Providing Web Services Security:
A Case Study on Multiparty Service Delivery

Joana Sequeira Torreira da Silva

A Thesis

In

The Department

Of

Electrical and Computer Engineering

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montreal, Canada

February 2005

© Joana Sequeira Torreira da Silva, 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-04394-6
Our file Notre référence
ISBN: 0-494-04394-6

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

ABSTRACT

Providing Web Services Security:
A Case Study on Multiparty Service Delivery

Joana Sequeira Torreira da Silva

Web Services Security, used to ensure data integrity and confidentiality, has been so far provided and/or implemented as a library or a proxy. Both means present serious drawbacks, be it in terms of flexibility, level of abstraction, dynamic coupling and others.

In this thesis we propose the use of Web Services to enforce Web Services Security. Available to both Web Service requestors and providers inside and outside its domain, Security Web Services can provide Web Services Security to many entities, including those that due to limited resources would not be normally able to secure their communications. Specific interfaces were devised for the proposed Security Web Service, giving access to granular security, through a high level interface, without pre-configurations.

We also present a Case Study where each entity, participating in a three party transaction, uses either libraries or a Security Web Service to secure their messages. Performance measurements were conducted, both in terms of time delays and network loads.

Despite the delay and the network load introduced, Security Web Service are an attractive method of providing WSS due to the high level of level and granularity provided, as well as the reduced local computation load (for encryption and signatures).

Acknowledgements

I wish to express my gratitude to my supervisors Dr. Ferhat Khendek and Dr. Roch Glitho for their support, guidance and great patience. Dr. Khendek, thank you for believing in me from the beginning to the end and for pushing me to do and be the best I can be. Your dedication and kindness, both as my teacher and supervisor, will always be remembered. Dr. Glitho, thank you for giving us a different and more business oriented view of our work. Your comments, advice and ideas proved not only useful but provided all of us with a new learning experience. Thank you so very much.

To my fellow students, past and present, you are the greatest team I've worked with. On common projects, each one of us pulled our load, making this adventure possible. When on different projects, you still had time to support and help me when I most needed it. For this and so much more, I thank you.

I would also like to acknowledge funding from the Concordia Research Chair in Telecommunications Software Engineering, and Ericsson for making their facilities and resources available to us.

Last but not least, I would like to thank my parents for their never wavering support. You have witnessed this adventure from the very beginning, always encouraging and never doubting me. You showed me to hold on to my objectives, bearing the unavoidable setbacks one encounters and making the most of my successes. You are an inspiration to me and I shall always be grateful for your guidance.

Table of Contents

LIST OF FIGURES .... **IX**

LIST OF TABLES .... **XI**

LIST OF ACRONYMS AND ABBREVIATIONS .... **XII**

CHAPTER 1 INTRODUCTION **1**

1.1 RESEARCH AREA OVERVIEW 1

1.2 PURPOSE OF THE THESIS 3

1.3 ORGANIZATION OF THE THESIS 5

CHAPTER 2 REVIEW OF WEB SERVICES AND DEPLOYMENT PATTERNS **6**

2.1 WEB SERVICES 6

2.1.1 Web Service Characteristics *7*

2.1.2 Web Services Application Domains *8*

2.2 DEPLOYMENT PATTERNS 8

2.2.1 The Routing Pattern *9*

2.2.2 The Gateway Pattern *10*

2.2.3 The Proxy Pattern *11*

2.2.4 The Interceptor Pattern *11*

2.2.5 The Adapter Pattern *12*

2.2.6 The Delegate Pattern *13*

2.2.7 The Filter Pattern *13*

2.2.8 The Orchestrator Pattern *14*

2.2.9 The Referral Pattern *15*

2.2.10 The Sequence Pattern *16*

2.2.11 The WorkFlow Pattern *17*

CHAPTER 3 WEB SERVICES SECURITY: THE STATE OF THE ART **19**

3.1 WEB SERVICES AND SECURITY 19

3.1.1 Authentication *20*

3.1.2 Authorization/Access Control *20*

3.1.3 Data Integrity *21*

3.1.4 Data Confidentiality/Privacy *21*

3.1.5 Mitigation of Denial of Service Threat *22*

3.1.6 Non-Repudiation *22*

3.2 SECURITY LEVELS 22

3.2.1 Transport Level Security *23*

3.2.2 Message Level Security *24*

3.2.3 Application Level Security *25*

3.2.4 Security Level Conclusions *26*

3.3 WEB SERVICES SECURITY: STANDARD AND TECHNOLOGIES 27

3.3.1 XML-enc *27*

3.3.2 XML-DSig *28*

3.4 PROVIDING WEB SERVICES SECURITY 28

3.4.1 The Library *28*

3.4.2 The Proxy/Gateway and Application Servers *30*

3.5 CONCLUSION 31

CHAPTER 4 SECURITY WEB SERVICES **33**

4.1 THE GLOBAL ARCHITECTURE 33

4.2 PROPOSED INTERFACES 39

4.2.1 The elementSecurity Interface *40*

4.2.2 The secureSendRemove Interface *45*

4.3 DEPLOYMENT OF SWS 49

4.4 ISSUES ON SWS 50

4.5 CONCLUSION 51

CHAPTER 5 A CASE STUDY 53

5.1 THE CUSTOMER-STORE-BANK CASE STUDY 53

5.2 SECURED CUSTOMER-STORE-BANK APPLICATION 55

5.3 IMPLEMENTATION 57

5.3.1 Main Issues 58

5.3.2 The Components 59

5.3.2.1 The SWS 59

5.3.2.2 The Customer 60

5.3.2.3 The Store 60

5.3.2.4 The Bank 61

5.3.3 The Keys 62

5.3.3.1 The Signature Key 62

5.3.3.2 The Encryption Key 62

5.4 THE TESTBED 63

5.5 THE TESTS 64

5.5.1 Individual Testing 64

5.5.2 Chain Testing 65

5.5.3 Full Chain Testing 67

5.6 PERFORMANCE MEASUREMENTS 68

5.6.1 Delay Overhead: Results and Analysis 69

5.6.2 Network Load Overhead: Results and Analysis 72

CHAPTER 6 CONCLUSIONS AND FUTURE WORK 74

6.1 CONTRIBUTION OF THIS THESIS 74

6.2 FUTURE WORK 77

CHAPTER 7 REFERENCES 78

List of Figures

Figure 1: Web Services - Registries, Providers and Requestors 7

Figure 2: The Routing Pattern 9

Figure 3: The Gateway Pattern 10

Figure 4: The Proxy Pattern 11

Figure 5: The Interceptor Pattern 12

Figure 6: The Adapter Pattern 12

Figure 7: The Delegate Pattern 13

Figure 8: The Filter Pattern 14

Figure 9: The Orchestrator Pattern 15

Figure 10: The Referral Pattern 16

Figure 11: The Sequence Pattern 17

Figure 12: The WorkFlow Pattern 17

Figure 13: Security Scope when spanning two Web Services 24

Figure 14: General Settings 35

Figure 15: Security Web Services - The Service Requestor Point of View 37

Figure 16: Security Web Services - The Service Provider Point of View 38

Figure 17: Security WS elementSecurity Interface 40

Figure 18: Example of a Security Web Service - Routing Pattern 45

Figure 19: Security WS secureSendRemove Interface 47

Figure 20: Use Case 53

Figure 21: Secured Use Case 56

Figure 22: The Store's Purchase Interface 61

Figure 23: The Bank's Debit Interface 61

Figure 24: DSA Key Parameters 63

Figure 25: Individual Testing using SWS 65

Figure 26: Chain Testing using the elementSecurity Interface 66

Figure 27: Chain Testing using the secureSendRemove Interface 67

Figure 28: Full Chain Testing using secureSendRemove interface 68

List of Tables

Table 1: Average Delay Overhead 69

Table 2: Average Network Load Overhead 72

List of Acronyms and Abbreviations

3G:	3rd Generation
BPEL4WS:	Business Process Execution Language for Web Services (BEA, IBM, Microsoft)
BPML:	Business Processing Modeling Language
BPSS:	Business Process Specification Schema
BTP:	Business Transaction Protocol (OASIS)
CORBA:	Common Object Request Broker Architecture (Object Management Group)
CPXe:	Common Picture eXchange Environment (International Imagery Industry Association)
DES:	Data Encryption Standard
DoS:	Denial of Service
ebXML:	Electronic Business XML
HTTP:	HyperText Tranfer Protocol
JDF:	Job Definition Format
OASIS:	Organization for the Advancement of Structured Information Standards
OMA:	Open Mobile Alliance
OWSER:	OMA Web Services Enabler Release (OMA)
PIP's :	Packetstar Internet Protocol Services (Lucent)
SAML:	Security Assertion Markup Language (OASIS)
SHA:	Secure Hash Algorithm
SOAP:	Simple Object Access Protocol

UDDI: Universal Description, Discovery and Integration standard

W3C: World Wide Web Consortium

WfMC: WorkFlow Management Coalition

XPDL: XML Process Definition Language (WfMC)

XML Processing Description Language

Wf-XML: WorkFlow XML (WfMC)

WS: Web Service

WSCI: Web Service Choreography Interface

WSCL: Web Services Conversation Language

WSDL: Web Service Description Language

WSEL: Web Services Endpoint Language

WSFL: Web Services Flow Language

WSS: Web Service Security (OASIS Standard)

XLANG: Web services for business process design

XML: eXtended Markup Language

XML-DSig: XML Digital SIGnature (IETF)

XML-enc: XML ENCryption

Chapter 1 Introduction

This chapter is intended to present the reader with a synopsis of the work accomplished and reported in this thesis. First, it gives the reader an overview of the research area tackled in this thesis. The problem at hand is then introduced, as well as the proposed solution. Finally, the thesis organization is presented in Section 1.3.

1.1 Research Area Overview

Easy creation, deployment and integration of new services have been a major preoccupation for a long time. Indeed, if creating and distributing a new service needs more resources, and therefore money, than the service will generate, no company will wish to create it. With 3rd generation (3G) networks [27][28], where network operators wish to make their network capabilities available to 3rd parties that will develop and host their own new services, an easy, secure and straight-forward manner of accessing and integrating applications as well as network resources is needed.

Over the years, many techniques, languages, architectures and protocols have been devised to make these steps easier and more affordable. One only has to think in the shift from procedural to Object-Oriented programming or the creation of CORBA [21]. More recently, in the telecommunication's domain, the Parlay Architecture [18][19][20] was developped. Despite their benefits, these solutions have often proved to be either incomplete (e.g. lack of security), inappropriate (domain dependency, too heavy, too slow, etc.) or too complex (requiring a considerable knowledge of a domain that represents a very small portion of the new service). Furthermore, many of these solutions

often implied a close interaction (high-coupling) between the new service/application and the service/application/resource it used.

On the other hand, Web Services (WS) loose-coupling, high-level of abstraction and use of SOAP (an XML-based protocol) seemed to be the answer to the problem. Indeed, a proper WS interface will provide a high level of abstraction, reducing coupling, which in turn eases the transition between two services (should the one being used fail or a new/better service appear). This high level of abstraction can also simplify the developers work by hiding the service architecture, requiring less domain-specific knowledge of the developer and even reducing the lines of code necessary to call a service.

So far, the main road-block to WS use across mistrusted networks has been the lack of security standards. Indeed, the lack of end-to-end, granular and durable security in WS has forced many companies to either wait for a mature and well-established standard or to develop their own proprietary solutions. An standard of the Organization for the Advancement of Structured Information Standards (OASIS) since March 2004, Web Services Security (WSS) is a standardized set of SOAP extensions, which can be used to enforce message content integrity, hence securing Web Services [4]. Through these extensions, it is not only integrity and confidentiality that can be achieved, but also authentication, non-repudiation, mitigation of Denial-of-Service (DoS) threats. Through authentication, authorization-rules and accountability can be enforced.

1.2 Purpose of the Thesis

Allowing access to network resources, services and/or applications requires the ability to secure/protect what is being exposed from unauthorized/abusive users. For users to feel safe when submitting private information (address or credit card number for billing), communication between the user/requester and the provider must also be protected. Only when satisfactory security exists for both the service requesters and the service providers can a service be successful. Although the security level necessary may vary according to users, domains and applications, security is always a concern.

So far, WSS has been provided as libraries or proxies. Security libraries are mostly directed towards developers, enabling them to secure their applications, independently of any other protection mechanisms ran by the application owner. Security proxies are mostly directed towards those running the applications so they can enforce their own security measures independently of those already applied by the application. The main problem with libraries lies in the fact that they are not easy to use, must be present wherever the application is run and are resource expensive. Proxies, on the other hand, must be pre-configured and are most of the time reduced to message level security, i.e. the whole message is secured, but no sub-parts can be secured individually. Furthermore, proxies must be configured ahead of time and require from those configuring it a knowledge that usually lies with the application developer despite the fact that these developers usually do not have access to these proxies.

This thesis inspects and analyses the use of the Web Service technology to provide WSS to both service requesters and service providers. By exposing WSS capabilities as a WS, providers and requestors are presented with a granular, accessible from anywhere, easy to use and light (no libraries to be packed) means of enforcing WSS. At the same time, the user will need fewer resources in terms of memory (no libraries) and in term of CPU power seeing that the most CPU expensive calculations (signing and encrypting) will be done remotely.

In the case study presented, high level web service interfaces were defined to allow WS developers an easy access to WSS enforcement capabilities. These interfaces were implemented as a library based Security Web Service. In order to observe and evaluate the performance of these interfaces, three applications were built: a Customer, a Bank and a Store, where the Customer purchased some item from the Store using a credit card and the Store would ask the Bank to debit a certain amount from the Customer's debit card.

On top of reducing the programming necessary by the developer to secure his application, the high level of abstraction also makes it easier for new comers to secure their applications without having to learn the all of WSS rules, nor those of SOAP messaging.

Unfortunately, the security between the SWS requester and his provider are still an issue when they belong to different domains and must be the object of further studies.

Nonetheless, the benefits derived from an easier and faster application development, even by WSS new comers, along with the greatly reduced footprint make web services an attractive method of providing WSS.

1.3 Organization of the Thesis

Chapter 2 offers an overview of Web Services and Web Services deployment patterns. Deployment patterns are important when considering security, because each pattern has a specific set of security issues associated with it and which must be addressed by Web Services deployed according to that pattern.

In Chapter 3, Web Services Security is introduced and the state of the art in Web Service Security provisioning is presented and analyzed, libraries first, then proxies.

Chapter 4 introduces Security Web Services, the architecture within which they should be used, as well as a set of comprehensive interfaces for the Security Web Services proposed in this thesis.

Chapter 5 describes the use case investigated and all of its components. Emphasis will be put on the Security Web Service interfaces and how it can be used. It then presents the prototype used and the implementation issues faced during development. Performance tests, results and analysis are also presented in this chapter.

Finally, Chapter 6 concludes this thesis by giving a synopsis of the work presented in this thesis and the lessons learned from it, as well as directions for future work.

Chapter 2 Review of Web Services and Deployment Patterns

This chapter quickly introduces Web Services and their deployment patterns (Section 2.2). While the former describes the Web Services paradigm and architecture, the latter addresses the different ways of deploying Web Services. Deployment patterns are particularly useful when considering security, because each pattern is associated with a specific set of issues.

2.1 Web Services

According to the World Wide Web Consortium (W3C) [29], one of the many standardization bodies involved in Web Services, a "Web service is a software system designed to support interoperable machine-to-machine interaction over a network"[23]. In other words, Web Services consist of services whose interfaces that can be published, discovered and bound to over a network.

Once a Web Service has been deployed, its description (as a WSDL document) can be published to a service registry (based on UDDI specifications) or distributed through files, e-mails or others. The WSDL documents provide information as to the location of the Web Service as well as to the interfaces available. Once a Web Service Requestor, user, has uploaded this document from a server or has obtained it by some other means, he can use its information to bind to the Web Service and use it, supposing no registration

is required. This relationship between registries Web Services and Users is illustrated in Figure 1 [5].

