Master's thesis

Web Services Security

Robert-Jan Boezeman

University of Nijmegen:
Security of Systems (SoS) group
University of Oxford:
Software Engineering Programme of the
Oxford University Computing Laboratory

June 10, 2003

Contents

List of Figures

Abstract

This thesis describes the different security aspects of web services and the technologies they use. It describes a framework to introduce message level security to the Simple Object Access Protocol (SOAP), the protocol used by web services. This is done by using digital signatures and the encryption of elements of SOAP elements to setup a security context. Furthermore, this thesis describes a system designed for the Software Engineering Programme (SEP) of the Oxford University Computing Laboratory (OUCL). The purpose of this system is to enable an end-user, via a website, to locate documents on other servers and to access a database on another server. Once the documents have been located, the web services provide the end-user with a way to create a printable format of each document and to send it to different printers. Also, the web services enable the end-user to transfer a password file from the database on one server to another server in order to merge the retrieved password file with the existing one. Once they are merged a web service can deploy the new password file. Finally, this thesis discusses the problems with certain security aspects of achieving message level security within SOAP. U

Preface

This thesis is the end result of an internship I did at the Software Engineering Programme of the Oxford University Computing Laboratory, from September 2002 till April 2003. This internship has been the last fase of my study of Computing Science at the University of Nijmegen. I have done my research under the supervision of Dr. Andrew Martin from the University of Oxford and Dr. Jaap-Henk Hoepman at the University of Nijmegen. I would like to take the opportunity to thank Andrew, Jaap-Henk and everybody at the computing laboratory for their help and support. Also, I would like to thank my family and friends very much for supporting me. Without them, this thesis would not have been written. I would not have learned as much, not only about computing science, as I have in the last half year.

Chapter 1

Introduction

This thesis is a report of the research done for the Software Engineering Programme (SEP) of the Oxford University Computing Laboratory (OUCL). It describes the security aspects of web services and the technologies they use. It describes a framework to introduce message level security to the Simple Object Access Protocol (SOAP), the protocol used by web services. This is done by using the latest specifications[1] to incorporate digital signatures and the encryption of elements of SOAP to setup a security context. Furthermore, this thesis describes a system, designed for the SEP, that makes use of these specifications and a security context to setup a secure network of web services. The purpose of this system is to enable an end-user to locate documents via a website on other servers and to access a database on another server. Once the documents have been located, the web services provide the end-user with a way to create a printable format of each document and to send it to different printers. Also, the web services enable the end-user to transfer a password file from a database on one server to another server in order to merge the retrieved password file with the existing one. Once they are merged, a web service can deploy the new password file. First, this introduction will give a general introduction to web services and the way they evolved from XML-Remote Procedure Calls into the SOAP protocol and why SOAP is by far the most widely used protocol by web services.

1.1 Web services

A web service is an application running on a computing system that can be accessed from a network. It can perform any number of operations and can be accessed using messages sent to it, that are described using the eXtensible Markup Language (XML, see [1]) or an XML-based protocol, such as SOAP. Before SOAP was developed, XML-based implementations of remote procedure calls were already in use (the protocol being named XML-RPC). Unfortunately XML-RPC was quite verbose and did not have good data typing. Its successor is SOAP, the de facto standard for web services today. The protocols used by web services are highly standardised and most of them are being developed and

[1] More on these later.

maintained by the World Wide Web Consortium (W3C)[2] and the Organization for the Advancement of Structured Information Standards (OASIS).
Web services provide a way to do distributed computing in a platform-independent way. In having this property, they can connect different computing systems so that they are able to use each others resources (such as information or processing time). There are other technologies available that do exactly the same thing, however web services have certain benefits over other technologies:

- Web services are very suitable to integrate completely different computing systems which each other, for example Unix platforms and Windows platforms. They try to achieve high levels of interoperability between applications by using Web standards. This advantage is one of the biggest appeals for big IT companies such as IBM and Microsoft to improve e-business.

- Web services are very fast and cheap to develop. There are many tools available that make use of a certain description of a web service, specified in the Web Service Description Language (WSDL)[3]. For an explanation on how WSDL works, see [3]. Using such a description, these tools generate nearly compilable code in a certain programming language; for example WSDL2Java is a tool that generates Java code from a WSDL-description. The developer only has to fill in the blanks and the web service has been created. There are even more advanced packages such as Apache Axis[4] that enable developers not having to think of SOAP or the other technologies web services use. They just write the operations they want their web services to have and Axis builds web services out of those operations.

- Web services are easy to deploy. There are various containers for different transport layer protocols such as FTP, SMTP and especially HTTP to quickly deploy a web service. With most of them, simply inserting the web service in the container and reloading the container is sufficient.

- Web services are easy to discover and invoke. Since every web service that uses a Remote Procedure Call message exhange model[5] can have a WSDL-description, these descriptions can be published in registries. The advantage of doing this is that clients of web services can retrieve these descriptions and invoke them accordingly. An example of such a registry is a registry that conforms to the "Universal Description, Discovery and Integration" or UDDI standard, see [4]. UDDI is a standard that defines how a (client) web service can *publish*, *find* and *bind* descriptions of web services. In binding a description, a client knows exactly how to invoke a web service. More information on binding a description can be found in section 2.2.1.

These advantages combined let endpoints (businesses) connect their computer systems after they have been designed and built. Inter-application communication across a network (the Internet) can be established at run-time instead of

[2]Website can be found at http://www.w3c.org/
[3]WSDL is also developed by the W3C, see [2] for a formal description
[4]For Apache Axis, see http://ws.apache.org/axis/index.html
[5]More on this later.

having to design it, resulting in loosely coupled systems. Each application can be changed or modified more easily without upsetting the communication with other applications.
SOAP, WSDL and UDDI are the cornerstones of web services today. The reason that computer systems can be connected in such a way, is a result of the standards emerging from organisations such as the W3C and OASIS[6]. As long as everybody uses these standards, the level of interoperability stays the same.

1.2 Architecture of web services

Web services use open Internet standards as the basis of communication. Consider the following diagrams:

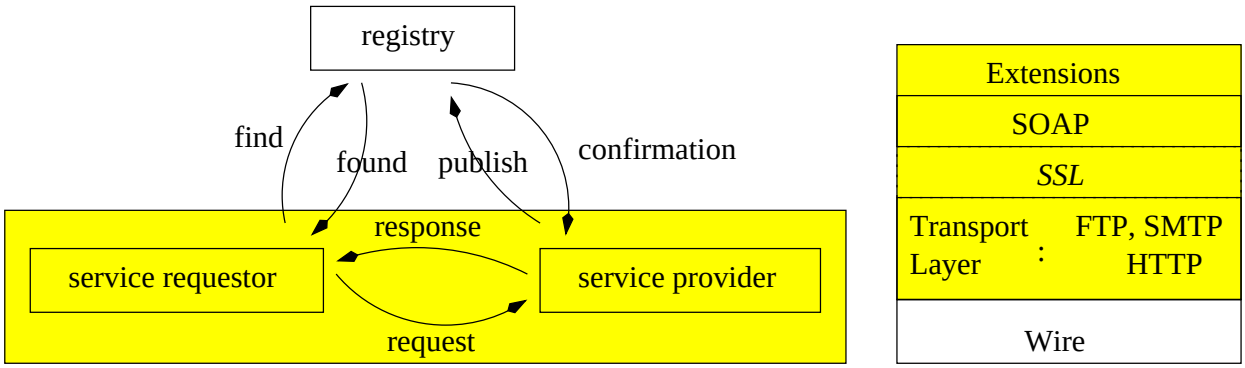

Figure 1.1: The discovery of web services and the SOAP stack.

The yellow parts depict the research done for the SEP, the topics discussed by this thesis. The left figure illustrates how web services can be found and invoked dynamically by a service requestor. The service provider publishes a description of the operations it provides to a registry. Whenever a service requestor needs to invoke one of those operations, it can look them up in the registry. A requestor retrieves the WSDL-description and in doing so, it knows exactly how to call the operation. It knows the format and the syntax of each message/request it can send to the provider. This setup is very common in the web services world. An example of such a registry is the "IBM UDDI registry"[7]. Requests and responses to web services are carried over standard transport layer protocols such as SMTP, FTP and HTTP but are protocol independent, a simple TCP-socket connection can be sufficient. SSL can be added to that stack as an option to increase the level of security. There is a paper [5] about SOAP and the use of digital signatures with SOAP in combination with SSL to take care of security issues such as authentication, integrity, confidentiality and non-repudiation. In many cases this gives a good enough solution for secure web services, however not always. More on this later.

[6]Website can be found at http://www.oasis-open.org/
[7]The IBM UDDI can be found at http://www-3.ibm.com/services/uddi/

1.3 Research and the problem description

This thesis is a report of the research done for the SEP and has the following topics:

- It focusses on the communication between the service requestor and the service provider and how the proposed security extensions can be used or improved upon to establish secure communication between them. The way web services can be published and found is beyond the scope of the research done for the SEP. The question here was how the communication between the service requestor and the service provider works and how it can be secured.

- To do so, it gives a thorough analysis of SOAP in chapter 2 and the way it binds to the transport layer protocols. An important part of the research was to analyse SOAP and to determine if web services and SOAP can be used to design a system for the SEP to connect their computing systems that they use for their administration. The system designed and built for the SEP is discussed in detail in chapter 4. The question here was whether or not it was possible to design a secure network of web services. The (security) requirements for this system have been analysed and can be found in [6].

- Finally, the proposed (security) extensions to SOAP, discussed in chapter 3 and how they can be used to setup a session/security context within SOAP are discussed in detail. It was a big part of the research done for the SEP. The system for the SEP uses such a SOAP session and is discussed in chapter 5. The last chapter draws conclusions about the performed research and the results.

Chapter 2

The Simple Object Access Protocol

The Simple Object Access Protocol (SOAP) is the standard protocol used by web services today. It is an XML-based messaging protocol that enables computing systems to communicate with each other, without having to think of the type of operating system or the environment each system is in. It is a lightweight protocol of which the structure of the contents of messages can be defined by the Web Services Description Language (WSDL) and XML Schemas[1]. This chapter explains what the structure and syntax of a SOAP message is, how SOAP messages must be processed, which features SOAP has, how it binds to the underlying protocols and how peers can use these protocols when exchanging SOAP messages. The focus here lies on the advantages and disadvantages. Finally, a brief explanation of WSDL, the Web Services Conversation Language (WSCL) and the Business Process Execution Language (BPEL) is given and the ongoing work on SOAP.

2.1 Using SOAP

SOAP is a messaging protocol designed to let a SOAP-message sender and a SOAP-message receiver exchange information, without having to know any details about each other on forehand. It is also a simple protocol because it leaves out many features that can be found in distributed systems, such as reliability, attachments, quality of service, routing, privacy and security. Instead, there are extensions to SOAP to cope with some or parts of those features. This makes SOAP a very small protocol and makes it possible for designers to completely define their own applications with SOAP. Furthermore, SOAP is extensible via the use of XML. This means that SOAP can be used/adapted for any situation or application, without going beyond the boundaries of the latest SOAP specification[2] by the W3C, see [7].

