# Decentralized Authorization for Web Services

Paraskevas Stefas

A dissertation submitted to the University of Dublin, in partial fulfilment of the requirements for the degree of Master of Science in Computer Science

September 6, 2005

## Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university

Signed: ____________________

Paraskevas Stefas

September 6, 2005

## Permission to lend and/or copy

## Acknowledgements

I would like to thank my supervisors Professor Donal O’Mahony and especially Patroklos Argyroudis for his valuable help throughout the duration of this project and his essential comments concerning this document.

# Abstract

*Web Services is an area that encountered a very sudden and huge explosion on its popularity during the latest years. The advantages Web Services provide compared to traditional distributed computing methods such as Remote Procedure Calls and Distributed Objects were found to be very appealing by researchers and users and this lead to a huge development effort resulting in the creation of many standards and specifications in a rather short period of time. This fact made a lot of security experts to believe that the supporters of Web Services had not given the appropriate consideration on the security aspects of their use. Many analysts showed that the security risks were numerous from such a rapid development without the appropriate attention on security. The response from the Web Services community was the creation of many Workgroups that concentrated their efforts in providing common security mechanisms to Web Services.*

*This dissertation investigates a different approach to Web Services security. We focus on distributed authorization and access control for the development of a security system for Web Services. The result is WebÆTHER which is a trust management system for Web Services based on the ÆTHER system that was originally designed for use in a ubiquitous environment. The system is relies on current technologies for the implementation of the Web Service used and is composed by the Web Service and the extra layer that provides the ÆTHER core functionality. This approach makes the evaluation of this effort easy and comparable to current solutions in the field of Web Services security.*

*The implemented Web Service is a file sharing application. The implementation results are 2014.54 milliseconds average time for the transfer of a 10.4 megabytes file that was used in the Web Service and the average overhead from the inference engine was found to be 6%.*

# Table of Contents

## Table of Figures

# 1 Introduction

## 1.1 Motivation

Web Services is an area that encountered a very sudden and huge explosion on its popularity during the latest years. The advantages Web Services provide compared to traditional distributed computing methods such as Remote Procedure Calls and Distributed Objects were found to be very appealing by researchers and users and this lead to a huge development effort resulting in the creation of many standards and specifications in a rather short period of time.

This fact made a lot of security experts to believe that the supporters of Web Services had not given the appropriate consideration on the security aspects of their use and many analysts showed that the security risks were numerous from such a rapid development without the appropriate attention on security.

The result was the creation of many work groups and a great effort in order to create standards concerning the security of web services along with the evolution of new protocols used especially for security purposes. Because of the fact that Web Services had advanced further than their security, most of the work groups concentrated their efforts to give Web Services the security assurances that traditional security mechanisms provide.

The highly distributed environment that Web Services consist also made many researchers sceptical about the effectiveness of traditional security mechanisms. Many of them found that traditional security mechanisms don’t always apply correctly on this environment and especially as the systems scale the security management often becomes impossible. The proposed solution to the problem of scale in distributed systems security management was a series of trust management systems that provided distributed authorization and access control. This introduces the challenge of investigating the application of these systems in the Web Services area.

## 1.2 Proposed Goals

The main goal of this dissertation project is the design and implementation of a trust management system in the area of Web Services. The system is going to provide access control for a Web Service while being able to handle a large scale application effectively. In order to achieve the main goal of this project, the following parts have to be achieved:

- In depth understanding of the current efforts and technologies on Web Services security.
- Investigating how appropriate is the application of a trust management system on web services compared to traditional security mechanisms.
- The definition of a trust management model that is appropriate for use in Web Services.
- To measure the performance of the implemented system in a real world usage scenario and locate possible issues.

## 1.3 Document Overview

The rest of the document is organized as follows.

- Chapter 2 is consisted of two parts with background information. The first part of the chapter provides information on security concepts and mechanisms. The second part of the chapter presents web services and the most common technologies they use and finally the current efforts on web services security.
- Chapter 3 presents the state of the art on distributed authorization and access control and analyzes the features of two currently available systems.
- Chapter 4 the design of WebÆTHER system is analysed.
- Chapter 5 provides the implementation details of both the clients and servers used in the WebÆTHER system for a specific Web Service that was chosen.

- Chapter 6 presents the evaluation of the WebÆTHER system. The metrics that were used in measurements are presented along with diagrams to illustrate the performance of the system's components.
- Chapter 7 is the final chapter that contains the conclusions and propositions for future work.

# 2 Background

## 2.1 Access control

By the term Access Control we mean the set of policies and mechanisms that restrict or allow access on a set of resources. Access Control is composed by two separate processes which are Authentication (determining the identity of a user by using enough provided proof) and Authorization (allowing access to the appropriate resources for the specific user).

### 2.1.1 Authentication

Authentication is the process of a user proving to a system his identity. Authentication can be performed by using a knowledge that only the user and the system share (for example a username-password pair) or by using something only a specific user has and is either physical (fingerprints, voice pattern etc) or technical (for example a smartcard). The most commonly used traditional Authentication mechanism is password systems, while other common mechanisms are Kerberos [1], Certificates in SSL connections, Biometrics [2] and other. Authentication mechanisms have nothing to do with access rights of a user and only ensure that the user is the one he claims to be.

### 2.1.2 Authorization

The second process in access control is Authorization. Authorization is the process of determining whether or not a user has access in a specific resource and what kind of access that is based on his identity that was proven in the Authentication process. The most common Authorization mechanism is Access Control Lists (ACLs). ACLs are usually tables where columns are resources and rows are the privileges for every user on a specific resource. Using ACLs and user groups, specific security policies are able to be implemented. Figure 2-1 illustrates an authorization mechanism example.

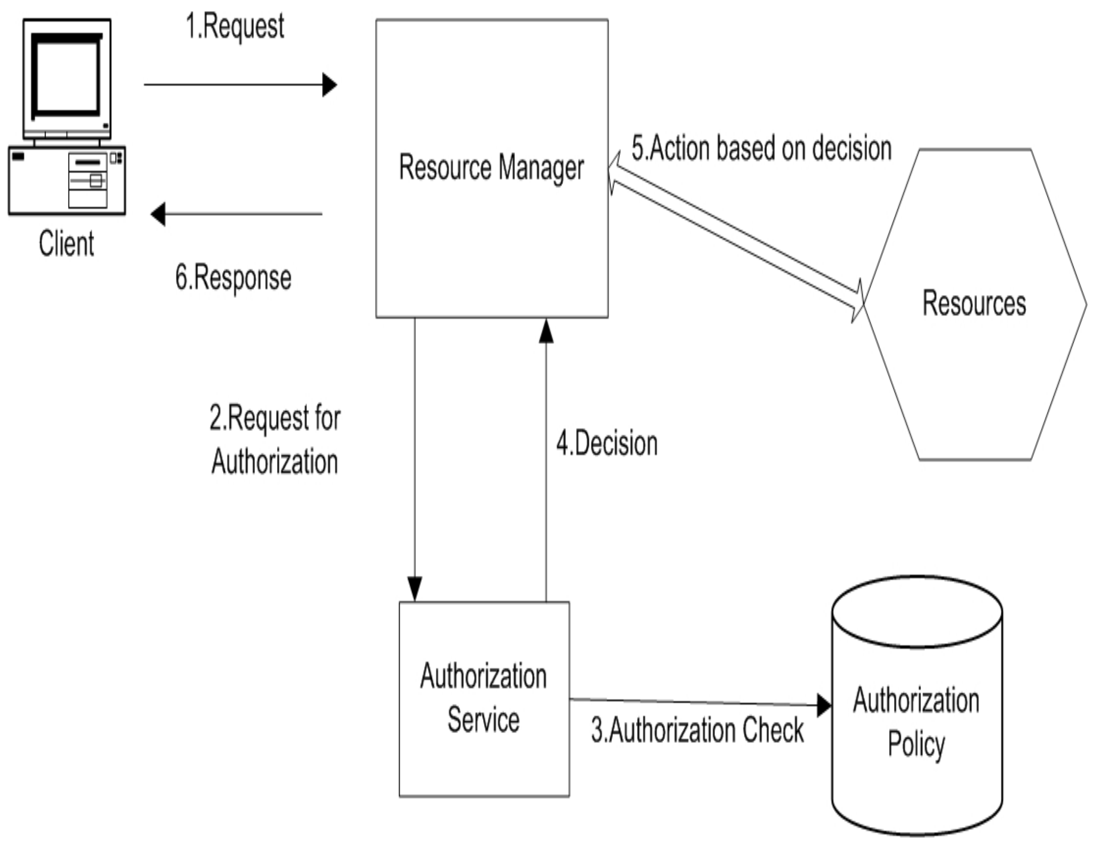


**Figure 2-1: Authorization Example**

## 2.2 Public Key Cryptography Access Control

Public key cryptography which was invented by Whitfield Diffie and Martin Hellman in the late 1970’s allowed among others the evolution of digital signatures. Public key cryptography is based on cryptographic keys coming in pairs, a public and a private key. The public key is distributed to many people and allows the decryption of messages that are encrypted with the private key which is kept secret. This way it is possible to make sure of the sender’s identity.

*Digital signatures* are small checksums of a message encrypted with the sender’s private key which are then verified using the sender’s public key by the recipient.

This way the recipient can make sure that the original message was not modified during transmission.
A *digital certificate* is the attachment of a digital signature to an electronic message for security purposes such as verifying the sender’s identity. A *public key certificate* is a specific kind of digital certificate that binds a public key with an identity and can be used to verify that a public key belongs to an individual.

### 2.2.1 PKI

PKI (Public Key Infrastructure) which is described in X.509 standard [3] provides a framework to verify the identities of each entity in a given domain. It uses public key certificates [4], which are data structures binding a name with a public key that are digitally signed by a trusted third party. The framework includes the requesting, issuing, signing and validating the public key certificates. The third party’s signature can be verified by every principal in the system and a trusted third party is called Certificate Authority (CA).
An X509 public key certificate contains the following information and is also digitally signed by the issuer in order to provide integrity:

- *Version*: the version number of the certificate.
- *Serial number*: a unique integer issued by the CA for every certificate.
- *Signature*: identifier for the algorithm and the hash function used by the CA in signing the certificate.
- *Issuer*: the entity that issued the certificate.
- *Validity*: a time interval in which the CA warrants for the validity of the certificate.
- *Subject*: the entity associated with the public key found in the certificate.
- *Subject public key info*: the subject public key and the algorithm which this key is instance of.
- *Issuer unique identifier*: used to uniquely identify the issuer in case of name reuse.

- *Subject unique identifier*: used to uniquely identify the subject in case of name reuse.
- *Extensions*: allows extra fields in the data structure
- *Certificate Signature Algorithm*: the algorithm that was used to sign the certificate.
- *Certificate Signature*: the digital signature.

The public-private key pair can be generated by the user, a third party or the CA. The creation and validation of public key certificates can only be done on a CA according to X.509.

The public key certificates can expire or be revoked by the CA prior to the expiration date. In order for the relying parties to be informed of revocations a mechanism called Certification Revocation List (CRL) is used. CRL is a data structure containing a list of revoked certificate serial numbers that is time-stamped and digitally signed by the Certification Authority.

According to PKI standards there are two types of certificates, *user certificates* and *CA certificates*. A user certificate is issued to a subject that is not an issuer itself and CA certificates are issued by CAs to subjects that are also CAs. The latter kind of certificate is called *cross certificate* and the lists of cross certificates are called certification paths. The decision if a CA is going to be trusted can be done by examining its certificate. The chain ends at the root certificate that is issued by the *Root CA* that can be considered as the ultimate certification authority and root certificates are implicitly trusted and this trust model is called Hierarchical.

This model is illustrated in figure 2-2. Trust extends from Root CAs to CAs and then to users. We can consider this as a tree where the leaf certificate's validity is verified by tracing backward from its certifier to other certifiers until a directly trusted root certificate is found.

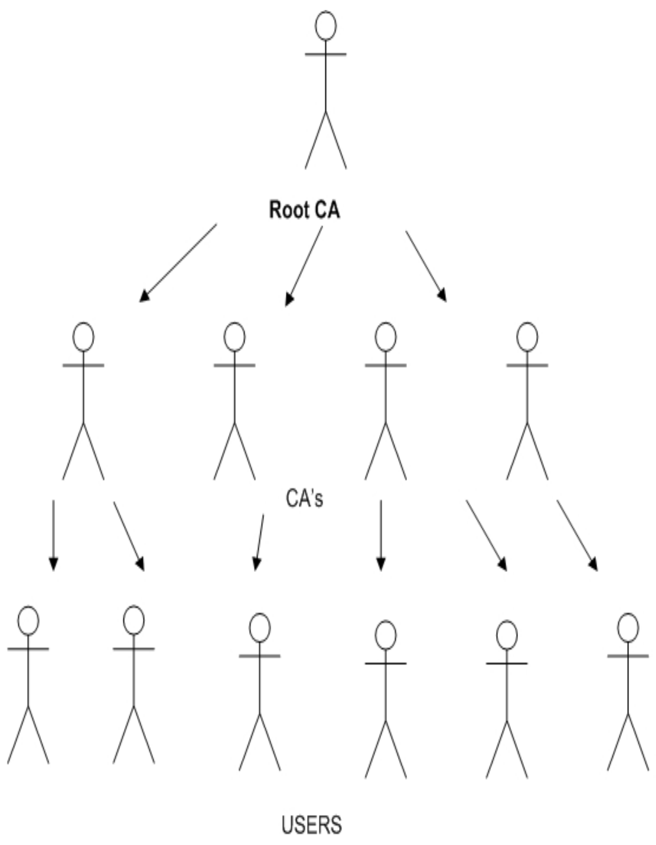


**Figure 2-2: Hierarchical Trust Model**

### 2.2.2 PMI

PMI (Privilege Management Infrastructure) [5] provides a framework to determine if an entity with a public key certificate is authorized to access a specific resource. PMI includes the issuing and validation of attribute certificates. Attribute certificates are digitally signed data structures which bind privileges to entities. Attribute Certificates are issued by Attribute Authorities (AA). Privilege Management Infrastructure can be established and managed independently from a PKI since it does not provide a mechanism to trust certificates' holders. Instead, PKI is used to authenticate issuers and holders of attribute certificates.

An attribute certificate contains the following information and is also digitally signed by the issuer in order to provide integrity:

- *Version*: the version number of certificate.
- *Holder*: the identity of the attribute certificate 's holder
- *Issuer*: the identity of the AA that issued the certificate
- *Signature*: the cryptographic algorithm used to digitally sign the attribute certificate.
- *Serial number*: the serial number that uniquely identifies the attribute certificate within the scope of its issuer.
- *Validity*: the time period during which the attribute certificate is considered valid.
- *Attributes*: the attributes for the holder that are being certified.
- *Issuer unique ID*: used to uniquely identify the issuer where the issuer component is not sufficient.
- *Extensions*: allows extra fields in the data structure.
- *Attribute Certificate Signature Algorithm*: the algorithm that was used to sign the attribute certificate.
- *Attribute Certificate Signature*: the digital signature.

In PMI Attribute Authorities (AAs) and Certification Authorities (CAs) are independent. There is an entity called Source of Authority (SOA) which is itself an Attribute Authority and, in analogy to the root CA in PKI, is the entity that is trusted by a privilege verifier as the ultimate entity responsible for privilege assignment. A SOA can issue certificates to other entities that can also act as AAs and further delegate the privilege. This feature is called privilege delegation and may continue through several intermediate AAs.

The authentication model used in PKI and authorization model used in PMI can be viewed in figures 2-1 and 2-2.

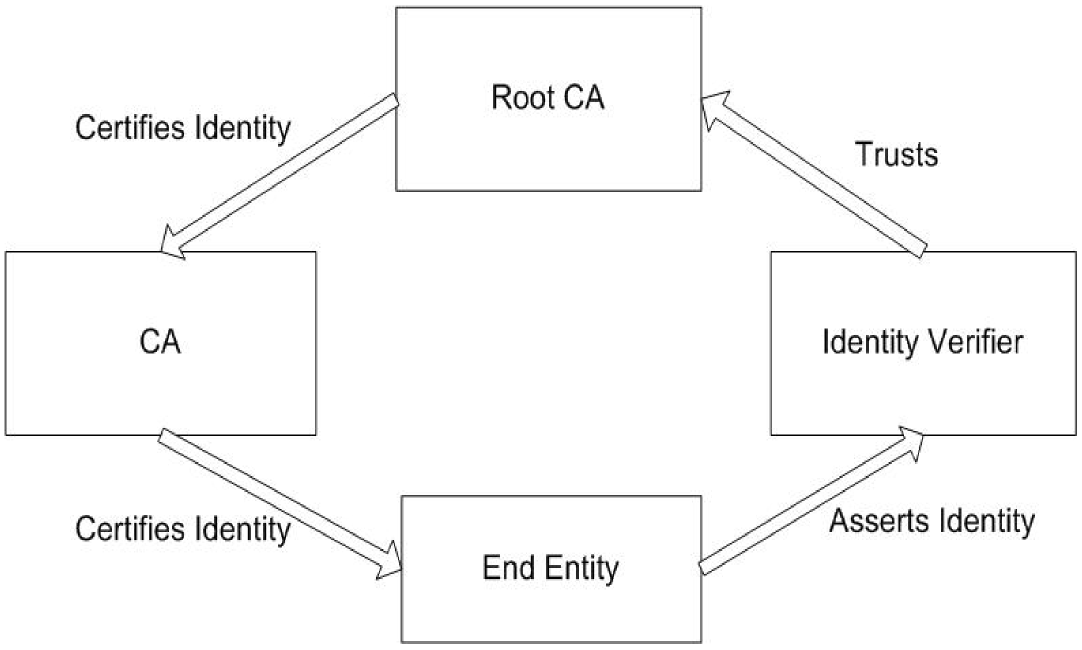


**Figure 2-3: PKI model**

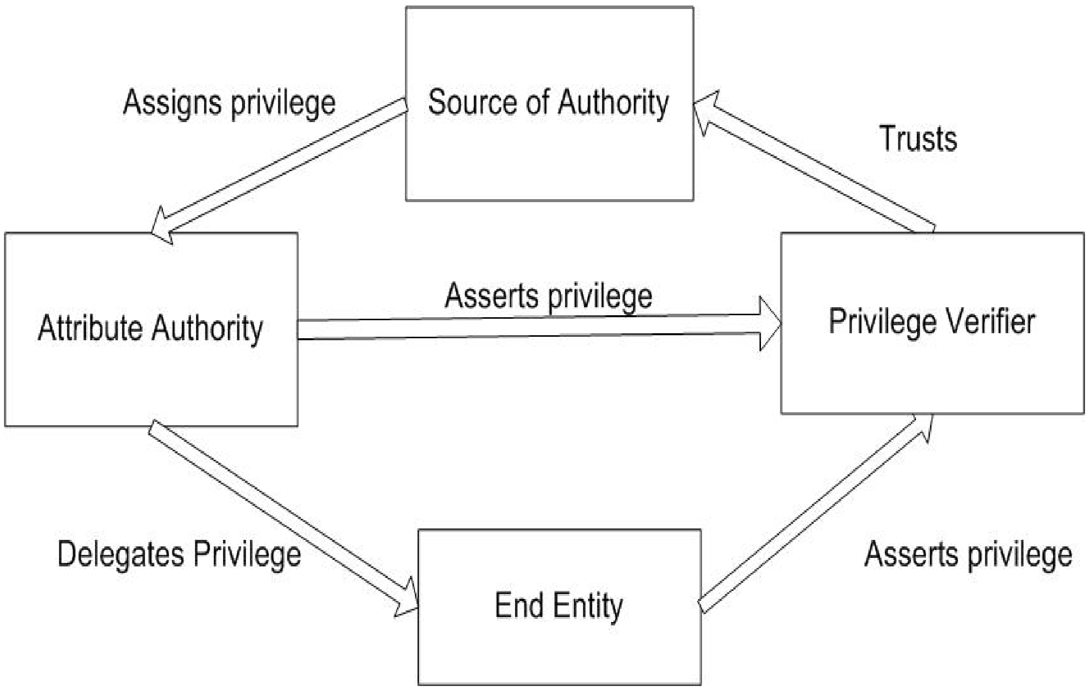


**Figure 2-4: PMI model**

## 2.3 PGP

PGP [6] is a cryptographic system used for authentication and encryption that was designed and developed by Phil Zimmermann in 1991. It uses public key cryptography and symmetric key cryptography. In PGP a sender uses a recipient's public key to encrypt a shared secret key that is going to be used to encrypt the plaintext message with a symmetric cipher algorithm. In order to ensure that a message was not altered during transmission, PGP uses digital signatures. The users' public keys are stored in special data structures called key rings. There are public and secret key rings, depending on the type of keys they include. Every PGP user will eventually add in his public key ring the public keys of his recipients. To prevent man-in-the-middle attacks PGP uses digital certificates and for the distribution of certificates storage repositories called certificate servers. Also PKI systems can be used. Both X.509 and PGP certificates are recognized by PGP.
PGP certificates have the following structure:

- *PGP version number*
- *Certificate holder's public key*: the public key together with the algorithm that was used
- *Certificate holder's information*: information about the user
- *Digital signature of the certificate owner*: also called a self-signature, this is the signature using the corresponding private key of the public key associated with the certificate.
- *Certificate's validity period*: the time interval in which the certificate is valid.
- *Preferred symmetric encryption algorithm for the key*: indicates the encryption algorithm to which the certificate owner prefers to have information encrypted.