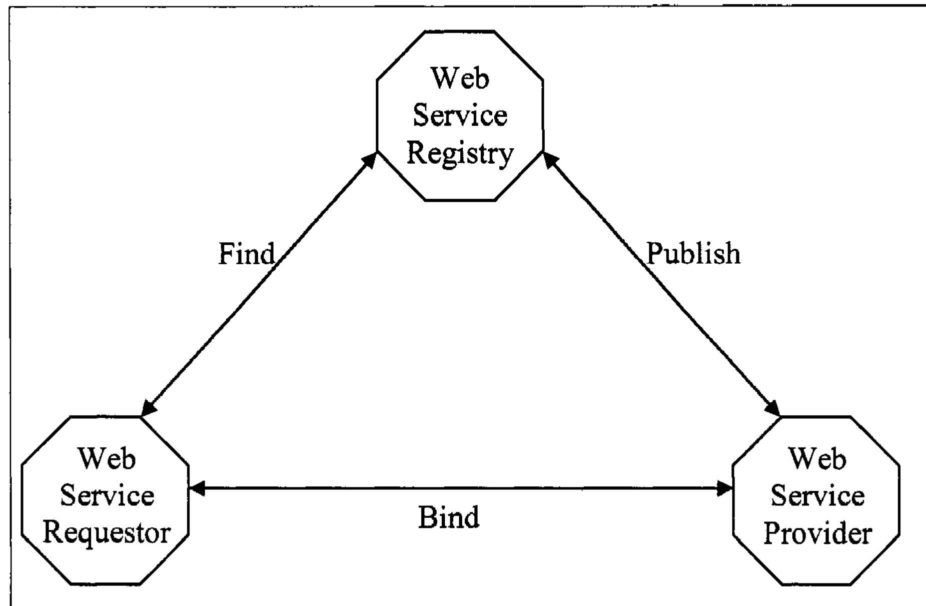

Figure 1: Web Services - Registries, Providers and Requestors

Communication with Web Services is done through SOAP, usually on top of HTTP although it could be on top of any other layer. Both SOAP messages and WSDL documents are written using the standard XML notation. For a short introduction to Web Services consult [1].

2.1.1 Web Service Characteristics

Among the many characteristics of Web Services ("small", autonomous, easily accessible pieces of software), the main ones are its coarse grained approach, loose coupling and the availability of both synchronous or asynchronous communication modes [22].

The coarse grained approach, often called high level of abstraction, allows hiding the software's architecture and simultaneously simplifies the use of the said software.

Web Services loose coupling is in part due to this high level of abstraction. This type of coupling allows users to "quickly" change from one service to another which comes in handy both when a new more performing service arrives and when the one that has been used is no longer available.

2.1.2 Web Services Application Domains

Web Services can be used in all sorts of domains such as telecommunications, imaging, maintenance, etc. For example, Boeing already uses Web Services to make their instruction and maintenance manuals available to South-Eastern Airlines. This way, they do not need to print and ship their manuals, and, whenever and update to such a manual is made, the newer version is immediately available for consultation [25]. Also, many imaging companies have come together and created the common picture exchange environment (CPXe) [24], which offers all sorts of digital imaging services.

2.2 Deployment Patterns

The Open Mobile Alliance (OMA), a standardization body mainly dealing with telecommunication networks, has identified in its OWSER Overview [5] 11 possible deployment patterns for Web Services. This Section presents each one of these patterns, their particularities and applications.

2.2.1 The Routing Pattern

The Routing Pattern consists in the use of intermediary Web Services between the Web Service Requester and the Web Service Provider, see Figure 2 [5]. Intermediaries can be used in the request message path, the response message path or in both. Request and response message paths can go through the same intermediaries or different ones, allowing the request to be sent to one Service and the response received from another. Similarly, the request may be received from one entity and the response sent to another.

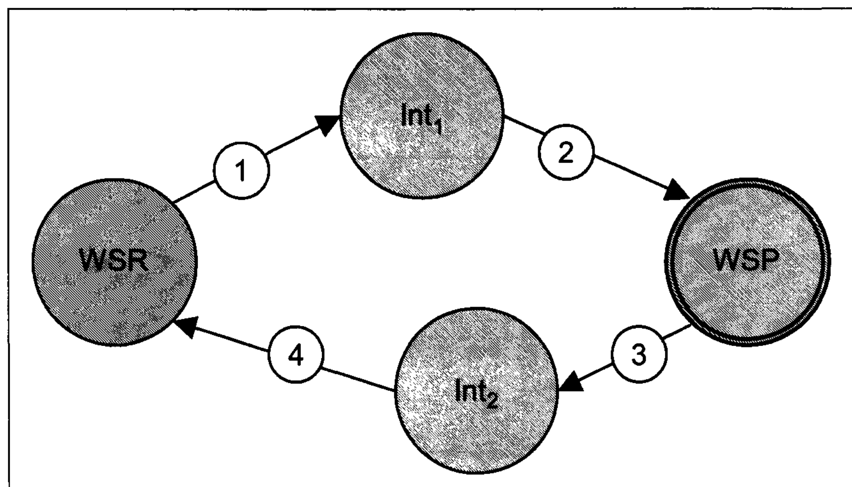

Figure 2: The Routing Pattern

The entities involved in this pattern (requester, provider and intermidiaries) may or may not know the entire path to be followed by the SOAP messages. Web Service Routing must be supported by all intermediaries, and preferably also by the endpoints (Web Service requester and provider). Due to the lack of routing standards in this domain, the Web Services thus deployed have been using proprietary solutions consisting of extensions to the SOAP message or of pre-configured profiles. Furthermore, because

related requests and messages can be sent and received from different entities, this pattern may be difficult to deploy across firewalls.

The Routing Pattern is most suitable when the intermediary services always perform the same tasks (e.g. in distributed business logic or adding/enforcing security checks) or when a specific message path should or must be used.

2.2.2 The Gateway Pattern

The Gateway Pattern shown in Figure 3 [5] consists in deploying a Gateway that within the same trusted domain as the Web Service, exposing the latter's interface at the Gateway and publishing the Gateway address instead of the Web Service's address. This pattern allows providers wishing to expose several Web Services to do so from one single address even if the Web Services are hosted on different servers and to easily change the Web Services without affecting the exposed interfaces. Furthermore, it allows one single entity, the Gateway, to provide consistent (though not necessarily the same) security as well as management features to all Web Services within providers network in relation to other mistrusted networks, while using a simpler security scheme, if any, within the provider's network, making the Web Service simpler and lighter.

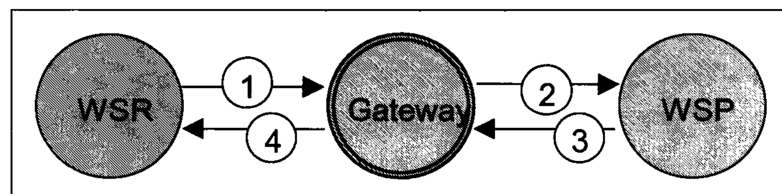

Figure 3: The Gateway Pattern

The main disadvantage of the Gateway pattern is that the Gateway must be populated with Web Services interfaces and preconfigured as to the actions to be taken upon reception of a message before it can be used.

2.2.3 The Proxy Pattern

The Proxy Pattern is used when software that is not SOAP enabled wishes to use a Web Service, as shown in Figure 4 [5]. In this case, the proxy lies in the requester's network and communicates with the former through some means other than SOAP. The proxy then translates the received request into a SOAP request that it forwards to the Web Service Provider. Upon reception of the response message from the Service Provider, the Proxy translates this into a message that the requester can understand.

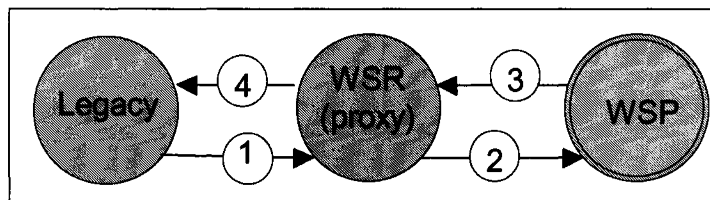

Figure 4: The Proxy Pattern

In short, the proxy poses as a service to the legacy system and as a Web Service Requester to the provider, and maps messages between both domains.

2.2.4 The Interceptor Pattern

The Interceptor Pattern, as shown in Figure 5 [5], can be both applied in the Web Service Requester and Provider's domain.

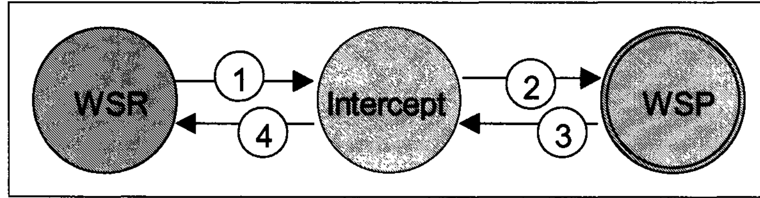

Figure 5: The Interceptor Pattern

Unlike the Gateway in the Gateway Pattern, the Interceptor does not pose as the Web Service Provider, its address not published and it does not expose the Provider's interfaces.

This pattern can be used as an "invisible Gateway" or to add service logic in a seamless manner. However, if transport level security is used, this security will not be end to end and the Interceptor may then become visible to one or both end-points.

2.2.5 The Adapter Pattern

The Adapter Pattern shown in Figure 6 [5] is simply a reversal of the Proxy Pattern explained in Section 2.2.3. Instead of allowing a legacy system to use web Services, the Adapter allows a Legacy System to be called by a Web Service Requester.

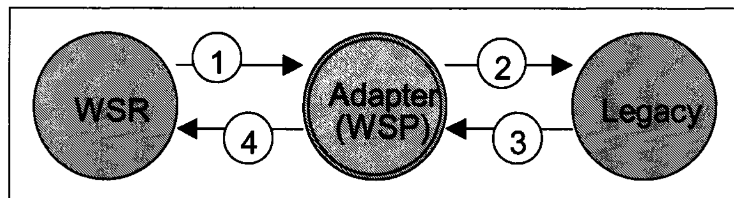

Figure 6: The Adapter Pattern

2.2.6 The Delegate Pattern

The Delegate Pattern allows a Web Service (WSP) to use a second Web Service (WSP1) in order to provide the requested service as represented in Figure 7 [5]. For security reasons, the WSP1 usually belongs to the same trusted domain as the WSP.

The Delegate Pattern is of particular interest when dealing with distributed logic, when redundant functionality exists among several services offered by the same provider as well as when a new service is created which includes functionality already available as a Web Service.

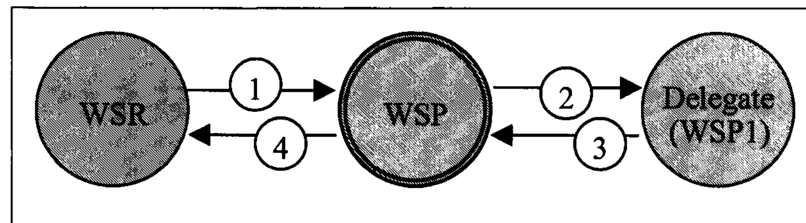

Figure 7: The Delegate Pattern

Note that usually, the majority of the logic lies in the WSP while only a small portion is delegated.

2.2.7 The Filter Pattern

The Filter Pattern discussed in this Section and shown in Figure 8 [5] is similar to the Gateway (2.2.2) and the Interceptor (2.2.4) patterns discussed earlier in that the Filter entity represents a single point where SOAP messages must be analyzed and some decision must be taken as to what is to be done with each.

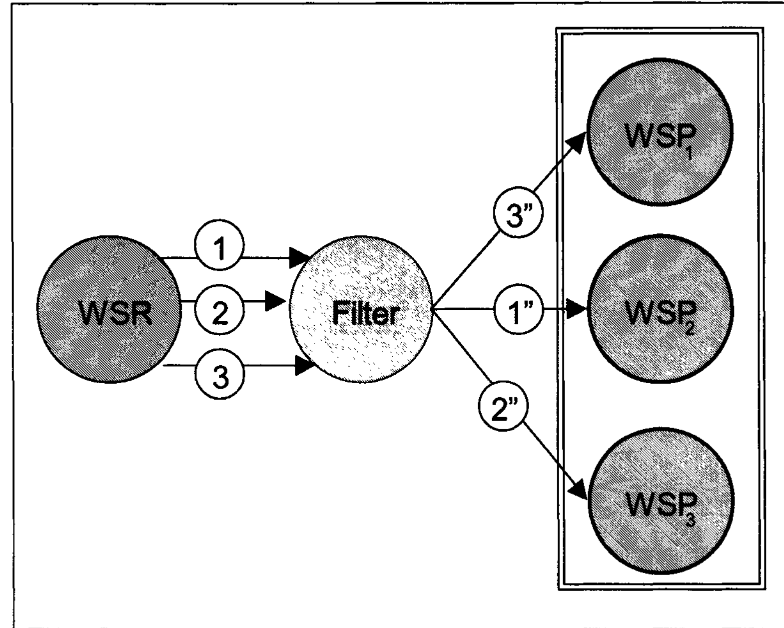

Figure 8: The Filter Pattern

Unlike the Gateway pattern that processes some part of the message before sending it to the Web Service Provider, the Filter analyses the message before deciding to which instance of the Web Service the message should be sent. This decision may be based on the load capacity of each instance, on some data context (the complete service requires several calls to the same provider and it is logical the same instance would handle all the calls), on data content (e.g. one instance only handles debits cards and the other credit cards), etc. This pattern can be easily combined with the Gateway or the Interceptor patterns.

2.2.8 The Orchestrator Pattern

The Orchestrator Pattern is similar to the Delegate Pattern (Section 2.2.6) in that it uses other Web Services, but instead of delegating a small part of its business logic, it

coordinates 2 or more services in order to offer a Value Added Service, i.e. a new service whose functionality represents more than the simple combination of the original services.

In this pattern shown in Figure 9 [5], it is the Orchestrator that is the real service (unlike the Gateway), and its address and interfaces are published.

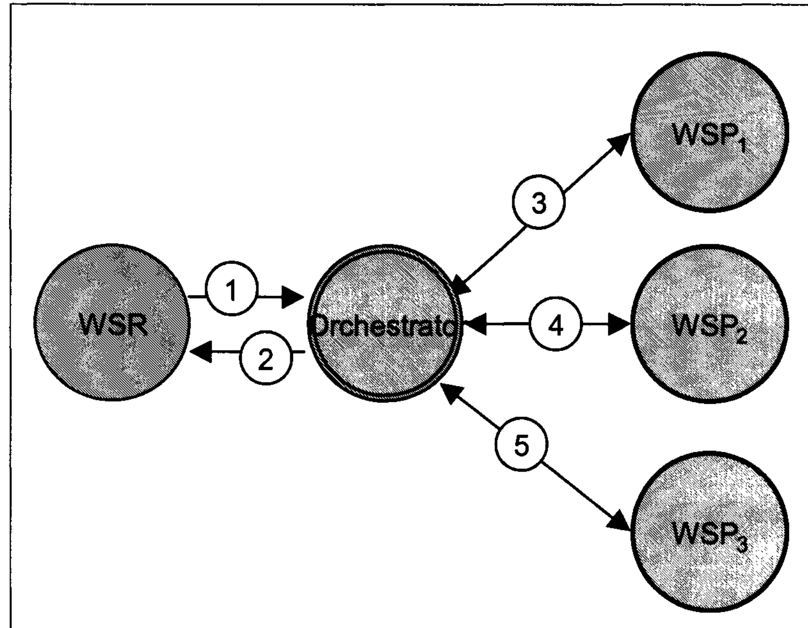

Figure 9: The Orchestrator Pattern

2.2.9 The Referral Pattern

Figure 10 [5] presents the Referral Pattern as described by OMA [5]. This patterns aims at solving the problem posed when a Web Service Requester invokes a WS and the latter recognizes that the request, although valid, is missing some information (e.g. third party authorization). The Web Service (WS1) then refers the requester to a second Web Service (WS2) which after invoking WS2 can go back to WS1 and see his request completed.