[1] How XML and XML Schemas work are beyond the scope of this thesis, more information on both of them can be found in [1]

[2] The latest recommendation is version 1.2 at the time of this writing.

SOAP consists of three parts:

- An envelope in which you can define what is inside a SOAP message and how you must process it.
- A set of encoding rules for defining and processing datatypes that can be defined using XML Schemas.
- A method of making Remote Procedure Calls and Reponses to those calls.

2.1.1 Structure and syntax

A SOAP message consists of an envelope with two sub-elements, a header and a body. The header element is optional and is included in the SOAP specification to make it possible to include information or meta-data about the contents of the message. This can be anything from routing information (as used in the routing extension to SOAP [8], more on this later) to processing rules on how to process the SOAP message. When using a header, SOAP can be extended so it can be used in various scenarios. Note that the version 1.0 in the example is the version of XML, not the version of SOAP.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://www.ox.ac.uk/reservation"
                 env:actor="http://www.ox.ac.uk/reservation"
                 env:mustUnderstand="true">
    <dateAndTime>2003-06-12T13:36:50.000-05:00</dateAndTime>
    <passenger env:mustUnderstand="true">
       <name>Robert-Jan Marijn Boezeman<name>
    </passenger>
  </m:reservation>
 </env:Header>
 <env:Body>
   <p:itinerary xmlns:p="http://www.comlab.ox.ac.uk/reservation/travel">
     <p:departure>
       <p:departing>AMS</p:departing>
     </p:departure>
     <p:return>
       <p:arriving>LHR</p:arriving>
     </p:return>
    </p:itinerary>
 </env:Body>
</env:Envelope>
```

Figure 2.1: A SOAP example of a flight reservation

The (mandatory) body element of a SOAP envelope contains the actual information being transferred, intended to be processed by the final recipient of the SOAP message. However, the SOAP specification does not say anything about the structure and syntax of the sub-elements of the header or the

body element. In fact, SOAP can be used to exchange *any* XML-document as long as it is well-formed XML. Appendix H illustrates a SOAP envelope with a header and a body element with attributes. As with normal XML, an element of a SOAP envelope can have zero or more attributes, giving more information about the element or referencing others. In the example the envelope itself is in the namespace `http://www.w3.org/2003/05/soap-envelope`, indicating which version of SOAP is used. A SOAP application must be able to process all SOAP namespaces in messages that it receives. Whenever a message contains an invalid namespace in the SOAP parts, the application can discard the message as invalid or send a SOAP-fault message back, more on this later. After the `<env:Envelope>` element, the optional `<env:Header>` element contains three sub-elements which are relevant to the application, not to SOAP itself. Therefore those elements can exist in their own namespace. The `<m:reservation>` element has two attributes: the `actor`-attribute and the `mustUnderstand`-attribute. The `actor`-attribute indicates the final actor that is supposed to process that header element. The `mustUnderstand`-attribute means that the header element has to be understood by the final recipient, it is not optional. Both attributes are in the same namespace as the envelope itself, therefore they are a part of the SOAP communication and not only intended for the final recipient. If there are intermediaries involved, any header element may be removed or changed by an intermediary.

2.2 Message Exchange Patterns

SOAP is a message exchange protocol that can have an unlimited number of setups, for example one-way messaging, request/response messages and conversations. SOAP itself does not define a specific way, it is essentially designed for sending a single message from one sender to a receiver. However, it can be used for various message exchange patterns. With web services this is done via the use of the Web Services Description Language (WSDL) or Web Services Conversation Language (WSCL). However WSCL is, at the time of this writing, being superceded by the Web Services Choreography Interface (WSCI) which in its turn is being superceded by the Business Process Execution Language (BPEL). Keeping up with these seems an endless challenge.

2.2.1 Web Services Description Language (WSDL)

SOAP itself does not provide a way of predefining the format of a particular message. When an application wants to publish a service to the outside world, users of those services must know exactly how to invoke them. WSDL has been designed specifically for that purpose. It is a language that describes the format of the SOAP message a requestor must send in order for the recipient web service to understand the message. Furthermore, WSDL defines how SOAP can be used with the underlying protocols (such as SMTP, HTTP, etc.), where a web service can be found, at which port and what operations there are available. It defines four basic operations:

- One-way messaging, a client only sends information to a web service and never receives any messages back.

- Notification, the inverse of one-way messaging where a web service only sends information to the client and never receives information back.
- Request/Response messaging, this is the most widely used setup, also known as Remote Procedure Calls, where a client makes a request to a service and gets a response back. WSDL defines the structure and syntax of the requests and responses.
- Solicit/Response, the inverse of request/response messaging, the service pushes output to the client and the client sends information back. Also, WSDL defines the format of the messages.

These are the only methods of messaging WSDL provides. To go into detail how exactly WSDL defines these methods or what the structure and syntax of WSDL-definitions is, is not the purpose of this paper; just an understanding of what WSDL does, will suffice. For more information on WSDL, please see [2] or [3].

2.2.2 Web Service Conversation Language (WSCL)

Since WSDL only allows for four message exchange patterns, the W3C is developing WSCL. WSCL can be used to design entire "XML conversations". With SOAP, it defines the sequence of the XML documents being exchanged and what the format of those documents must be. Using this language, requestors and providers are no longer limited to two messages at a time. WSCL may use WSDL as a building block to exchange those XML documents. WSCL is especially useful for business that want to describe their business processes. WSCL is relatively new and it is hardly being used anywhere at the time of this writing. It may be used to address the problem of providing support for transactions that involve the exchange of multiple messages, one of the shortcomings of SOAP see section 2.3. For more information on WSCL, please see [9].

2.2.3 Business Process Execution Language (BPEL)

BPEL is a new language for describing business processes that make use of web services. With BPEL, business processes that consist of the sequential operation of web services can be specified. It enables you to model a business process in XML in such a way that the operations of web services are called in the right order and with the right parameters. Where WSDL describes the grammar of a single web service and its operations, BPEL enables a business to combine several web services as one. This is a relatively new specification (May 2003) and provides the designers of web services with another way of having a conversation between web services.

2.3 Shortcomings of SOAP

The intent of the designers of SOAP was to keep it a small protocol that could be extended so that it can be used for any situation or application. Therefore, SOAP has a number of shortcomings/features that it lacks. This section discusses which features it lacks and which of those are a part of the research. For

some of these features, there are extensions to SOAP that try to take it into account, but not for all of them:

2.3.1 Reliable messaging

Reliable messaging means that when a client and a web service are exchanging messages, that it is possible to guarantee delivery. SOAP does not have any method of detecting whether or not a message has been lost in transit nor does it have a method of asking to resend a message. Also, when an adversarial intermediary has duplicated a message and resends it (replay attack), there is no method in SOAP to detect this. In chapter 5 a security context is introduced where message numbers and session id's are used to address these problems.

2.3.2 Attachments

Whenever two entities want to use SOAP to exchange non-XML or binary data, there is no way in SOAP to do this. There is already a preliminary specification from the W3C, see [10]. Within this specification, content is encoded using MIME. This shortcoming is not a part of the research done for this thesis.

2.3.3 Routing/intermediaries

SOAP messaging is specifically designed with intermediaries in mind. The initial SOAP specification [11] already mentiones SOAP applications processing SOAP messages that have not reached their "ultimate destination". The latest version of SOAP [7] explains how intermediaries should perform relaying of SOAP messages in terms of processing the headers of the messages. It does not specify however what the structure and syntax of those routing elements is. There is an extension designed by Microsoft that tries to incorporate routing into SOAP, see [8]. When routing SOAP messages, new security issues arise. These issues are also a part of the research done, please refer to section 5.1.

2.3.4 Quality of Service

When a business provides services to its customers, those customers can expect a certain level of availability and performance from those services. At the time of this writing SOAP does not have any mechanisms to provide for availability, nor can it influence performance. During the design of the network of web services for the SEP (see section 4) it was an important issue, because many users may use the designed system. With web services, quality of service has to be dealt with at another level.

2.3.5 Transaction support

Very often business processes involve exchanging multiple messages. Whenever web services need to perform a group of operations it might be necessary to let that group of operations succeed or fail. Currently SOAP does not provide a way of doing this. Now, all transactions with SOAP are short in the way they can only perform request/response transactions. WSCL (see 2.2.2) and BPEL (see 2.2.3) might provide a solution to this problem in the future. It is not a part of the research done for the SEP.

2.3.6 Security

The latest version of SOAP [7] [3], does not specify how to handle authentication in SOAP, nor does it predecessor version 1.1, see [11]. Other security related issues such as integrity, confidentiality and non-repudiation have also not been included in SOAP to keep it a lightweight protocol.
Furthermore, SOAP messaging is specifically designed with intermediaries in mind. This means that SOAP messages traversing a certain path, can be intercepted, read, modified and removed. However, the problem of making SOAP messaging confidential between two computersystems can be addressed by Transport Level Security (TLS) with for example Secure Sockets Layer (SSL). SSL appends Message Authentication Codes (MAC's) to the transmitted messages to ensure message integrity, which would preserve the integrity of the SOAP messages transmitted with SSL as well.

However, when various intermediaries (may be adversaries) have to read the SOAP message to determine who's next in the communication chain, TLS is not sufficient. When using SSL to encrypt communications, a separate SSL connection between each pair of nodes on the path of the message is needed. In this situation you would only have hop-to-hop security instead of end-to-end security. It is possible that security has been breached on one of the web services along the way. Therefore secure SOAP messaging is necessary. Also, the idea behind SOAP messaging is to provide loosely coupled systems with a way that they can communicate with each other in a connectionless way. For example, there could be a message queue that a receiver has to process first. This way the sender of a message must wait for the receiver to finish the queue, here SSL is not a good solution to this problem.
This thesis goes indepth on the (proposed) extensions of SOAP to achieve a higher level of security and is large part of the research done for the SEP.

[3] The latest version is 1.2, the working draft from the W3C

Chapter 3

SOAP security extensions

A big part of the research done for the Software Engineering Programme was to build a secure network of web services. Since web services use SOAP, it became necessary to find a way to use SOAP in a secure way. This chapter gives a detailed analysis of the extensions to SOAP used to create secure SOAP conversations. The security extensions combined, address the relevant security issues. These issues are:

authentication
: Web services communicating sensitive information with each other must know without a doubt, that they are communicating with the web service they think they are. Authentication is therefore needed at the message level: this issue should not be resolved at another level, say for example the transport layer, because SOAP messages may be routed through various intermediaries.

integrity
: In computing science, integrity means information sent from a sender should arrive at its intended receiver in the exact the same way it was sent. No adversary must be able to change its contents/meaning along the way, without it being detected. For example, web services may perform critical tasks when receiving a SOAP message. This message must not be altered along the way, making a web service perform the wrong action. Therefore the integrity of SOAP messages must be preserved in transit. The integrity of a SOAP message is preserved, when an adversary fails to keep the validaty of a SOAP message when trying to change it.

confidentiality
: Confidentiality means protecting information in a way that only the intended receiver of the information can interpret its semantics. Web services communicating sensitive information must be protected from adversaries that are eavesdropping. Confidentiality for web services means that SOAP messages must be protected (encrypted) so that eavesdropping doesn't result in exposure of the information being transmitted.

To take care of these security issues, the following specifications are relevant. First, the specification of digitally signing a SOAP message (SOAP-DSIG). This specification can be found in [5]. This paper extends XML Signatures [12] into

SOAP to digitally sign messages. It provides a method of associating keys with referenced data and is therefore used in SOAP-DSIG. To ensure that SOAP messages stay confidential if needed, this thesis uses another specification-paper produced by IBM and Microsoft, about web services and encryption [13]. These additions to SOAP enable the web services in the system for the SEP to communicate in a secure *enough* way for that system, meaning having addressed the risks as explained in [6]. That is, SOAP messages must be exchanged making sure that a requesting web service has been authenticated, the integrity of the message is preserved in transit and that the message is confidential. The different elements added to the SOAP messages to make it secure are specified in this chapter.

3.1 Securing SOAP

In this section, the way the SOAP messages will be constructed is explained. Adding security to SOAP involves making use of a new element in the header of a SOAP message, the `<Security>` element. This element can be separated from the rest of the SOAP message, because it must be in the following namespace: `"http://schemas.xmlsoap.org/ws/2002/04/secext"`. The prefix `wsse` to the `<Security>` element stands for "web services security extension", but it may be replaced by any other. To give an example of how this element resides in a SOAP envelope, consider the following:

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope"
            xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
   <env:Header>
      ...
      <wsse:Security S:actor="..." S:mustUnderstand="...">
         ...
      </wsse:Security>
      ...
   </env:Header>
   <env:Body>
      ...
   </env:Body>
</env:Envelope>
```