With PGP certificates anyone is able to do the validation because of the self-signature field but a hierarchical structure of CAs is also supported. A unique aspect of the PGP certificate format is the ability to contain multiple signatures in a single PGP certificate. This allows a trust model (a way of establishing certificate validity) called *web of trust* to be used. The idea is that at first everyone signs his friends' and people he trusts PGP certificates. Next, a user checks if a PGP certificate of an unknown user is signed by people he trusts and if this is true he also signs the certificate and trusts the holder.

For implementing the trust model, PGP includes levels of trust. The highest trust level is implicit trust, which is the trust you have in your own key pair. All keys signed by your implicit trusted key are considered automatically valid. There are also the following levels of trust that you can assign to another person's public key:

- Complete trust
- Marginal trust
- No trust

Along with trust levels there are three levels of validity which are:

- Valid
- Marginally valid
- Invalid

To define another person's key as trusted it must be a valid key signed by you or signed by another trusted entity and then you define the level of trust that you feel the key is entitled. PGP requires one completely trusted signature or two marginally trusted signatures to establish a key as valid. PGP is most commonly used for e-mail but it can be used for encryption of any kind of data and files. The PGP design was made an Internet standards track specification called OpenPGP [7] which is now an open standard. In Figure 2-5 there is an example for web of trust model. Bob marginally trusts Alice and John and doesn't trust Helen. Andy's PGP key is signed from both Alice and John, so Bob considers it valid and can assign a level of trust. Jack's PGP key is signed by Helen who is un-trusted and Alice who is marginally trusted, so his PGP key is considered invalid by Bob.

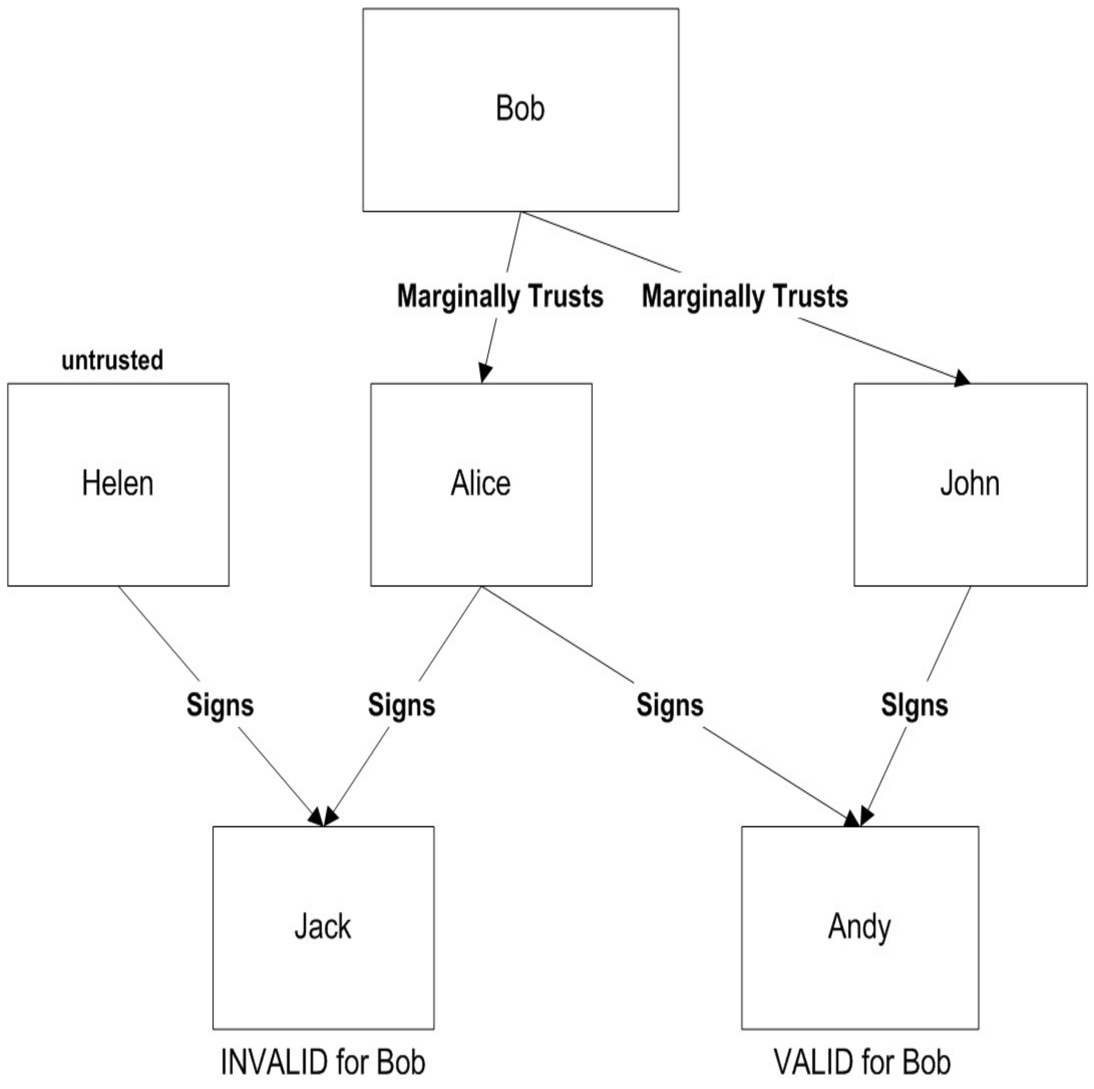


**Figure 2-5: Web of trust example**

## 2.4 Role Based Access Control

Role Based Access Control [8] is an approach for restricting access to authorized users that was evolved by the traditional types of access control. Traditional types of access control as defined in Trusted Computer System Evaluation Criteria [9] are Discretionary Access Control (DAC) for commercial systems and Mandatory Access Control (MAC) for military systems.

In Discretionary Access Control, users may allow or deny other users (subjects) permissions to specific resources (objects) by using the identity of subjects and/or groups they belong. That means that in DAC, a subject that owns an object is able to decide to whom it gives access on the object.

In Mandatory Access Control, permissions are controlled by the system based on the sensitivity of the information contained in objects and the clearance of subjects to access information of that sensitivity. That means that a subject cannot give access on an object to a subject that is not qualified by the system.

In RBAC, the main idea is that in an organisation there are roles created for various job functions and permissions are assigned to specific roles instead of the users themselves. The users can acquire permissions by acquiring roles and management of individual user rights becomes management of specific roles (Figure 2-6). Only the administrator can assign roles on users and users cannot pass access permissions on to other users in their discretion. That means that RBAC is a form of MAC but is not based in multilevel security requirements.

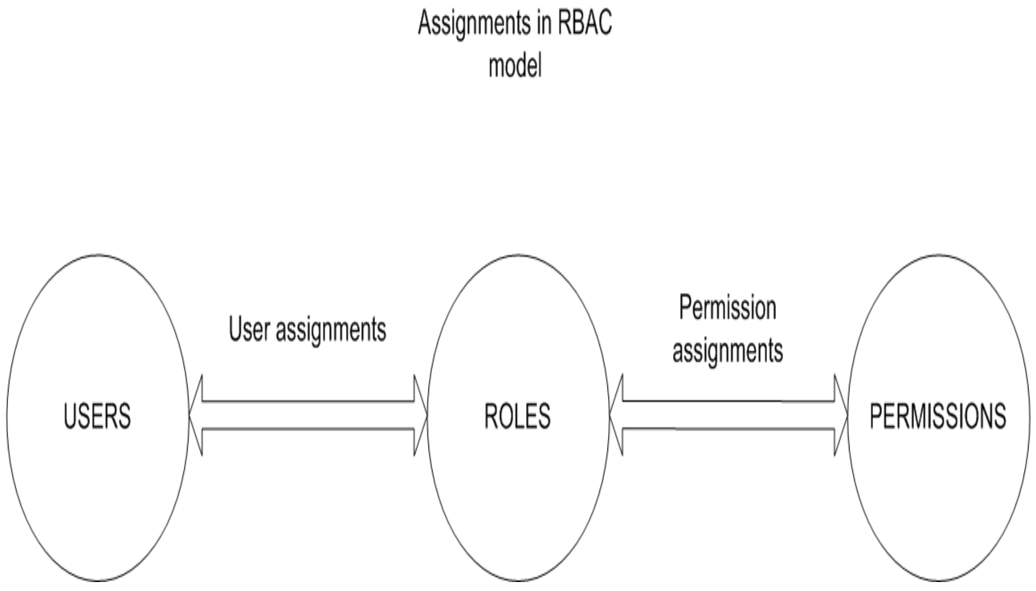


**Figure 2-6: RBAC model**

In order for RBAC to be implemented correctly, a user must not be given any more privilege that is needed to perform the given job. This concept is called *least privilege* and requires the identification of user’s job functions determining the minimum set of privileges required to perform those functions.

RBAC allows the separation of duty principle implementation. According to separation of duty the possibilities of fraud are minimized when collaboration between job-related capabilities in critical business functions do not exist. That means no single user can be allowed to perform all the operations associated with a business function. For example, a company might have the financial officer role and the payroll clerk role. These two roles cannot be assigned to the same user at the same time.

## 2.5 Web Services

Web Services are the result of Distributed Computing evolution. Starting from Inter Process Communication, continuing to Remote Procedure Calls and evolving to Distributed Objects, along with the evolution of XML to ensure data portability, Web Services emerged. The evolution of the above technologies made possible to provide self-contained, self describing, modular applications that can be published, located and invoked across the Internet that are called Web Services.

Web Services provide a way of interaction that is called Service Oriented Computing [10]. The basic steps are publishing the services on registries that are available for consumers to search, find the service and invoke it. In order to achieve this, Web Services must be described using a contract definition language (such as Web Service Description Language - WSDL), be published and located in a standardized way (such as Universal Description Discovery and Integration - UDDI),  encode and invoke the application data (such as XML and Simple Object Access Protocol - SOAP over HTTP/SMTP).

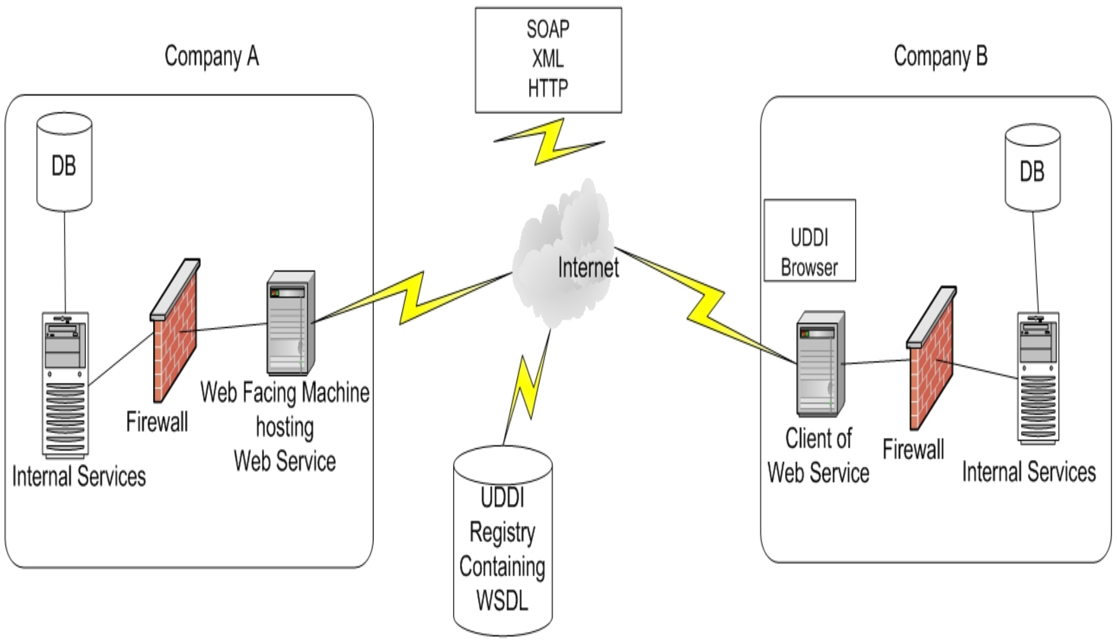


**Figure 2-7: Web Services Example**

### 2.5.1 Web Services standards

The standards and protocols used to implement a web service are considered as the web service protocol stack [10]. The leading organizations in the development of web services standards are OASIS and W3C. Also the Web Services Interoperability (WS-I) organization is promoting interoperability between vendor solutions by developing a series of profiles that refine the different standards and their use in order for better integration to be achieved between implementations. The web service protocol stack is composed of four main areas:

1. *Service Transport*: this area is responsible for message transport between networks. Popular protocols such as HTTP, SMTP and FTP are being used in service transport. The reason is that these protocols can bypass firewalls without requiring changes to firewall rules.
2. *Messaging*: this area is responsible for encoding messages in a XML format which makes messages understandable at both ends of the network connection. Well known protocols in this area are XML-RPC, SOAP and REST.
3. *Service Description*: used for describing the interface to a specific web service. WSDL is the definition language typically used for this purpose.
4. *Service Discovery*: the role of this area is the creation of a centralized common registry for the web services to publish their location and description. At present, the UDDI protocol is used for service discovery.

#### 2.5.1.1 SOAP

SOAP is the most popular standard for exchanging XML based messages in web services usually using HTTP. The SOAP specification is maintained by the XML Protocol Working Group of the World Wide Web Consortium [11]. SOAP provides a messaging framework that more abstract layers can build on. A SOAP message is contained in an envelope and has two sections, the header (where relevant information about the message can be contained) and the body which contains the payload. An example of a client SOAP message requesting information about a fictional product is the following:

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getDetails xmlns="http://company.example.com/ws">
       <productNAME>productA</productNAME>
     </getDetails>
   </soap:Body>
 </soap:Envelope>
```

An example response can be:

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getDetailsResponse xmlns=" http://company.example.com/ws ">
       <getDetailsResult>
         <productNAME> productA </productNAME>
         <description>An example product for a sample
      message.</description>
         <price>96.50</price>
       </getDetailsResult>
     </getDetailsResponse>
   </soap:Body>
 </soap:Envelope>
```

#### 2.5.1.2 REST

An alternative standard for messaging is Representational State Transfer (REST). REST is referred to a collection of architectural principles for distributed hypermedia systems and can be used to describe simple web based interfaces that use XML and HTTP without the message exchange patterns based approaches that are used in SOAP [12]. In REST clients work with the standard HTTP operations GET, POST, PUT and DELETE on resources that are represented with XML records on specific locations.
An example REST application could have the following record:

```
<user>
 <name>John Doe</name>
 <age>30</age>
<gender>male</gender>
 <location
href="http://www.example.org/locations/new_york_city.xml">New York
City </location>
</user>
```

To update a user’s location, a REST client has to download the above record using HTTP GET, then modify it to change the location and upload it using PUT. SOAP and other applications using the RPC approach focus on operations that are represented as verbs (for example *getDetails)*. REST in the other hand focuses on the diversity of resources which are represented as nouns (for example *user*). This means that because HTTP has not any LIST or FIND operations or any other method of resource discovery, REST data applications must create sets of search results as another type of resource, requiring users to know additional URLs for listing or searching each type of resource.

## 2.5.2 Security in Web Services

Web Services are based on XML which by itself provides no security mechanisms at the messaging layer and relies on transport layer security provided by SSL. However SSL has some serious drawbacks when used for securing web services which are due to the fact that SSL is all or nothing because it doesn’t allow the users to provide different levels of security in the various parts of the XML document. SSL also encounters problems in complex, high volume transactions because the systems must decrypt data every time they arrive in a new web server and then encrypt in order to transmit to the next web server. These facts have lead to the creation of various XML security standards, in order to be able to develop confidentiality, integrity and access control mechanisms in the messaging layer.

### 2.5.2.1 XML Encryption & Signatures

Encryption is the basic way of providing confidentiality. An XML document may be encrypted completely as with SSL but many times each part of the document must be treated differently. For example credit card information contained in a specific part of a document shouldn’t be allowed to be viewed by any other party than a specific bank. In addition to this, some already encrypted parts might need to be encrypted again to provide a wider context and issues arise when the different parts must be signed to ensure sender authentication.

XML encryption defines a vocabulary and processing rules in the form of XML tags that enclose the parts of data being encrypted, for protecting confidentiality of whole or parts of XML documents and non XML data. The idea is to place the element tag `EncryptedData` whenever and as often is needed.

In analogy, XML signature describes a specific XML syntax for the representation of associations between cryptographic signatures and XML documents. The Signature tag encloses the whole document, followed by `SignedInfo` and `SignatureMethod` nested tags to describe information and the algorithm used to sign the message. The XML Signature concept is a joint effort of W3C and IETF and has been standardized [13]. XML Encryption is in recommendation status [14].

#### 2.5.2.2 SAML

Security Assertion Markup Language (SAML) is an XML based framework for the exchange of authorization and authentication information. This is useful in applications that do not share the same authentication and authorization infrastructure and gives the ability to create single sign on systems between different platforms. SAML statements include information referring to authentication, authorization and attributes such as credentials and group memberships and generally describe the results of previous authentication actions. The use of SAML requires the participants to be trusted by each other but it doesn’t include any mechanisms to establish trust. For this it relies to other mechanisms such as XML signature and XML encryption [15].

#### 2.5.2.3 XACML

Extensible Access Control Markup Language (XACML) is an XML specification for expressing access policies in XML documents. The XACML specification defines ways to encode rules and policies by using access control lists composed by:

- Subject
- Target object
- Permitted Action
- Provision

Subject can be user ID, group or role name. Target object allows granularity down to a single XML document element. Permitted Action can read, write, create or delete. Provisions are actions that must be executed upon a rule's activation. Such action may be for example initiating login. XACML basically defines a language that can formulate such actions [16].

#### 2.5.2.4 XKMS

XML Key Management Specification is a framework for supporting basic PKI functions for Web Services. XKMS basically defines an interface written in XML that is usually developed by PKI providers and allows the use of already present PKI infrastructure by users. The framework includes:

- Registration: XKMS services can be used to register key pairs. The generation of the keys can be performed by the client or the registration service. After registration the service manages the revocation and recovery of the registered keys. Some additional functions are reissue, revoke and recover.
- Locating: This service is used in order to retrieve a public key that is registered with an XKMS-compliant service. The public key can be then used for signature verification and encryption.
- Validation: Validate service is used to ensure that a registered public key is valid and has not expired or been revoked.

The main advantage of XKMS is that it hides the complexity of a PKI from the users. The users only need to know how to access the appropriate interface and the whole PKI functionality resides into server-side components. XKMS is currently a recommendation by the W3C XKMS working group [17].

### 2.5.2.5 WS-Security

Web Services Security (WS-Security) is a standard that extends SOAP in order to provide message confidentiality, integrity and single message authentication [18]. This standard makes use of XML signatures and XML encryption specification and defines the use of digital signatures, encryption and message digests in a SOAP message. It doesn't provide a complete security framework but is rather a set of basic mechanisms that may be incorporated into other security concepts or protocols like SSL and Kerberos. In order for users to be able to use a wide variety of security models, the following components have been implemented by the WS-Security group:

- tokens for authentication and authorization
- trust domains
- encryption technologies
- end-to-end security in the messaging layer

WS-Security defines a vocabulary and processing rules at the form of additional XML tags included in a SOAP message header. This standard's main purpose is to allow end to end security by using XML signatures and encryption in SOAP, rather than point to point security provided by SSL.