This scenario is of particular interest when third-party authentication or authorization is used by the Web Service Provider. Unfortunately, no standardized way of automatically accomplishing this is known.

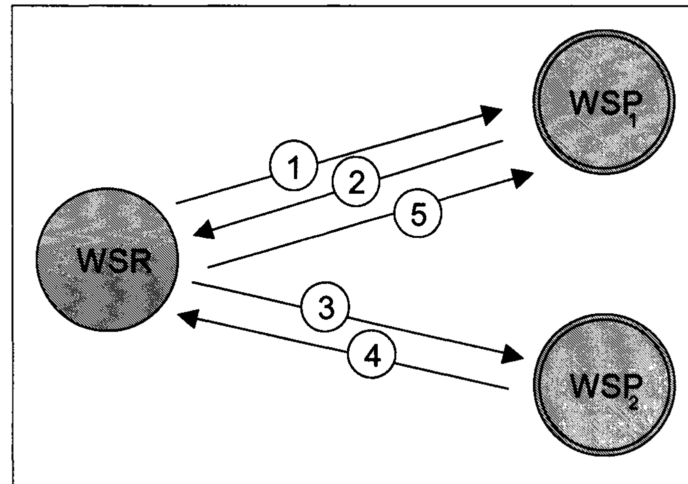

Figure 10: The Referral Pattern

2.2.10 The Sequence Pattern

Although the Sequence Pattern shown in Figure 11 [5] seems very similar to the Referral Pattern described in the previous Section, it has in fact more in common with the Orchestrator (see Section 2.2.8) than with the Referral Pattern. Indeed, after discovering the appropriate services, the Web Service Requester must then orchestrate them so as to obtain the desired Service.

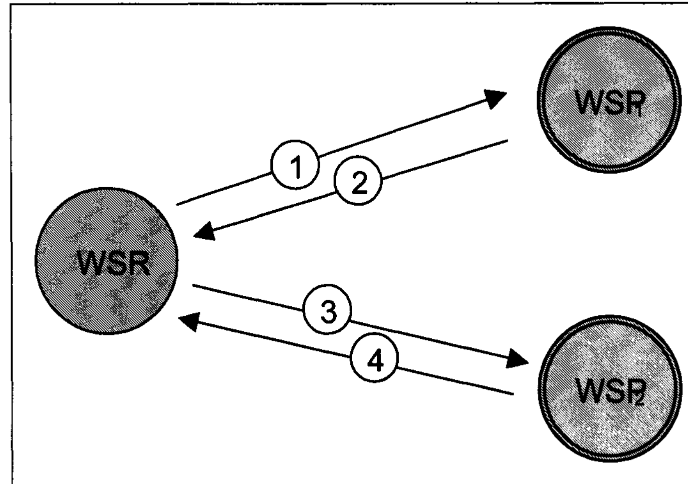

Figure 11: The Sequence Pattern

2.2.11 The WorkFlow Pattern

The Workflow Pattern represents the ability of several Web Services, spread over different networks, to work together in order to attain a specific goal. Note that in Figure 12 [5] each Web Service Provider is also a Web Service Requester.

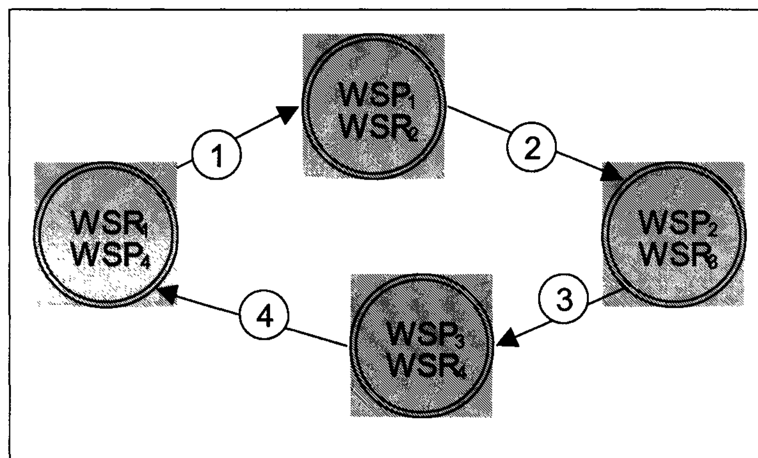

Figure 12: The WorkFlow Pattern

Although many Workflow solutions have been proposed, none of them have yet emerged as a de facto standard. These propositions include Wf-XML, WSFL, XLANG,

BPEL4WS, ebXML BPSS, WSCI, WSCL, PIP's, JDF and others. For a short description of each of these solutions see [17].

Chapter 3 Web Services Security: The State of the Art

Security is a critical component of any technology aiming to integrate applications, providing services or exposing capabilities across networks, especially when they, or the networks traversed to reach it, cannot be implicitly trusted. It is even more important when the providers wish to charge users or when users need to communicate sensitive information (e.g. credit card number) to a party (e.g. a store), so the latter can complete a transaction by communicating with another party (e.g. a bank). Although the store may need the Customer's information so the bank knows from which account to debit the amount, it is not essential that the store be able to actually read this information. Furthermore, the bank wishes to be sure that this transaction was accepted by the user.

The aim of this chapter is to provide an overview of the Web Services Security and of the means currently available to enforce it. The first section defines security with respect to Web Services. Section 3.2defines the different levels of security that can be used to secure Web Services. Section 3.3introduces the Web Services Security standard itself and 3.4the current means of enforcing the said standard.

3.1 Web Services and Security

According to OMA's OWSER Core Specifications, authentication, authorization, access control, data integrity, confidentiality, privacy, key management, security policies and the mitigation of denial of service attacks are important security services [6]. This Section describes shortly authentication, authorization/access control, data integrity, data

confidentiality/privacy, mitigation of Denial-of-Service (DoS) threat, as well as non-repudiation, the main security aspects that this thesis is concerned with.

3.1.1 Authentication

Authentication is the verification of a party's identity. This party may be the sender of a message, the receiver or the signer of data [6]. The identified entity could be a device, an application, a service requester, etc…

Authentication can be achieved through several means such as Username/Password, use of SSL/TSL, XML Digital Signatures (XML-Dsig), X.509 certificates, SAML, security proxies [12] and so on.

3.1.2 Authorization/Access Control

Although often assimilated to authentication, authorization is dependent on authentication but not equivalent. Once a party has been identified, it must be determined whether or not this party is allowed to access a resource, a service, a subset of operations or not [6].

Authorization, as opposed to authentication, does not aim at identifying the party but at determining what the party is allowed to access. Although very often combined with authentication, these two operations are different because before authorization can be decided upon, identity must be established. SOAP security proxies such as Xtradyne's [12] can handle authorization in a pre-configured fashion. SAML assertions allow third parties to convey authorization statements (using WSS SOAP extensions) and authorization rules may be defined using XAML.

3.1.3 Data Integrity

Data integrity is the property that allows a receiver to verify whether or not a message has been tempered with, whether intentionally or not [6].

It can be achieved at the transport level though SSL/TLS, but in this case, it will not be permanent. In other words, trace of this integrity cannot be preserved at the SOAP level, and should several parties be involved in a transaction, they must rely on the other parties to preserve the integrity of the original data.

Although SSL/TLS must be supported by Service Providers, the best way to secure data is through XML Digital Signature (XML-Dsig), because can secure the whole message, or parts of the message independently from one another and that being above the transport layer, the SOAP-messages (or relevant parts) can be stored, forwarded or included within other messages destined to third parties and the latter will be able to check data integrity from the original requestor.

3.1.4 Data Confidentiality/Privacy

Confidentiality refers to the ability of limiting information viewing to intended parties [6]. It is achieved through encryption either at the transport level (SSL/TLS) or at the message-level (XML encryption).

3.1.5 Mitigation of Denial of Service Threat

When a service becomes unavailable, or falls below a certain level which is required, it is considered to be in Denial of Service [6]. This type of threat is usually achieved by flooding the service with unwanted/invalid messages. To mitigate such a threat, messages should be screened before reaching the actual service so that suspicious requests may be discarded. This usually requires some sort of client authentication below the application level.

Several ways of achieving this include the use of SSL/TLS, WSS and [16] even proposes the use of stateless service architectures whenever possible. Because stateless services do not need to keep track of states, they are likely to be overwhelmed by too many incoming messages, which would lead to Denial-of-Service.

3.1.6 Non-Repudiation

Repudiation happens when one or more entities involved in a transaction deny participation in part or in the whole communication [6]. Because SSL/TLS signatures are transient, they do not offer long term non-repudiation capabilities and this is why some more permanent signatures, such as XML-DSig, must be used.

3.2 Security Levels

Security can be enforced at several levels of the network. This Section presents the type of security introduced at three levels -transport, message and application- and their characteristics.

3.2.1 Transport Level Security

As its name suggests, Transport Level Security is provided at the network's transport layer. This type of security is used to secure a conversation (set of message's exchange), but not a message properly, i.e. all the messages are secured in the same fashion.

The main advantage of this security level is that communication can be secured at the transport level without developer's intervention. Furthermore, because it is the communication channel that is secured, all messages coming through this channel can be trusted.

The main disadvantages of implementing security at this level lies in the security scope provided. Yes transport level security, such as SSL/TLS, provides end-to-end security, but only between the two initial end-points of data exchange. For example, Figure 13 [79] represents a Web Service transaction spanning two Web Services. Because Transport Level Security only secure messages between 2 end-points, the data loses it's secured state at the middle Web Service, to be then secured by this very same service before being sent to the final (in the figure, right-most) Web Service. In other words, the last Web Service only knows that the data was not corrupted between itself and the middle Web Service, but it must implicitly trust the middle Web Service to not corrupt the data coming from the original sender. Hence, it is secured end-to-end between any two directly communicating entities, but not between those communicating via intermediaries.

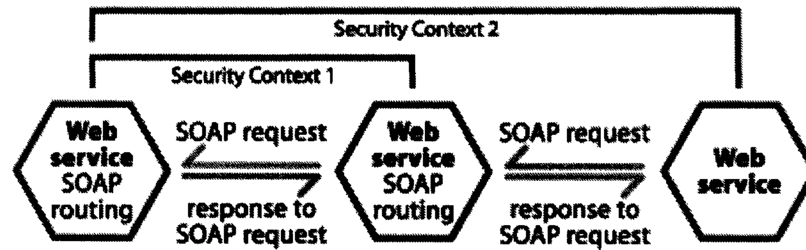

Figure 13: Security Scope when spanning two Web Services

Another disadvantage lies in the lack of granularity. Taking again the example of Figure 13, each message exchanged between the first two Web Services is secured as a whole. So if the security properties were to somehow be preserved, the whole message would have to be forwarded to the last Web Service. Indeed, when no granularity is available, messages must be secured as a whole and only as a whole can security be checked. As soon as data is extracted from this message, this data looses its secured state.

The lack of granularity and of end-to-end security (when messages span 2 WS or more), are the two main reasons why, despite OMA stating that Web Service Application Servers must support SSL/TLS, this type of security is not sufficient for Web Services.

Other disadvantage is the lack of path flexibility of such security, i.e. once the secured tunnel has been established it cannot be changed.

3.2.2 Message Level Security

On the other hand, Message Level Security does not require a security tunnel and, as such, is more flexible as to the path the messages can use. It also has the advantage of granularity, allowing messages to be secured as a whole, to have parts of the message

secured independently of the rest as well as to do a combination of both (securing part of the message for the third party alone and securing the message as a whole when sending it to a third party).

Another advantage of this sort of security is its permanent aspect. A company, wishing to keep track of the incoming requests may store them and still preserve the secured state of the message. It may also choose to only store the secured part of the message that is relevant to itself and forward the rest to a third party. This third party has now not only access to security information from the middle service, but also from the original data provider, ensuring that the data has not been tempered with and/or viewed by the middle Web Service.

3.2.3 Application Level Security

As its name implies Application Level Security is applied and dependent on the application used. This type of security is independent of the transport means used to communicate the data. It is also protocol independent.

The main issue with this type of security is that it relies solely on the application applying the security and, as such, all applications built to communicate with the former must support the same type of security. Now in a worst case scenario, this means that if an entity must communicate with 100 other applications, each applying their own type of security, this entity needs to support these 100 different security formats.

Furthermore, on top of being cumbersome when communicating with many different entities, this also means that it is virtually impossible to dynamically switch from one application to another even if they accomplish the same work. Indeed, because they are possibly from two competing companies, the security they will enforce is different and the entity cannot learn a new security format on the fly.

These are the two reasons why application level security is highly inappropriate to Web Services.

3.2.4 Security Level Conclusions

In short, because Transport Level Security cannot secure data across several Web Services and because the security provided is neither granular nor durable (it cannot be stored), security at this level is wholly inappropriate for WS.

Application Level Security is at the other extremity. Yes it can be granular, stored and preserved across multiple Web Services, but it is intrinsically related to one (or a group of) application and therefore it is hard for an entity to dynamically switch Providers, or accommodate new Requestors.

Message Level security allows for granular and durable security, whether the Service spans one or more Web Services and it will be applied to a large group, all those using the same messaging protocol.

3.3 Web Services Security: Standard and Technologies

In order to overcome the main drawbacks of Transport Level Security, IBM, Microsoft and Verisign originally united their forces and created WS-Sec, a specification proposal that was submitted to the OASIS group. It was soon renamed WSS and has now become an OASIS standard specification, a standardised set of SOAP extensions used to secure WS by implementing message content integrity and confidentiality [4]. It mainly uses XML-DSig [3] and XML-enc [2] for message content integrity and confidentiality, respectively.

Although designed specifically for data integrity and privacy, WSS can also be used to ensure authentication, mitigate Denial-of-Service attacks as well as repudiation. The Digital Signature (XML-DSig) used to check content integrity is associated to one entity only. In other words, if the message integrity is confirmed, then so is its author's identity. Because the key is unique, the author cannot repudiate the message and can be legally charged and forced to pay. Also, if the content has been tempered with, the author is unknown/unauthorized, the message cannot be decrypted or a signed timestamp is expired (in case of attacking a server by sending several copies of one valid message), these can all be used to reject a SOAP request and therefore mitigate the risk of Denial-of-Service attacks.

3.3.1 XML-enc

XML encryption is a W3C recommendation that specifies how to encrypt data and represent it in XML [2].

3.3.2 XML-DSig

XML Digital Signature (XML-DSig) is W3C recommendation [7] and an IETF draft standard [8] on how to sign an XML document, or part of the document. Through this means integrity, authentication and/or signer authorization can be provided [7]. [3] provides a good overview of XML-DSig.

3.4 Providing Web Services Security

So far, WSS has been mostly provided as libraries, proxies and application servers that apply security profiles to SOAP messages. This Section not only describes and discusses these ways, but introduces the reader to a novel way of providing WSS: Security Web Services.

3.4.1 The Library

Libraries are a very common way of providing frequently used functionality and of packaging it with other related functionality that might be useful. WSS is no exception to the rule with such libraries as Sun's Java Cryptography Architecture (JCA) [10], Java Cryptography Extension (JCE) [11] and others. Although libraries have their advantages (such as granularity), they are ill-suited to the Web Services world.

Indeed, libraries are tightly coupled to the application that uses them. For this reason, if an application –or its user- wishes to use a different library, this must usually be done offline and often implies changing the source code before recompiling it. Another disadvantage is that they are resource expensive in terms of memory needed to store

them, to use them and of the CPU computational power needed for certain operations such as encryption and signing. Furthermore, these libraries often require the use of several function calls before one single task, such as encrypting a sentence, is accomplished. Knowing all the libraries and the most advantageous way of using them requires a minimum knowledge and understanding of the library, its architecture and very often of the protocols/standards being implemented by these libraries. For example, when using a library to sign a message, one must not only know which objects to initialize, but must be aware that a signature is obtained by first creating a message digest and then signing this digest [3][10]. This last part is not so much related to the library but much more to the XML-Dsig standard itself.