Figure 3.1: A SOAP envelope containing a security element

The `<Security>` element is put in the header block of the SOAP envelope and may have two attributes, in this case `S:actor` and `S:mustUnderstand`. When a SOAP message is routed via various intermediaries, those intermediaries are allowed to add and remove any elements contained within the header of the SOAP message. The `S:actor` attribute indicates the intended recipient of the `<Security>` element. If there are multiple `<Security>` elements in a header block, only one may omit the `S:actor` attribute and no two `<Security>` elements may have the same actors. The one omitting the `S:actor` attribute

is intended to be processed by the final recipient. The `S:mustUnderstand` is truly optional and indicates whether the intended actor must be able to process the element. Note that these attributes are in the namespace of the envelope itself and not the `<Security>` element, because the actors specified should be the same actors as specified in the routing path, as defined in the WS-Routing specification [8]. Within this `<Security>` element, it is possible to define digital signatures and specify the way elements in the body are encrypted.

3.2 SOAP D-Sig

The SOAP D-Sig specification [5] is a document that describes the rules for the syntax and the processing of an element that can be added to the header of a SOAP message, to digitally sign portions of an envelope. The web services built for the SEP make use of this specification to ensure secure communication between each other. A digital signature included inside a SOAP message looks like this:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Header>
   <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
      <ds:Signature xmlns:ds="http://www,w3.org/2000/09/xmldsig#">
         <ds:SignedInfo>
            <ds:CanonicalizationMethod Algorithm= "http://.../xml-exc-c14n#"/>
            <ds:SignatureMethod Algorithm= "http://.../xmldsig#hmac-sha1"/>
            <ds:Reference URI="#msgbody">
               <Transforms>
                  <ds:XPath>child-or-self:RobertsOperation</ds:XPath>
               </Transforms>
               <ds:DigestMethod Algorithm= "http://.../2000/09/xmldsig#sha1"/>
               <ds:DigestValue>LyLsF0Pi4wPU...</ds:DigestValue>
            </ds:Reference>
         </ds:SignedInfo>
         <ds:SignatureValue>KiGF9JK7G4Tyu...</ds:SignatureValue>
            <ds:KeyInfo>
               <ds:KeyName>robert's public key</ds:KeyName>
            </ds:KeyInfo>
      </ds:Signature>
   </wsse:Security>
</S:Header>
<S:Body Id="msgbody">
    <This is not signed/>
    <RobertsOperation/>
</S:Body>
</S:Envelope>
```

Figure 3.2: An example of a digital signature within a SOAP envelope.

The interesting part of this example is the `<Signature>` element inside the `<Security>` element. Within this signature element it is possible to specify

which parts of the body are signed, which algorithms are used to create the digests[1] and which key is used to calculate the signature value. The first sub-element of the `<ds:Signature>` element is the `<ds:SignedInfo>` element. This element specifies which canonicalisation method[2] is used to normalise the entire `<ds:SignedInfo>` element. Normalisation has to be performed, because it is possible for XML documents which are semantically equivalent to differ in physical representation. For example, they may differ in their structure, ordering of attributes and character encoding. For the validation and creation of a digital signature the input must be the same.
The next element, the `<ds:SignatureMethod>` specifies which algorithm is used to calculate the signature value from the canonicalised `<ds:SignedInfo>` element. The `<ds:Reference>` element has an `URI` attribute that refers to the element being signed. This element can be included more than once in a `<ds:SignedInfo>` to digitally sign multiple parts of a SOAP message. This `<ds:Reference>` element contains a optional `<Transform>` element, which specifies the algorithm used to perform additional transformations on the information being signed. In this example, an XPath transformation is included. This one means that the element `<RobertsOperation>` is being signed instead of the entire body element! This may pose a security risk, see 3.2.1. The other two elements in the `<ds:Reference>` element are the digest method and the digest value. Since a signature value is computed *only* over the `<ds:SignedInfo>` element and not over the SOAP elements themself, each `<ds:Reference>` element must have a digest method and a digest value specified. This digest value is digitally signed to ensure that an adversary cannot change the data (of which the digest value has been computed) en route without invalidating the signature. After the `<ds:SignedInfo>` element, there must be a `<ds:SignatureValue>` element which is the signed version of the entire `<ds:SignedInfo>` element. The `<ds:KeyInfo>` element specifies which key is used to compute the signature value. Note that the method of calculating the signature value is specified inside the `<ds:SignedInfo>` element, so that an malicious intermediary cannot change it to a weaker algorithm without corrupting the signature value.

The Signature element

The `<Signature>` element in combination with the setup of a session[3] was a big part of the research and is used as a part of the authentication mechanism in the system for the SEP. In short: the web service creating a SOAP message signs the body element with a nonce with its own private key and verifies the identity of the creator of the SOAP message by using the creator's public key. Note that it cannot verify who sent the SOAP message, because it might have been resent by a third party. Upon receiving this envelope, the recipient creates a random session id and includes it, along with the same nonce and its identity, in a SOAP envelope which is sent back. The first web service reads the session id and verifies the nonce and responds by sending a SOAP envelope back in which the session id is signed. In doing this, the first web service gives proof of possession of its private key. The session is now setup and the recipient can perform its task. If the private keys of both web services is not compromised, this is sufficient for

[1] Both parties in a session must decide whether or not to use a certain algorithm
[2] The method to "normalise" the XML to ensure that the digest is computed correctly
[3] A session is setup to prevent replay attacks, as explained in more detail in section 5.3

authentication. The recipient web service determines whether or not to trust a public key by inspecting the list of public keys of web services that have access to it.

3.2.1 Issues with creation/validation

There are a couple of considerations one should think of, when using a digital SOAP signature that could compromise the level of security. First of all, when web services are communicating with each other, they must trust the security model they are using. The way they trust public keys, algorithms and communication in general is very important. They must trust the policies they establish for communication to be secure. When creating a signature, only what is relevant and used by the receiving application should be signed. Also, the application should only output data that was signed. The following considerations become relevant when using the SOAP-DSig specification:

Transformations
: Because it is possible with SOAP to perform transformations on the information before creating a signature, the information in the SOAP message that is lost by performing a transformation is not secured. Only the portion from which a digest is calculated is secured. These transformations can be any kind of transformation that is allowed in the SOAP-DSig specification: XSLT, XPointer, XPath or even a custom made one is possible. In the example on page 23, the body element `<This is not signed/>` is not signed, because there is an XPath transformation inside the signature that only takes the `<RobertOperation/>` element as input. The `<This is not signed/>` element can be changed en route. Therefore, only what is signed is secured.

Canonicalisation
: One advantage of canonicalising the SOAP elements before calculating a digest is that all references to elements or attributes and all namespaces are replaced by their actual values. This ensures that those elements/attributes are also signed, even when the source being signed has references to elements. If one does not use canonicalisation before creating a signature value, those elements are not protected and can be changed in transit.

Headers
: By definition, headers of SOAP messages may be removed by intermediaries at will. If an adversal intermediary decides to remove the signature or elements relevant to it, the endpoint or application does not have a method of validating the SOAP message. When the receiving application processes the message without checking for signatures, integrity might be compromised. Two applications communicating with each other should agree upon signatures and signature methods.

Signature/digest methods
: The strength of a particular signature depends heavily on the algorithms used to create the signature and the message digest. Weak algorithms may be broken or their output predicted. Also, the keys used for creating

the signatures must be created using secure random numbers because adversaries may predict them by reproducing the environment they were created in resulting in exposure of the private key. Furthermore, long keys are preferrable over short ones, because using a long key creates more possibilities for an attacker to try and therefore makes it more difficult to break.

3.3 XML-Encryption

To make SOAP message confidential for intermediaries, this thesis introduces a subset of the XML Encryption standard [14] which has become a W3C recommendation December 2002. Since a SOAP envelope is just another version of a XML-document, this specification can be used to encrypt elements inside a SOAP envelope as well. The recommendation specifies how an element in a XML-document may be encrypted and replaced by its original value. It makes use of the `<Security>` element as well to specify which element is encrypted. The encryption of SOAP elements is done by establishing a session key (using a symmetric cipher) per session (explained in section 5.2), which can be used to encrypt arbitrary elements of SOAP elements. A SOAP message that has an encrypted element, is shown in figure 3.3. For brevity, the signature is omitted.

Note that in the example in figure 3.3 an encrypted key has also been included in the SOAP envelope. However, this is optional. It is included only to illustrate what XML Encryption provides. Using these encrypted elements, the contents of parts of a SOAP message can be protected. The encrypted elements must be in a specific namespace to conform with the specification: "`http://www.w3.org/2001/04/xmlenc#`". A list of every encrypted element is kept in the `<Security>` element. Whenever a web service in the system for the SEP needs to keep information confidential, it uses these encryption methods. To improve the confidentiality, the entire body of the message is encrypted and not only the "sensitive" elements.

3.3.1 Issues with XML-Encryption

Also with the XML-Encryption specification, there are a few issues and security considerations to be thought of when using it. The following issues have become relevant during the research and the design of the system of the SEP.

Using both signatures and encryption

The web services for the SEP use digital signatures as well as the encryption of elements of SOAP. A new issue arises when there are encrypted elements that must also be signed or vice versa. There is a difference between signing encrypted elements or encrypting elements after a digital signature has been computed. First of all, what has been done first, signing or encryption, has to be very clear. Otherwise validation of a signature will fail. There is another specification released (see [15]) by the W3C in december 2002 which specifies the advantages and disadvantages of signing before encrypting and vica versa. Also, it specifies how you can make it clear in a SOAP message what has been

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
            xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext"
            xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
   <S:Header>
      ...
      <wsse:Security>
         <xenc:ReferenceList>
            <xenc:DataReference URI="#body"/>
         </xenc:ReferenceList>
         ...
      </wsse:Security>
      ...
   </S:Header>
   <S:Body>
      <EncryptedKey Id="sessionkey-582af99" xmlns="http://www.w3.org/2001/04/xmlenc#">
         <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
         <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <ds:KeyName>public key of recipient web service</ds:KeyName>
         </ds:KeyInfo>
         <CipherData><CipherValue>xy87BH65Vuyto89zabc</CipherValue></CipherData>
         <CarriedKeyName>
            sessionkey-582af99
         </CarriedKeyName>
      </EncryptedKey>

      <xenc:EncryptedData Id="body">
         <xenc:CipherData>
            <xenc:CipherValue>J8Jh7HGF4H89Kh...</xenc:CipherValue>
         </xenc:CipherData>
      </xenc:EncryptedData>
   </S:Body>
</S:Envelope>
```