### 2.5.2.6 WS-Policy

Web Services Policy Framework (WS-Policy) is a framework that defines the ways of defining policies in web services that can be advertised and be used by security mechanisms. The framework so far supports only how the policies should look like by providing a set of XML structured elements that indicate how a web service will express the associated policy. These elements enclose the policy assertion for specific security aspects such as which type of authentication is needed [19].

#### 2.5.2.7 WS-Trust

Web Services Trust Language (WS-Trust) is a specification that is based on WS-Security and defines additional primitives and extensions for issuing, exchanging and validating security tokens [20]. In other words it defines extensions to WS-Security in order to be able to establish and access trust relationships between communicating parties.

In order for this to work, WS-Trust introduces two additional entities besides client and server which are a SOAP gateway and a *security token service* (STS). When the client sends a request to the service, it will be processed by the SOAP gateway which will extract the security token which is based on XML Signature and Encryption that the client included. Then it will issue a request to the STS to validate the original token and generate a new one valid for the service that will be sent back to SOAP gateway and will be incorporated in the original SOAP message before it is forwarded to the service itself.

#### 2.5.2.8 Discussion

The above standards and specifications are not the only ones being in development. There are many other proposed specifications in order to address other issues associated with Web Services security such as WS-Privacy, WS-Secure Conversation, WS-Federation and WS- Authorization and also other XML based languages for the implementation like XrML for digital rights management. A common place in all the above is the use of SOAP for the messaging transport layer.

In figure 2-8 an overview of the web services security standards, protocols and mechanisms is illustrated.

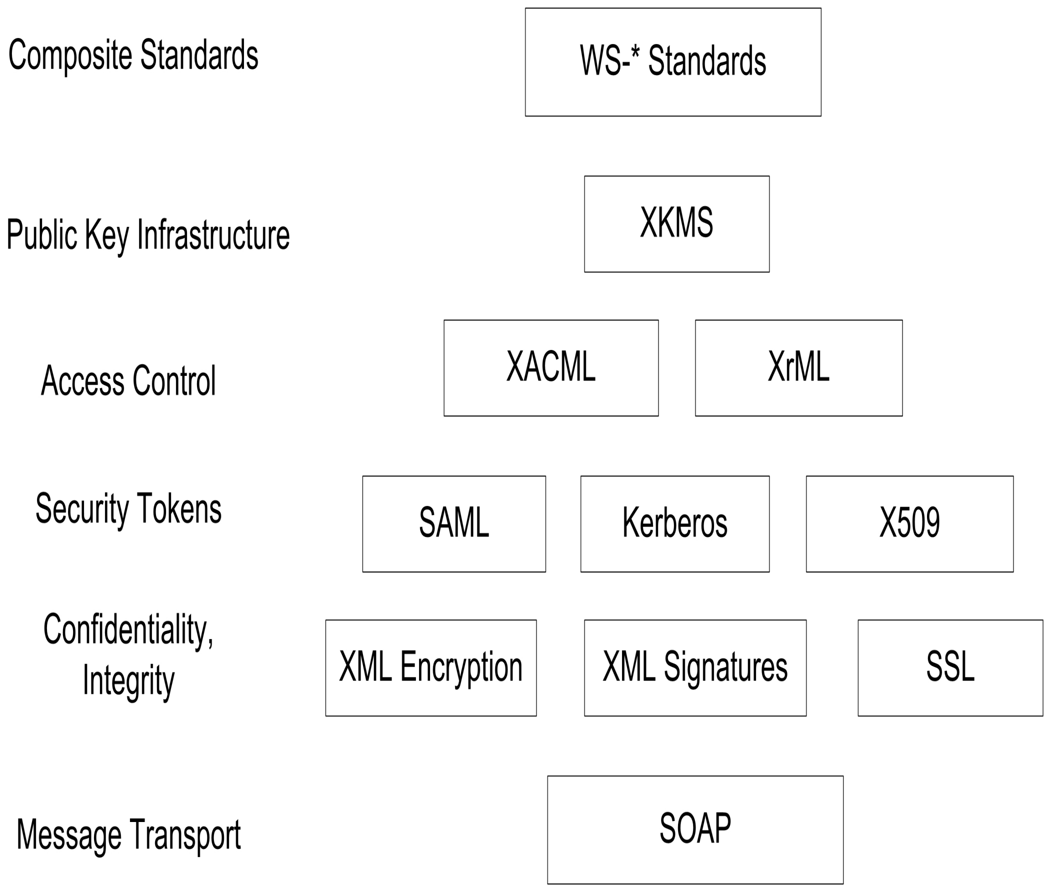


**Figure 2-8: Web Services Security overview**

## 2.6 Summary

In this chapter the traditional security mechanisms were presented along with general web services technologies and standards and finally the area of security in web services was presented, showing the directions that the researchers are following nowadays.

The proposed solutions in Web Services security are concentrated in providing the traditional security mechanisms in this new area of computing. It is easy to realise that the major effort is given on the creation and adoption of standards in order to promote platform independent solutions that would also help interoperability while providing the security they are intended to. The area of Web Services is rather new and many of the proposed solutions haven’t been tested in real world scenarios and others haven’t been implemented at all. Thus, the selection of the protocols and/or the security scheme that is going to be used in a real world Web Service should be carefully and thoroughly considered.

# 3 Distributed Authorization & Access Control

## 3.1 Introduction

Traditional authorization mechanisms like the ones described in chapter 2 are identity-based, which means that they bind a user’s identity to a public key in order to build trust and authorize service requests. X.509 and XKMS which are proposed in order to implement X.509 in Web Services use these identity bindings to public keys and authentication is employed using a centralized architecture.

However, in a highly distributed environment the users are not known beforehand and an identity could have no meaning on authorization decisions because naming becomes very complex and locally scoped.

In RBAC, role assignment is managed in a centralized way which means that there is a single point of failure/attack and performance issues arise as the systems scale and more participants with different roles are added. X.509’s centralized architecture has the same drawbacks in large scale applications.

Trust management, first defined in PolicyMaker [21], is an approach that addresses the problem of decentralized authorization by using bindings of access rights to public keys instead of identities. These authorization certificates are used to delegate permissions directly from the key of the issuer to the key of the subject enabling access control decisions between strangers without the need for a universally trusted third party.

The five key components of a trust management system are:

- A language for describing `actions': actions are operations that are to be controlled by the system and have security consequences.
- A mechanism for identifying `principals': principals are entities that can be authorized to perform actions.

- A language for specifying application `policies': policies are rules containing the actions that principals are authorized to perform and this language defines them.
- A language for specifying `credentials': this language allows principals to delegate authorization to other principals.
- A `compliance checker': it provides a service to applications for determining how an action requested by principals should be handled, given a policy and a set of credentials.

Trust management approach has many advantages when security policy is distributed over a network and helps avoid the need for application-specific distributed policy configuration mechanisms, access control lists, and certificate parsers and interpreters.

## 3.2 KeyNote

KeyNote [22] is a trust-management system designed to work in a variety of large and small scale Internet-based applications. It provides a single, unified language for both local policies and credentials. KeyNote policies and credentials are called `assertions' and describe the trusted actions permitted by the holders of specific public keys like Trust management approach defines. There are two kinds of assertions, *policy* and *credential*. Credential assertions can be sent over an untrusted network and also serve the role of certificates. They have the same syntax as policy assertions but are also signed by the principal delegating the trust.

The basic service provided by KeyNote is compliance checking which means checking whether a proposed action conforms to a policy. Actions are specified in *action attribute sets* which are sets of name-value pairs. Policies are written in KeyNote assertion language and can be broken up and distributed as credentials to a network. The local policy then can refer to a credential when making a decision.

Keynote engine functionality can be seen in figure 3-1. Keynote engine takes as input Certificates, Policies, Actions’ environments and Requests and provides a decision as output in the format of yes/no.

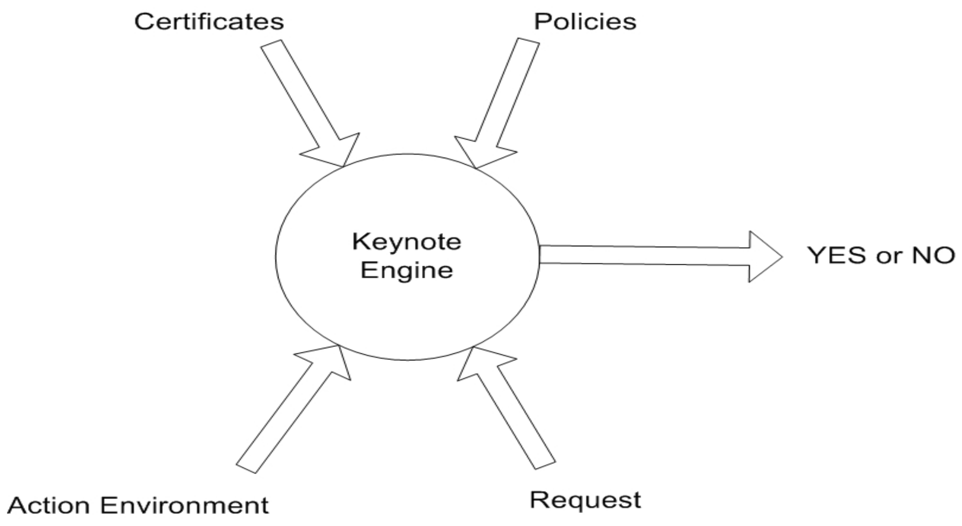
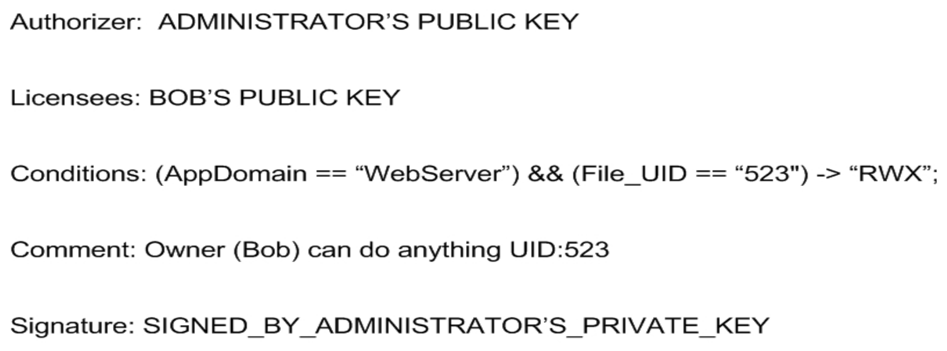


**Figure 3-1: Keynote input/output**

An example of two credentials forming part of a delegation chain is illustrated (figures 3-2 and 3-3). In the example illustrated, user Bob is granted access to a file through a credential shown in figure 3-2 which is issued by the administrator.


Authorizer: ADMINISTRATOR'S PUBLIC KEY

Licensees: BOB'S PUBLIC KEY

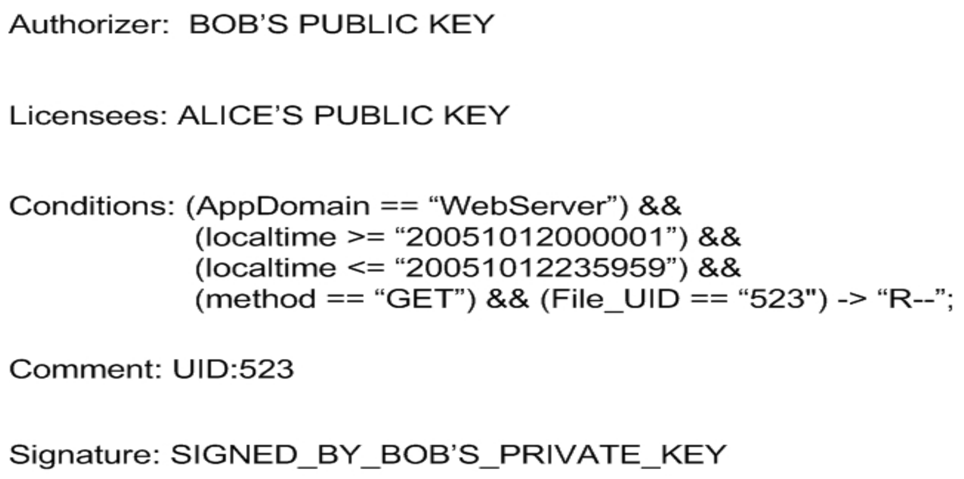
Conditions: (AppDomain == "WebServer") && (File_UID == "523") -> "RWX";

Comment: Owner (Bob) can do anything UID:523

Signature: SIGNED_BY_ADMINISTRATOR'S_PRIVATE_KEY

**Figure 3-2: Credential giving Bob access to a file**

Figure 3-3 then shows how Alice is granted read-only access by Bob to access the same file during October 12 2005. KeyNote allows this delegation because the authority granted to Alice is a subset of the one granted to Bob.



```
Authorizer:  BOB'S PUBLIC KEY

Licensees: ALICE'S PUBLIC KEY

Conditions: (AppDomain == "WebServer") &&
              (localtime >= "20051012000001") &&
              (localtime <= "20051012235959") &&
              (method == "GET") && (File_UID == "523") -> "R--";

Comment: UID:523

Signature: SIGNED_BY_BOB'S_PRIVATE_KEY
```

**Figure 3-3: Credential from Bob giving Alice read-only access to a file**

The example shows that the users are identified only by their public keys. KeyNote allows using the system without previous knowledge of the user base. Users can propagate access by delegating rights to other users and this allows the system to associate access requests with keys by using the information in credentials making possible the reconstruction of the authorization path from the administrator to the user making the request.

In the example shown in figures 3-2 and 3-3 the system can log that Alice’s public key was used to get a file and Bob’s public key authorized the operation leaving an audit trail that can be used to validate that the appropriate policy is being followed. A final note is that the user can at most pass on the privileges he holds and can delegate only a subset of his privileges or the complete set of privileges he holds but not more.

## 3.3 ÆTHER

ÆTHER [23] [24] is a decentralized architecture for handling the complex requirements of trust establishment and authorization in pervasive computing. ÆTHER is designed specifically to address dynamic smart environments where *a priori* knowledge of the complete set of participating entities and global centralized trust registers cannot be assumed. The general ÆTHER model is instantiated into two distinct architectures, namely ÆTHER$_0$ and ÆTHER$_1$. ÆTHER$_0$ has been designed to address the authorization needs of small pervasive computing domains whose management requirements are simple. ÆTHER$_1$ has been designed to support the authorization requirements of large pervasive computing domains that have multiple owners with complicated security relationships. This means the existence of large numbers of devices whose ownership rights can be shared among many principals. ÆTHER$_0$ is out of scope for this dissertation and from now on when ÆTHER is mentioned it refers to the ÆTHER$_1$ instantiation.

In ÆTHER the users embed policy entries into their devices that allow access to protected services only to entities that present the correct attribute certificates. Also definitions of *attribute authority sets* whose members are trusted to certify the corresponding attributes are created. The design of ÆTHER supports dynamic membership in the attribute authority sets in order to facilitate distributed administration and attribute mapping to allow roaming among authority domains.

An authority domain (AD) in the terminology of ÆTHER is the initial set of relationships between attributes and principals specified in a security policy and is a logical representation of a smart environment. The owner of several pervasive devices creates an authority domain by specifying in a policy which principals are trusted to certify which attributes. Also the owner creates policy entries for controlling which attribute credentials a principal must present in order to get specific access rights to a resource provided by a device [24].

The policy constructs that define the attributes of the authority domain and the principals that act as sources of authority for these attributes are called attribute authority sets (AASs). In ÆTHER permissions are modelled as the rights of an AAS as seen on figure 3-4.

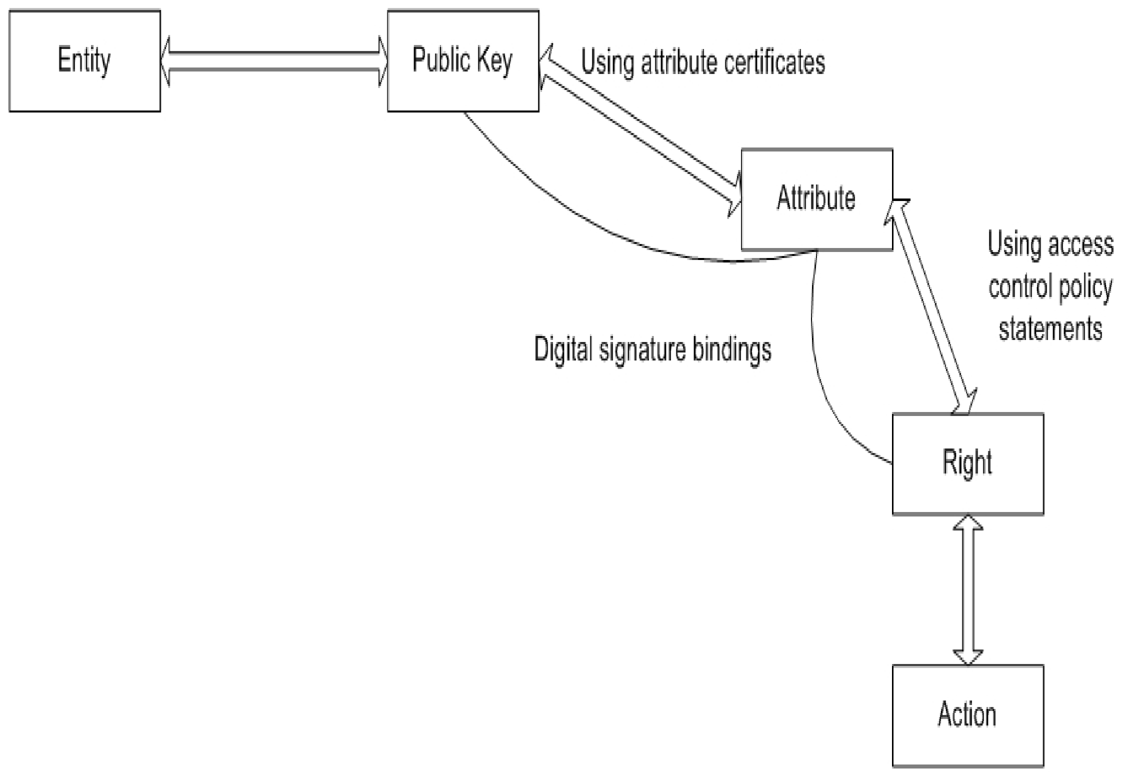

Entity
Public Key
Using attribute certificates
Attribute
Using access control policy statements
Digital signature bindings
Right
Action

**Figure 3-4: Permission modelling in ÆTHER**

Rights are associated with actions, meaning that possession of an authority attribute permits the certified principal to perform a certain action. The certification of the authority attributes is performed by the members of the corresponding AAS. ÆTHER supports dynamic and static AASs. Static attribute authority sets have as sources of authority only principals specified in the initial policy entry whereas dynamic AASs can grow without requiring the explicit change of a policy entry or the issuing of a new one, providing this way decentralized administration of authority domains and facilitating the introduction of principals that were previously unknown to the system.
Below is an example of an AAS used in ÆTHER:

```
aether version: 1
type: authority attribute set
issuer: “304802...1532c” (public key Key0)
attribute name: group
attribute value: visitor
membership threshold value: 2
delegation depth: 2
sources of authority: “304802...1532c” (public key Key0),
“802...ab32c” (public key Key1)
signature: “5eac6d9...7cfd575e68” (Key0’s signature)
```

In this example, the issuer (Key0) defines the sources of authority for giving an attribute certificate with name *group* and value *visitor* which will be valid in his domain. Delegation depth and threshold value are 2 which means that a trusted visitor is able to introduce another visitor to the house, and in order for a visitor to be able to do so he must be trusted by two existing members of the AAS. That means that by using specific values for threshold and delegation the AASs can grow dynamically.
ÆTHER also provides a mechanism to allow the linking of ADs by mapping corresponding attribute sets for supporting secure interactions between different smart environments by using attribute mapping certificates and a naming mechanism for allowing users to specify names instead of keys in interactions with their devices. For more information see [24].
In ÆTHER bindings of public keys to attributes are being done by using attribute certificates (ACs) as shown in figure 3-4.
An attribute certificate example is the following:

```
aether version: 1
type: attribute certificate
issuer: “304802...1532c” (public key Key0)
subject: “1819c...2cf50bbb17c075e” (public key Key4)
attribute name: device
attribute value: family_object
not valid before: 2004/04/12-12:21
not valid after: 2004/04/12-12:31
signature: “198b73ed9...7cfd1b1575e68” (Key0’s signature)
```

In the above example the certificate binds the attribute (`device, family_object`) to the principal Key4 and is certified by principal Key0. Currently in ÆTHER instead of using CRLs a short expiration time period and refreshing mechanisms for the issued credentials are being used.