If one considers devices such as PDA's, cell phones, Backberrys and others, it is obvious that these devices have a limited amount of library and CPU power and that these can be used more efficiently than to store and run libraries.

Furthermore, developers must always take into account the time-to-market of a product, if a product comes out too late, the market will be taken by someone else. Learning to properly use new libraries will in certain cases take more time than actually implementing a new service. This is quite obviously counterproductive and a simpler way of securing their services could quicken service development by 3rd parties, and other developers in the 3G world.

3.4.2 The Proxy/Gateway and Application Servers

Proxies, such as Xtradyne's [12], and Application Servers (e.g. JBuilder, BEA) provide WSS through security profiles. Although the former is an independent entity placed at the edge of a network and the latter are actually the server on which applications are deployed, they are discussed here as one because they very much work in the same way.

Proxies and Application Server (from now on referenced as proxies) solve part of the problem posed by libraries, i.e. use of local resources. Because messages are secured and security is checked/removed at the proxies, WS requestors and providers technically speaking need not to explicitly secure the data to be transmitted.

The main disadvantage of proxies lies in the fact that they must be pre-configured. This off-line work might be considered part of normal load for WS providers, used to secure their network and only dealing with a fixed known number of services (their own) deployed once, but it is trouble and tiresome to a user that just discovered an amazing service and must now configure his proxy before being able to use the service.

From developers' point of view, proxies are a big unknown. Very often, a developer does not have access to the security proxy through which his Web Service will communicate and must completely rely on the proxy manager. Proxies also add a delay to WS that may not need security functionality or have decided to handle it themselves. Furthermore, security proxies are, like other single point of passage entities, prone to congestion. Finally, the use of proxies requires a profound knowledge of the proxy, its

particularities and, in the rare cases where they allow for granularity, a good knowledge of the SOAP Message structure which may be too much for common users.

3.5 Conclusion

Although there already exists ways of enforcing WSS, truth is that they are ill-suited to both WS and 3G networks.

Libraries are too heavy for low resource devices. Furthermore, applications using libraries are very tightly coupled to them. From developers' point of view, libraries are cumbersome as they require several invocations to do what is logically considered as one operation and libraries architectures require that developers not only know what they wish to accomplish, but very often how, both according to standards and within the specific library being used.

Proxies, and Application Servers, require the use of profiles which must be defined before the use of any given Web Service. Furthermore, not only do most of these proxies lack granularity (securing the message as a whole but incapable of securing specific message's elements), they also take away from the developer the capability of securing his WS. Indeed, proxies are usually not configured by developers but by network managers. Finally, proxies remove the security information from the message, which may be counter-indicated in many applications where the response depends on the security context.

For all of these reasons, libraries and proxies are inefficient in the WS and 3G world and another means of enforcing WSS must be devised.

Chapter 4 Security Web Services

As a solution to libraries and proxies shortcomings (see Chapter 3), we propose the use of Web Services to provide WSS. These new WS will be henceforth called Security Web Services (SWS).

Based on Web Services technology, our SWS provide access to WSS in a loosely coupled manner through the use of high level interfaces. These interfaces, simpler than libraries and reduced in number, will allow a faster service development. Furthermore, expensive calculations (encryption and signatures) will be handled remotely, allowing limited resource devices to secure their WS communication.

A great advantage of our SWS over proxies is that they do not need to be pre-configured and will offer granularity through a high level but complete interface. Another advantage, if a proxy crashes, the services relying on it become unavailable, but entities using a SWS will still be available and able, through Web Service Composition, to use a different SWS to continue their normal operations, should the one they are using crash.

The goal of this chapter is to introduce the global architecture of SWS use, the proposed SWS interfaces and to situate SWS within deployment patterns. SWS related issues such as security will also be discussed.

4.1 The Global Architecture

Before an entity, application or otherwise, can use a Web Service, it must know, if nothing else, its location and its interfaces. The simplest way is to consult the service's

WSDL description. This description, published by the Web Service provider can be found in service registries (see Figure 1). SWS are no exception to this rule. Their providers must publish the services description to a UDDI server and then wait for a Service Requestor to invoke the service.

The general setting where all entities are set in their own trust domain is the one in which the issue of web service security must be dealt. Figure 14 represents this setting with full lines for WS Requestor invocation calls and dashed lines for the ones made by the WS Provider. In this figure, it is assumed that the WSDL descriptions of all WS have already been published and that the WS Requestor has already found the WS Provider he wishes to use.

The WS Requestor and Provider wish to secure the messages they will exchange during their transaction but are incapable of doing so. As a solution, they choose to use a SWS according to the scenario described below:

1. First, the WS Requestor in Domain 1 consults a UDDI registry located in Domain 3 to find a SWS that it may use. The result of this search is the SWS1 residing in Domain 4. The WS Requestor then binds to (invokes) SWS1 to secure the message he wishes to send to the WS Provider. Once the message is secured, the WS Requestor send this message to the WS Provider.
2. When the WS Provider, in Domain2, receives the secured request message from the WS Requestor, he realizes that he needs to use WSS standard in order to understand the message. Because he can't do it, he looks for a SWS by consulting a UDDI registry. In Figure 14 it happens to be the same UDDI that

was consulted by the WS Requestor. From this operation, it finds SWS2 in Domain 5 and then binds to it in order to check and/or decrypt the secured message.

3. The WS Provider can then process the SOAP request and compose a Response.

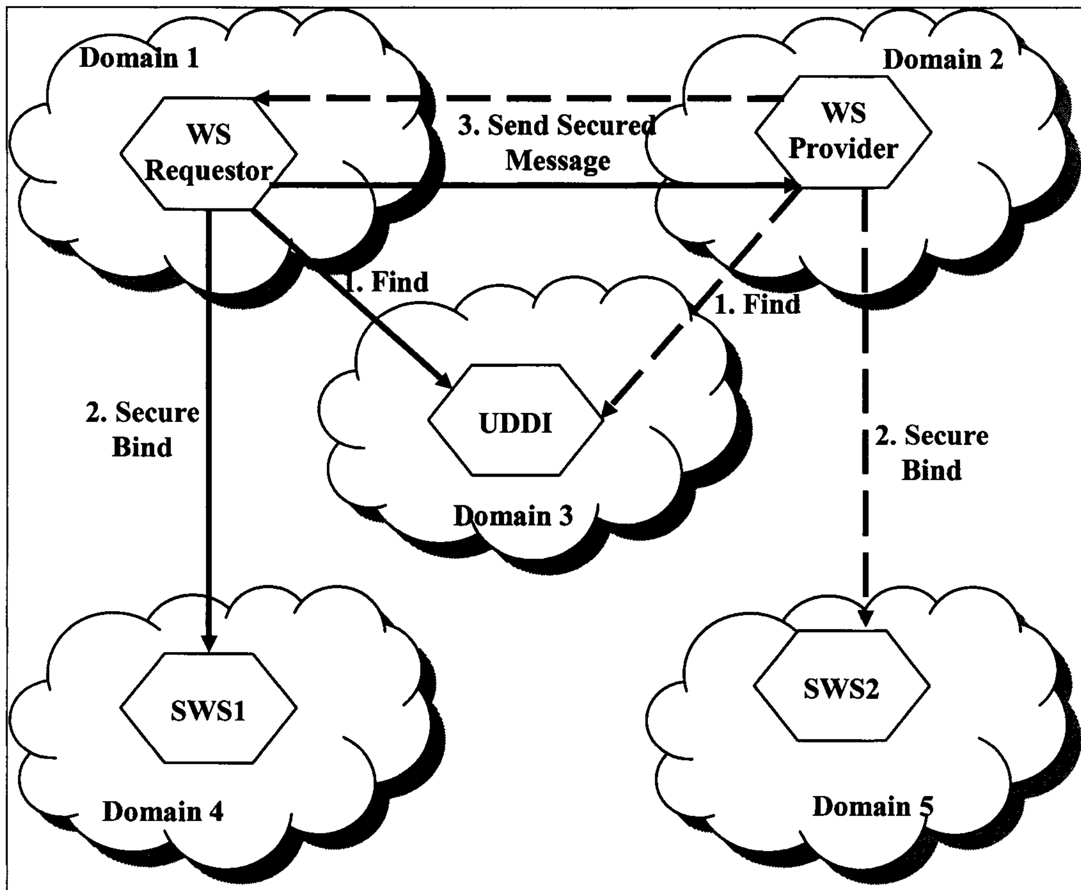

Figure 14: General Settings

Figure 15 represents the use of SWS from the Service Requestor point of view. When the Service Requestor has composed his message, he decides that he must secure it. Because he does not support WSS locally but needs message level security, he decides to contact a UDDI service registry in search of a suitable SWS. Once one has been found,

he asks the SWS to secure his message. The Web Service Requestor then sends his message to the Provider and awaits an answer. Upon reception of a secured SOAP response, the Requestor may decide to use the same SWS to check the response's security headers and certify that the message is secure. Based on the SWS response, the requestor can know whether his SOAP request was accepted, rejected or whether the message received should be disregarded because it has been tempered with or simply does not come from the provider.

Figure 16 presents the same scenario but this time from the Service Provider point of view. In this case, the search for SWS is triggered by the arrival of a secured SOAP request. The SWS is used to check the SOAP request and then to secure the SOAP response before the latter is returned to the Web Service Provider.

It is interesting to note that in both figures only the entity using the SWS is aware of this. In Figure 15, the WS Requestor is aware of using a SWS but does not know how the Provider has secured his message, it only knows that the message originated at the provider and is secured. In Figure 16, the Provider that is unaware of how the Requester secured his message. Note also that although both the Requestor in Figure 15 and the Provider in Figure 16 use SWS to secure their respective messages, they do not have to use the same. SWS is a generic name for Security Web Services. In other words, both interacting entities need not use the same SWS to be able to communicate (the same logic applies to the UDDI registry server).

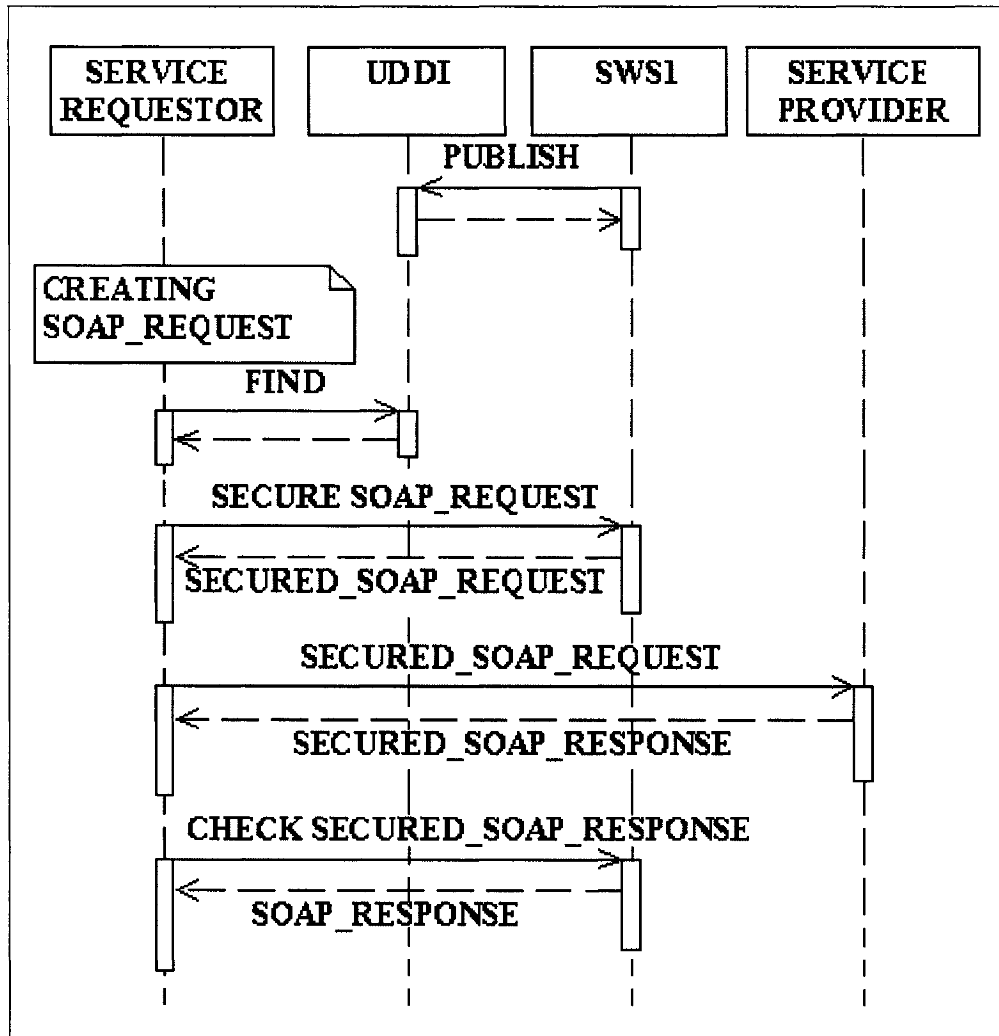

Figure 15: Security Web Services - The Service Requestor Point of View

Furthermore, the work accomplished by the SWS is invisible to any entity other than the one interacting with it. This is possible because SWS applies WSS, a standard set of SOAP extensions. Because it enforces a standard, and that this standard can be enforced in numerous ways, the other participants in a conversation do not know how it was enforced. This also allows all entities involved in a transaction to apply WSS in a way that is most appropriate to them (libraries, proxies, SWS, etc.). The usage of SWS is

therefore flexible in the sense that it does not force all communicating entities to use SWS.

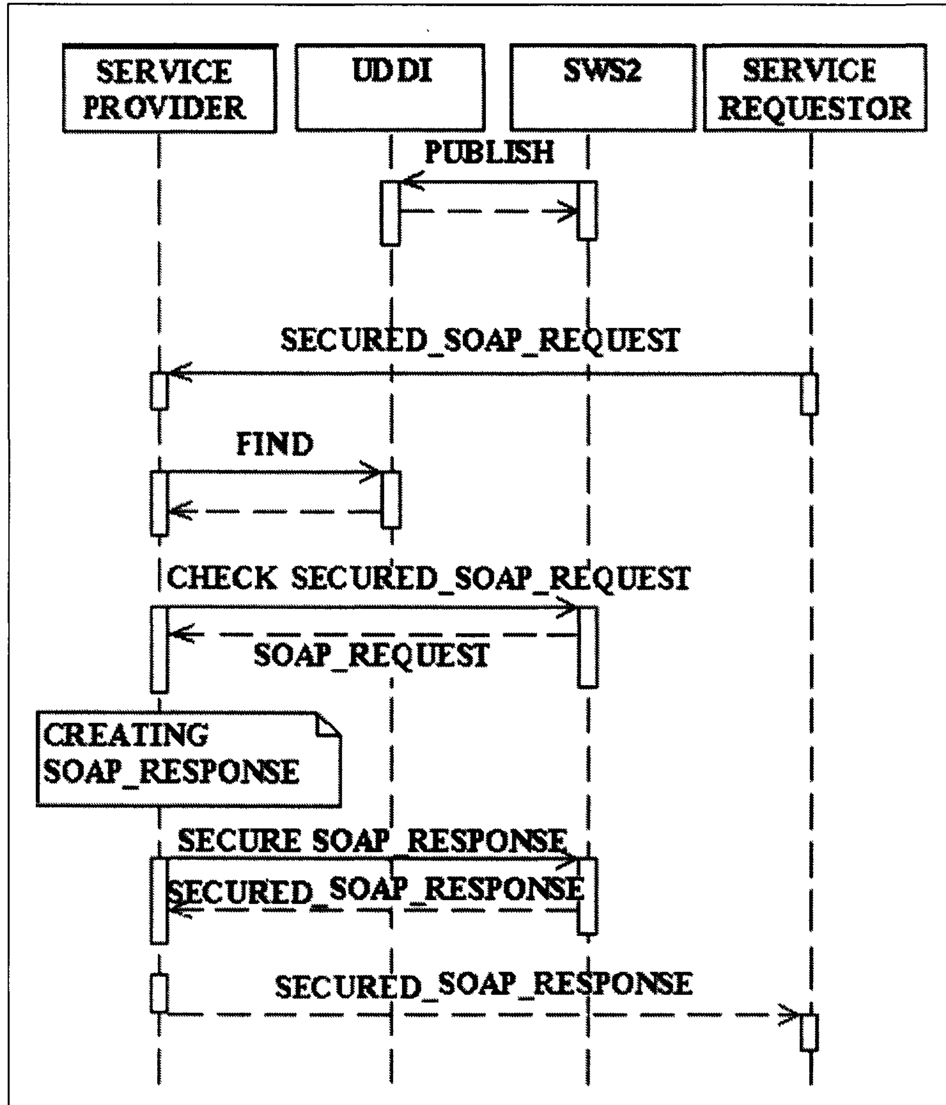

Figure 16: Security Web Services - The Service Provider Point of View

4.2 Proposed Interfaces

In the previous Section we saw that SWS are flexible in terms of deployment architecture. In this Section, we define a SWS for the architecture previously presented.