Figure 3.3: An example of a SOAP envelope with an encrypted body.

done first. It is not necessarily the case that a signature has always been created before elements are encrypted. For instance, when an actor A signs a message and encrypts a few elements and actor B in its turn also signs the entire envelope and includes the signature. Usually it is the case that encryption after signing is desirable, because then it is clear that the signer understood the plaintext. Another disadvantage of including a signature over encrypted elements is, that the signature may reveal information about the encrypted elements so that plain text guessing attacks might be possible by inspecting the digest value. Such a value is always included in a signature. For the system of the SEP, the web services will *always* sign a SOAP envelope *after* the body of the SOAP message has been encrypted. Doing it this way means that an invalid or changed SOAP message can be detected as soon as possible, without having to decrypt

the encrypted elements first and also consumes less resources, which becomes relevant when there are few resources available:

Availability/Denial of Service

Digitally signing and encrypting SOAP elements, consumes resources. It might be possible for an adversary to send bogus SOAP messages with signatures and encrypted elements to force the endpoint or application to spend many resources before invalidating a message. Furthermore, the XML-Encryption specification allows for recursive processing. The decryption of a symmetric key A might require the decryption of another key B which in its turn needs key A. This creates a deadlock situation and must be dealt with. Also, a SOAP message might reference items necessary for decryption that take a lot of time to retrieve. These three issues can heavily reduce the availability of a web service and might even result in the denial of service of a "valid" SOAP message.

Chapter 4

The System at the SEP

A very large part of the research done for the SEP was to design a system useful for the SEP by making use of web services. The purpose of the system is to enable an end-user to locate documents via a website on various servers and to access a database on another specific server. Once the documents have been located, the system provides the end-user with a way to create a printable format of each document and can send it to different printers. Also, the system enables the end-user to transfer a password file from the database on one server to another server in order to merge the retrieved password file with the existing one. Once they are merged the new password file must be deployed.
The purpose of this chapter is to give an understanding of the process and the design of the system and to explain the design decisions made. Furthermore, after reading this chapter, one should be able to explain what the system does, how it works and what its added value is to the SEP. The specification of each component in the system can be found in section 4.5, along with its models and diagrams. These models have been made using the Unified Modelling Language (UML), for more information on the UML-diagrams used in this design please refer to [16]. First, an introduction to the system is given, to provide a general idea of how the end-user interacts with the system followed by how it is deployed and designed.

4.1 Functionality

First the different functionalities that the end-user would benefit from having[1], are depicted in the diagram below. These functions were performed manually. The system that is designed in this document implements them. The use case diagram in figure 4.1 depicts the way the end-user sees and interacts with the system. Note that the end-user does not see the security measures used by the system[2].

End-users have three pieces of functionality at their disposal:

- It can perform a passwordfile transfer from a database called the "Server for Support for Learning and Teaching" database (SSTL-database).

[1]For the reasons why the end-user would benefit, see section 4.4
[2]End-users have no notion of the ***Security* web service** explained later in this chapter

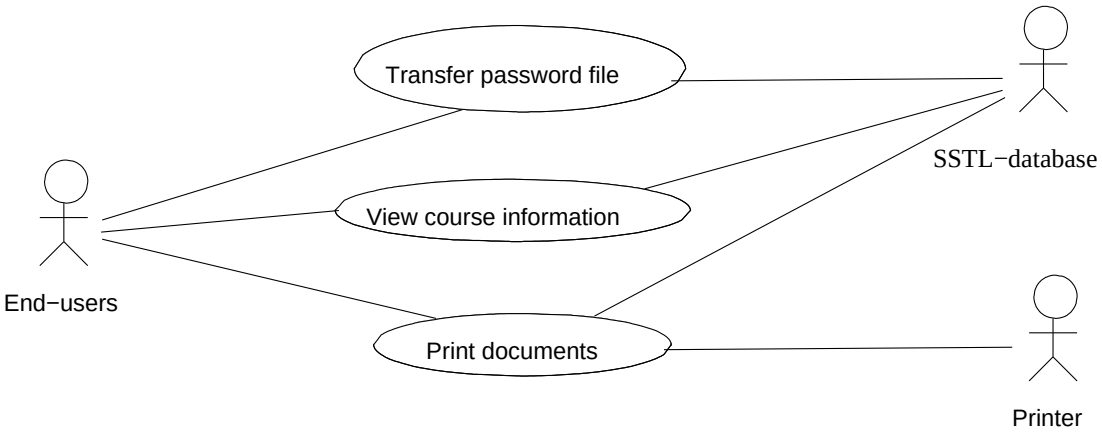

Figure 4.1: Use case diagram of the different functionalities of the system.

- It can view information on a specific course.
- It can print documents that belong to a certain course.

These are the three main functionalities of the system. There are also non-functional ones such as scalability, availability, stability and ease of use. The security requirements of this system are very also very important, because the information being transferred is of a sensitive nature. All these requirements have been analysed and documented in [6].

4.2 Previous situation

The existing computer systems and infrastructure to perform these tasks were sufficient to perform these task manually. However, the information and data needed to perform these tasks are distributed over different computer systems. The SSTL-database for example, is running on a Solaris server called the *sed*. This database was only easy accessable via a website. Also, the password files that need to be changed and deployed regularly on another Solaris server called the *sef*, are stored in the SSTL-database. The documents that belong to a course are also stored on Solaris server *sef*. The diagram in figure 4.2 explains the setup of the situation in more detail.

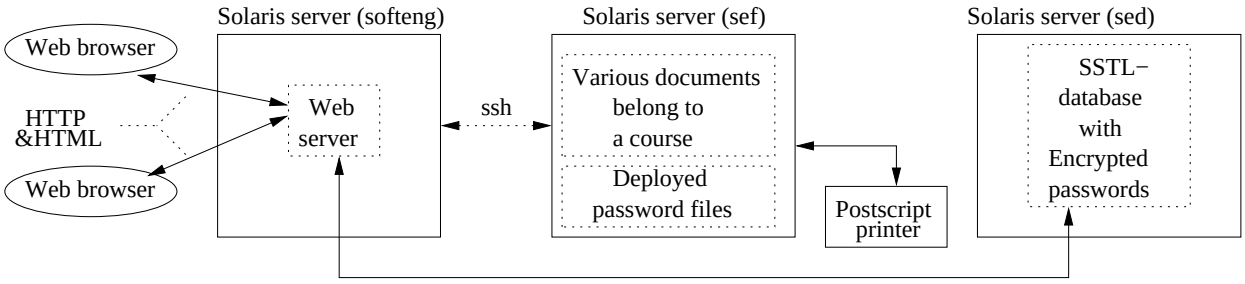

Figure 4.2: Component diagram of current systems at SEP

In the previous situation, the end-user had to log in to the server *sed* to manually convert the documents to a printable format and to send them to a printer. Furthermore, the password files had to be first retrieved from the database (via the website) manually, copy-pasted from the page and sent to the administrator of the *sef* via email.

4.3 Design

Since all the resources are distributed on different servers, web services were very suitable to solve the inconveniences of the previous situation. This section describes how a network of web services is built to perform the tasks. The web services that implement the desired functionalities have to contact a separate web service that is the only web service that has access to the SSTL-datbase. The end-user never has access to it. A component diagram is drawn in figure 4.3 to give a general idea of how the web services are deployed on the Solaris servers. It consists of five web services deployed on four different Solaris servers, all communicating with each other. The diagram in figure 4.3 depicts the different web services and their place in the system.

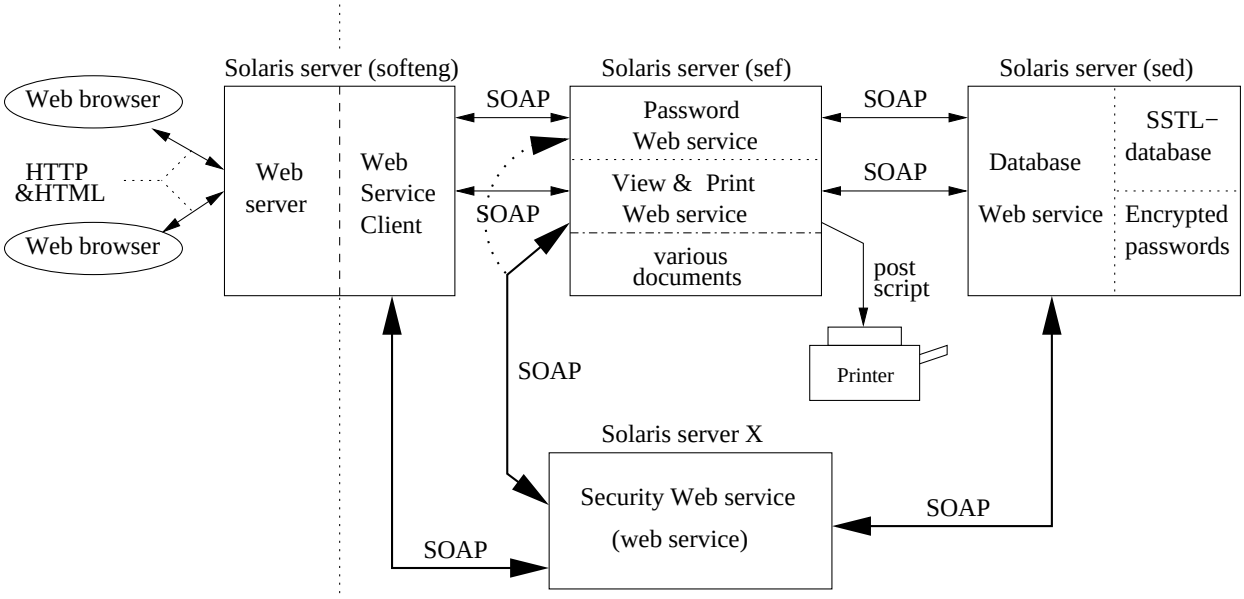

Figure 4.3: Component diagram of current systems at SEP with web services

In abstract, these web services combined let an enduser specify which of the documents on server *sef* are to be printed and how many times (performed by the *Print* web service). Since all the information needed to perform that task is dispersed on all the four different servers, the web services call each other to retrieve information. Instead of just printing documents, another web service, the *Password* web service, is deployed on server *sef* to transfer and deploy password files. The database can be accessed when necessary by calling the *Database* web service. Since web service are designed for program-to-program

interaction, meaning the end-user does not directly contact them, the *Client* web service is deployed on server *softeng* to interact with the end-user and to make requests to the other web services.
In this network of web services, the *Security* web service (SWS) is trusted by all the other web services. It provides the web services with a way to authenticate each other. A more detailed description and explanation of this communication model is given in chapter 4.5. Every web service holds a list of public keys of the web services that have access to it. This list is maintained by the administrator of the SWS per web service. All the lists are stored in a database on the SWS. Whenever a list is changed by the administrator, the SWS contacts the corresponding web service and updates its list. When a web service requests a service from another, it sets up a session/security context to communicate with it. Section 5.2 goes into detail on how such a session can be setup in order to have a secure *enough*[3] information exchange.