Bindings of attributes to rights are being done by using data structures called authorization statements. The authorization statements are further divided into positive and negative. The positive statements define the resources that are provided by the device and the attribute credentials that are required to access them.

The negative authorization statements explicitly deny access to a resource based on the attributes of the requesting principal and are used in ÆTHER to enforce separation of duty requirements for limiting the authority that can be acquired by a specific principal. A positive authorization statement is the following:

```
aether version: 1
type: positive authorization
issuer: “304802...1532c” (public key Key0)
resource: TV_set
operation: control_state
requires: (group == family_member) || (group == visitor)
signature: “5426d9...7cfd1b1575e68” (Key0’s signature)
```

This authorization example specifies that the action (`TV_set, control_state`) can be performed only by principals certified to have the attribute (`group, family_member`) or (`group, visitor`).

The input and output of the ÆTHER engine is illustrated in figure 3-5.

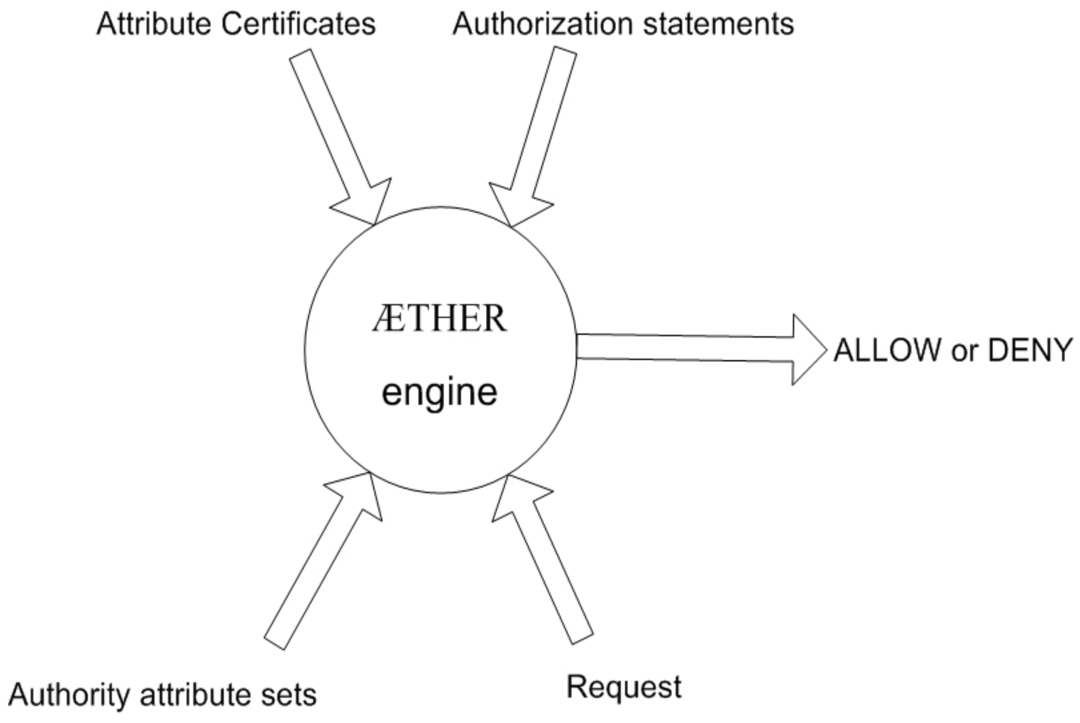


**Figure 3-5: ÆTHER input/output**

## 3.4 Summary

Trust management systems solve the difficulties met when using traditional authorization in highly distributed environments such as web services where participants are not known beforehand. The ÆTHER trust management system provides easier management of policies and provides decentralized administration by introducing the use of attribute based authorization in authority domains. Policies are kept in authority domains and only attribute certificates are being distributed. Capability-based systems like Keynote lack the ability to define complex policy expressions like for example “any principal that has an attribute is able to access a service”. Instead they must delegate the access right specifically for each user.

However there are environments that could introduce hundreds of services and this would mean issuing a delegation for each user that wishes to access the service and this would introduce a big administrative overhead and responsibility. A Web Services environment is similar and tens or hundreds of services could be hosted in the same machine. An example would be a University that wants to provide access to an online library to all the students of a specific department. Using Keynote, a certificate that verifies that a student is able to access the specific online resource must be issued to every one of the students that belong to the department (figure 3-6).

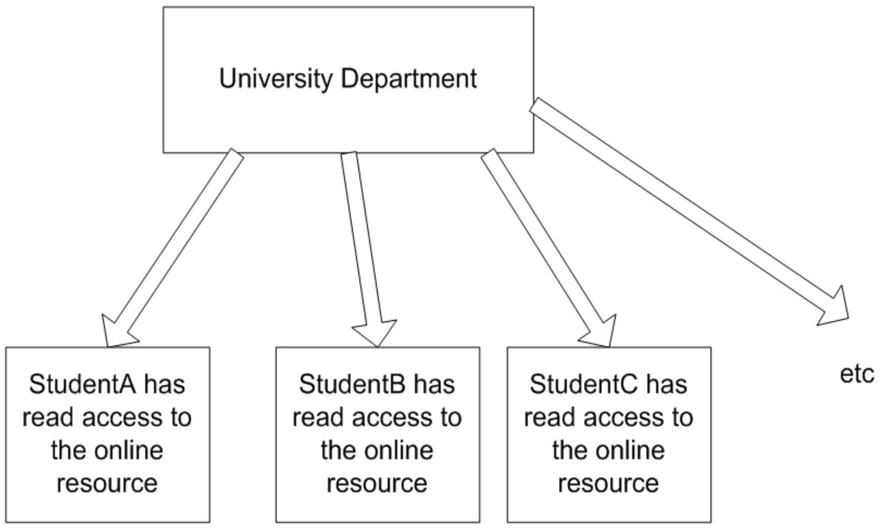

University Department
StudentA has read access to the online resource
StudentB has read access to the online resource
StudentC has read access to the online resource
etc

**Figure 3-6: Keynote implementation**

This means that every time the department wants to provide access to a new online resource the operation of issuing certificates for the new resource to each student separately must be repeated. The same operation must be repeated in case the department wants to grant a different kind of access to the resource, for example if the post graduate students must have write access also, a new kind of certificates granting read-write access to the post graduates must be issued to them. And for every new resource the whole operation has to be done from the beginning, involving great management costs as the system scales.

On the other hand, using ÆTHER, it is possible to implement policies saying that "every student has read access to the resource" and "post graduate students have read-write access to the resource". In order to implement this, the department first has to issue ACs validating the students and the post graduates.

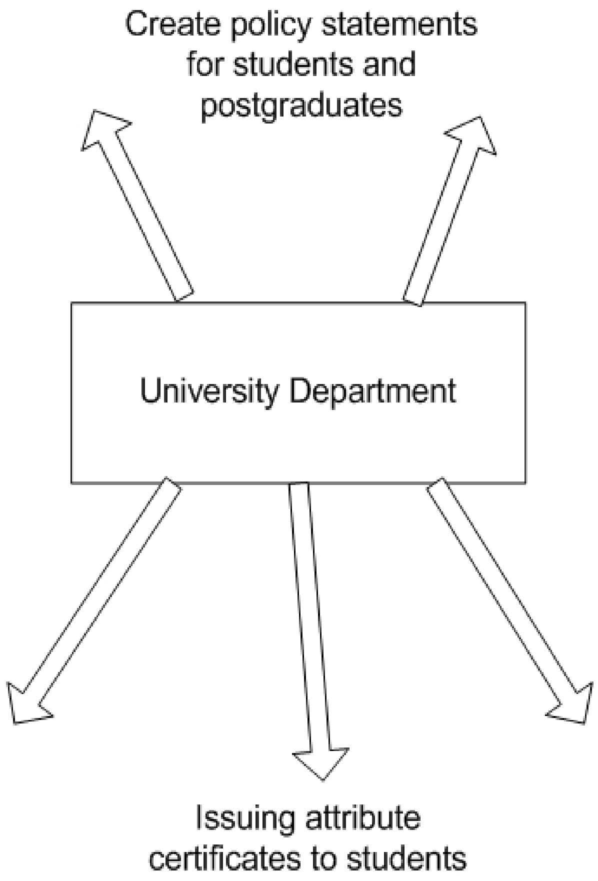

Create policy statements for students and postgraduates
University Department
Issuing attribute certificates to students

**Figure 3-7: ÆTHER implementation**

Then it must create a corresponding access control statement that would give read access to the online resource on students and another one giving read-write access on post graduate students. When the department wants to provide access to a new online resource, it has only to create a new access control statement saying that all the students have access to it which in the case of using Keynote is impossible.
This ability of ÆTHER to implement more complex policies and manage them easily makes it flexible enough and a more appropriate trust management system to be used in web services environment.

# 4 Design

## 4.1 System Overview

WebÆTHER is a security management system that focuses on access control for Web Services. The approach of WebÆTHER to web services security is different than the WS-* standards series. It focuses on trust management and the ability to distribute policies between previously unknown entities by using the ÆTHER decentralized architecture rather than XKMS where a traditional PKI is being used. Attribute based authorization is identical to ÆTHER and authority domains are considered the servers that provide the Web Services. Policies and attribute authority sets are specific for each Web Service and attribute certificates are being distributed to requesters of the service.

Encryption is being used in WebÆTHER at the transport layer in order to provide point to point confidentiality in terms of Web Services and integrity. This means that the architecture currently targets Web Services that will not use intermediate nodes that can access the same message thus requiring different encryption in parts of the message. This means that each message in WebÆTHER has only a single recipient.

WebÆTHER does not address problems like denial of service attacks and is not able to handle stealing of private keys used in creating the public keys included in attribute certificates. Moreover definition of conflicting policies, issuing wrong attribute certificates and inappropriate administration are considered outside of this dissertation's scope.

## 4.2 Architecture

In WebÆTHER, a server providing Web Services is an authority domain and its administrator is responsible for specifying the sources of authority for the service and the policies themselves. Users can get access to a service by providing proof that they hold the appropriate attributes. In order to achieve this, WebÆTHER model uses three kinds of ÆTHER statements and four mappings. In more detail WebÆTHER model is consisted of:

- A set of Subjects: this is the set of the requesters of web services
- A set of Attributes (As)
- A set of Attribute Authority Sets (AASs): the list of principals that act as authorities for a given attribute name and value
- A set of operations
- A set of Web Services
- A set of permissions
- A mapping of As to permissions: this is done using the policy statements
- A mapping of Subjects to attributes: this is done using AC statements
- A mapping of Subjects to AASs: this is done using AAS statements
- A mapping of AASs to As: this is also done using AAS statements

Policy statements are used in order to define permissions. In other words this means expressing the access control policies which are basically the rules on which a decision on giving access or not is made. The kind of access is described by the binding of an operation to the Web Service. The Attribute Authority set is a new concept of ÆTHER that defines the set of keys that are sources of authority for the specified attribute. In a similar way in Web ÆTHER Attribute authority sets are the means of defining the principals that are able to validate that a principal has an attribute. Attributes are similar to the concept of roles in traditional RBAC. A subject is a member of a role only if it has been assigned the authority attribute that represents the role. Attribute certificates are used by subjects that need to proof the possession of an attribute. The attributes are not static for a subject and they need to be obtained dynamically prior an access request. ACs are related to permissions in the context that in order to have a permission the subject must have the appropriate ACs that are defined in the Policy Statement.
The WebÆTHER model is illustrated in figure 4-1.

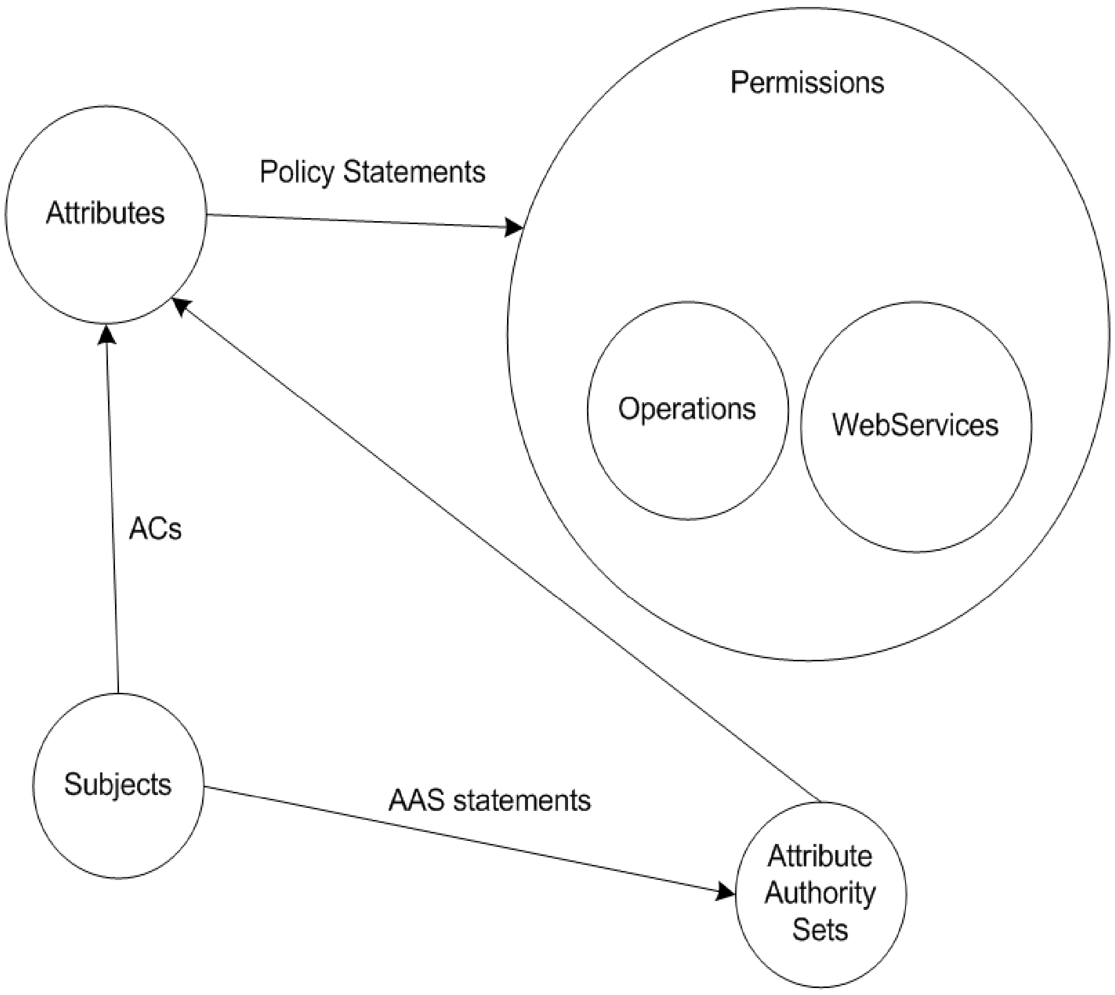


**Figure 4-1: WebÆTHER model**

### 4.2.1 Policy Statements

Policy statements are the means of specifying requirements in order to provide access to a service. A policy statement contains the action and the attributes needed to be certified. For example `hostA` may provide a service called `special_serviceA` which would be available only to the clients that are certified to belong in the group `employees` or group `co-workers` specified by the administrator of the host. The policy statement is going to look like:

```
aether version: 1
type: positive access control policy
issuer: “304802...1532c”(hostA’s public key)
resource: special_serviceA
operation: get
requires: @group == employees || @group == co-workers
```

The public keys in all the statement files used in WebÆTHER are in hexadecimal format like in ÆTHER. In the given examples only a minor parts of the keys are displayed. The requesters of the service are going to be granted access as long as they provide an attribute certificate that can be validated and contains the attribute name `group` and either the value `employees` or `co-workers`. For each Web Service that `hostA` provides there must be a correspondent policy statement. WebÆTHER keeps the notion of positive and negative policy statements but in the context of rather simple Web Services that do not require separation of duty to be applied, only positive statements are useful. In more complex policies that for example would require public keys to belong in a specific group but have to make sure they don’t belong to another, both kinds of policy statements can be used. The task of including the correct attributes in the policy statements are the responsibility of the administrator.

One advantage of using policy statements is the ability to ask the requesters of a service to provide only the related attribute certificates instead of sending every attribute certificate they have when they want to access a Web Service. Notice that policy statements in WebÆTHER do not need to be signed because they are valid only locally to the host that provides the specific service.

### 4.2.2 Policy Distribution

In WebÆTHER policies are distributed in order to provide easier management of access control. For example `hostA` from the previous example is a machine in a software development company (called `companyA`) headquarters that provides a service called `special_serviceA` and must be available to all the people working in the team. The model is illustrated in figure 4-1.

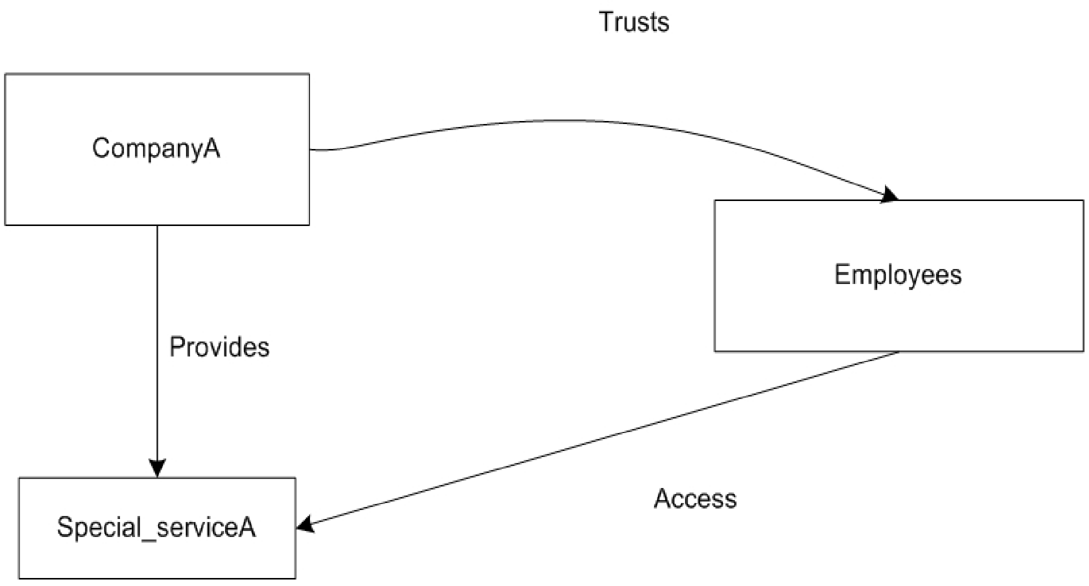


**Figure 4-2: CompanyA example**

In WebÆTHER this is being done by providing the attribute certificates to the employees that certify their status of being company employees. Such an example attribute certificate will have the following form:

```
aether version: 1
type: attribute certificate
issuer:  “304802...1532c” (hostA public key)
subject: “1819c...2cf50bbb17c075e” (an employee’s public key)
attribute name: group
attribute value: employees
not valid before: 2004/04/12-12:21
not valid after: 2004/04/15-12:31
signature: “5426d9...7cfd1b1575e68”(hostA’s signature)
```

The administrator of `hostA` will distribute this kind of certificates to every employee in the company in order for them to be able to prove to `hostA` that they are employees of the company. To be able to check the validity of these attribute certificates the system must be able to know which keys are able to certify the group. In WebÆTHER this is being done by using another kind of statements which are attribute authority sets. If in `companyA` exist two hosts (`hostA` and `hostB`) that can certify that a public key belongs to an employee the AAS is going to look like:

```
aether version: 1
type: authority attribute set
issuer: “304802...1532c”(hostA public key)
attribute name: group
attribute value: employee
membership threshold value: 0
delegation depth: -1
sources of authority: “304802...1532c”(hostA’s public key),
“5426d9...7cfd1b1575e68”(hostB’s public key)
```

In the above example `hostA` and `hostB` are defined as sources of authority which means that WebÆTHER is going to consider valid the ACs that are issued by any of these two hosts. Delegation depth -1 and threshold value 0 mean that a trusted employee is not able to introduce another employee to the company and only the employees with ACs signed by either of the two sources of authority are considered valid. If the number of employees is big and the two sources of authority are considered very few, the list of sources of authority can dynamically grow by increasing those two values. For example increasing to delegation depth 1 and threshold value 2, means that when a public key that has been issued an AC from each of the sources of authority in the AAS, can also act as a source of authority and the ACs it issues are considered valid from hostA. The value 0 for delegation depth means there is no restriction on the maximum delegation.