Our specific SWS is also flexible in the term of the services offered. Indeed, through its interfaces, a user may decide to secure a message or only an element of this message. Furthermore, it allows the user to decide whether he wants to secure through timetamping or signing or encryption or any combination of these three methods. Also, because our SWS need not be preconfigured, it ends up offering the type of granularity expected from libraries while doing operations remotely (like proxies).

Secondly, our SWS offers two comprehensive interfaces (extensively presented in Sections 4.2.1 and 4.2.2) that allow its clients to secure a SOAP message and/or one of its elements through time stamping, XML-Signatures, XML-encryption, or any possible combination of these three in one single WS call. The reverse operations (checking timestamps checking signatures and decrypting), and possible combinations, are also available in a single invocation. This is a neat advantage over libraries where several calls must be made to simply encrypt a sentence.

Sections 4.2.1 and 4.2.2 present the *elementSecurity* and the *secureSendRemoved* interfaces proposed in this thesis and used in the Case Study.

4.2.1 The elementSecurity Interface

The *elementSecurity*, shown in Figure 2, interface was designed to allow SWS users to secure an element of a SOAP message in one single invocation call, reducing overheads (delay and network) generated by using a WS.

```
boolean      elementSecurity(
    String in_mode,                   // insert, remove

    String in_SOAPmsg,
    String in_ElementID,

    String in_FromActor,
    String in_ForRole,

    boolean in_Timestamp,  //Non-Repudiation Info
    String in_DelayXSDDateFormat,

    boolean in_Authentication,      //Authentication Info
    String in_SignatureKey_SecurityToken,
    String in_CanonicalizationMethodAlgorithm,
    String in_SignatureMethodAlgorithm,
    String in_DigestMethodAlgorithm,
    String in_SignatureKey,

    boolean in_Crypting,            //Crypting
    String in_SharedCryptingKey_EncryptionMethodAlgorithm,
    String in_SharedCryptingKey_KeyInfo,
    String in_SharedCryptingKey_Key,
    String in_IncludedCryptingKey_EncryptionMethodAlgorithm,
    String in_IncludedCryptingKey_KeyInfo,
    String in_IncludedCryptingKey_Key,

    boolean out_Timestamp,
    boolean out_Authentication,
    boolean out_Crypting,

    String   out_SOAPmsg
)
```

Figure 17: Security WS elementSecurity Interface

Although the method name refers to an element, it can be used to secure the message as a whole. Indeed, when the body element of the SOAP message is secured, it is said that

the message is secured. The rest of this Section is devoted to a comprehensive description of the interface's signature.

The *elementSecurity* interface is composed of in-parameters and out-parameters denoted by the *in_* and *out_* prefixes respectively. It also returns a variable of type *boolean*. This last variable indicates whether or not the invocation call was successful. Should the function return *false*, the user can use the *out_Timestamp*, *out_Authentication* and *out_Crypting* to determine where the invocation failed. These three Boolean out-parameters are set to *true* when their respective actions where successful and to *false* if they failed or were not requested by the user. If the function returns *true* the user can get the new message from the *out_SOAPmsg* variable. The message is returned as a String and must then be parsed into a SOAP message.

The *in_mode* is used to specify whether the client wishes to insert or check/remove the security. To secure the client must pass "insert", and to check timestamp and signature or decrypt he must pass "remove".

The next two elements are used to identify what must be secured, or checked. The *in_SOAPmsg* is the SOAP message as its name indicates. The *in_ElementID* is the value of the *ID* attribute of the SOAP Element to be secured. The SOAP message must be passed so the appropriate security headers may be added inside the message in the proper order and at the proper location. The element's ID is used because no two elements can have the same ID in a SOAP message, but they can have the same name.

Messages, and their elements, can be secured by several entities, and hence be referenced in several security headers. The security headers can all be addressed to the same entity or to different ones. The ones securing the message are known as Actors and the ones using these headers play Roles. Entities receiving SOAP-messages are only allowed to consult headers that are specifically destined to one of the Roles they play, or that are destined to anyone, in which case the Actor attribute of the Security Header is left empty. To illustrate this, in our Case Study, the Customer is an Actor that secures his credit card for any entity playing the role of a Financial Institution and his message for a WS playing the Role of a Store. The Store upon reception of the Customer's request can consult the Security Header with "Store" as a Role, but not the one specifying "Financial Institution". For more information on Actors and Roles consult [4].

The next variable is Boolean *in_Timestamp* used for non-repudiation [4] and to further secure message interaction. Indeed, if a Bank receives a request to debit a client's account and the client signed the authorization one hour ago, the message could be considered suspicious and the bank may decide to refuse the transaction. On the other hand, the user may be communicating information that is only valid for a certain lapse of time (e.g. location, availability). The user may then wish to timestamp a message and to specify the delay during which the information is valid (*in_DelayXSDFormat*). The choice of XSDFormat, for the delay and the timestamp printed by the SWS, was based on [4] and on the fact that it also allows specification of the time zone to which the time refers. Thanks to the time zone specification, all entities receiving the message can correctly enforce delay policies even if they are in different time zones. If the user is

checking the timestamp and specifies a delay, then the SWS will use this delay to check the validity of the message.

Once time stamping information has been filled by the SWS user, he must specify his Authentication information (provided authentication is wished). SWS uses XML-DSig for authentication. Once the user has set *in_Authentication* to *true*, he must give, as a String, the Security Token related to the signature (see [4] for format). From this token, SWS knows whether the Signature Key comes from a PBKS system or other, being then able to choose the appropriate headers. The three *MethodAlgorithm* variables are used to specify the type of Canonicalization, Signature and Digest methods to be used. If they are left blank, the SWS should choose default methods. Note that these three variables can be left empty when checking signatures because the values are specified in the header itself. Finally the *in_SignatureKey* parameter is used to transmit the Key that is to be used.

Last but not least, there are the parameters dealing with encryption. Once more, the *MethodAlgorithm* parameters can be left empty for decryption. As it is the case for Signing –or checking signatures-, the SWS client must pass the *KeyInfo* (equivalent to signature's *SecurityToken*) element as well as the Key itself. When asking for decryption, the *KeyInfo* value is used to associate the Key to the proper Security Header. When it is to encrypt, the *KeyInfo* elements will be placed in the new Security Header. When encryption is used, it must be using a key that is shared between both concerned entities or at least accessible by both (*SharedCryptingKey* parameters). If the encrypting party decides to use a key to which the other does not have access but must use, he must pass

this key within the message and then encrypt this part with a shared key. The information related to this key is passed in the *in_IncludedCryptingKey* parameters. For more details on how to used Shared and non-shared keys see [4].

It can be argued that forcing the user to write their own *KeyInfo* and *SecurityToken* is a load that should not be placed on the user. Truth is that this information is always the same for all messages using the same key and can, once determined, be saved and re-used, just like the Key value that must be passed to the SWS. It should not be saved at the SWS, just as the Key and authorization rules should not be stored there. These are specific to each user entity and would only overload the SWS with information that would need some kind of pre-configuration or update, finding ourselves once more with the pre-configured profiles that we wished to avoid in proxies.

Thanks to the return Boolean the user can know whether the operation was or not a success. If the operation is successful the user can retrieve the modified message from *out_SOAPmsg*, otherwise the user can check which operations were not completed and decide what actions to take next (e.g. reject message, try again, pass on to a different application, etc..).

Thanks to the *in_mode*, the user can use the same interface whether he wishes to secure a message or check its security information, reducing the number of interfaces to learn. Because the all securing operations on one element can be done in one step, whether individually or in any possible combination, this interface offers the flexibility and granularity of libraries while using proxies main advantage, non-local calculation of resource expensive operations.

4.2.2 The secureSendRemove Interface

As seen in Figure 15, the use of SWS will not only create a delay overhead, but also a network load overhead each time SWS is called. In order to reduce both delays, especially in cases where the security elements applied to the response message are known ahead of time, as well as the keys used, the *secureSendRemove* interface was created. This interface allows the user to secure one last element and have the SWS send it to the desired Web Service (see Figure 18). In this case SWS falls into the Routing Pattern (Section 2.2.1).

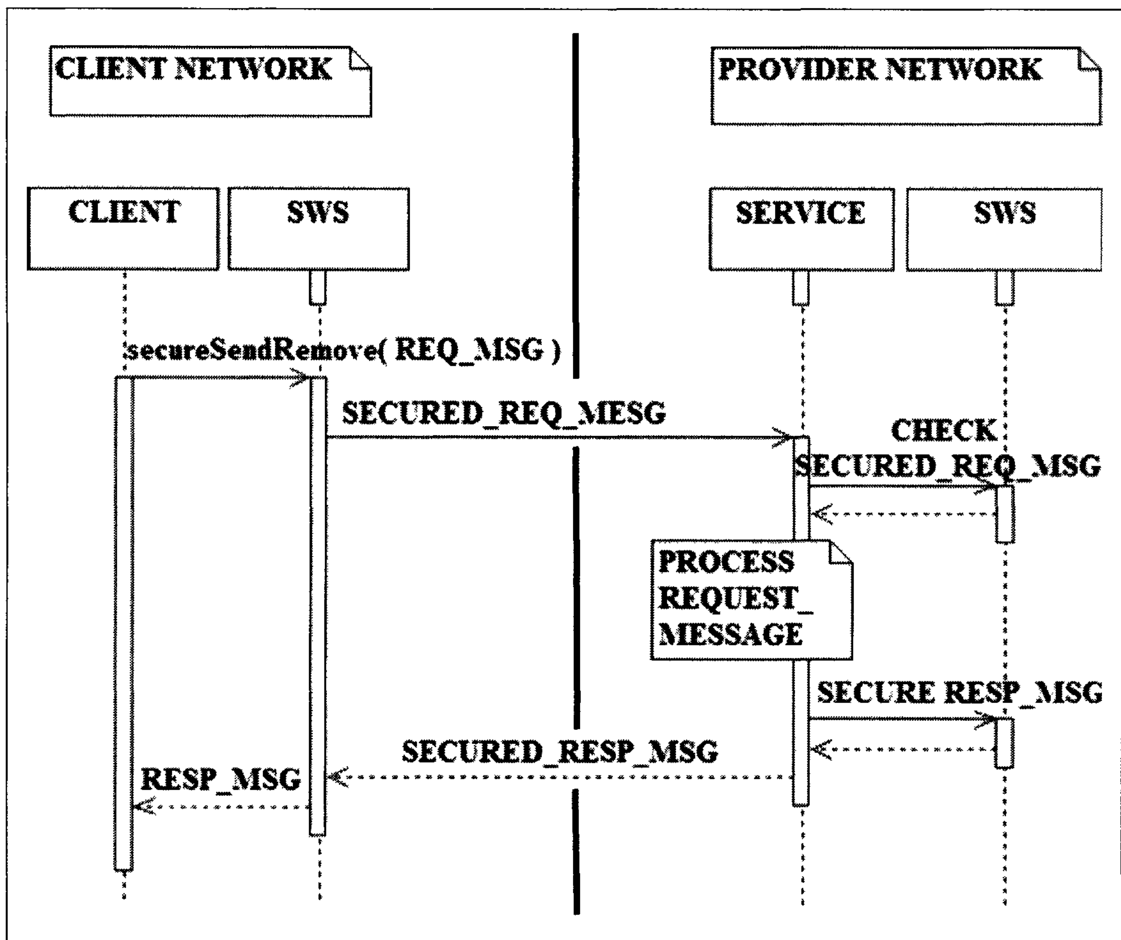

Figure 18: Example of a Security Web Service - Routing Pattern

Although this thesis does not explore key creation and distribution, according to [4] the entities wishing to encrypt their data must do so using a Shared Key and therefore it is logical that the Service Provider would know what encryption key will be used by the WS. It is not far fetched that, when getting the decryption key, the user would also get the key necessary to check the signature of the provider.

Unfortunately, due to the very nature of this interface, it cannot be used by WS providers unless they are sending a request message to another WS and use it for this request and its response. To use it for a response that the WS would be sending, it would need to be pre-configured, like a proxy, which we wish to avoid.

The rest of this Section is devoted to a detailed explanation of the interface itself, as shown in Figure 19, and its parameters.

The first half of this interface, up to *in_destinationAddress*, is similar to the *elementSecurity* interface (Section 4.2.1) and this part will therefore be quickly summarized here. The *in_SOAPmsg* and *in_ElementID* identify the element to be secured, while the *in_FromActor* represents the requesting entity and the *in_ForRole* represents the type of responding entity allowed to access this header. The *in_Timestamping* and the *in_DelayXSDDateFormat* are used for non-repudiation purposes and the next 5 parameters to sign the Request SOAP message. Finally, parameters *in_Encrypt* through *in_IncludedEncryptingKey_Key* are used to pass the data necessary for the SWS to encrypt the SOAP message.

```
boolean      secureSendRemove(
    String  in_SOAPmsg,
    String  in_ElementID,

    String  in_FromActor,
    String  in_ForRole,

    boolean in_Timestamp,
    String  in_DelayXSDDateFormat,

    boolean in_Authentication,
    String  in_SignatureKey_SecurityToken,
    String  in_CanonicalizationMethodAlgorithm,
    String  in_SignatureMethodAlgorithm,
    String  in_DigestMethodAlgorithm,
    String  in_SignatureKey,

    boolean in_Encrypt,
    String  in_SharedEncryptingKey_EncryptionMethodAlgorithm,
    String  in_SharedEncryptingKey_KeyInfo,
    String  in_SharedEncryptingKey_Key,
    String  in_IncludedEncryptingKey_ EncryptionMethodAlgorithm,
    String  in_IncludedEncryptingKey_KeyInfo,
    String  in_IncludedEncryptingKey_Key,

    String  in_destinationAddress,
    String  in_rcvdFromActor,
    String  in_rcvdForRole,

    boolean in_checkTimestamp,
    String  in_checkDelayXSDDateFormat,

    boolean in_checkAuthentication,
    String  in_rcvd_SignatureKey_SecurityToken,
    String  in_rcvd_SignatureKey,

    boolean in_Decrypt,
    String  in_SharedDecryptingKey_KeyInfo,
    String  in_SharedDecryptingKey_Key,
    boolean in_decryptIncludedKey,

    boolean out_insertTimestamp,
    boolean out_insertAuthentication,
    boolean out_encrypting,
    boolean out_sentSuccessfully,
    boolean out_checkTimestamp,
    boolean out_checkAuthentication,
    boolean out_decrypting,

    String   out_SOAPmsg,
    String   out_checkedSOAPmsg
)
```

Figure 19: Security WS secureSendRemove Interface

The SWS must then use the *in_DestinationAddress* parameter to specify the endpoint of the Web Service to which this request must be sent. The rest of the parameters passed to this interface deal mostly with the response received at the SWS from the WS provider and the answer it will provide to its user.

in_FromActor specifies the actor who inserted the security header we wish to check. The *in_ForRole* specifies the role the user is now playing and therefore the headers he is allowed to access [4].