4.4 Improvement

This system is an improvement for the situation in the SEP of the OUCL. Transferring password files by hand or by email and deploying them manually is insecure and error prone. Doing this whenever users are added/removed or when passwords are changed, is very cumbersome. The *Password* web service automates this process. Also, the implementation of the *Print* web service is an improvement. The person responsible for printing the documents, has to convert them manually to a printable format which may vary per document. To do this, that person also has to retrieve information from the SSTL-database manually to retrieve the number of assessors, observers, lecturers and students that are involved in a course in order to know how many times documents need to be printed. In the future, when the knowledge of how to use web services and SOAP securely becomes available, other systems may be connected as well.

4.5 Specifications and models

This section describes the details of the network of web services. First, it explains the communication model used and the way trust is being provided for the web services in the network. Second, the main web service class is introduced. Every web service being specified in the design, has certain functionality in common with others. This common functionality is designed into an abstract **ComlabWebService** class. After that, there is a section for each web service that describes how the web service is constructed.

4.5.1 Communication model

The preliminary security whitepaper by IBM and Microsoft (see [17]) proposes various architectures of deploying a security service or token service which provides the necessary tokens in a network of web services to manage access and authentication. In the design one of those architectures has been chosen and

[3] Secure enough means covering the risks as analysed in [6]

eventually implemented. The diagram in figure 4.4 should explain the architecture used.

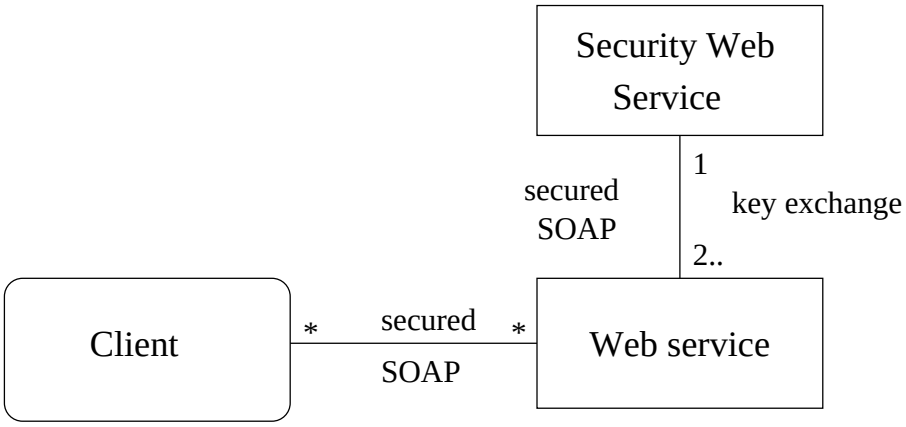

Figure 4.4: Architecture of web services setup

In this architecture each web service in the system has a public and a private key. Of course, the private key is kept secret. The public key of each web service is stored in the *Security* web service. Also, every web service that provides a service to others has a list of the public keys of the web services that have access to it. Whenever the list of a web service in the *Security* web service is updated by the administrator, the SWS contacts the web service by establishing a session with it and updates its list. This is done by encoding the latest list inside the body of the SOAP envelope. The web services that make requests to other web services do not to contact the SWS. This architecture has been chosen for scalability and availability reasons: when a lot of web services in the network need to communicate with each other, the SWS may be flooded with requests for security tokens. In this architecture, the web services only need to be updated when something changes in the setup of the web services or in the way they communicate with each other.

4.5.2 The main web service

This section describes the abstract **ComlabWebService** class and its dependencies and associations with the other necessary classes to setup a secure SOAP communication.

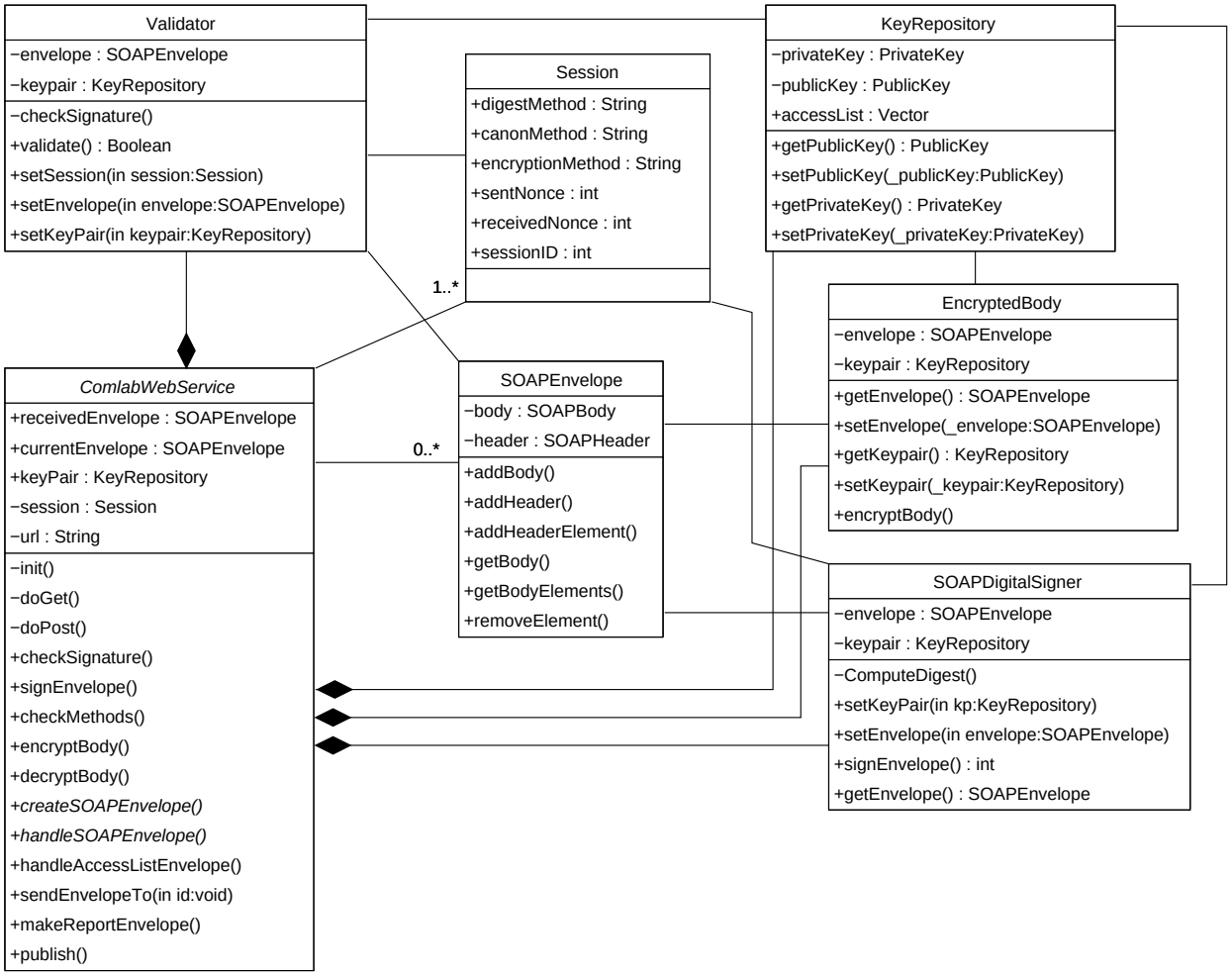

Figure 4.5: The main ComlabWebService class which every web service extends

In figure 4.5 the abstract **ComlabWebService** class is depicted. It is the main class that every web service in the system for the SEP extends. It provides all the basic functionality of digitally signing an envelope, reading signatures, setting up a messaging framework, setting up a session and a security context. It comprises of four other subclasses. The **Validator** class, the **EncryptedBody** class, the **SOAPDigitalSigner** and the **KeyRepository** class. The **Validator** class is responsible for validating a digital signature included in a SOAP Envelope. Whenever a web service receives an envelope, it has an instance of the **Validator** to check the signature. The **Validator** class must check this signature according to the processing rules as defined in [12]. The **KeyRepository**

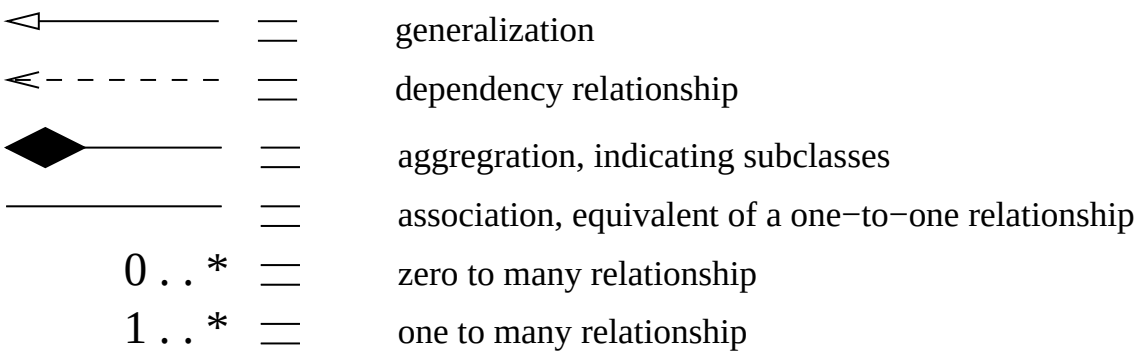

Figure 4.6: Legend for the UML diagrams

class holds the public and the private key of the web service and it also contains a list of the public keys of the web services that have access to it. Each element of this list is an object which holds an ID and the public key of the web service. The **KeyRepository** class is always instantiated whenever the **ComlabWebService** class is. When a web service is first started, the **KeyRepository** class determines its access list by reading the XML-file *accesslist.xml* which is stored in the same directory as the web service is deployed in, more on this later. The **EncryptedBody** class has but one function. It takes a SOAP envelope and a key pair and encrypts the body of that envelope using the private key. The **SOAPDigitalSigner** computes the digest of the body of the SOAP envelope. It then encrypts the digest by using the settings from the session of the web service and its keypair to create and add a `<Signature>` element in the `<Security>` element of the envelope it has to sign.

In the future, more web services can be added to the network of web services by deploying a web service class that extends the **ComlabWebService**. Future classes only have to build their own envelopes for their purpose or application and call the functions of the parent to digitally sign or encrypt and send the envelope.