Let’s assume now that the company has contracted with another software development company (called `companyB`) and these two have now become a single company that must collaborate in order to develop a new product. In order for this to happen, `companyB`’s employees must also have access to the service called `special_serviceA`.

The new model of the two merged companies is illustrated in figure 4-2.

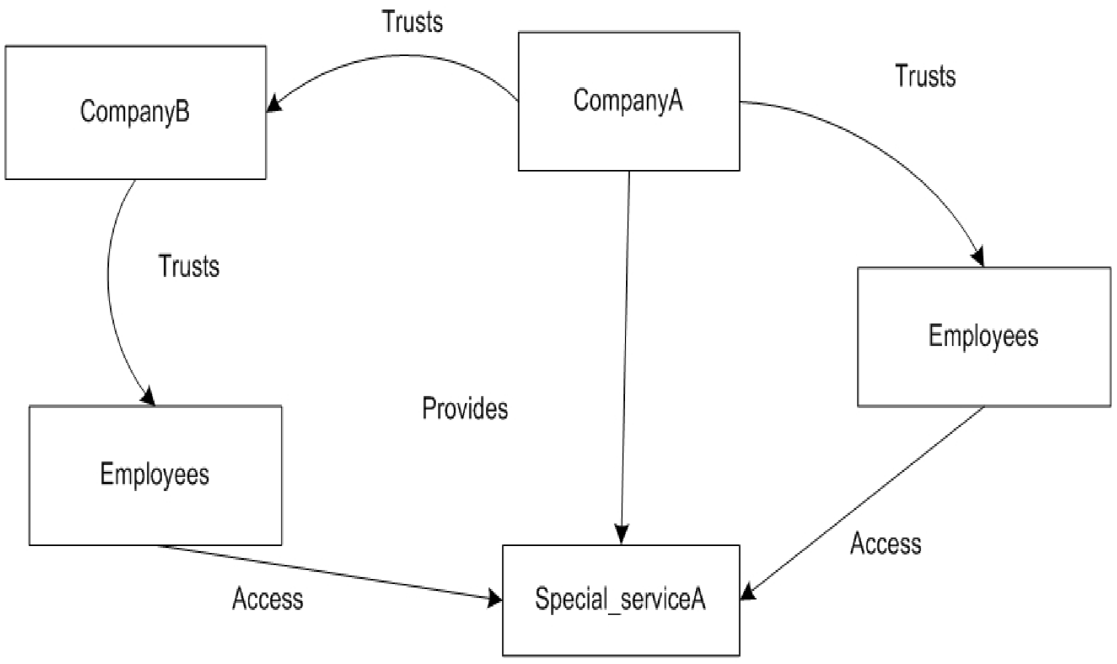


**Figure 4-3: merged companies example**

In WebÆTHER this can be done in three different ways:

- Create a new host or select a host in `companyB` to act as a source of authority in `companyB` and add his public key in the existing AAS.
- Sign `companyB` employees’ ACs using either `hostA` or `hostB` who are currently in the AAS.
- Increase delegation depth and threshold value in order to permit a specific number of trusted employees of `companyA` at first and then from `companyB` to introduce new public keys as employees in the `companyB`.

It is easy to realize that the architecture is flexible enough to provide alternative ways of introducing new people as employees with different impact on administrative tasks and relying on different trust relationships. If the employees are fully trusted the increase of delegation depth and threshold value leads to minimum administrative effort. If such decision of who is an employee is not trusted to be made from current employees, then depending on the number of new employees one of the other two options can be selected.
There are other options to solve a real world scenario like this using WebÆTHER. For example instead of group employees, alternative groups can be used like `companyA_employees` and `companyB_employees` with the corresponding changes in AAS's sources of authority and policy statements.

### 4.2.3 Certificate Revocation

There are occasions when an attribute certificate's validity must be revoked. These occasions include the loss or compromise of the key that was used by a source of authority in order to sign an attribute certificate, the loss or compromise of a key to which a certificate was issued or when the attribute that was certified for a requester is not considered valid by the authority any longer. The main mechanisms to achieve this are:

- Using the validity period of a certificate
- CRLs
- OCSP

Using the validity period of the certificate, an issuer is able to set it to a short time interval. This way revocation happens when the period expires and there is no issuing of a new certificate. The window of revocation is limited to the validity period specified as part of the certificate. This approach encounters some problems. If the validity time period is very short, it can result in a frequent re-issuing of all the certificates. This would introduce a constant administrative task for the issuer and continuous traffic on the network that would downgrade the performance of the system and could harm other applications using the same networks. It can also lead to time periods during which access to a particular service is impossible. On the other hand, if the validity period is set to a very high value then a required revocation may not take place until this period expires.

The next mechanism is Certificate Revocation Lists (CRLs), which are lists that include all credentials that have been revoked. The issuer must create a CRL and distribute it to all participants of the system. The main problems with using CRLs are two. First of all the list must be distributed to all the interested parties. This can be done by using flooding techniques. Another solution is placing the CRL on a public repository that the interested parties would retrieve the CRL from. Both of these approaches require the issuer to always be available for access. The second problem is related to scalability. As the number of revoked certificates increases CRLs require more effort in order to manage them.

The final revocation mechanism is OCSP (Online Certificate Status Protocol) [25]. OCSP requires the certificate verifier to check the revocation status of the presented certificate at every verification request. When a user attempts to access a server, the server's OCSP implementation sends a request for information considering the certificate status to the issuer. The issuer sends back a response that classifies the certificate's status as current, expired, or unknown. OCSP allows expired certificates a grace period, so that the users that have them can access servers for a limited time before renewing. This means that the verifier must always be able to contact the issuer of every credential that is presented to him. Also, since the verifier is also a service provider, the process of checking the status of every credential at every access request introduces a significant additional computation overhead.

ÆTHER uses the approach of using the certificates' validity period. This is due to the fact that CRL implementation and management is difficult in a ubiquitous environment with a large number of participants. Also, the computation overhead introduced in the use of OCSP is also very expensive for small devices and there is the requirement that the issuer is always accessible by every verifier, a requirement that is not compatible with the disconnected nature of pervasive computing. The responsibility of setting a correct expiration period is assigned to the administrators of these environments and a suggested value for it is 1 hour.

In WebÆTHER the environment is consisted of more powerful nodes and the computation overhead of using an OCSP approach would not have a big impact in performance. The administrative effort in the other hand of using CRLs or OCSP would be at the same level. In WebÆTHER services and participants can be numerous and the implementation of such mechanisms would be very difficult because of the distributed environment. Therefore, the certificate validity period is considered the best solution to certificate revocation, with the suggested expiration values being higher than the ones used in a ubiquitous environment, for example 24 hours or more.

### 4.2.4 Service Discovery

WebÆTHER being a trust management system for use in Web Services uses the typical service discovery mechanism for Web Services which consists of WSDL descriptions of the provided remote methods that compose the service and can be contained in a UDDI registry for easier publishing and acquisition. The WSDL descriptions can also reside on static URLs for the interested parties to acquire them.
The advantage of this method for service discovery is that it is based in a currently used approach. The disadvantages are that WSDL provides description of the input and output of the remote methods but it doesn't provide information on how they can be implemented. This means that familiarity with WebÆTHER is required in order to implement the methods described in the WSDL descriptions.

### 4.2.5 Certificate Acquisition

In order for WebÆTHER model to work, users that need to acquire ACs must be able to contact the issuers in order to send a request and then receive a signed by the issuer AC. In WebÆTHER this is done by issuers providing an AC issuing service. For example, `issuer_A` is able to issue a number of ACs. `requester_B` wants to access a service and needs the correspondent attributes to be assigned to him. First he has to communicate with `issuer_A` in order to ask for the attributes by email or by a telephone call, stating his identity. Then `issuer_A` has to decide if he trusts `requester_B` based on his own criteria.

If the decision is yes, he will create a temporary URL that will be valid only for a short period of time and will contain the AC issuing service. Finally he will send a reply giving the URL `requester_B` (figure 4-4).

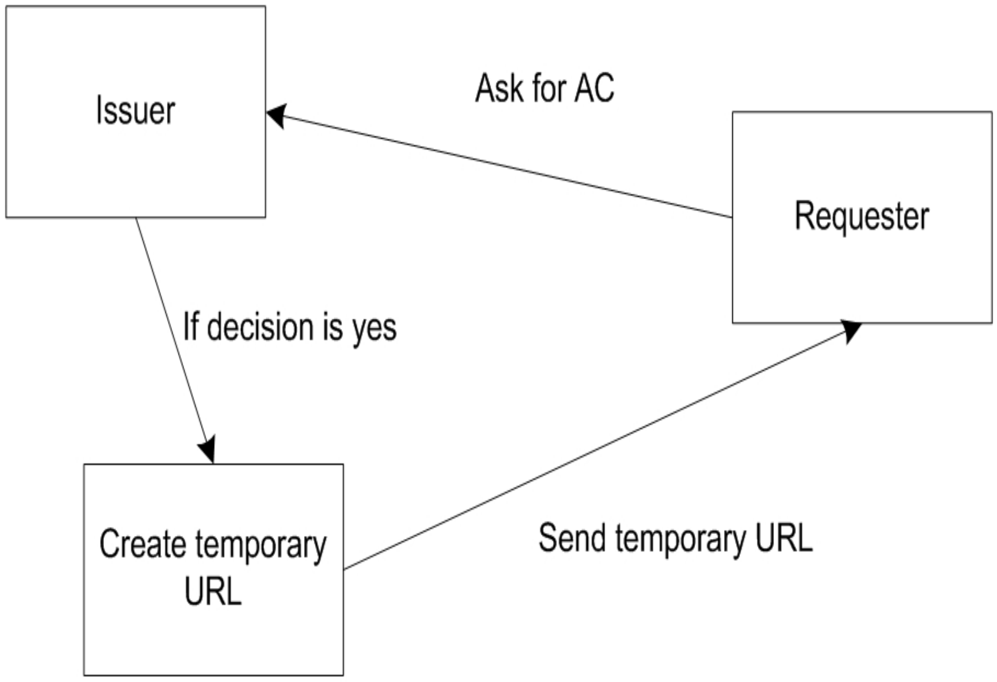


**Figure 4-4: request handle approach**

After this step the requester knows the URL that he is going to use to acquire the AC. The AC issuing service works in four steps (figure 4-5). In order to prevent man in the middle attacks, the whole operation is done by using SSL and the requester has to use his public key that will also be used in the AC for the SSL session establishment. The operation after the establishment of the SSL channel is:

- First the requester makes a call on the remote method to signal the start of the request.
- Then the Issuer sends back a data structure that contains the attribute name and value that is going to be in the AC when the procedure finishes.
- The requester adds his public key to the structure and sends it back to the issuer.

- Now the issuer is able to verify that the public key used in the SSL session is the same to the provided one, create the AC using the attribute name-value pair, the requester's public key and his own public key, sign it, verify it and send it back to the requester.

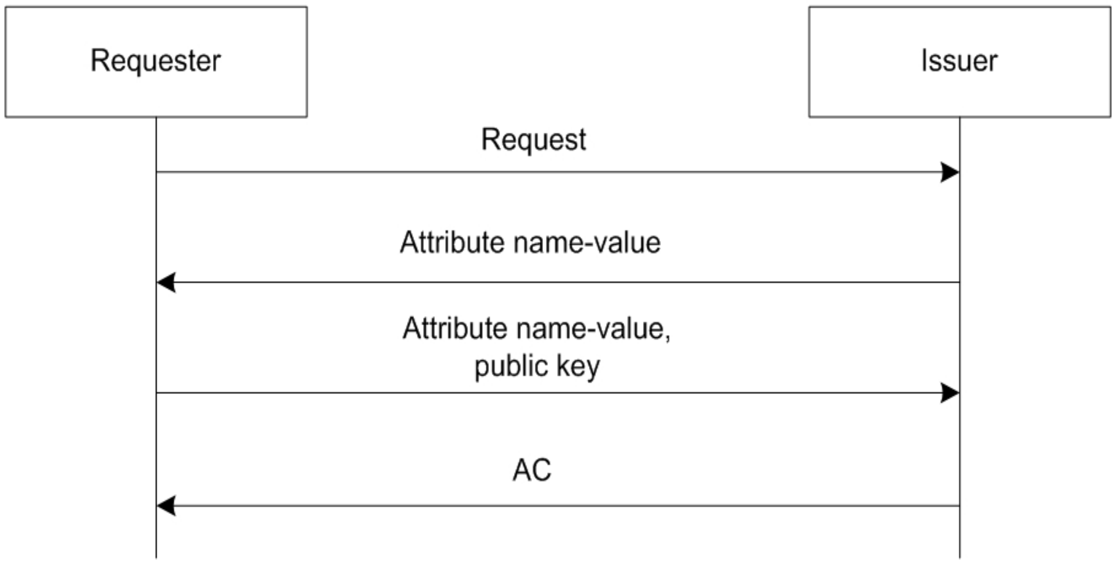


**Figure 4-5: AC service**

The requester is then able to save the newly issued AC in his repository that holds all his certificates. This approach for certificate acquisition has the advantage of simplicity and ease of use. The possible complications have to do with the fact that the requester must verify his identity to the issuer before he can access the URL that provides the service. The two suggested methods are telephone calls and email. Telephone calls are useful in cases like small companies where the users are few and they possibly know each other. Email has to be digitally signed in order to provide assurance that the requester's identity is the one he claims to be. The use of a temporary URL introduces a security risk. If an attacker manages to learn the temporary URL and use it while it is active, he can acquire an AC which validity will be verified by the system. Therefore the temporary URL is strongly recommended to have a random name each time it is used by the issuer and the time period that it is available to be kept to only a few minutes after the sending the email that contains it so the security risk will be minimized.

### 4.2.6 System Components

The WebÆTHER is consisted of 3 major components in the server side and 2 major components in the client side. The components found in the server side are the proxy component, the ÆTHER component and the Web Service resources component. The components found in the client side are the proxy component and the ÆTHER component.

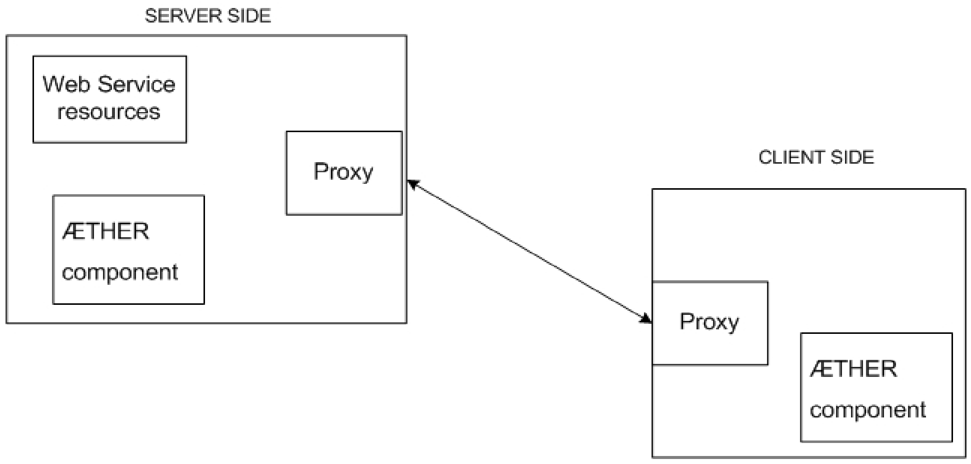


**Figure 4-6: System components**

The proxy components in both sides are responsible for the creation of the request/response messages and data serialization/de-serialization in order to be available for transmission. The ÆTHER component provides the trust management functionality to the server and is responsible for handling the attribute certificates at the client side. The Web Service resources are the set of resources that the system is able to provide. For example this can be access to a file that is located at the server side.

### 4.2.7 System Logic

WebÆTHER system logic is divided in three main parts. The first part is Certificate Acquisition which was described in paragraph 4.2.5. The remaining two parts form the web service and include the establishment of an SSL session before the logic itself takes places. The two parts are:

- Service Query
- Service

Service Query is the part of the system that is used in order to provide the list of needed attribute names for accessing the service to the requester. There is a service query method for every provided service in the system. The requester has to invoke this method in order to acquire the list of needed attribute names before he attempts to access the service itself. The service provider when he receives a service query message, he searches the list of policy statements for the correspondent to the service one. After parsing the field `requires` that is found, he is able to construct a *reduced ACL* (RACL) which is a set of all the attribute names found in the policy statement without the predicates that apply on them.
For example, if the policy statement is:

```
aether version: 1
type: positive access control policy
issuer: (hostA's public key)
resource: special_serviceA
operation: get
requires: @group == employees && @speciality == engineers
```

The correspondent *reduced ACL* is (`group, speciality`) and a list containing these two attribute names must be sent to the requester. Service Query for the above example is illustrated in figure 4-7.

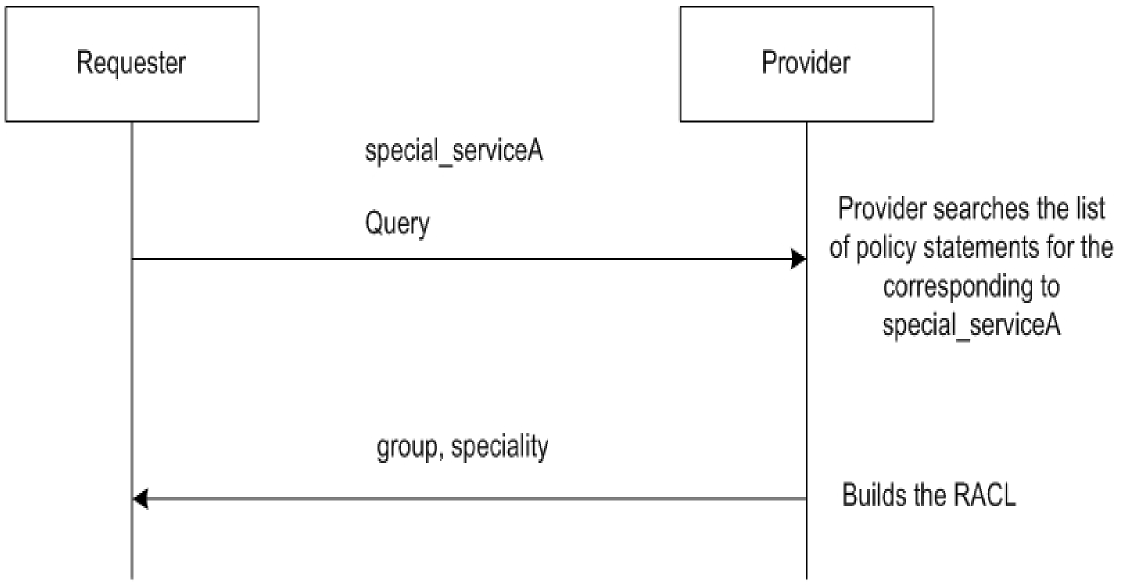

Requester
Provider
special_serviceA
Query
Provider searches the list of policy statements for the corresponding to special_serviceA
group, speciality
Builds the RACL

**Figure 4-7: Service Query**

The last part of WebÆTHER system logic is the Service itself. In order to access the service the requester must send a message containing a list of the needed ACs. The way to define which ACs to send is done by using the *reduced ACL* that was received in the previous part. In the above example, after invoking service query the requester knows that he has to send ACs with attribute names `group` and `speciality` to the provider in order to access the service but he is unaware of the predicates specified in the policy statement. Therefore he will send all the ACs he possesses that include these two attribute names.

