fact that XML is very flexible, sometimes serializing data as XML is not the best option. In order to transmit a custom data-type, the programmer must define new serializers and deserializers and both the client and service implementation class must register the new type mappings. In this case it is much more convenient to transmit the binary representation of a PublicKey object. The binary attachment can be send together with the SOAP message.

One way to support opaque data/binary data in XML is to encode objects as base64 or hexadecimal text and embedding them directly in the XML message as elements or attributes. This approach is known as “by value” transfer. The other solution is called “by reference transfer” and involves attaching the data externally, outside the XML message which will contain an additional element or attributes representing the URI to that data. The advantage is saving processing power, the disadvantage is working with two data models. This approach is similar to e-mail attachments.

MTOM (SOAP Message Transmission Optimization Mechanism) is a specification for sending binary data in Axis2, that tries to combine the advantages of the two mechanisms mentioned before. It uses one data model, and the attachment becomes in-line with the message, although its physically separated [22].

7. Experimental results

7.1 Functional testing

Test setup

We have deployed ten demo services (named $Service_i$) and their corresponding proxy services (named $SecService_i$). A successful invocation of $SecService_i$ will return a message from $Service_i$ with the following format: Response from: $Service_i : operation_j$ $i=1,10$ $j=1,5$.

Two users are registered: user1 and user2. User1 is the owner of $SecService_1 \ldots SecService_7$. Services 1 through 7 with operations 1 through 5 are the only available services at the moment. User2 has been granted access to all operations of services 1 to 6 for the next 300 seconds.

The available commands for the client application are:

- register: r username password
- auth: a username password time
- access: g service_name operation_name
- own perm: o service_name [operation_name]+
- delegate: d username service_name op_name time
- invoke: i service_name operation_name
- list: l all|perm
- exit: e

Test output

Each invocation response is followed by the corresponding command and commentaries on the expected output. The test case contains almost every possible conditions for each type of operation.

```
11:49:34> Command: r user2 user2                                    # unavailable username
11:49:37> Register phase1 response: The requested username is unavailable.
11:49:38> Command: a invalid invalid 120                            #  invalid username
11:49:38> Authentication operation failed. Invalid username. Error opening key file.
11:49:39> Command: a user2 invalid 120                              # invalid password
11:49:39> Authentication response: Invalid username or password.
11:49:40> Command: a user2 user2 300                                # success
11:49:40> Authentication response: OK
11:49:41> Command: a user2 user2 300                                # another session is open
11:49:41> Authentication response: Another session is open.
11:49:42> Command: l all                                            # list all available services
11:49:42> GetAllServices response: OK
        SecService7
           operation1
           operation2
           operation3
           operation4
           operation5
```

```
        ...
        SecService1
            ...
11:49:42> Command: l perm                                   # list services whit permission
11:49:42> GetAllServices response: OK
        SecService6
           all          u           30.06.2011 11:50:30
        ...
        SecService1
                     all          u           30.06.2011 11:50:30
11:49:43> Command: i SecService1 operation1                 # success
11:49:43> Invocation response from SecService1:operation1: Response from Service1:operation1
        ... # test permission for services 2, 3, 4.
11:49:48> Command: i SecService5 operation5                 # success
11:49:48> Invocation response from SecService5:operation5: Response from Service5:operation5
11:49:49> Command: i SecService6 operation6                 # unavailable operation
11:49:49> Invocation operation failed. The service/operation might not exist or is temporally unavailable.
11:49:50> Command: i SecService8 operation1                 # unavailable service
11:49:50> Invocation response from SecService8:operation1: Unavailable service or operation.
11:49:51> Command: i SecService7 operation5                 # no rights to access this service
11:49:51> Invocation response from SecService7:operation5: No rights to access this service
11:49:52> Command: g SecService7 operation1 120             # success
11:49:52> GrantAccess response for SecService7:operation1: OK
11:49:53> Command: i SecService7 operation1                 # success
11:49:53> Invocation response from SecService7:operation1: Response from Service7:operation1
11:49:54> Command: i SecService7 operation2                 # no rights to access this operation
11:49:54> Invocation response from SecService7:operation2: No rights to access this service
11:49:55> Command: g SecService7 all 120                    # success
11:49:55> GrantAccess response for SecService7:all: OK
11:49:56> Command: i SecService7 operation3                 # success
11:49:56> Invocation response from SecService7:operation3: Response from Service7:operation3
11:49:57> Command: g SecService8 all 120                    # unavailable service
11:49:57> GrantAccess response for SecService8:all: The requested service does not exist.
11:49:58> Command: o SecService7 operation1 operation2      # unavailable service name
11:49:58> GrantOwnerRights response for SecService7: A service with this name already exists.
11:49:59> Command: o SecService8 operation1 operation2      # success
11:49:59> GrantOwnerRights response for SecService8: OK
11:50:00> Command: i SecService8 operation1                 # success
11:50:00> Invocation response from SecService8:operation1: Response from Service8:operation1
11:50:01> Command: i SecService8 operation3                 # unavailable operation
11:50:01> Invocation response from SecService8:operation3: Unavailable service or operation.
11:50:02> Command: d user4 SecService8 operation1 120       # inexistent user
11:50:02> DelegateRights response for user4: The user is not registered.
11:50:03> Command: d user1 SecService9 operation1 120       # unavailable service
11:50:03> DelegateRights response for user1: The requested service does not exist.
11:50:04> Command: d user1 SecService1 operation1 1000      # time exceeds access right time limit
11:50:04> DelegateRights response for user1: No rights to access this service
11:50:05> Command: d user1 SecService8 operation1 600       # success
11:50:06> DelegateRights response for user1: OK
11:50:07> Command: s 200                                # sleep
11:51:47> l perm                                        # access to services 1-6 expired
11:51:47> GetAllServices response: OK
        SecService8
           operation1          owner
```

```
            operation2          owner
11:51:48> Command: i SecService1 operation1              # access rights expired
11:51:48> Invocation response from SecService1:operation1: No rights to access this service
11:51:49> Command: e                                     # logout
11:51:49> Logout response: OK

Check delegation for user1
11:54:01> Command: a user1 user1 10                      # check delegation
11:54:02> Authentication response: OK
11:54:03> Command: i SecService8 operation1              # success
11:54:03> Invocation response from SecService8:operation1: Response from Service8:operation1
11:54:04> Command: i SecService1 operation1              # fail
11:54:04> Invocation response from SecService1:operation1: No rights to access this service
```