The *in_checkTimeStamp* variable is used to specify that the SWS user wishes the SWS to check the Response Message Timestamp and the *in_checkDelayXSDDateFormat*, the delay the user will accept as valid, if it is inferior to the one specified in the message. If the delay specified in the message is shorter than the one specified by the user, the former should be used and the latter ignored.

in_checkAuthentication is used to enforce authentication using the passed *in_rcvd_SignatureKey_SecurityToken* and *in_rcvd_SignatureKey_Key*. As you can notice, there are no *MethogAlgorithm* parameters because this information can be found in the message and therefore needs not be passed to the SWS. In the *elementSecurity* interface, these parameters could be left blank when removing security for the same reasons.

The SWS user sets *in_Decrypt* to true when he wishes to see the incoming SOAP_response decrypted. *in_decryptIncludedKey* is used to express the will to decrypt all elements which might have been encrypted with a Key that was included in the

message. Because these keys have been encrypted with a shared key, only on once they have been decrypted can their elements be decrypted. Therefore, once decryption with a shared key has been done, the SWS checks if this decryption has revealed any *IncludedKey*, and if it has, it decrypts all elements related to this Key. For the decryption with the shared key to take place, the Key and KeyInfo element must be passed in *in_SharedDecryptingKey_Key* and *in_SharedDecryptingKey_KeyInfo* respectively.

The *out_insertTimestamp*, *out_insertSuthentication*, *out_encrypting*, *out_sentSuccessfully*, *out_checkTimestamp*, *out_checkAuthentication* and *out_decrypting* are used to communicate which functions were successfully carried out. The first three relate to the SOAP request passed in *in_SOAPmsg* by the client. *out_sentSuccessfully* determines whether or not the message was successfully sent and a response received. The last three Boolean variables refer to the received SOAP response message.

Finally, *out_rcvdSOAPmsg* contains the received SOAP-response while *out_SOAPmsg* contains the processed SOAP-response, if processing was successfull. Why using both, because the SWS user may wish to be able to store the secured response as a "proof of purchase".

4.3 Deployment of SWS

In general, SWS are used by entities wishing to delegate the security functionality because they do not locally support WSS, they do not have enough CPU power or memory to enforce it, their usual means of enforcing WSS is unavailable and so on. This

way of "sub-contracting" part of the SOAP-message processing clearly falls under the Delegate Deployment Pattern described in Section 2.2.6 (see page 13).

From another point of view, if the SWS is based on libraries, it also falls in the Adapter Deployment Pattern (Section 2.2.5). Indeed, if one considers libraries as the legacy system and since SWS expose some of the libraries functionalities as a WS, the SWS then become the Adapter allowing other applications to use the libraries even if the libraries do not understand WS invocations.

Finally, SWS can also fall under the Routing Deployment Pattern (Section 2.2.1) if they are built to apply WSS and then route the secured message to another WS. The *secureSenRemove* interface presented in Section 4.2.2 (page 45) allows the SWS to route the secured messages and, as such, all SWS implementing this interface, or an equivalent, will fall into the Routing Pattern.

4.4 Issues on SWS

OMA recommends that WS falling within the Delegate Pattern be deployed within the same trusted network as the delegating party [5]. Indeed, this avoids the necessity to secure the messages among these two parties, reducing SOAP messages size and the delay necessary to process them.

Security Web Services is no exception to this rule. In fact, this rule should be even more stringent because if a party is using SWS, it is probably because it is incapable of securing its messages locally according to WSS SOAP extensions. It is then safe to

assume that messages sent to SWS have not been secured even though they may contain sensitive information.

In order to be able to use a SWS outside one's trusted network, the messages exchanged must be secured at the transport level. This does not go against OMA's security rules, since OMA states that all Web Service Providers must support SSL/TLS [5][6].

However, due to transport level end-to-end security issues discussed in Section 3.2.1, one should only use SWS outside its own trusted network if does not need durable security between itself and the SWS.

4.5 Conclusion

In short, SWS are flexible in terms of deployment because they are independent from the transactions which they secure. The entities using them do not have to use the same instance of SWS, nor do they have to all use SWS, as long as all enforce WSS.

SWS are also flexible in terms of functionality offered. A message's element, or the message itself, can be secured through signature, times tamping and encryption in one single step, but the user may choose to only use on or two of these functionalities. SWS offer the flexibility of libraries without its complexity.

Non-Secured messages do not have to go through the SWS (unless the *secureSenRemove* interface is used), saving time when compared to a proxy where all messages must be analysed by the proxy even when it isn't necessary.

Finally, SWS located outside the user's trusted domain can be used, as long as the user is aware of the risks/issues associated with such a choice.

Chapter 5 A Case Study

This chapter will present the Case Study explored in this thesis (Section 5.1) and the associated security issues. Section 5.2 will then describe a Secured version of the same Case Study and present the assumptions made. The rest of the chapter describes the implementation of the proof-of-concept prototype (Section 5.3) and issues faced, the environment in which the tests were performed (Section 5.4), as well as the tests performed (Section 5.5) and finally the results obtained (Section 5.6).

5.1 The Customer-Store-Bank Case Study

The Case Study presented in this thesis is based on a Web Service interaction spanning two Web Services (see Figure 20). Because in such cases transport level security such as SSL/TLS is insufficient to secure the general transaction (see Section 3.2.1), this Use Case requires the use of WSS and will therefore be a perfect example of why and how to use SWS.

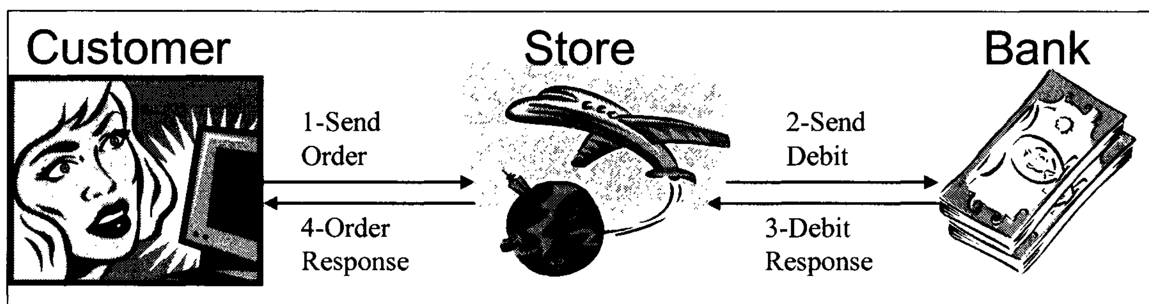

Figure 20: Use Case

In this Use Case, a Customer wishes to buy some merchandise via Web Services from a Store. In order for the transaction to be completed, the Customer must submit his Credit

Card Information to the Store, so the Store may in turn notify a Banking Institution of the transaction.

The Store will then contact the Bank with both the customer's Credit Card Information and information about its own account. Based on the Customer's credit margin and the amount of the purchase, the Banking Institution will then inform the Store of the acceptance or refusal to debit the Customer's account and credit that of the Store.

The Customer, Store and Bank lie in three different trust domains.

As previously mentioned, the main issue lies in the fact that the Web Service used by the Customer relies on passing information provided by the said Customer to a 2nd WS, the Banking Institution. Indeed, if the message was to be secured only at the transport layer, the security attributes of this message (data privacy and integrity) would be lost once above the said layer. Because it deals with money, it is obvious that the store will want to keep track of these security elements, proving that not only the Customer did make an order, but that he also approved of the amount to be charged.

From the Banking Institution point of view, it will want some form of guarantee that the Credit Card Information it receives from the Store did originally come from a Customer making a purchase and was not simply created by an entity pretending to be in contact with the Credit Card Holder (the Customer).

Finally, as far as the Customer is concerned, he may wish to secure his Credit Card Information independently from the rest of the message for many reasons. First, he may not trust the Store entity and fear that it will try to receive more than the authorized

amount. He will then want to at least sign the amount he is authorizing so the Banking Institution will know how much has been indeed approved by the Customer for the transaction. For the same reason, the Customer will want to timestamp his Credit Card Information and the amount so the Store cannot reuse the message later on pretending that it is a second transaction. Indeed, time stamping allows the Bank to decide whether too much time has elapsed, since the transaction was approved by the Customer, for the transaction to still be considered valid. Thirdly, the Customer may wish to encrypt this same information because he does not trust the Store to properly secure it when sending to the Banking Institution, leaving it in clear in the message. Finally, the Customer may believe the Store to be trustworthy, as well as to properly secure its transmissions, but it may not trust the Store's network, databases, servers or others to safely store this sensitive information and therefore the Customer will encrypt, sign and timestamp it in order to prevent possible hackers from entering the Store's databases and using their personal information.

All these motivate the use of WSS security rather than Transport Level Security to secure the credit card information in the message sent.

5.2 Secured Customer-Store-Bank Application

Now that the issues have been covered in this particular Use Case, let us look at how SWS can be used to secure the proposed transaction.

In Figure 21 the Customer uses a SWS to secure his credit card information so that only the Bank may view it (1 secure-request, 2 secure-response). In step 3, the Client sends it to the Store.

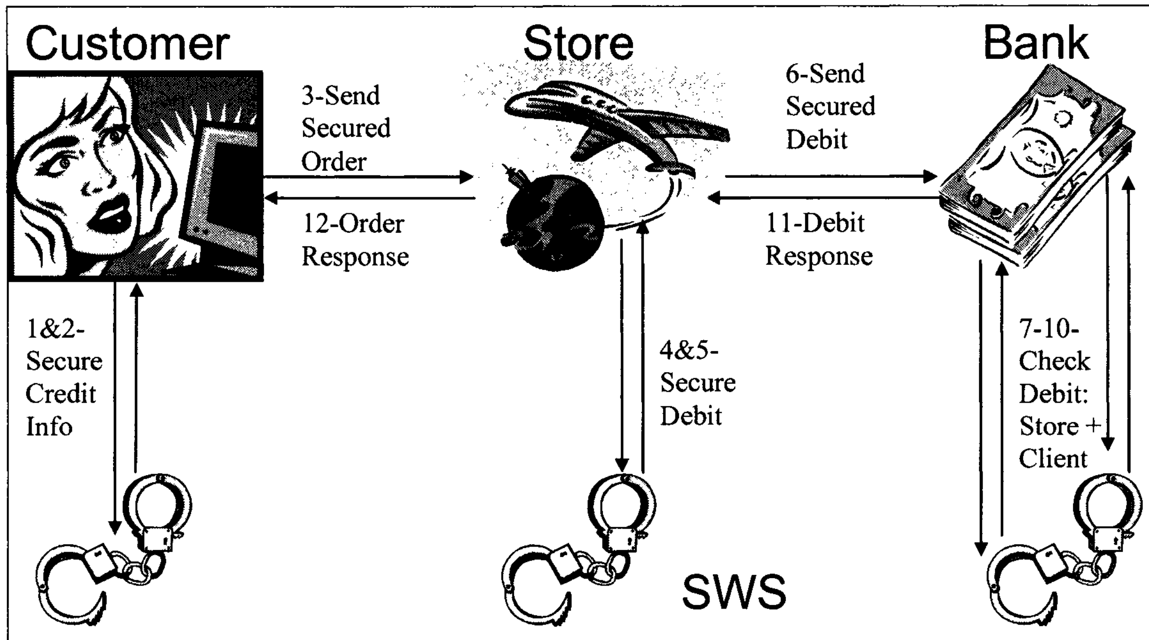

Figure 21: Secured Use Case

The Store then builds a Debit request to be sent to the bank. It copies the secured credit card information from the client's Order Request into his Debit Request, as well as the security header related to it. Once the Debit Request is complete (including the Store's account information), the Store uses the SWS to secure his message and once the message is secured, the Store sends it to the Bank (Steps 4 and 5).

Upon receiving the Debit Request from the Store, the Bank uses its SWS to check the Store's security and then the Client's security (Steps 7-10). Once the message has been checked and decrypted, the Bank can analyze the request and return the appropriate

answer (Step 11). Note that in this example (for simplicity's sake) the responses are not secured, but they could very well be.

After receiving the Bank's response, the Store processes the Client's order and responds to it in the 12th and last Step. Once again, the responses are not secured in this example for simplicity's sake.

The assumptions behind this use case as presented in Figure 21 are the following:

- The Client, Store and Bank are each located in their own trusted domains.
- The Client, Store and Bank have access to a SWS within their own trusted domain, eliminating the need to secure the messages between the SWS and its user.
- The Client Store and Bank are all aware of their own SWS and therefore need not look it up in a UDDI registry.
- For simplicity's sake, it is assumed that the Store and Bank's responses are not tempered with and need not be secured.

5.3 Implementation

This Section describes the implementation of the prototype's components: SWS, Client, Store and Bank (Section 5.3.2.1 to 5.3.2.4) as well as the Keys used to sign and encrypt the messages (Section 5.3.3), but first it takes a look at implementation issues that where faced and which influenced the prototype development and design.

5.3.1 Main Issues

The first problem faced during the implementation was related to *out* and *in/out* parameters. Indeed, although they are officially supported by BEA WebLogic 8.1.2 Application Server, we were unable to activate this feature. To circumvent this problem, the SWS interfaces were changed to return the processed SOAP message rather than *true* or *false*. Of course, this reduces the size of the messages returned by the SWS since all the other *out-parameters* are now omitted.

In the case of the *elementSecurity* interface (Section 4.2.1), this function will return a SOAP message containing the processed-SOAP-message. Because the size of the latter is much larger than that of the Boolean variables, this change will not greatly influence the network for the *elementSecurity* interface. The *secureSendRemove* (Section 4.2.2) interface, however is more greatly affected because the original interfaced returned the processed message (*boolean out_SOAPmsg*) but also the message received at the SWS (*boolean out_rcvdSOAPmsg*). In this case, the network load is, at least, halved.

The second issue lies in the handling of XML-enc and XML-Dsig. Although BEA WebLogic 8.1.2 Application Server handles signatures and encryption, it only does so when using the Server as a proxy to enforce security policies. If a developer decides to secure his message himself, the encrypted and/or signed message raises a *com.bea.xml.marshal.XmlEncodingException* and is never sent. Different encodings were tried (UTF-8, UTF-16, ISO-8859-1, etc…) but none worked.

If using the SWS, the secured message can't be returned to the SWS user for the same reason. Only one instance of an encrypted message was found to be returned from the SWS, but even this message could not be sent.

5.3.2 The Components

The Client, Store and Bank could use either the SWS to enforce WSS or they could use the JCA [10] and JCE [11] libraries (the same used by SWS) to enforce WSS locally.

All components were implemented in Java and BEA WebLogic 8.1.2 Application Server was used to deploy all Web Services.

5.3.2.1 The SWS

The Security Web Service prototype was implemented using the Sun's Java Cryptography Architecture (JCA) [10] and Java Cryptography Extension (JCE) [11] libraries for Signatures and Encryption respectively.

For the reasons explained in Main Issues (Section 5.3.1), both SWS interfaces were modified to return the processed SOAP message. These interfaces deployed on the BEA Weblogic 8.1.2 Application Server and invoked through the stub classes generated by this server.

5.3.2.2 The Customer

The Client's role is to place an order with the Store. Because the Client does not trust the Store with his credit card information, the Client secures it so that only the Bank may read it.

Due to the lack of support by BEA Weblogic 8.1.2 Application Server of XML-enc and XML-DSig, the security enforced was time stamping (for non repudiation) of the credit card information, addressed to the Bank. However, as will be later demonstrated in Section 5.6.1 (*Delay Overhead: Results and Analysis*), this does not invalidate the measurements taken.