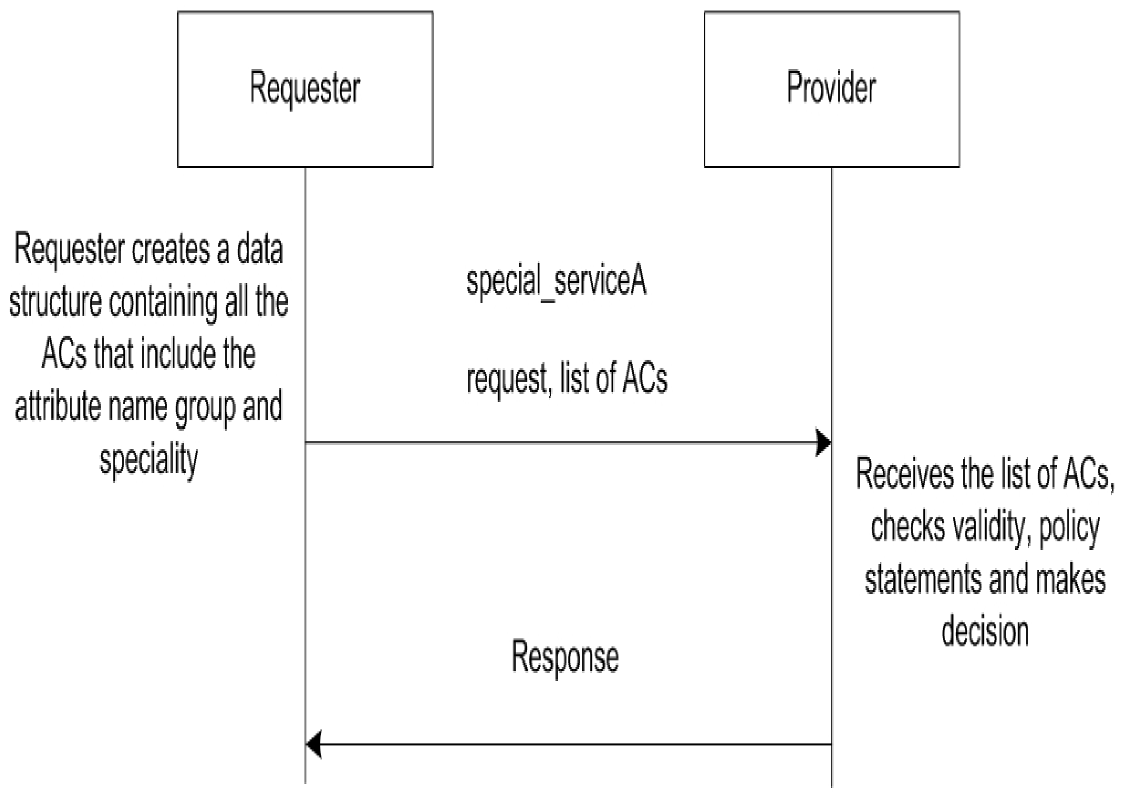

Requester
Provider
Requester creates a data structure containing all the ACs that include the attribute name group and speciality
special_serviceA
request, list of ACs
Receives the list of ACs, checks validity, policy statements and makes decision
Response

**Figure 4-8: Service Request/Response**

When the provider receives the requester’s ACs he is going to check the subject public key field in every AC to be the same to the one used in the establishment of the SSL session. This is done in order to prevent man-in-the-middle attacks. In a man-in-the-middle attack, the attacker will have to use his key in the establishment of the SSL session. Forging the ACs will result in invalidating them because the digital signature will not correspond to the message. That leaves the attacker the option to try and send stolen ACs but checking the session key prevents him from getting access. The next step is to check each AC’s validity and finally check the policies in order to decide whether or not to give access to the requester. In order to give access in the example, the requester must include in the list of ACs that he sends to the provider a valid AC with attribute name `group` and value `employees` and a valid AC with attribute name `speciality` and value `engineers`.

## 4.3 Summary

The main design of WebÆTHER trust management system for Web Services was presented in this chapter. The main parts of the system address the issuing and acquiring of certificates, revocation, validation and finally service discovery and the service messaging protocol itself. Web Services environment seems to be compatible to the ubiquitous computing one and changes were made only on the messaging protocol compared to ÆTHER while the heart of the trust management system remains the same. The implementation details of the system and the performance evaluation are demonstrated in depth in the following two chapters.

# 5 Implementation

## 5.1 Overview

In order to implement and evaluate WebÆTHER the Web Service that was chosen is a file transfer service. The host of the service is going to provide a file that when the requester meets the requirements will be available for downloading. This scenario was chosen in order to demonstrate the ability of WebÆTHER to be implemented in the very popular area of file sharing while providing a secure framework for the service.
For the Service Transport layer, HTTP was selected because it is the dominant protocol for this layer in the Web Services area. The creation of a new XML based messaging protocol for the messaging layer implementation was considered to be outside of scope to this dissertation, therefore the options were limited to the ones with the best support and documentation. These options were SOAP and REST. SOAP was finally selected between the two, because of the features of the existing tools, the overall support from the Web Services community and the ease of implementation using the existing development tools.
Apache web server, being the dominant web server was selected and the whole implementation would work over an established SSL session using Apache and OpenSSL. All the toolkits to be used are going to be open source projects.
One of the primary aims during implementation was to keep as much of the original ÆTHER infrastructure as possible in order to demonstrate the suitability of this trust management system to be used in the area of Web Services.

## 5.2 Apache Web Server & gSOAP

In order to meet the requirement of using the Apache web server and SOAP for the messaging layer of the Web Service, a SOAP toolkit had to be selected. All SOAP toolkits provide a proxy component, which parses and interprets the SOAP message to invoke application code. When a SOAP message arrives to the proxy component by a listener, the proxy must do three things:

1. If necessary, De-serialize the message from XML into some native format suitable for passing off to the code.
2. Invoke the code.
3. Serialize the response to the message (back into XML and hand it back to the transport listener for delivery back to the requester.

The following were the main choices that were available for open source toolkits to be used in the implementation:

1. Apache - AXIS (was Apache-SOAP) C++ version. AXIS [26] is a web service framework consisting of an implementation of the SOAP protocol and various utilities and API's that hides the details of SOAP and WSDL from a user thus facilitating the creation of web services servers and clients. There are two versions of Apache – AXIS, one using C++ and one using Java. The fact that this version uses C++ instead of Java makes it capable of greater performance and more efficiency oriented compared to the Java version. The installation of the framework over a normal apache web server works by using a provided apache module (called libaxiscpp_mod.so). However it also needs Java in order to run wsdl2ws tool, which is a tool that creates skeletons and wrappers from WSDL files for the server side and also stubs for the client side. That means that the development would have to use the two computer programming languages in both server and client side. The implementation of SSL in Apache – AXIS C++ version is done by using a provided shared library (called libaxis2_ssl_channel.so). Axis2Transport which is the part of Apache – AXIS that handles the transport layer, loads this SSL channel library when HTTPS is used instead of HTTP and sends the requests using SSH tunnelling. The framework includes an API to provide the ability to write alternative SSL channel libraries (using a library other than OpenSSL), but the main SSL implementation uses OpenSSL.

2. Apache – AXIS Java version. The Java version of Apache – AXIS has by far the greater support among the open source toolkits currently used for web services in the apache web server because of the fact that it is widely used. In this version of AXIS the web service is written in Java and deployed as a Tomcat *webapp* (servlet) on the server side. The framework implements the JAX-RPC API which is a way to provide Web Services written in Java and provides the WSDL2Java tool in order to create stubs and skeletons. In the java version of Apache – AXIS, an HTTPS listener and sender are not yet implemented at the time this dissertation was carried out.

3. SOAP::Lite [27] [28]. This is a collection of Perl modules that provides an interface to SOAP both on client and server side. There are no special requirements for the Apache web server in order to use this framework and Web Services are cgi scripts written in perl that automatically use SOAP for the messaging layer and are also able to use SSL.

4. gSOAP. gSOAP [29] is a toolkit that provides a transparent SOAP API for the creation of Web Services through the use of compiler technologies. These technologies are being used by the toolkit in order to map XML schemas to C/C++ definitions. The toolkit uses a compiler that generates efficient XML serializers for native and user-defined C and C++ data types. As a result, SOAP/XML interoperability is achieved with a simple API and hides WSDL and SOAP details from the user, thus enabling him or her to concentrate on the application-essential logic. Finally gSOAP supports SSL.

Among these toolkits, gSOAP was selected because of the fact that it uses C++ and this facilitates the use of as much as possible of the original ÆTHER infrastructure which is also written in C++ without the need to re write the code of the core system in a different programming language. Apache – AXIS C++ version was also considered for this reason but the need of also using Java was not appealing and could result in unpredictable difficulties. The Java version doesn’t support HTTPS and the core ÆTHER system code would have to be re-written, therefore it was not taken into

account when the decision on the toolkit had to be made. SOAP::Lite was an interesting solution but invoking C++ from inside perl scripts was considered less convenient than using C++ for the whole implementation.
gSOAP is consisted of a stub and skeleton compiler and a WSDL parser. The stub and skeleton compiler (named *soapcpp2*) is a pre-processor that generates the necessary C++ sources to build SOAP C++ clients. The input to *soapcpp2* is a standard C/C++ header file that can be generated from a WSDL documentation of the service using the gSOAP WSDL parser (named *wsdl2h)*. SOAP service methods are specified in the header file as function prototypes and the stub routines in C/C++ are automatically generated by *soapcpp2* for these function prototypes.
The resulting stub routines allow C and C++ client applications to seamlessly interact with existing SOAP Web Services. In a same manner *soapcpp2* generates skeleton routines in C++ source form for each of the remote methods specified as function prototypes in a header file that is processed by *soapcpp2*. The skeleton routines can be then used to implement the remote methods in a new Web Service. The gSOAP compiler automatically generates serializers and deserializers for the data types that are being used in order to enable the generated stub and skeleton routines to encode and decode the contents of the parameters of the remote methods in XML.
The gSOAP compiler also generates a remote method request dispatcher routine that will serve requests by calling the appropriate skeleton when the SOAP service application is installed as a cgi on a Web Server.

## 5.3 WebÆTHER Model Implementation and system components

The Web Service is installed as a CGI script on Apache2 web server version 2.0.54. CGI is a standard that enables a client to request data from a program executed on the web server. In order to implement the WebÆTHER model that was described in chapter four, the web server was first configured to require user certificates when HTTPS is being used. The file transfer web service CGI is designed to work only with HTTPS and client-side authentication. The components of the implemented system are illustrated in the figure 5-1. gSOAP and SSL form the proxy component that handles the message transport in both client and server side. The Apache component provides the service in the form of the CGI in the server and ÆTHER component provides the trust management to the server and AC handling to the client.

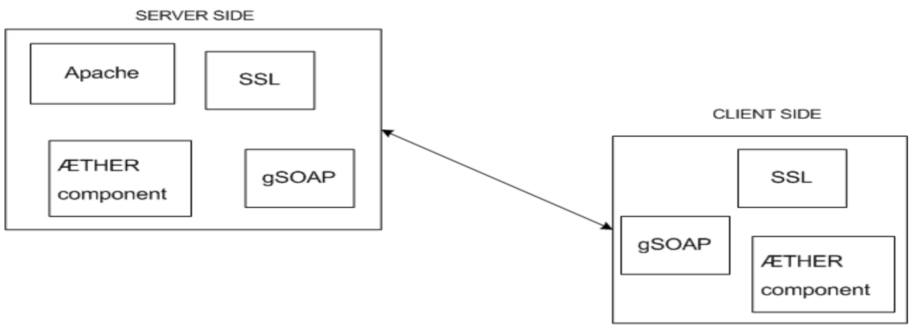


**Figure 5-1: Implemented system components**

The certificates that the clients use in client authentication must include the same public key to the one that is being used later in the WebÆTHER statement files. WebÆTHER only works with RSA 1024-bit keys. The file transfer service was named serviceA and the operations on the model can be viewed on figure 5-2.

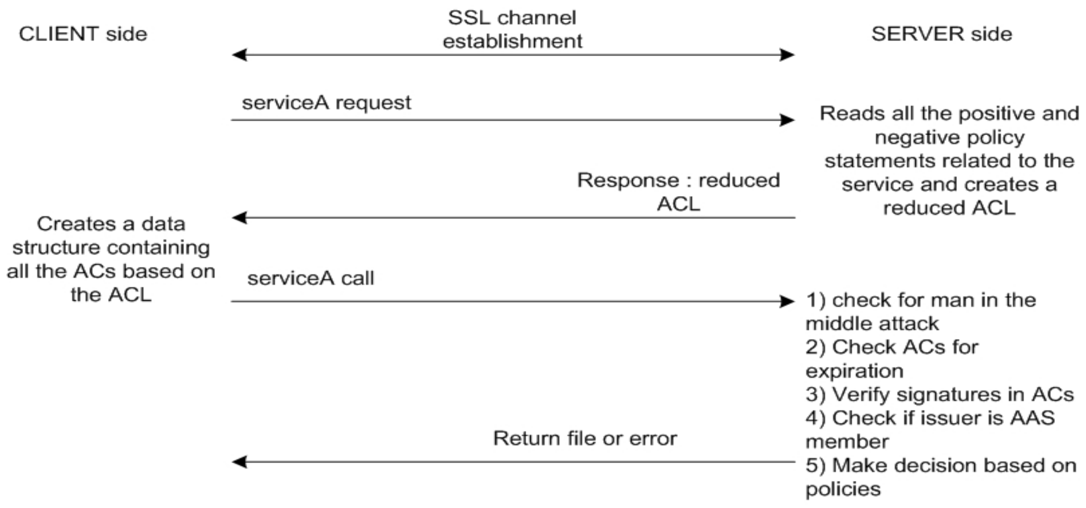

CLIENT side
SSL channel establishment
SERVER side
serviceA request
Reads all the positive and negative policy statements related to the service and creates a reduced ACL
Response : reduced ACL
Creates a data structure containing all the ACs based on the ACL
serviceA call
1) check for man in the middle attack
2) Check ACs for expiration
3) Verify signatures in ACs
4) Check if issuer is AAS member
5) Make decision based on policies
Return file or error

**Figure 5-2: model operations**

After the establishment of the SSL channel, the client sends a serviceA request in order to retrieve the reduced ACL as WebÆTHER system logic commands. The server creates the reduced ACL and returns it to the client. After this step the client knows the required ACs and searches his repository to find all that satisfy the requirements. Then the client is ready to make a call to serviceA itself while sending at the same time the ACs he has gathered from his repository. The server checks the validity of the statements and compares the subject keys of the ACs to the one used in the establishment of the SSL channel and takes into account only those who are identical to it. Then it checks the policy statement for serviceA to make a decision on sending the file back to the client. If the decision is positive, the file is returned else the client gets an error. The implementation of the model is consisted of a header file that contains the function prototypes and the declarations of the complex data types that are being used along with a source file for the server and a source file for the client. In order to meet the goal of using as much of the core of ÆTHER, the implementation uses the sources of ÆTHER for parsing statements, making the date checks, the verification of the statements and the logic control of the collected information in order to make decisions.

## 5.4 Implementation of the Service Query function

The service query function in the file transfer implementation is called `serviceAreq`. This is a remote method and according to gSOAP, a function prototype has to be written in the header file webaether.h. This method has no input and returns an array of strings as an output. The array of strings data type must also be defined in the header file as a structure that contains strings the number of elements and the offset. The definition of the data type is:

```
typedef char *xsd__string;
struct ArrayOfstring {xsd__string *__ptr; int __size; int __offset;}
```

The function prototype of `serviceAreq` is:

```
ns__serviceAreq(struct ArrayOfstring &_return);
```

The implementation of this function is written on the webaetherserver.cpp file that when compiled gives as output the cgi called webaether.cgi. The source file that is used for the client program is called webclient.cpp.

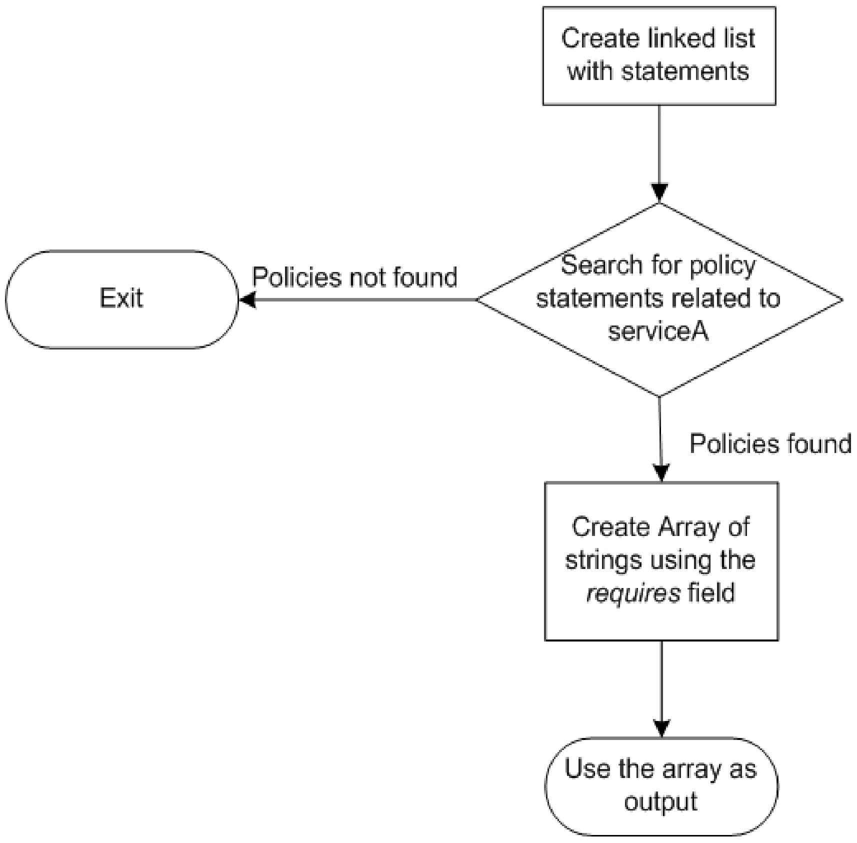

Create linked list with statements
Search for policy statements related to serviceA
Policies not found
Exit
Policies found
Create Array of strings using the *requires* field
Use the array as output

**Figure 5-3: `serviceAreq` server side**

The implementation of `serviceAreq` (figure 5-3) which resides in webaetherserver.cpp first reads the directory where all the server statement files are kept and creates a linked list with all the statements found. Then it searches the created linked list for the policy statements related to the specific web service which in this case is serviceA. After finding the policy statements it creates an `ArrayOfstring` structure and places the unique attribute names found in the `requires` field of the statements and returns it as output. In the client side (illustrated in figure 5-4), the client makes the remote call to `serviceAreq` and creates a linked list that includes all the attribute certificates in his possession. When he receives the array of strings he compares each of the elements to the attribute names he has in the ACs in his possession and counts them.

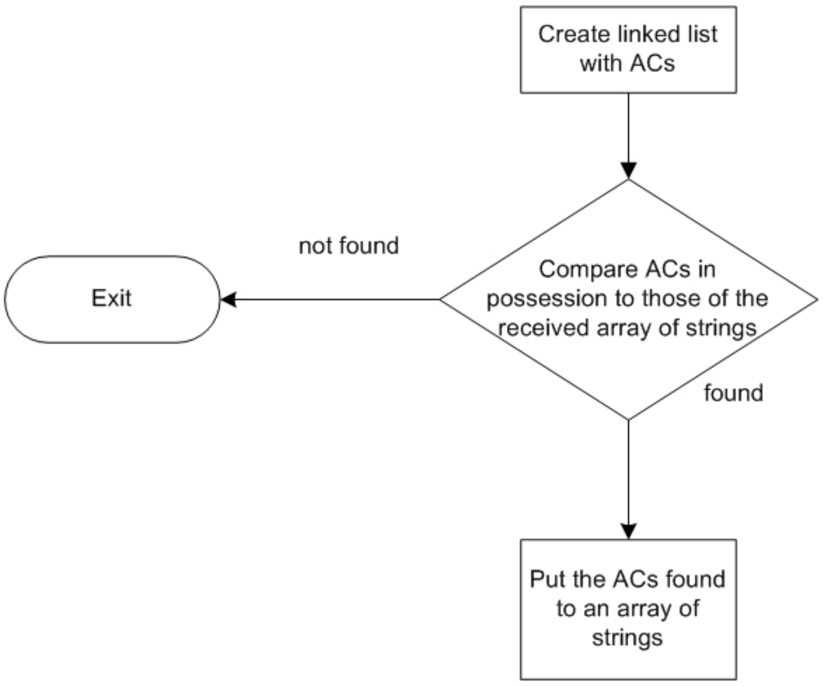

Create linked list with ACs
Compare ACs in possession to those of the received array of strings
not found
Exit
found
Put the ACs found to an array of strings

**Figure 5-4: `serviceAreq` client side**

Then the client creates a new array of strings using as size the number of ACs found in the previous step and starts searching again the linked list for the elements in the reduced ACL but this time when one is found, it is inserted in the newly created array. The client after finishing this task is able to continue to the next step which is making the call to the service function itself.