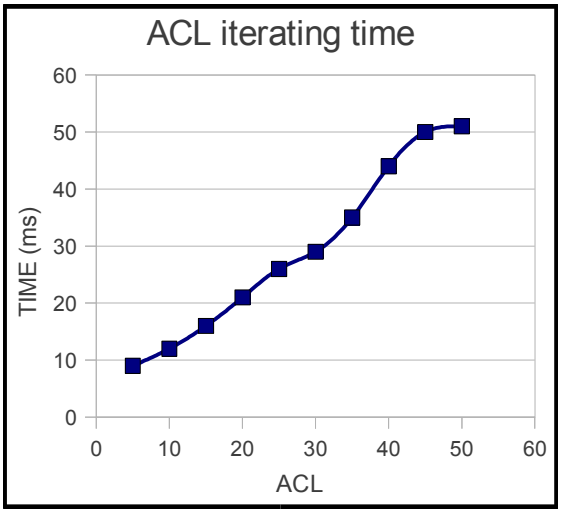
7.2 Performance testing

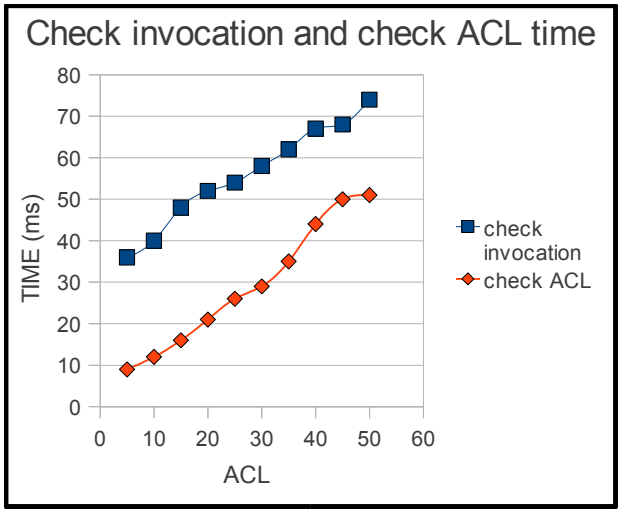
7.2.1 ACL scaling

Test setup

We have build an ACL with 50 entries and measured the ACL iterating time and the overall invocation security check time (authorization, service availability, access rights, writing to access logs) for service operations that correspond to the 5th, 10th, ... and 50th ACL entry.