The client was also used to measure the delays.

5.3.2.3 The Store

The Store is a Dummy Store in the sense that it does all the security checks, but it does not process the orders. If the Bank accepted to debit the Client's account, then the Store responds positively to the order received, else it returns a negative answer.

Figure 22 shows the Store's only interface, the Purchase Interface. This interface allows a client to place his order. Before any order received is filled, the Store checks the security information contained in the message headers concerning the Store. If the message is considered secure, the Store proceeds to request from the Bank that the client be debited. To do this, the store copies the credit card information contained in the

purchase request (along with its security headers) and adds his own account information. The Purchase request is then secured as a whole.

```
boolean     purchase(
    String description,
    double amount,
    String creditCardInfo
)
```

Figure 22: The Store's Purchase Interface

The Store WS was composed of two classes: Store and StoreHandler. The former is the dummy class where the purchase logic can be implemented. The StoreHandler, which extends *javax.xml.rpc.handler.GenericHandler*, is used to catch the SOAP messages so that the security information can be investigated on received messages and enforced on outgoing messages.

5.3.2.4 The Bank

The Bank is composed of one interface, the Debit interface shown in Figure 23. This interface allows an entity to be paid (in this Use Case the Store) by submitting the payer's information as well as the account to which the payment must be made.

```
boolean     debit(
    double amount,
    String payerAccountInfo,
    String payedAccountInfo
)
```

Figure 23: The Bank's Debit Interface

The Bank checks whether or not the security information is valid. If yes, it automatically accepts to debit the client. Otherwise, the debit is refused. For the security information to be valid, the bank checks that the entity paying has secured its credit card information and that the entity being paid has secured the message as a whole.

The Bank is also composed of two classes, the Bank which never checks the actual accounts and in that sense it is a dummy Bank, and the BankHandler which checks the security information on incoming messages and enforces security on outgoing messages. BankHandler also extends the *javax.xml.rpc.handler.GenericHandler* class.

5.3.3 The Keys

Two keys were used one for signing and one for encrypting. Section 5.3.3.1 gives the parameters of the Signature Key and 5.3.3.2 the parameters of the Encryption Key.

5.3.3.1The Signature Key

DSA Key [13][10]was used for signing, applying the SHA-1 algorithm described in [14]. The hexadecimal values of the DSA key parameters are shown in Figure 24 and come from [13].

5.3.3.2 The Encryption Key

A DES **[15][11]** Key used was an 8-byte array with the following value: 103, 32, 88, 81, 2, 59, 94, -89. Single DES Algorithm was applied.

```
P =  d411a4a0 e393f6aa b0f08b14 d1845866 5b3e4dbd ce254454
     3fe365cf 71c86224 12db6e7d d02bbe13 d88c58d7 263e9023
     6af17ac8 a9fe5f24 9cc81f42 7fc543f7

Q =  b20db0b1 01df0c66 24fc1392 ba55f77d 577481e5

G =  b3085510 021f9990 49a9e7cd 3872ce99 58186b50 07e7adaf
     25248b58 a3dc4f71 781d21f2 df89b717 47bd54b3 23bbecc4
     43ec1d3e 020dadab bf782257 8255c104

X =  80c36d28 822e436f 372c2aec 001fd854 57b133d9

Y =  a93d5f7e bb51262d 9edcd656 c69cd26f 0a05c64 850f2616
     bcc71dbf 1748c292 1b64cf4a d99dfb84 76de6465 d61c97ee
     00bb50e4 237b04df a640b457 982bb59b
```

Figure 24: DSA Key Parameters

5.4 The Testbed

All components were deployed on P4s with the following characteristics:

- OS: Windows XP
- HD: 60 GB
- RAM: 512 MB
- CPU: 2GHz

The Web Services were deployed on BEA Weblogic 8.1.2 Application Server and the Ethereal Protocol Analyzer [9] was used to observe the network traffic.

5.5 The Tests

The tests conducted for this thesis aimed at measuring the time and network load overhead introduced by SWS. Three tests were used: the Individual, the Chain and the Full Chain tests, Sections 5.5.1, 5.5.2 and 5.5.3 respectively. Each test was run 1000 times and the average of these results, as well as there analysis, can be found in Section 5.6.

5.5.1 Individual Testing

In this test, only the Client and SWS components are used. The client builds a purchaseRequest message and secures it. Figure 25 shows this test using the *secureElement* inter SWS to enforce time stamping, sign, encrypt, all three and to check timestamping. Due to the implementation issues previously explained, neither decryption nor signature can be checked. In order to measure signature delay, the signature was implemented but the message returned did not contain the signed messge because it could not be sent. Due to its very nature, the *secureSendRemove* interface cannot be used.

Time delay was calculated between the time the message was built and the time the message was secured. Network load was measured at the Client. The same test is also done using libraries locally.

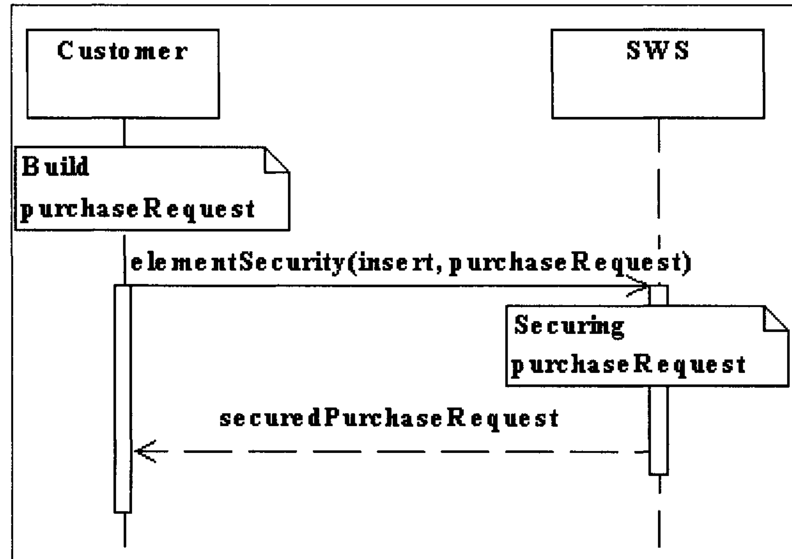

Figure 25: Individual Testing using SWS

5.5.2 Chain Testing

Chain Testing is the first test to involve all components. It consists of securing the Use Case presented in Chapter 5 where only the request messages are secured. Figure 20 on page 53 presents the Use Case studied and also represents the Use Case when WSS is enforced locally through libraries.

The Secured Use Case presented in Figure 21 (page 56) and whose Sequence Diagram is shown in Figure 26 represents this test when only the *elementSecurity* interface is used.

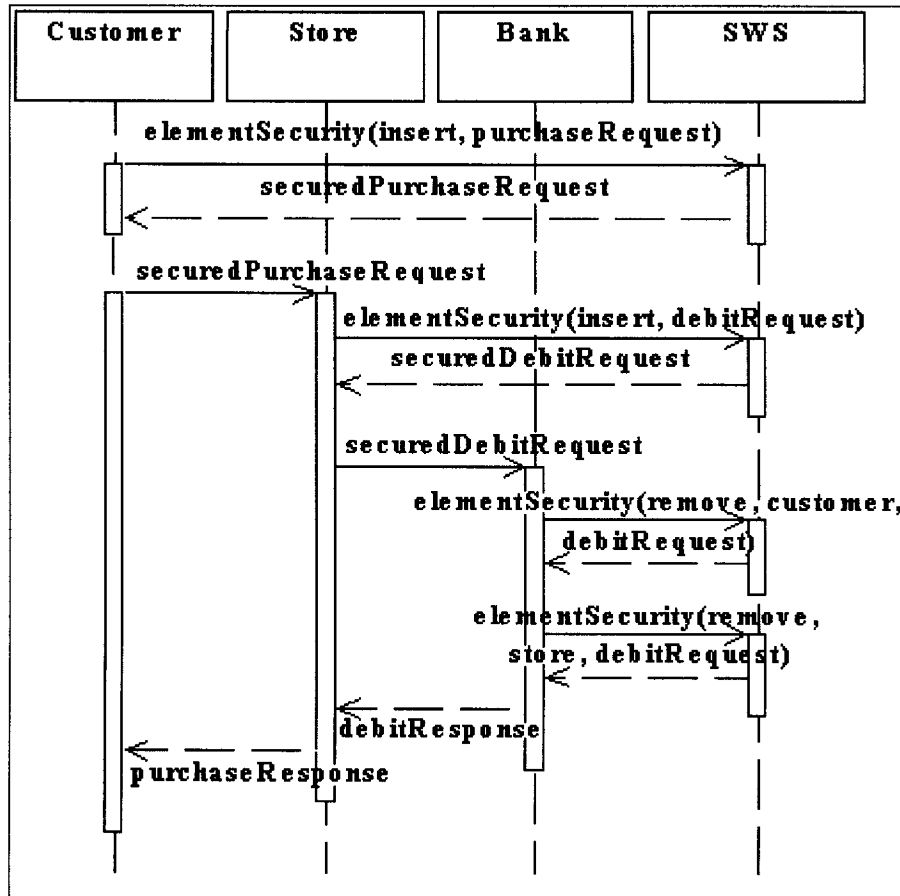

Figure 26: Chain Testing using the elementSecurity Interface

Although only one instance of SWS is represented on the picture for simplicities purpose. The Customer, Store and Bank each had their own SWS.

Figure 27 presents the same test using the secureSendRemove interface. Because this interface can only be invoked by an entity to secure an outgoing request and check its response, the *elementSecurity* interface must be used to check incoming requests and secure outgoing responses.

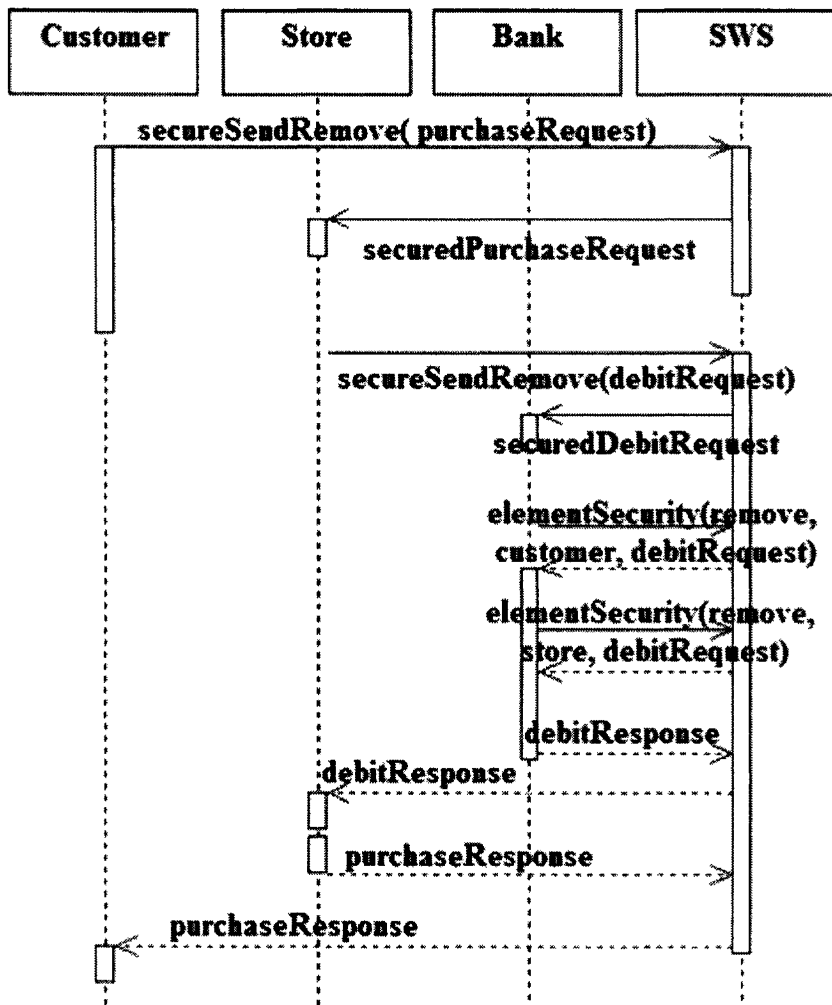

Figure 27: Chain Testing using the secureSendRemove Interface

5.5.3 Full Chain Testing

This test is very similar to the previous test, with for only difference the fact that response messages are also secured and must therefore be checked. Figure 28 presents the Sequence diagram of this test when the *secureSendRemove* interface is used.

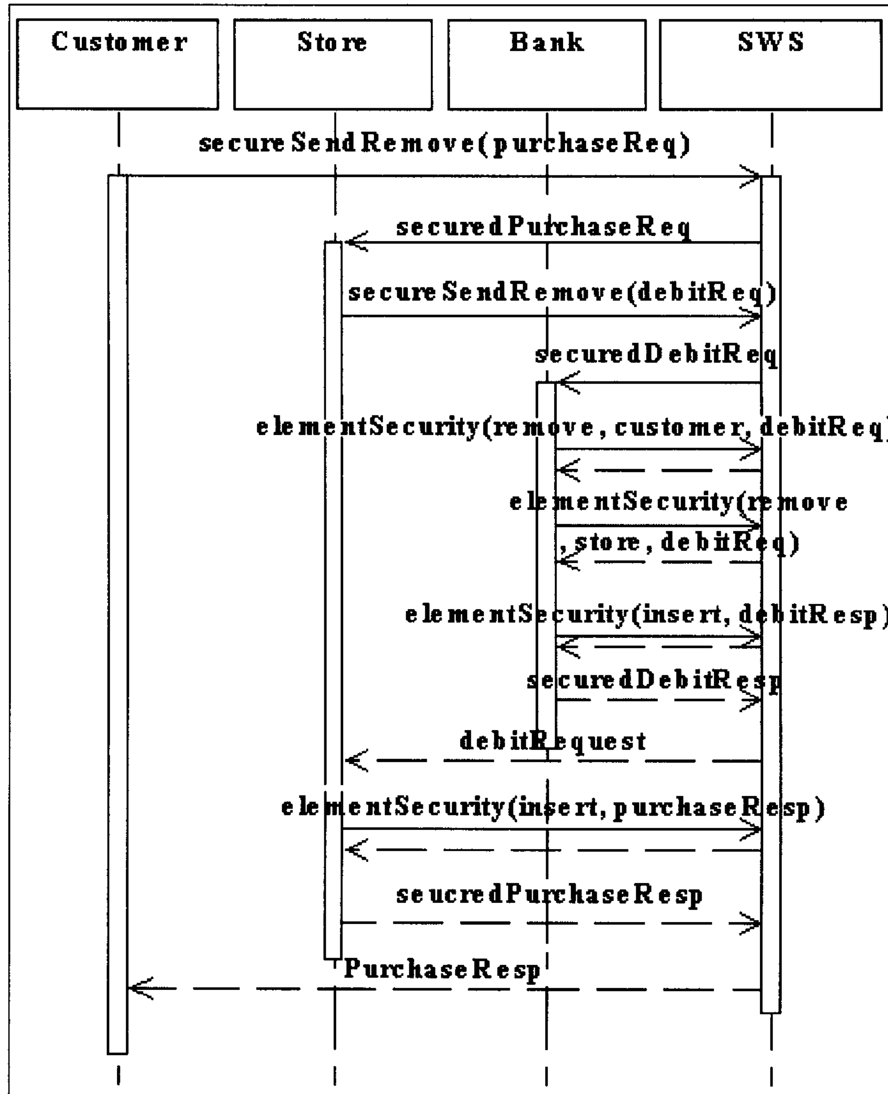

Figure 28: Full Chain Testing using secureSendRemove interface

5.6 Performance Measurements

This Section first presents and analyses the results of the tests presented in the previous Section. Delay Overhead is addressed in Section 5.6.1 and Section 5.6.2 deals with the Network Overhead. Note that each measurement was taken a thousand times and that the values here presented are the average of these measurements.