## 5.5 Implementation of the Service function

The service function is called `serviceA`. This remote method takes as input an array of strings that contains the clients ACs converted to strings and returns the file or an error message. The file sending used gSOAP's feature of streaming techniques for DIME binary attachments.

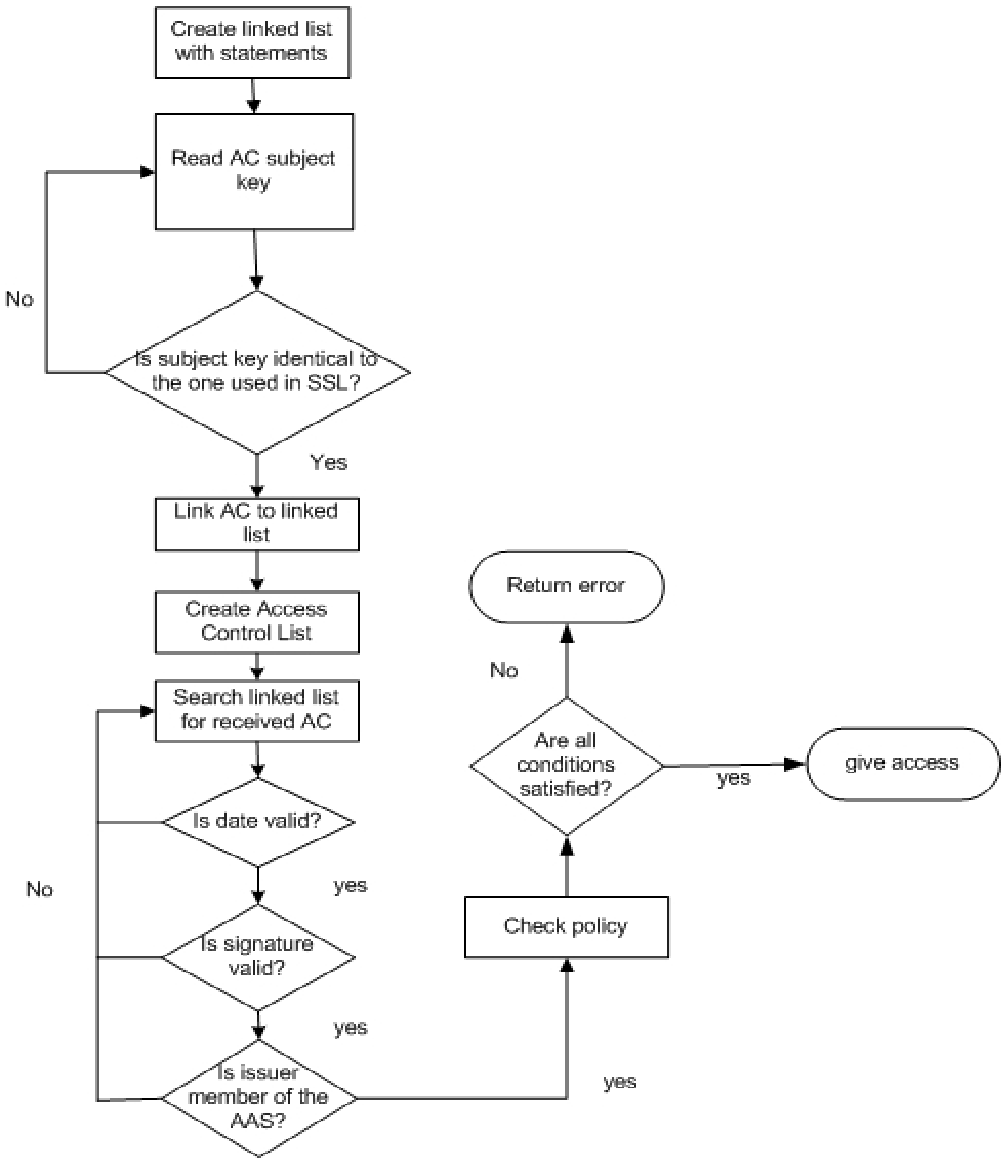
Direct Internet Message Encapsulation (DIME) is a specification method for sending and receiving SOAP messages along with additional attachments, like binary files, XML fragments, and even other SOAP messages. Usually binary attachments in SOAP are encoded as base64 XML and included in the SOAP message body but this approach has many difficulties. These issues are performance related when the size of the attachment is large and the situation becomes complicated when the attachment is digitally signed. DIME works by creating references on the SOAP messages showing to the attachments that are sent along the SOAP message using special tags to include metadata. This way the data doesn't have to be encoded in order to be sent. DIME is a similar approach to MIME but is less complex thus more performance oriented. The data type defined in the header file for DIME attachment is:

```
class ns__Data
{ unsigned char *__ptr; /* points to data */
  int __size;           /* size of data */
  char *id;             /*dime attachment ID */
  char *type;           /* dime attachment content type */
  char *options;  /* dime attachment options (optional) */
  ns__Data();
  struct soap *soap;    /* soap context that created this instance */
};
```

And the function prototype for `serviceA` is:

```
ns__serviceA(struct ArrayOfstring _inputStringArray,ns__Data &image);
```

The implementation of `serviceA` (figure 5-5) function that resides on webaetherserver.cpp source file, first reads the directory that contains all the statements that the server has and creates a linked list that includes them. Then in order to prevent man in the middle attacks it links to the list only the received ACs from the client that have as subject public key the same one that was used on the establishment of the SSL channel. Next it searches the linked list for policy statements in order to create the access control list related to serviceA. After determining from the created access control list the number of attribute name/value pairs that need to be satisfied by the ACs that were received, the function searches the linked list for the ACs received.



Create linked list with statements
Read AC subject key
No
Is subject key identical to the one used in SSL?
Yes
Link AC to linked list
Create Access Control List
Search linked list for received AC
Is date valid?
yes
Is signature valid?
yes
Is issuer member of the AAS?
No
yes
Check policy
Are all conditions satisfied?
No
Return error
yes
give access

**Figure 5-5: ServiceA implementation**

The next step is composed of the following parts for each AC:

1. Check that the received AC's date shows that the statement has not expired.
2. Verify that the signature in the statement is originated from the issuer.
3. Check if the issuer of the AC statement is a member of the AAS related to serviceA
4. If all of the above requirements are met, the function finds the logical operators in the access control list that was created in the previous steps and compares each of the name/value pairs in the access control list to the pair

found in the AC using the logical operators. If a name/value condition is satisfied, the number of remaining for satisfaction pairs is decreased and the function continues the whole operation for the next AC that was received.

The operation ends when all the ACs are checked and there are still not satisfied conditions or until all the conditions are met. If the latter happens the function changes a status indicator that shows the client has gained access to the file and from this point reads the file to be sent from the disk and uses it as serviceA output. If the status indicator shows that the condition is not met a SOAP error message is returned instead.

## 5.6 Implementation of Certificate Acquisition

Certificate acquisition is implemented also as a CGI web service called attribute.cgi. The service's main components are two functions named `attributeReq` and `attributeSend` and their prototypes are defined in the header file attribute.h. In the implementation of the certificate acquisition, first the administrator of the service has to decide the attribute certificates he is going to issue to the client according to the specification on the paragraph 4.5.2 of the design chapter. Then he is going to send the temporary URL that is going to be used to the client. The first remote method, `attributeReq` is being used by the client to initiate the service. It has no input and the output is an array of strings containing the name and value of the attribute that is going to be sent from the server. The function prototype along with the data type array of strings defined in the header file is:

```
typedef char *xsd__string;
struct ArrayOfstring {xsd__string *__ptr; int __size; int __offset;};

ns__attributeReq(struct ArrayOfstring &_return);
```

The implementation of the function, contained in the source file attributeserver.cpp, first creates a new array of strings and then inserts the name and value of the AC he is willing to send. This array is used as the output of the function.

On the client side found in the source file attributeclient.cpp, the client creates a new array of strings and inserts the name and value that has received and along with them his public key (converted in hexadecimal format in order to facilitate the server). The function `attributeSend` as expected takes as input an array of strings which is going to be the one that the client has created in the last step of the call to the previous function. The function prototype defined in the header file is:

```
ns__attributeSend(struct   ArrayOfstring   _inputStringArray,   struct
ArrayOfstring &_return);
```

The implementation of the function in attributeserver.cpp receives the array of strings from the client and creates a new attribute certificate that contains the name/value pair that was decided, checks that the public key that was received is the same to the one the client used in the establishment of the SSL channel and inserts it as the AC's subject field. Then it includes the validity period that is decided and parses the created statement in order to make sure that it is correctly structured and signs it. After signing the AC, the function converts the statement to a string and uses it as output. On the client side found in attributeclient.cpp, the client uses as input to the `attributeSend` function the array of strings that has created in the previous remote method call and receives a signed AC in string format as output that he is able to save as a file in the repository that he keeps all the ACs in his possession.

## 5.7 Summary

In this chapter was described the implementation of a file transfer web service that uses WebÆTHER. The implementation consists of the service query and the service remote methods along with the appropriate setup of the environment that uses Apache web server and gSOAP toolkit.
Finally another service handling attribute certificates distribution composed of a request for initializing the service and the service method itself was described in this chapter. The evaluations of the functions that compose the web service along with the evaluation of the whole protocol used are in the following chapter.

# 6 Evaluation

## 6.1 Overview

This chapter describes the tests that were performed in order to evaluate the performance of the WebÆTHER implementation. In the tests that were performed the server was a Dell Latitude D400 laptop equipped with a mobile Pentium 1.6 processor and 512Mbytes of RAM. The operating system used in the server was Fedora Core 4 Linux.

On the client side was an Acer Aspire 1350 laptop equipped with an AMD mobile Athlon XP 2400 processor and 256Mbytes of RAM. The operating system used on the client side was KNOPPIX 3.9. The network that was used in the experiments was a 100Mbps LAN and the file that was used in the file transfer was a 10.4Mbyte image in JPEG format. The measurements were made using three functions that implemented gettimeofday along with the following static structures:

```
static struct timeval _tstart, _tend;
static struct timezone tz;
```

The first two functions get the current time and store it in the static structures:

```
void tstart (void)
{
gettimeofday(&_tstart,&tz);
}

void tend (void)
{
gettimeofday(&_tend,&tz);
}
```

The next function was used in order to compute the difference in time between the two static values that were taken and convert it to milliseconds:

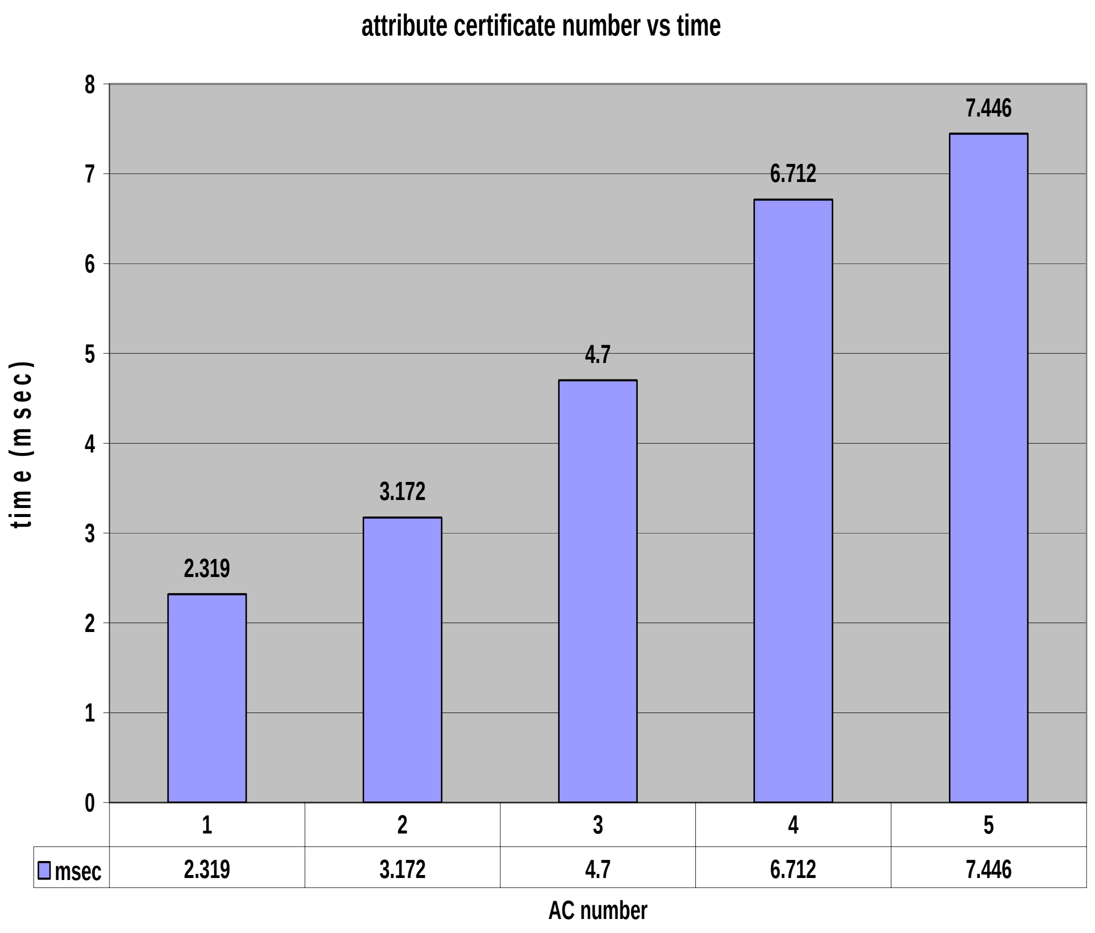
```
double tval()
{
double t1, t2;
t1 = (double)_tstart.tv_sec + (double)_tstart.tv_usec/(1000*1000);
t2 = (double)_tend.tv_sec + (double)_tend.tv_usec/(1000*1000);

return (t2-t1)*1000;

}
```

The measurements were made by carefully placing calls on `tstart` and `tend` functions at the start and at the end of the blocks of the code that needed to be measured. This stored the timestamps at the beginning and the end of the operations. Next a call to the function `tval` provided the difference in milliseconds between the two timestamps. The function `tval` is able to return the result also in seconds if the multiplication with 1000 is omitted from the last line of the function.

## 6.2 Performance of `serviceA` function

The performance of the function `serviceA` which is the function that provides the main ÆTHER functionality and is responsible for handling the statements and come to a decision is measured in this paragraph.
The first test in order to get an evaluation for the function’s performance was measuring the time it takes to handle a different number of required attribute certificates from the point of receiving the array of strings that includes them to the point that a decision on giving access is reached (figure 6-1).
This includes the process of reading each certificate, checking date and signature, determining if the issuer is a member of the corresponding AAS and making a decision on the policy that concerns the service. The function `serviceA` was described in detail on paragraph 5.5 of this dissertation.

The measurements were taken for handling from one to five attribute certificates and the average time in milliseconds for each case is shown in figure 6-1. Each test was repeated one hundred times in order to get the average time.



|  | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| ▢ msec | 2.319 | 3.172 | 4.7 | 6.712 | 7.446 |

**Figure 6-1: attribute certificate number versus time**

The second test in order to measure `serviceA` function's performance was related to delegation depth. The scenario that was chosen for this experiment was a case of threshold value equal to 2 and various delegation depths in order to show how delegation affects the performance of this function. The number of verifications needed for each case of delegation depth with a maximum threshold value of two can be seen on figure 6-2.

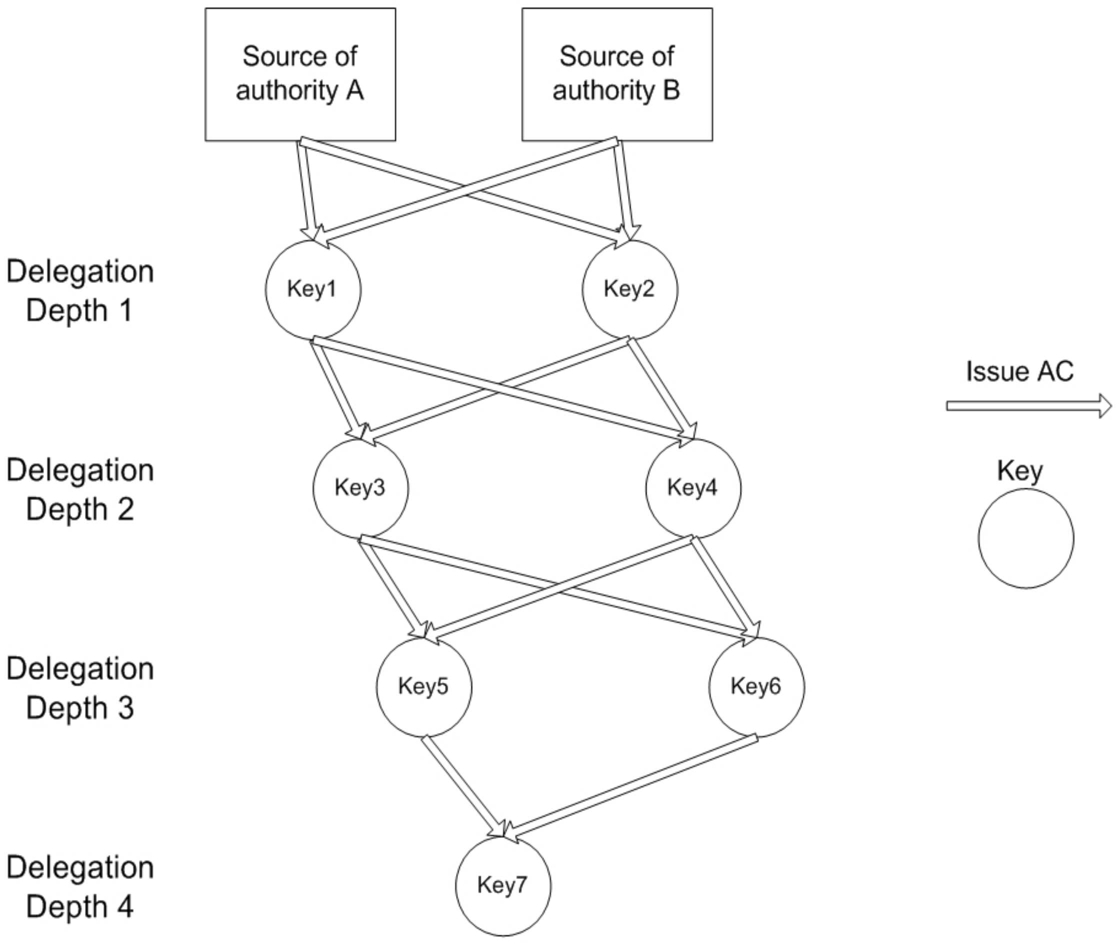
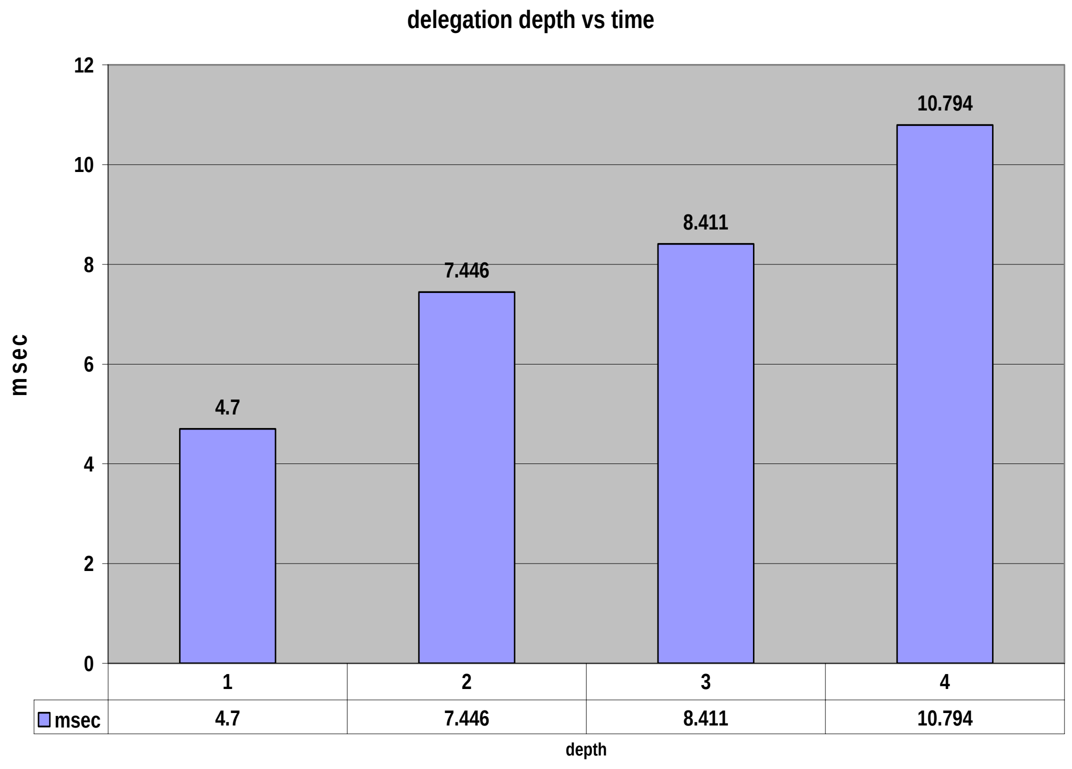


**Figure 6-2: delegation depth using threshold value 2**

The test was performed using delegation depths from one to four. In step one, the keys 1 and 2 get an AC from each of the sources of authority. That means that because we have set the threshold value to 2, they join dynamically the AAS and they are able to also issue valid ACs. The same step continues up to the maximum specified depth in the AAS statement. Each time a key gets one AC from two different keys that have already been inserted to the AAS they dynamically become members. For each case the verifications included the sources of authority keys along with the dynamically inserted to the AAS keys. This meant 3, 5, 7 and 9 attribute verifications respectively. The measured time in milliseconds for each case is illustrated on figure 6-3.



|  | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| msec | 4.7 | 7.446 | 8.411 | 10.794 |

**Figure 6-3: delegation depth versus time for threshold value 2**

## 6.3 System performance

The next set of tests is intended to evaluate the performance of the whole system including the protocol and the transport layer. The set is composed of four tests in order to illustrate the performance of the separate parts composing the file transfer web service implementation. The scenario chosen for the tests is serviceA controlling access to the 10.4Mbyte file using WebÆTHER. This can be viewed as a composition of SSL, gSOAP and ÆTHER core functionality. The policy that was selected for the set of tests requires the client to have two specific attribute certificates. The attribute certificates were directly acquired by the server which means there is no delegation involved in the verifications.