Test results





Because only one ACL entry is read at a time and due to the processing time for an entry, the access rights verification time grows proportional to the ACL size, with a 1:1 ratio. However, the size of the ACL has less impact on the total time of invocation verification.

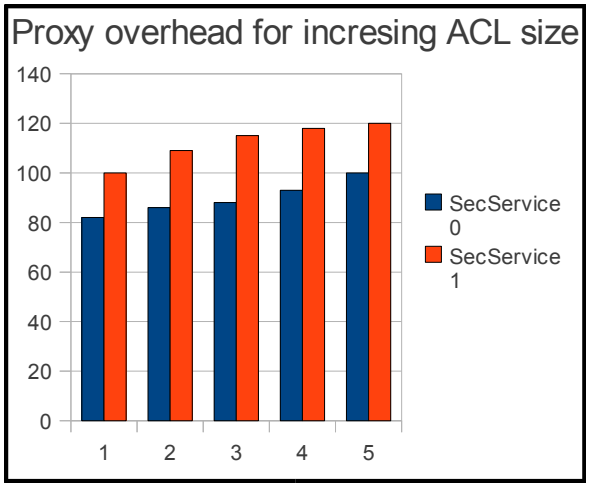
The time-ACL size ratio for service invocation is 1:5. The results are influenced by the read and write speed of BlobSeer files and by the read speed of the higher level write function, discussed in the *Implementation details* section. The best results were obtained using 4KB pages, considering that the record sizes are: 64B for status files, 128B for log files, 128B for ACL files, etc. The ACL size has insignificant impact on the overall invocation time, because of the web server response time, SOAP message processing and transport overhead.

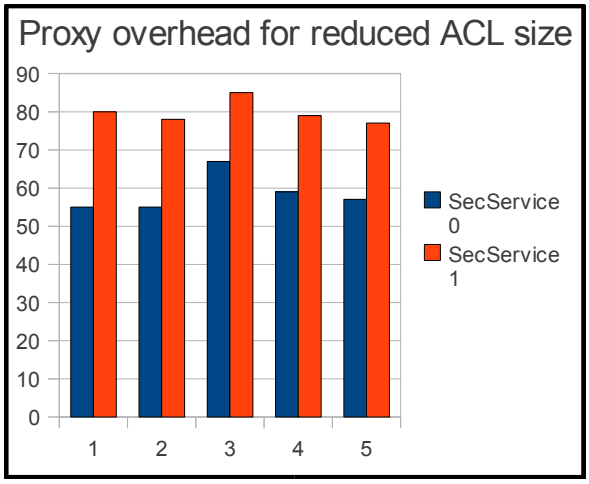
7.2.2 Proxy service overhead

Test setup

For this test, we have deployed a demo web service and corresponding proxy service (SecService1) and a secured service that embeds all the secure access logic (SecService0). In order to obtain the maximum overhead the proxy service is responsible for, the demo service is a “dummy” service that doesn't perform any additional computation. To eliminate the transport overhead, all tests have been performed on the local machine. We have performed five series of 30 invocations for both services and computed the average invocation time for a constant ACL size (small size). We have performed another series of tests for increasing ACL sizes (starting with 50 entries). The overhead percentage was similar for both cases.

Test results





SecService0	SecService 1	Overhead(%)
82	100	18
86	109	21
88	115	23
93	118	21
100	120	16
Average overhead		19.8

The 20% overhead represents the maximum theoretical overhead because we tested with dummy services that don't add to the overall computation time. When adding this security layer on top of real web services with large running times, the time needed for the security service will be the same but the overhead will be significantly lower. In conclusion, the performance loss is not significant. We have implemented a flexible and generic solution that offers secure access to any web service in a Cloud environment, by using proxy services, without adding much overhead to the system.

8. Conclusions and Future work

We developed a secure layer for web services in Cloud Environments. This novel approach offers a high degree of security through authentication, authorization and delegation.

This project has managed to integrate the BlobSeer storage system with web services. The experimental results have concluded that BlobSeer is very efficient in allowing transparent fine-grain access to massive security related data, as access to BLOBs data doesn't effect the system's performance.

The authentication mechanism of our solution allows single sign-on access control, simplifying access management for both the system and its users.

The Service Manager allows secure registration and authentication that protects both the client and the system. In order to authorize access to other services after login into the system, the security manager grants users security tokens associated with an expiration date, that need to be validated by each service the users will invoke. The authentication process uses encryption in order to protect the password and the access token, while the timestamp will prevent reply attacks.