The structure and format of a SOAP envelope

Apart from the security elements added by the **ComlabWebService** class, the internal structure of the body of a SOAP envelope is standardised in this system as well. Every SOAP envelope that is sent or received in the system for the SEP conforms to the same structure as the class it represents. This means that every attribute or structure of the class is stored in the SOAP envelope in the same way. This is done by using normal SOAP elements and SOAP arrays if multiple elements are needed. An example of such a SOAP envelope can be found in appendix H. For more information on the structure of SOAP and SOAP arrays, please see [7]. There are four different structures used in the SOAP messages in the system: the *Report*, *Course*, *AccessList* and the *PasswordFile* structured SOAP envelope. To see which web service uses which, see the class diagrams in the following sections.

4.5.3 The Client web service web service

This web service communicates with all the other web services to call remote procedures and to retrieve information. It acts as the interface between the end-user and the network of web services. Note that the authentication of the end-user is outside the scope of this design and has been taken care of by the website in which the **Client** web service will be deployed. The **Client** web service extends the **ComlabWebService** and has the necessary operators to perform its function. Since the purpose of this web service is to be accessible via a web browser, this web service must have the capability to process HTML-form requests. Most of the work has already been done by the servlet container (see "deployment" section 4.5.8), however it must translate requests from the user into SOAP Remote Procedure calls; therefore it has the two operations to do this. Before explaining exactly how the **Client** web service works, consider the diagram in figure 4.7.

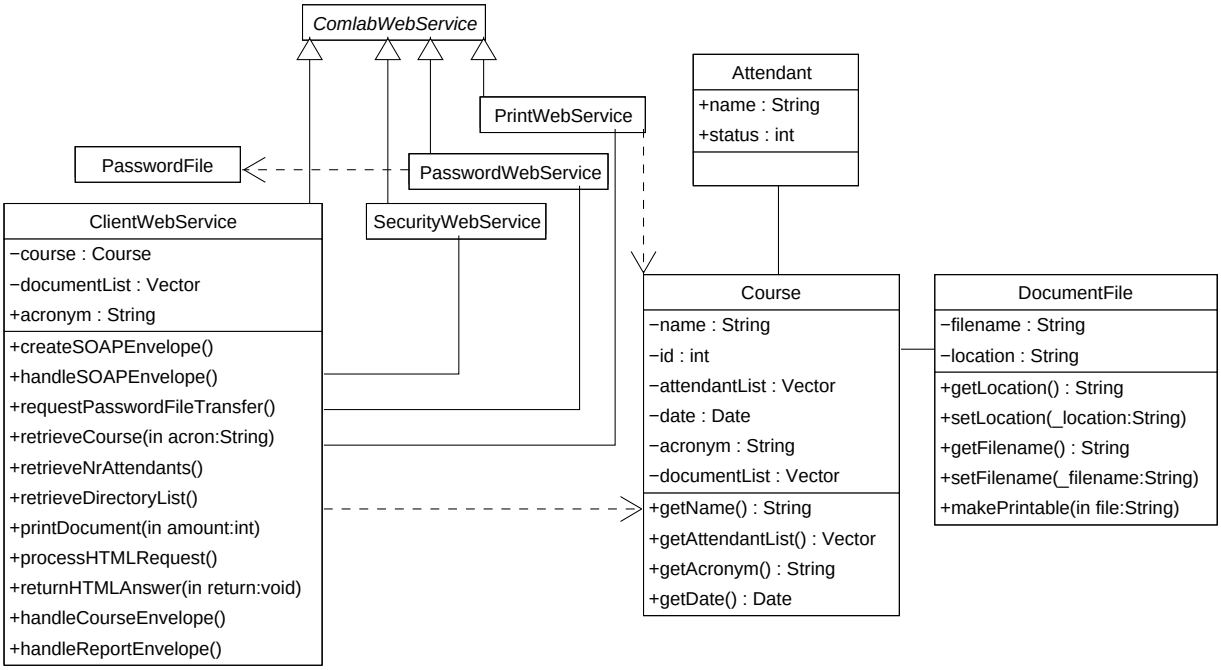

Figure 4.7: The **Client** web service web service and its associations.

The **Client** web service has a dependency relationship to the **Course** class, since it retrieves a SOAP envelope containing the information on a course along with a list of the deployed files.

Issueing print commands

The **Client** web service retrieves a course specification from the print web service by providing it with the acronym for the course. The end-user provides the **Client** web service with this acronym (by posting a form), which is the same acronym as used in the SSTL-database. This form is already integrated in the

website of the SEP. The **Client** web service translates this request into a SOAP envelope and sends it to the **Print** web service. After that, the print web service gives the course specification[4] back to the client. This specification contains a list of the attendants and of all the files deployed on the Solaris server **sed** for that course. The client returns this information back to the web browser of the end-user. The user is then able to select a file and issue a print command, specifying how many the times the document must be printed. The following printers are available to the end-user:

- sep2 on the server psi
- sep3 on the server psi
- sep4 on the server psi, which is the color printer

The **Client** web service translates this request into a SOAP message and sends this to the **Print** web service. For details on how exactly the messages are exchanged between the **Client** web service and the **Print** web service, the corresponding sequence diagram is included in appendix C.

Viewing the database

To issue print commands the end-user also has to query the database for information on courses. The **Client** web service provides the end-user with a way to retrieve information per course. It also gives meta-data about the course that might be of relevance.

Transfer of password file

The only thing the **Client** web service has to do in order to do perform a password file transfer is to issue this command, given by the end-user, inside a SOAP envelope to the **Password** web service (PWS). The necessary steps to complete this operation are done by the PWS itself. The **Client** web service sets up a session with the PWS, using the standard functionality which is already a part of the **ComlabWebService**. A session is needed, instead of a single SOAP envelope containing a SOAP remote procedure call, for security reasons[5]. After the session setup, it sends a SOAP message containing the request. It then waits for confirmation of the transfer of the password file and returns feedback to the end-user. A sequence diagram is included in appendix D that depicts exactly how this request is made.

[4] conform the internal structure of the Course class
[5] please refer to chapter 4.6.3 for details

4.5.4 The Print web service

This web service is responsible for the printing of the documents stored (per course) on Solaris server *sed*. It interfaces only with the client web service and the database web service.

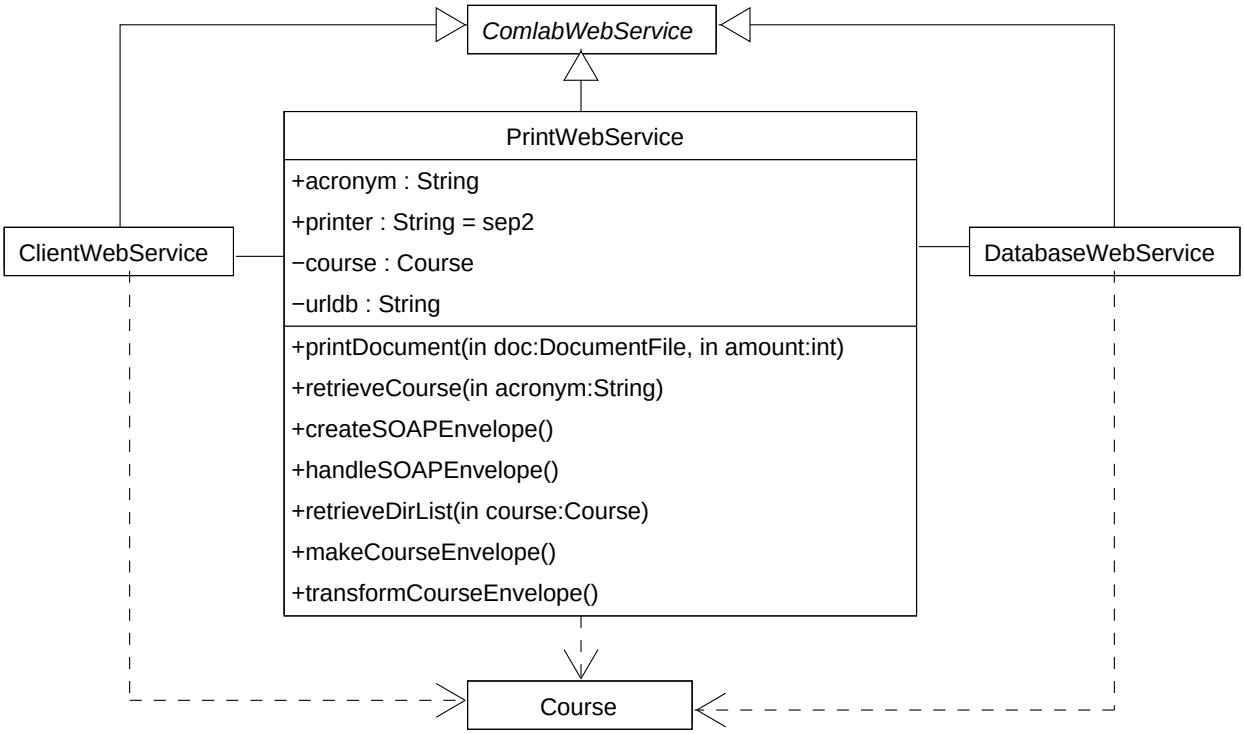

Figure 4.8: The **Print** web service and its associations.

Printing documents

The documents and course notes that can be printed are stored in several different directories per course. A directory name always starts with the acronym of the course for which a document needs to be printed, followed by the month the course is given in, followed by the year; making the format of the directory `<acronym>-<month>-<year>`. The web service assumes the document is stored in the directory starting with the provided acronym. It prints a document by using the details that are specified in the class **DocumentFile**, a parameter of the print operation. Before a print command is sent to the printer spooler, it may need to be converted. The **Print** web service assumes that there is a **Makefile** for every file that is not in the PostScript format, with extension `.ps`. It makes the system call `make <filename>` to convert it if necessary. If the document is not in the PostScript format and no rule in the **Makefile** exists to process it, the operation is aborted with a SOAP fault message, more on that later. The print commands are also issued by making system calls to the *sed* server. Since this server is a Unix server, this is done by the command `lpr -P<printer> -m -#<amount> <document>`.

The `-m` option indicates that progress of the print job is emailed, furthermore the printer is specified, the amount of times it needs to be printed and of course the document itself. If any of these steps fail, a SOAP fault envelope with the faultcode **Server.PrintFailure** is sent back to the client web service, specifying the reason as the **faultstring**. For more information on SOAP fault messages, see section 4.5.9 or [18]. When needed, the **Print** web service contacts the **Database** web service in order to provide the **Client** web service with the right details of a course.

The **Print** web service sets up a session with the **Database** web service when information is needed. The way this is exactly done, is depicted in the sequence diagram in appendix E. To make a request, it creates a SOAP envelope with the same structure as the Course class, but with empty elements. The **Database** web service returns the same envelope with the same elements, provided with the course details. The **Print** web service converts this SOAP envelope into its own session with the **Client** web service web service and sends it.

4.5.5 The Password web service

This web service is responsible for the complete transfer of a password file from the SSTL-database to the Solaris server *sed* on which this file is merged with the existing password file and deployed. In the SSTL-database each course has several students attending it. The **Password** web service retrieves the password file, which is created per course, via the **Database** web service. Consider the class diagram in figure 4.9, the SWS is omitted.