5.6.1 Delay Overhead: Results and Analysis

Table 1 presents the results obtained for Delay Overheads.

Table 1: Average Delay Overhead

	Lib (ms)	SWS (ms)	SWS – Proxy (ms)	Overhead (ms)	
				SWS	SWS - Proxy
Time Stamp (in)	~0.1	38.7	N/A	38.6	N/A
Time Stamp (out)	2.7	37.1	N/A	34.4	N/A
Encode	10.5	50.5	N/A	40	N/A
Decode	17.5	N/A	N/A	N/A	N/A
Sign	7.3	41.9	N/A	34.6	N/A
Check Signature	20.8	N/A	N/A	N/A	N/A
Timestamp, Encode And Sign	20.4	53.5	N/A	33.1	N/A
Chain Time Stamp	147.1	300	325	152.9	177.9
Full Chain Time Stamp	166.9	507.2	446	340.4	279.1

The column named Lib refers to measurements made using libraries. The columns named SWS, to measurements using the *elementSecurity* interface only and SWS-Proxy to measurements using the *secureSendRemove* interface combined with the *elementSecurity* interface.

"Timestamp (in)" represent inserting a timestamp while "Timestamp (out)" the checking of the timestamp. Encode and Decode refer to the use of XML-enc to secure a message, while Sign and Check Signature to the use of XML-Dsig.

The mention **N/A** means "Not Available" and refers to results that can't be obtained due to the limitations of the Application Server as explained in Section 5.3.1 (page 58). In

order to obtain values for "Sign" and "Timestamp, Encode and Sign", XML-Dsig was applied (so the time delay measured is valid) as well as the Security headers but the message returned does not contain the Signature and Digest values are set to null.

The overhead introduced for single operations (timestamp (in/out), Encode and signature) is lower than 40ms. The overhead introduced to "Timestamp, Encode And Sign" is also inferior to 40 seconds. This is mainly due to the fact that the SWS must convert the SOAP message received as a String into a SOAPmessage object and do the reverse at the end. These operations, and the time taken to communicate with the SWS, is what consumes most time when using a SWS and hence the overhead to accomplish three actions (time stamping, encoding and signing) is in the same range as the overhead for a single of these actions.

Furthermore, the overhead introduced when calling the *elementSecurity* is always smaller when removing/checking security features than when inserting them. This is because for a message to be secured using libraries, the message must be built from element by element and when this is done the developer can keep track of the necessary elements and then simply secure them. When doing so remotely, the String received must be parsed into a SOAP message, and the appropriate elements found before the same being done. However, when a message is receive, whether security is checked locally or not, the elements must be found and this is why the overhead in security checks is smaller than when inserting it.

In Chain Time Stamping, timestamps are applied twice and checked twice. If one looks at the overhead it takes to do this using the *elementSecurity* interface (SWS column), it

takes 152.9 ms, which is 6.9 ms superior to the theoretical value of 146 ms ((38.6 * 2) + (34.4 * 2) = 146 ms). When using a combination between both SWS interfaces, the overhead is of 177.9ms, 33.9 ms superior to the theoretical overhead of using only the *elementSecurity* interface.

In Full Chain Time Stamping, it is when the *secureSendRemove* interface is used, that the overhead is smaller 279.1 ms, instead of 340.4 ms when using *elementSecurity* interface alone. The explanation to this is quite simple. In Full Chain, security operations must be done on all messages, using the *secureSendRemove* interface to its optimal power. Because in the Chain Test only request messages were secured, the use of *secureSendRemove* interface forced responses to go through an interface that is superfluous, hence extra delay. But in Full Chain Test, the use of this interface reduces the number of messages, saving time.

Although no measurements were taken for a Full Chain using Encryption and Signature, From the individual tests, one can infer that it securing the same messages with all features would lead to a similar overhead. This overhead may seem large when compared to the time it takes to the same using libraries. However, because the Store and Bank were dummy services, this overhead (of less than 400ms for Full Chain Time Stamping) would seem smaller and smaller depending on the complexity of the implemented services.

5.6.2 Network Load Overhead: Results and Analysis

Table 2 presents the Network Load Overhead results for the same tests. Once again, "Lib" represents local libraries, "SWS" the use of the SWS *elementSecurity* interface and "SWS-Proxy" the use of *secureSendRemove* interface, in combination with the *elementSecurity* for the Chain and Full Cahin tests.

Table 2: Average Network Load Overhead

	Lib (KB)	SWS (KB)	SWS – Proxy (KB)	Overhead (KB)	
				SWS	SWS – Proxy
Time Stamp (in)	0	4.7	N/A	4.7	N/A
Time Stamp (out)	0	5.2	N/A	5.2	N/A
Encode	0	5.5	N/A	5.5	N/A
Decode	0	N/A	N/A	N/A	N/A
Sign	0	N/A	N/A	N/A	N/A
Check Signature	0	N/A	N/A	N/A	N/A
Timestamp, Encode And Sign	0	N/A	N/A	N/A	N/A
Chain Time Stamping	5	35.5	34	30.5	29
Full Chain Time Stamp	6.8	54	48.8	47.2	42

No Individual Tests using libraries generated a network load is generated, which means that any load generated by calling a SWS will be the overhead for this test. Due to the implementation issues explained in 5.3.1, some measurements could not be taken and are designed as "N/A". For the other individual tests, the overhead lies between 4.7 and 5.2 KB. Of course, this depends on the size of the message because the message itself is sent to be secured. In fact, the overhead is equal to the message size, plus the processed

message plus the size of the used options out (e.g. when signing, some variables must be filled which could otherwise be left as null).

For Chain Testing, even the tests using libraries generate a load because the Store and Bank are called. The overhead is therefore calculated as the network load generated when using SWS minus the network load generated when using libraries. The same is valid for Full Chain Testing.

Not that in both Chain and Full-Chain tests, the use of SWS *secureSendRemove* reduces the network load. This is due to the smaller number of messages necessary. However, if the interface could have been implemented as originally designed, this would not have been the case in Chain Testing (currently 1.5 KB difference only) and may have been or not the case in Full Chain (5.2 KB of difference). Indeed, implementing the interfaces as designed would mean that the *secureSendRemove* response messages would include the processed and the original response message rather than only the processed message as is now the case. Whether this difference would depend on the message size or not would require testing that cannot currently be done.

The main issue with the network load generated lies in the fact that it is directly proportional to the size of the message and the larger the message, the larger the overhead.

Chapter 6 Conclusions and Future Work

We conclude by presenting in this chapter, the main contributions of this thesis as well as the lessons learned from it. We also indicate possible research direction for future work.

6.1 Contribution of this Thesis

In this thesis we have explained the necessity of a new way of enforcing Web Services Security (WSS) in a granular, lightly coupled and flexible manner. We then introduced Security Web Services (SWS) as an alternative to the traditional libraries and proxies.

The proposed SWS is composed of two main interfaces: the *elementSecurity* and the *secureSendRemove* interfaces. These two interfaces allow the SWS user to secure (check/remove security of) a message element (signature, encryption and time stamping) in one single step. The second interface allows a user to reduce the network load by having the SWS forward the secured message and only return once it has received and processed the response. Thanks to these two interfaces, the entity wishing to secure its messages can do so remotely in a granular, flexible, lightly coupled and end-to-end manner.

A comprehensive Use Case was then presented and it was shown how it could be secured using SWS. A prototype of this Use Case, adapted to the available technology, was implemented and some performance tests were conducted. Performance was evaluated in terms of overheads for both time delay and network load.

Performance measurements were taken by comparing the time delays and the network loads between the prototype when it uses the library and when it uses the different SWS interfaces.

Through this Case Study and performance measurements, we have learned that:

- The first lesson is that the SWS approach seems suitable to secure SOAP communications, allying the advantages of both libraries and proxies. From the libraries, it retains the flexibility in terms of functionality, the lack of profiles to pre-configure and the granularity of the security provided. From the proxies, it retains the possibility of executing expensive calculations remotely and the possibility of swapping security provider in an easier way than with libraries.
- Secondly, the SWS can be used by either the Service Requesters, Service Providers or both, without forcing other parties involved in the transaction to also use a SWS. SWS can therefore be used with other means of enforcing security as long as they all support WSS. Furthermore, SWS could be used with another technology to secure the same message.
- Thirdly, the SWS allows easy secure service development. Using the SWS to secure messages with one invocation is shorter and requires less knowledge about the WSS technologies than using the libraries.
- The fourth lesson is that the time delay overheads introduced are not very significant. In the use case studied, it added at most 340 ms. This may seem considerable when compared to the time delay of ~167ms when using libraries, but one must take in consideration the fact that the Store and Bank where

dummies. So as soon as these two must go into databases to find the client, do the debit, process the order and so on, the processing time increases and the overhead seems even smaller. Furthermore, 340 ms, not even half a second, is a very small delay that most users will not notice and/or would be willing to wait in order to secure their personal and/or sensitive information.

- Fifthly, the overhead involved in applying all WSS features (signature, time stamp and encryption) is smaller than for time stamping alone. One can infer from this that if the overhead involved to apply all features would be sensibly the same, or smaller as the one obtained for time stamping.
- Sixthly, the current Application Servers do not yet support all the functionality necessary to implement the SWS as designed. The lack of in-out parameters can be worked around but it reduces the information available to the entity using the SWS when something goes wrong and, in the *secureSendRemove* interface, the SWS user no longer has access the secured message as received. If the user needs to keep track of this message, this may turn out to be problematic.
- Finally, the network load is directly proportional to the size of the message which is being handled. SOAP messages being XML documents and potentially extremely heavy, it may be considered a problem for certain messages. However, as the network bandwidths increase, this should not be a consideration in the long term.

6.2 Future Work

In the Use Case studied, the Client, Store and Bank called a SWS in their own trusted domain. This eliminated the need to secure the requests sent to the SWS and the responses returned by the same. Important attention must be brought to the case when the SWS and its user are in different domains and their communication must be secured in a granular and durable way. Message level security cannot achieve this (security information is transient) and if the entity wishes to use a SWS, one can safely assume that it cannot enforce WSS. Further investigation is required as no easy solution is foreseen.

As the current Application Servers do not support XML-enc or XML-Dsig at the programming level, only when the server itself is used as a gateway applying WSS, further testing must be done in order to properly measure delays and network load when using all the functionality.

As the current Application Servers do not support in-out, or out, parameters, further testing will also have to be conducted when these are available. SWS will then be re-evaluated.

Network load may be problematic in a transaction involving many entities, or many messages, as the overhead is proportional to the size of the messages used. This issue will need investigation as more people use Web Services requiring security.

Chapter 7 References

[1] J. Roy and A. Ramanujan, "Understanding web services", *IEEE IT professional Magazine*, November 2001, vol. 3, no. 6, pp. 69-73.

[2] D. Eastlake and J. Reagle, "*XML Encryption Standard and Process*", W3C Recommendation, December 10 2002, (current October 2004) http://www.w3.org/TR/xmlenc-core/

[3] E. Simon, P. Madsen and C. Adams, "*An Introduction to XML Digital Signatures*", XML.com, Aug. 8 2001, (Current October 2004) http://www.xml.com/pub/a/2001/08/08/xmldsig.html

[4] A. Nadalin, C. Kaler, P. Hallam-Baker and R. Monzillo, "*Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*", version 1.0, OASIS Standard 200401, March 2004, (Current October 2004) http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

[5] "*OMA Web services Enabler (OWSER): Overview 1.0*", OMA-OWSER-Overview-V1_0-20040715-A, Open Mobile Alliance (OMA), Mobile Web Services (MWS), 15 July 2004, (Current December 2004) www.openmobilealliance.org

[6] "*OMA Web Service Enabler (OWSER): Core Specifications - Approved version 1*" OMA-OWSER-Core-Specification-V1_0-20040715-A, Open Mobile Alliance (OMA), Mobile Web Services (MWS), 15 July 2004, (Current December 2004) www.openmobilealliance.org

[7] D. Eastlake, J. Reagle and D. Solo, "*XML-Signature Standard and Process*", W3C Recommendation, February 12 2002, (current October 2004) http://www.w3.org/TR/xmldsig-core/

[8] D. Eastlake, J. Reagle and D. Solo, "*XML-Signature Standard and Process*", IETF RFC 3275, March 2002, (Current October 2004) http://www.ietf.org/rfc/rfc3275.txt

[9] "*The Ethereal User's Guide*", ETHEREAL, (Current June 2004) http://www.ethereal.com/docs

[10] "*Java Cryptography Architecture: API Specification and Reference*", Sun Microsystems, August 2002, (Current August 2004) http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html

[11] "*Java Cryptography Extension (JCE) Reference Guide, for the Java 2SDK, Standard Edition*" version 1.4, Sun Microsystems, January 2002, (Current August 2004) http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html

[12] G. Brose, "Securing Web Services with SOAP Security Proxies", *In the Proceedings of the 2003 International Conference on Web Services - ICWS 2003*, (Las Vegas, USA, June 2003), pp. 231-234

[13] "*Digital Signature Standard (DSS)*", Federal Information Processing Standards Publication 186 (NIST FIPS 186), National Institute of Standards and Technology, 19 May 1994, (Current December 2004) http://www.itl.nist.gov/fipspubs/fip186.htm

[14] "*Secure Hash Standard*", Federal Information Processing Standards Publication 180-1 (NIST FIPS 180-1), National Institute of Standards and Technology, 17 April 1995, (Current December 2004) http://www.itl.nist.gov/fipspubs/fip180-1.htm

[15] *"Data Encryption Standard (DES)"* Federal Information Processing Standards Publication 46-2 (NIST FIPS 46-2), National Institute of Standards and Technology, 13 December 1993, (Current December 2004) http://www.itl.nist.gov/fipspubs/fip46-2.htm

[16] H.T.Kung, F. Zhu, M. Iansiti, "*A Stateless Network Architecture for Inter-enterprise Authentication, Authorization and Accounting*", *In the Proceedings of the 2003 International Conference on Web Services - ICWS 2003*, (Las Vegas, USA, June 2003), pp.235-242

[17] Virdell, M. "*Business processes and workflow in the Web services world*", 1 January 2003, http://www-106.ibm.com/developerworks/webservices/library/ws-work.html (Current March 2005)

[18] "*Parlay 4.1 Specification*", The Parlay Group, (Current June 2004) http://www.parlay.org/specs/index.asp

[19] A.-J. Moerdijk, L. Klostermann, "*Opening the Networks with Parlay/OSA: Standards and Aspects Behind the APIs*", IEEE Network Magazine, May 2003, pp. 58-64

[20] "*Parlay X Web Services Specifications, Version 1.0*", The Parlay Group, (Current March 2005) http://www.parlay.org/specs/index.asp

[21] "*CORBA/IIOP Specification*", Object Management Group, (Current June 2004) http://www.omg.org/technology/documents/formal/corba_iiop.htm

[22] "*A conversation with Adam Bosworth*", ACM Queue, Vol. 1, No. 1, March 2003

[23] W3C, Web Services Activity, (Current June 2004) http://www.w3.org/2002/ws/

[24] T. Thompson, R. Weil and M. D. Wood, "*CPXe: Web Services for Internet Imaging*", IEEE Computer Magazine, October 2003, pp. 54-62.

[25] John Fontana, "*Boeing lets single sign-on project fly*", Network World Fusion, July 14 2003, (Current March 2005) http://nwfusion.com/news/2003/0714boeing.html

[26] "The Vordel View - SSL and Web Services - myths and facts", Vordel Limited, (Current March 2005) http://www.vordel.com/knowledgebase/vordel_view4.html

[27] 3GPP, (Current June 2004) http://www.3gpp.org

[28] 3GPP Specifications, "*Open Service Architecture: Application Programming Interface*", (Current Aug. 2004) http://www.3gpp.org/ftp/Specs/html-info/TSG-WG--N5.htm

[29] W3C, Web Services Activity, (Current June 2004) http://www.w3.org/2002/ws/