In order o allow a flexible way to deploy new services, we have separated the functional layer of a service from the secure access layer without affecting the invocation performance.

Because Clouds provide services on a pay-per-use basis, the access control system allows a fine-grain representation of the user's permission to use these services. Our project uses a representation that is based on clients capabilities to access individual operations exposed by web services for a given amount of time.

This project offers access control management for individual users. The granularity of access rights is appropriate for Cloud Software as a Service clients. The access control lists contain individual permissions for service operations with a time limit. Because Cloud clients are both individuals and organizations, a possible extension of this project would be adding group access control management.

The role based access control model is more appropriate for organizations access control. A RBAC model for Cloud organization clients should be based on the following rules: permissions are associated with roles and not users, each user will be assigned one or more roles (that can also be revoked), a user can have more than one role simultaneously, a user can change its role during a session.

This project is focused on the Software as a Service aspect of Cloud computing. The access control model allows a user to become owner of a new web service, but the deployment process is manual. In order to offer Platform as a Service functionality, the deployment process should be automatic.

BIBLIOGRAPHY

[1] B. Nicolae, G. Antoniu and L. Boug´ e, “BlobSeer: Efficient Data Management for Data-Intensive Applications Distributed at Large-Scale ”, published in “24th IEEE International Symposium on Parallel & Distributed Processing: Workshops and Phd Forum (IPDPS '10) (2010) 1-4 ”

[2] B. Nicolae, G. Antoniu and L. Boug´ e, ”Distributed Management of Massive Data: an Efficient Fine-Grain Data Access Scheme ”, October 2008

[3] B. Nicolae, G. Antoniu and L. Boug´ e, “BlobSeer: How to Enable Efficient Versioning for Large Object Storage under Heavy Access Concurrency ”, published in "2nd International Workshop on Data Management in Peer-to-peer systems (DAMAP 2009) (2009)”

[4] B. Nicolae, G. Antoniu and L. Boug´ e, “Enabling High Data Throughput in Desktop Grids Through Decentralized Data and Metadata Management: The BlobSeer Approach ”, published in "Euro-Par 2009 (2009) 404-416"

[5] B. Nicolae, G. Antoniu and L. Boug´ e, Diana Moise and Alexandra Carpen-Amarie , “BlobSeer: Next Generation Data Management for Large Scale Infrastructures ”, August 24, 2010

[6] D. Chappell and T. Jewell , “Java Web Services”, O'Reilly , First Edition March 2002 , pp. 6-7, 77, 100, 21, 231-245

[7] E. Cavanaugh , “Web services: Benefits, challenges, and a unique, visual development solution ”, Altova, 2006

[8] R. Englander , “Java and SOAP”, O'Reilly, Edition May 2002 , pp 18-33.

[9] B. Spies, http://www.ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest , posted on May 2nd, 2008

[10] J. Newmarch , “A RESTful Approach: Clean UPnP without SOAP ”, IEEE Consumer Communications and Networking Conference, 2005

[11] C. Pautasso, O. Zimmermann and F. Leymann , “RESTful Web Services vs. “Big” Web Services: Making the Right Architectural Decision ”,
[12] The Apache Software Foundation, http://axis.apache.org/axis/java/user-guide.html
[13] *** “Introduction to Cloud computing ”, ThinkGrid, 2008
[14] G. Boss, P. Malladi and all, “Cloud Computing ”, Copyright IBM Corporation 2007
[15] T. Winans and J. Brown, “ Cloud computing - A collection of working papers ”, Deloitte Development, 2009
[16] B. Khaund, http://www.codeproject.com/KB/webservices/CloudSaaS.aspx , posted on Dec 30 2009
[17] M. Bishop, “Computer security: art and science” , Pearson Education, Inc, 2003, pp 31-32.
[18] E. Simon, P. Madsen and C. Adams, “An Introduction to XML Digital Signatures” August 08, 2001
[19] D Gollman, “ Computer sercurity”, 3rd edition John Wiley & Sons, Ltd, 2011, pp. 71.
[20] S. Seely “Understanding WS-Security”, Microsoft Corporation, October 2002
[21] J. Crampton and H. Khambhammettu , “Delegation in Role-Based Access Control ”
[22] The Apache Software Foundation, http://axis.apache.org/axis2/java/core/docs/mtom-guide.html
[23] W. Khattak, “SOA Pattern (#14): Logic Centralization”, InformIT online magazine, June 2010