After this web service received a valid request for a transfer of the password file of a particular course, it creates a SOAP envelope with the same internal structure as the **PasswordFile** class with empty elements inside the envelope. The only element that is set in the envelope is the *acronym* attribute. This indicates to the **Database** web service which password file is needed. It then starts a session and retrieves the password file. For security reasons, the session that is setup with the **Database** web service uses encryption of the body of the SOAP envelope. For a more detailed description of how exactly this communication takes place, a sequence diagram is included in appendix F. After it has retrieved the password file, there are two possible scenarios.

If the web service has sufficient rights to merge the existing with the retrieved file, it proceeds. This merging process is done in the following way. For each entry in the retrieved password file, it looks up the corresponding entry in the existing file. If it does not exist, it adds the new entry. If it does exist and it is changed, it updates the entry in the existing password file; else it does nothing. The **Password** web service never removes an entry. For each entry made or changed, a report is sent back to the **Client** web service.

If it does not have enough rights to merge the password files, it saves the file in the same directory and changes the file rights to read-only for itself[6]. The administrator of the existing file has to manually merge the files afterwards.

[6]On the Solaris server this corresponds to rights **200**

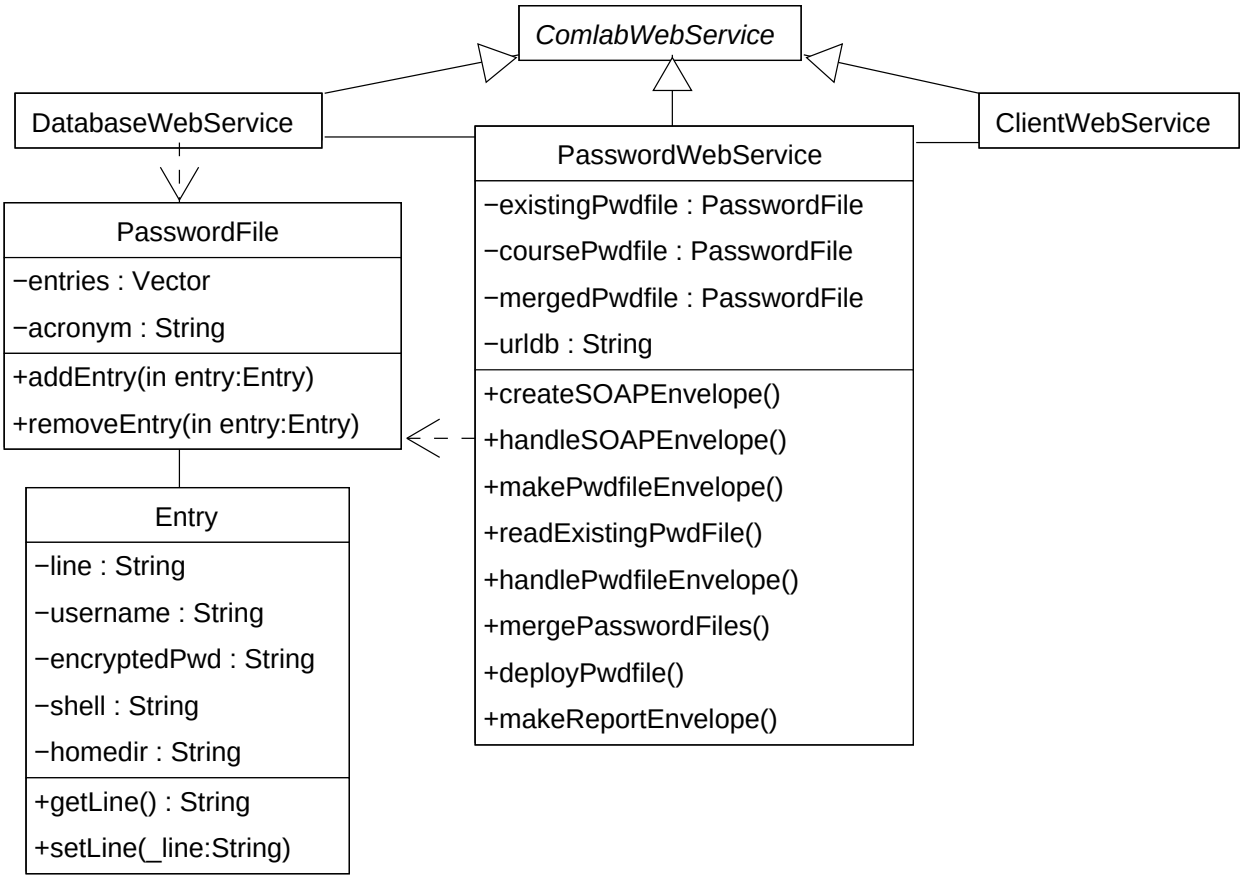

Figure 4.9: The **Password** web service and its associations.

4.5.6 The Database web service

The purpose of this web service is to access the SSTL-database on the Solaris server *sed*. There are two types of information the web services retrieves, a description of a course and the password file that belongs to a course. Therefore, the structure of the SOAP envelopes it creates correspond to the **Course** and the **Password** class. It also has an interface with the SSTL-database on which it can execute queries. First, consider its class diagram in figure 4.10, the SWS is omitted.

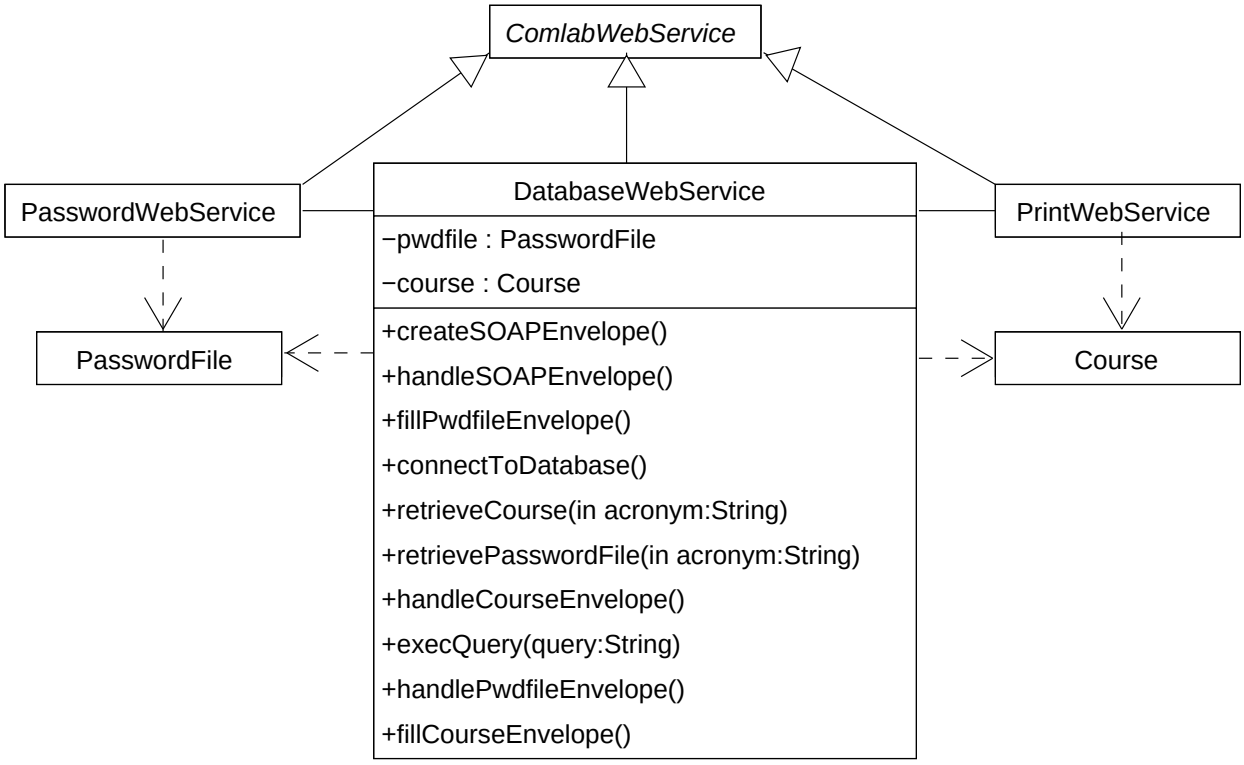

Figure 4.10: The **Database** web service and its associations.

The query that is executed on the database that will enable the **Database** web service to retrieve the necessary information is:

```
FROM Course
SELECT courseSubject.acronym, courseStartDate,
       courseLecturers, courseAssessors,courseObservers,
       FROM attended
       SELECT studentContact.contactAccess.username,
              studentContact.contactAccess.password
       END
WHERE courseSubject.acronym = [what?]
END
```

Note that this is not normal SQL; a different query language is used on the SSTL-database. The latest course and the latest password file that have been accessed in the database are kept in memory by the web service. This is done so that another database lookup is not always necessary. Whenever a request is made to the web service to transfer a password file, it always encrypts the body to ensure confidentiality using the operations of its parent class, the **ComlabWebService**. The way this web service communicates with the other web services can be found in appendices E and F.

4.5.7 The Security web service

The SWS is responsible for providing trust within the network of web services. It does this by maintaining a database of all the web services deployed in the network. For each web service, it has a list of the web services that have access to it. Whenever this list is changed by the administrator, it updates the corresponding web service(s) with their new list. A list, the **AccessList** class, consists of a public key and a **KeyRepository** class. The keyrepository holds all the entries of the web services that have access. See the class diagram of the SWS in figure 4.11.

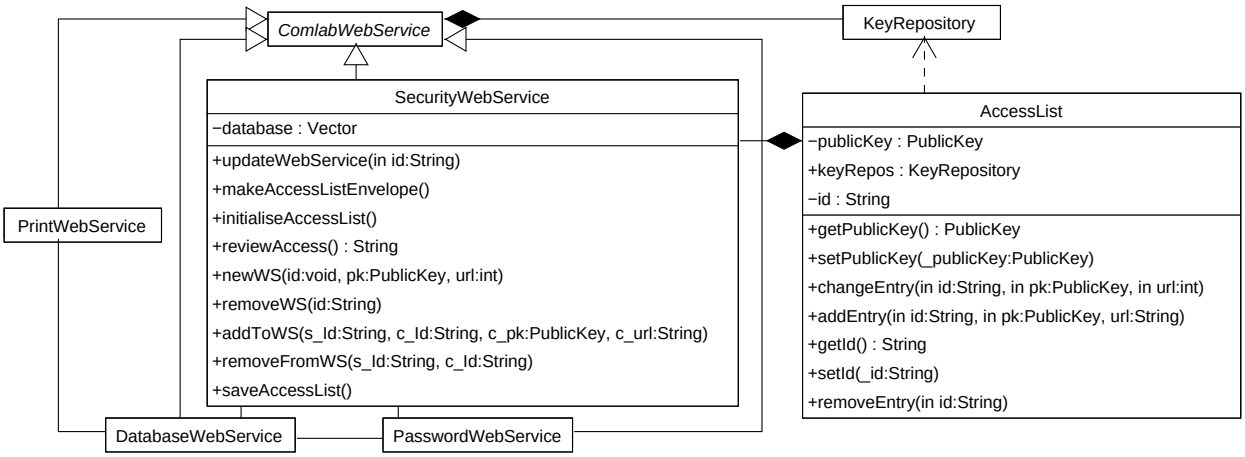

Figure 4.11: The *Security* web service and its associations.