The first test in the set illustrates the performance of the handshake between the client and the server, which includes SSL establishment, a call to `serviceAreq` in order to receive the reduced ACL (implementation details can be found in paragraph 5.4), a call to `serviceA` remote method in order to send the attribute certificates that are required and the whole set of operations of `serviceA` until the decision is made but it doesn't include the sending of the file. The average time that was computed in 100 performed measurements was 272.812 milliseconds.

The next test was a client meeting the requirements of the policy statement that would eventually get the file. This test shows the overall performance of the system for a 10.4Mbyte file transfer using SSL, gSOAP and the trust management functionality provided by the ÆTHER implementation combined. The result was an average of 2014.51 milliseconds for the whole operation. Figure 6-4 presents the handshake compared to the complete implementation.

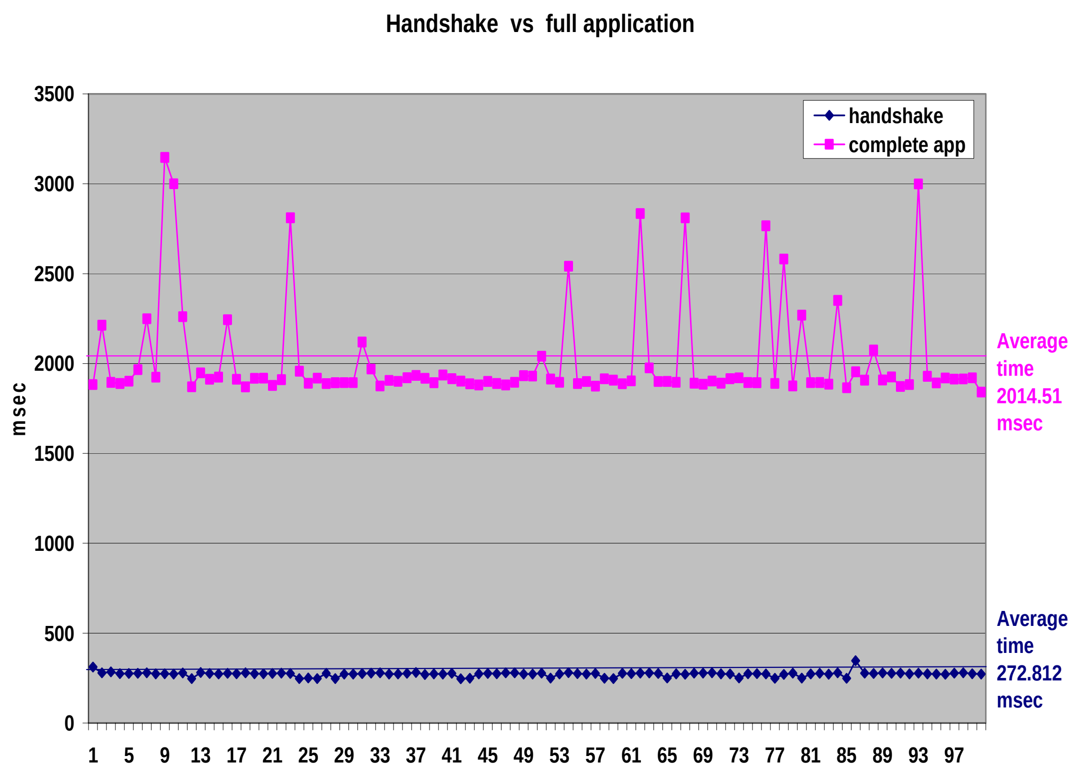


**Figure 6-4: handshake compared to complete implementation**

In order to get a better view of the impact on performance of each part consisting WebÆTHER, first the file transfer was measured without the ÆTHER functionality, thus consisting only of a file transfer using SSL and gSOAP.
The result was an average time of 1891.809 milliseconds. This means that the operation not including the trust management layer was 122.701 milliseconds faster. The result of this measurement showed that the performance drop was not in analogy to the measurement of the handshake that was found to be 272.812 milliseconds and this meant that the use of SSL had significant importance for the performance results. The inference engine overhead is illustrated in figure 6-5 by comparing the full application to the case when the ÆTHER functionality is stripped.

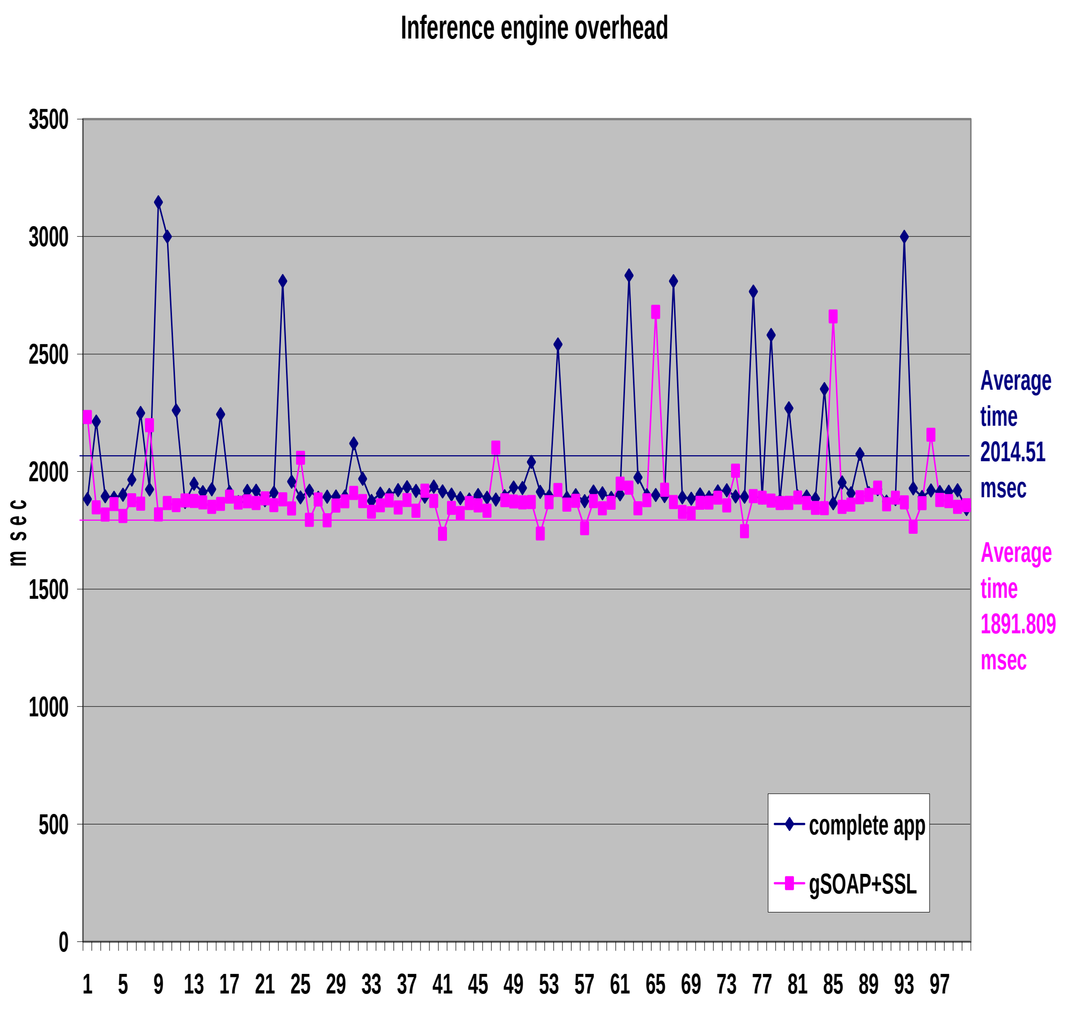


**Figure 6-5: Inference engine overhead**

It was decided to perform another test to get a better image of the system performance. In order to evaluate the impact in the performance from the use of SSL in the transport layer, the test had to be performed without using SSL. This final test was making the file transfer using only gSOAP and resulted in an average time of 1194.949 milliseconds for the same file used in all the previous measurements. The difference from the experiment using both gSOAP and SSL was 696.86 milliseconds.

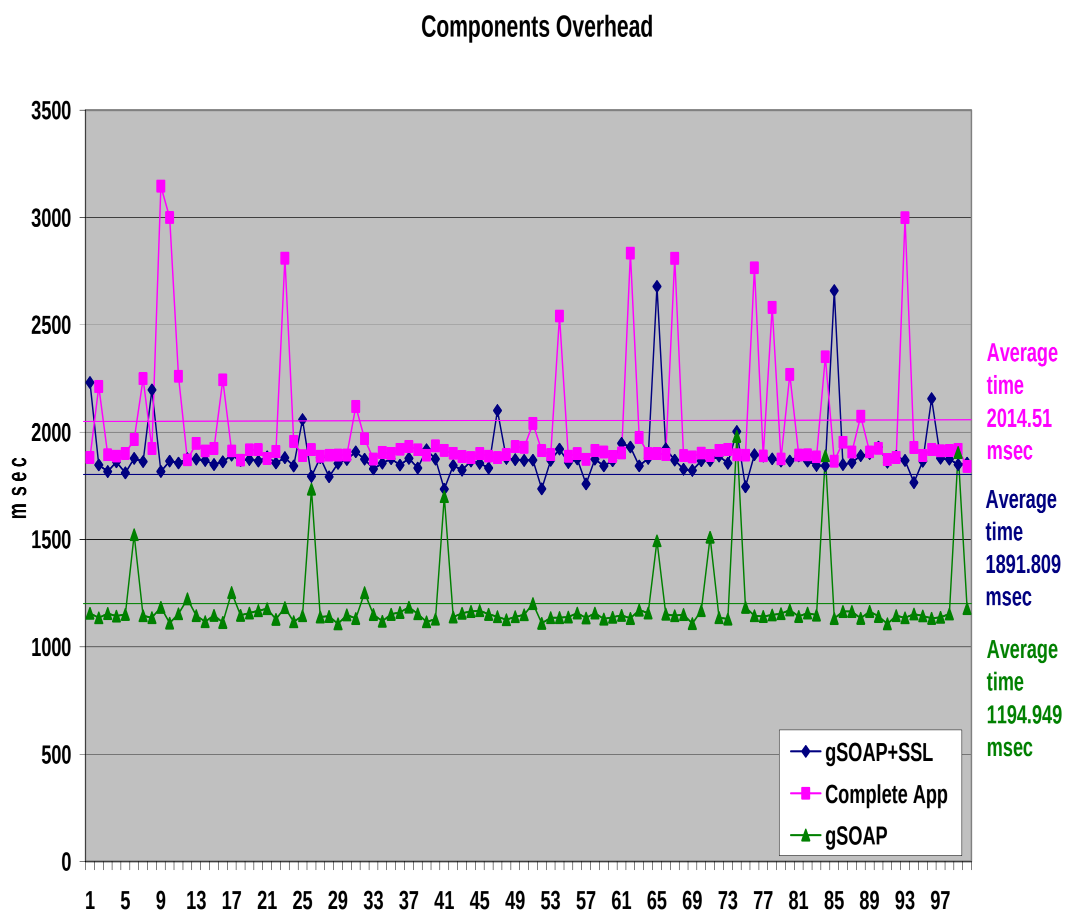
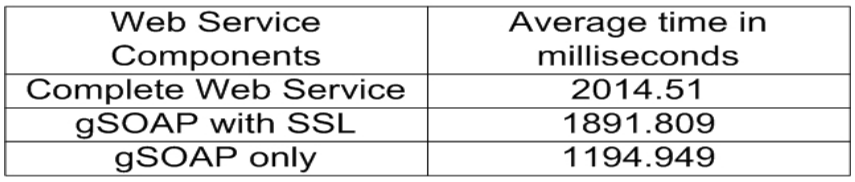


**Figure 6-6: components overhead**

Figure 6-6 illustrates the time that was measured in each of the tests for the file transfer presenting this way the overhead that each component introduces to the overall system performance.

## 6.4 Summary

This chapter presented the tests that were performed in order to evaluate the components of the WebÆTHER implementation. The conclusion of the tests is that the most serious impact on performance in the implementation comes from the use of SSL. When the ÆTHER functionality was removed from the web service, the performance gain was computed to be 6% in the 10.4 megabyte file transfer. This means that the percentage of the trust management layer impact on performance becomes less important as larger files are being used for transfer. The impact of SSL use in the transport layer was expected to be high and the tests of a web service using only gSOAP showed a gain of 36.8% to performance compared to a web service file transfer using gSOAP and SSL without the ÆTHER functionality and a significant 40.7% compared to the complete WebÆTHER implementation (figure 6-7).
While the impact of using SSL for file transfers is a topic that could be debated we believe that the overall performance of the system is considered to be sufficient for use in real world scenarios.



| Web Service Components | Average time in milliseconds |
|---|---|
| Complete Web Service | 2014.51 |
| gSOAP with SSL | 1891.809 |
| gSOAP only | 1194.949 |

**Figure 6-7: results of tests**

# 7 Conclusions & Future Work

In the first chapter of this dissertation in order to meet the primary goal which was the design and implementation of a trust management system for use in the web services area, four objectives were identified.

The first objective was to understand in depth the current technologies and philosophy in web services security. Chapter 2 of this dissertation presented an extensive review of the technologies, standards and protocols involved and in Chapter 3 the disadvantages of these approaches were described.

The second objective was to investigate the application of trust management systems in web services. Chapter 3 described how these systems handle the biggest problem of traditional centralized mechanisms which is scaling and offered examples of real world scenario and how two of the available trust management systems would respond if they had been used. This Chapter also described the reasons why ÆTHER is appropriate to handle the requirements of the web services environment.

The next objective was the definition of an appropriate for web services trust management model, which was carried out in Chapter 4. In Chapter 4, the design that was based on ÆTHER was described in detail and examples presenting the responding of the design to a real world scenario were presented in order to provide a better understanding of the choices that were made.

The final objective was to implement and measure the performance of the created system in order to demonstrate its ability to be used in real world scenarios providing good results. In Chapter 6 a series of measurements of the components of the implemented system was carried out showing the capabilities of the system for real world use.

Below is some suggested further work that could be completed as a development for this project:

- Create an Apache web server module to provide WebÆTHER framework instead of using specific for each web services cgi applications. This along with the development of a graphical user interface for making the administration of the system easier could result in the adaptation of the system in a large series of web services applications.

- Develop an automated tool for creating, deploying and deleting temporary URLs for use in certificate acquisition along with a GUI for the administrator of the server and for the user.
- Investigate the use of XML encryption and the new kinds of web services that could be implemented using it.
- Optimize the use of SSL to reduce the impact on performance it has.
- Research the validity time period that is suitable for major kinds of web services along with the investigation of ways to reduce the administrative overhead of OCSP so it can be used in WebÆTHER.
- Research the implementation of ÆTHER Context Attributes in WebÆTHER in order to provide access control based on context rather than attributes. For example limiting access to a specific IP range will be easy to implement using Context Attributes.

# 8 Bibliography

[1] Kerberos: The Network Authentication Protocol
http://web.mit.edu/kerberos/www/
[2] The Biometric Consortium
http://www.biometrics.org/
[3] Public-Key Infrastructure (X.509) (pkix)
http://www.ietf.org/html.charters/pkix-charter.html
[4] Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, R. Housley, W. Polk, W. Ford, D. Solo, April 2002, http://www.ietf.org/rfc/rfc3280.txt
[5] An Internet Attribute Certificate Profile for Authorization, S. Farrell, R. Housley, April 2002, http://www.ietf.org/rfc/rfc3281.txt
[6] PGP: Pretty Good Privacy, Simson Garfinkel, 1st Edition, December 1994
[7] OpenPGP Message Format, J. Callas, L. Donnerhacke, H. Finney, R. Thayer, November 1998, http://www.ietf.org/rfc/rfc2440.txt
[8] D.F. Ferraiolo and D.R. Kuhn, "Role Based Access Control" 15th National Computer Security Conference (1992), http://csrc.nist.gov/rbac/Role_Based_Access_Control-1992.html
[9] Trusted Computer Security Evaluation Criteria, DOD 5200.28-STD. Department of Defence, 1985
[10] IT Web Services: A roadmap for the Enterprise, Alex Nghiem, 1st edition, October 2002, chapter 2
[11] SOAP Version 1.2 specification, http://www.w3.org/TR/soap12
[12] Architectural Styles and the Design of Network-based Software Architectures, Roy Thomas Fielding, Chapter 5
[13] (Extensible Markup Language) XML-Signature Syntax and Processing, D. Eastlake et al, March 2002, http://www.ietf.org/rfc/rfc3275.txt
[14] XML Encryption Syntax and Processing, W3C Recommendation December 2002, T Imamura et al, http://www.w3.org/TR/xmlenc-core/
[15] OASIS SAML Specification,

http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf
[16] OASIS XACML Specification,
http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf
[17] XML Key Management Specification (XKMS 2.0) Version 2.0
W3C Recommendation 28 June 2005,
http://www.w3.org/TR/2005/REC-xkms2-20050628/
[18] Web Services Security: SOAP Message Security 1.0,(WS-Security 2004)
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf
[19] Web Services Policy Framework (WS-Policy), September 2004
ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf
[20] Web Services Trust Language (WS-Trust),
http//www.verisign.com/wss/WS-Trust.pdf
[21] M. Blaze, J. Feigenbaum and J. Lacy. Decentralized trust management. In Proceedings of the 1996 IEEE Symposium on Security and Privacy, pp. 164-173, 1996.
[22] The KeyNote Trust-Management System Version 2, M. Blaze, J. Feigenbaum, J.Ioannidis, A. Keromytis, September 1999,
http://www.cis.upenn.edu/~angelos/Papers/rfc2704.txt
[23] Securing Communications in the Smart Home, Patroklos G. Argyroudis and Donal O'Mahony, In Proceedings of 2004 International Conference on Embedded and Ubiquitous Computing (EUC'04), LNCS 3207, Springer-Verlag, pp 891-902, 2004
[24] Authorization management for pervasive computing, Patroklos G. Argyroudis, phd thesis, Trinity College Dublin
[25] X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP, M. Myers et al, June 1999, http://www.ietf.org/rfc/rfc2560.txt
[26] Apache Web Services project – AXIS, http://ws.apache.org/axis/
[27] SOAP::Lite for perl, http://soaplite.com/
[28] Programming Web Services with SOAP, James Snell, Doug Tidwell, Pavel Kulchenko, 1st Edition December 2001, Chapter 3 - Writing SOAP Web Services
[29] gSOAP SOAP C++ Web Services, http://gsoap2.sourceforge.net/