The operations to add, update and remove an entry on the list of a web service, can be called by the administrator on demand. The database is stored in memory the entire time the SWS is deployed. Whenever the administrator changes this list, the SWS also saves this list in a XML-file (**accesslist.xml**), which is easily parsable, to store the database. Whenever needed, the SWS can read this XML-file to create the **AccessList** classes. This file is stored in the same directory as the SWS is deployed in. The SWS does not have to communicate with an encrypted SOAP body, because the keys that are exchanged are public keys anyway. When the SWS sets up a session with a web service, it can securely update its list. Please refer to appendix G for a sequence diagram on how the SWS updates a web service.

Bootstrapping web services

Whenever a web service is to be added and deployed in this network of web services, it publishes its public key to the SWS. Each web service can do this by calling the *publish()* method from the **ComlabWebService**. This causes its public key to be encoded inside a SOAP envelope and is then sent to the **Security** web service by setting up a session. The SWS then notifies the administrator of the new web service. The administrator has the option to issue the command to the SWS to instruct other web services to let the new web service have access to them.

4.5.8 Deployment

The web services are not standalone services. They have to be deployed inside a servlet container to handle HTTP-GET or HTTP-POST requests, or via any other protocol for that matter. However, these web services are designed to process HTTP requests. That is the reason that each web service inherits the *doPost()* and *doGet()* methods from the **ComlabWebService**. However, other operations can be easily added to support different protocols, such as SMTP or FTP. However, HTTP is probably the most widely supported protocol on the Internet and virtually all firewalls let HTTP through. With this idea in mind, the web services in the system for the SEP must be deployed in a servlet container that supports HTTP requests. Since all servers on which they will deployed are Unix servers, they will use the servlet container called *Apache Tomcat*. *Tomcat* is the servlet container that is used in the official implementation for the Java Servlet and JavaServer Pages technologies developed by Sun[7]. This servlet container is a Java based one, meaning that it can be deployed on any platform. In the future, computer systems running different operating systems/platforms at the Oxford University Computing Laboratory can be connected using *Tomcat*. It is released under the Apache Software License, meaning it is free for download; it is easy to install and use[8].

4.5.9 Faults and errors

Whenever a fault or error occurs, the requesting web service must receive feedback of the exception. This is done in two ways. First, the web service that notices the fault or error, creates a SOAP fault message with the correct `<Fault>` element with a faultcode, a faultstring specifying the reason, and the faultactor. The second method of error detection the web services use, is looking at the HTTP response codes. Since SOAP is bound to HTTP in this case, every SOAP envelope is exchanged inside a HTTP request. The HTTP request is used to determine what the type of the SOAP envelope is. This is done inside the protocol specific methods *doGet()* and *doPost()* in the **ComlabWebService** class. For example, a typical HTTP response code is the `500 Internal Server Error` message, a required response by the HTTP transport binding as defined in the SOAP specification [11]. A HTTP response with a SOAP fault message used by this system, looks like this:

[7]The website of Sun can be found at http://www.sun.com
[8]*Apache Tomcat* can be downloaded from http://jakarta.apache.org/tomcat/

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnn

<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
            <Faultcode>Client.MustUnderstand</faultcode>
            <Faultstring>Did not provide digital signature</faultstring>
            <Faultactor>
                <Id>pwdws</Id>
            </faultactor>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 4.12: An example of a SOAP fault message bound to HTTP

The *Faultcode* element inside the *Fault* provides an insight for the web service where the problem lies. "Client" means that the the requesting web service made an error, "MustUnderstand" means that the web service did not understand a certain header in the header element of the SOAP message. These values can be: "Client" or "Server" for the first part and "VersionMismatch" or "MustUnderstand" or any other namespace that is appropriate. The *Faultstring* is not processed but displayed to the end-user. The *Faultactor* element specifies the *Id* of the web service where the error occurred.

4.5.10 Communication in sequence

To visualise the way a secure communication must be set up in the system for the SEP, a sequence diagram is shown in appendix B. This diagram consists of classes that each perform a task of the secure SOAP messaging framework. The functions and attributes for these classes are explained in more detail in chapter 4.5. For this communication sequence, the function of each step and class and what part of the framework it addresses, will be discussed. This sequence diagram depicts a typical SOAP communication. This sequence is a "normal" sequence, no errors occur like, for example, faulty signatures and other errors. These are discussed in section 4.5.9.

4.6 Risks and security issues

This section explains how the risks, as analysed in [6], are addressed by the system. A brief explanation is given for each risk and the part of the system in the system that addresses it.

4.6.1 Performance

The web services are deployed inside a servlet container (*Tomcat*). This servlet is a Java-based servlet; Java programs are known to be somewhat slower than other programs. Fortunately, for each request to a web service, the servlet container calls the instance of the web service, making it unnecessary for the web services to be multithreaded. If the servlet dies, it is respawned by the container leaving the web service in tact. There should not be a significant performance risk here. However, in the future, the servlet may provide for many web services. This poses a performance risk to the servlet container itself, not to the Solaris servers. The performance of the web services decrease significantly when multiple web services are deployed within the same servlet container.

4.6.2 Implementation/schedule risks

The "hard" part of implementing the system as specified in previous sections was to make the main messaging framework between the web services work. The **ComlabWebService**, see section 4.5.2, performs all the necessary transformations to the SOAP envelopes: such as digital signatures, the encrypting of body elements and such. It sets up the communication between the web services, making sure that an particular web service that extends it, only has to create the SOAP envelopes that are specific to its application. There was no precedence for implementing this framework and therefore posed a high implementation and schedule risk. Unexpected problems or difficulties occurred, while implementing it. The problem during implementation was that new technologies that had not been used before had to be used or even implemented. The advantage of this framework is however, that future web services can be easily added to the network of existing web services at the OUCL.

4.6.3 Security risks

The messaging framework used by the system has been specifically created in this way to incorporate message level security into web services communication. The security issues of authentication, integrity and confidentiality are addressed in this system, via various methods. Before explaining how they are addressed, the system relies on the following assumptions. Each assumption must be upheld now and in the future.

- The public and private keys can not be cryptographically broken within a reasonable amount of time. To ascertain this, the web services use well-known and established algorithms. The algorithms used can easily be changed by changing the default settings of the **SOAPDigitalSigner** and the **EncryptedBody** class as specified in section 4.5.2 if they are not sufficient.

- The private keys themselves are not compromised. When an adversary gets hold of the private key of one of the web services, it can pose as it. This risk becomes specifically dangerous when, for example, the private key of the SWS is compromised.

- The servers on which the servlet container is running, are secure. Whenever a malicious third party has access to one of those servers, changes can be made to the web service itself or the way it communicates. This assumption can be asserted by making sure that nobody[9] can change anything inside in the *Tomcat* directory or in any of its subdirectories and it is impossible for a third party to access the computer systems memory where the web service is running. In the event that a server and the keypair of a web service *is* compromised, the manager of the SWS can then revoke a keypair by letting the SWS update the lists of the other web services.

Authentication

Since authentication is done via a public and private key scheme, web services are authenticating each other by trusting the public keys the SWS provides. Since it is the task of a serving web service to determine a session id and every SOAP envelope has a unique session id or nonce (during setup of) in a session, it is not possible for an adversary to resend SOAP envelopes without the serving web service being able to detect it. This implies that the creator and the sender of every SOAP envelope can be authenticated, proving the identity of both.

Confidentiality

Confidentiality means that the contents of a message, or a transmission for that matter, is never exposed when an adversary listens in (eavesdrops). For this SOAP messaging framework the sensitive information being transferred is the password file stored in the SSTL-database. The password file itself, if previous assumptions are upheld, is not exposed. However, it is obvious to an intermediary if it can see every SOAP envelope transmitted in the network, that a certain SOAP envelope contains the password file. This is one of the reasons that the system uses the encryption of the entire body element of a SOAP message, instead of for example encrypting only the encrypted password in the file. The level of confidentiality of the content of the SOAP envelopes that the system provides is considered to be confidential *enough* for this particular situation.

Integrity

The **ComlabWebService** class, if extended by a web service, ensures that every SOAP envelope transmitted in the network is provided with a digital signature and checked upon delivery. Also in the security context of a SOAP communication (see section 5 for details), a SOAP envelope can not be changed in transit. If previously mentioned assumptions hold, the integrity of a SOAP envelope is preserved.

[9] except the trusted administrator

Implementation faults

The risks that remain are risks such as buffer overflow attacks and exploits and for example DoS-attacks. Buffer overflow attacks are very hard to countermand. This is especially hard, because the **ComlabWebService** makes use of higher level functions to parse SOAP envelopes such as JAXP[10]. DoS-attacks rely on the flooding of a single server/web service with a large amount of requests.

[10]JAXP, Java API for XML Processing, see [18]

Chapter 5

Setting up a session

The latest version of SOAP [7] version 1.2, the working draft from the W3C, does not have a notion of a session context, or a security context for that matter. This means that each and every SOAP message must have an elaborate `<Security>` element to address all the security issues at the message level. A big part of the research done for the SEP was to design web services that have message level security. This chapter explains how this can be achieved by setting up a session context with SOAP between two web services so that they can communicate securely. This is done by making use of the security extensions to SOAP explained in chapter 3. Later on in this chapter, a proof is given that no adversary can tamper with the protocol or (re-)send messages to compromise security, using the verification tool Casper[1], more on this later.

5.1 Need for secured SOAP

The problem of making SOAP messaging confidential between two computer-systems can be addressed by Transport Level Security (TLS) for example Secure Sockets Layer (SSL). SSL appends Message Authentication Codes (MAC's) to the transmitted messages to ensure message integrity, which would preserve the integrity of the SOAP messages transmitted with SSL as well.

However, when various intermediaries (may be adversaries) have to read the SOAP message to determine who's next in the communication chain, message integrity and confidentiality must be preserved. SSL does not provide for non-repudation to start with. You only have hop-to-hop security when you are using SSL to encrypt communications instead of end-to-end security; because it is possible that security has been breached on one of the web services along the way. Also, the idea behind SOAP messaging is to provide loosely coupled systems with a way that they can communicate with each other in a connectionless way. For example, there could be a message queue that a receiver has to process first. This way the sender of a message must wait for the receiver to finish the queue, here SSL is not a good solution to this problem.

[1] Casper has been developed at the Oxford University Computing Laboratory by Gavin Lowe.

5.2 Security context

This section introduces a security context into SOAP. The properties of this context and the schema type definitions of new elements are given in more detail in the SOAP D-Sig and the XML Encryption specification. The web services in the system for the SEP make use of the `<Security>` element and of a new element, the `<Continue>` element, to initiate a session and to agree on the way they will communicate; call it a "handshake". The first message a client sends to a web service may look like the example on page 23, however the body element contains the `<Continue>` element as the *last* element. Adding this element does not have implications to the validaty of the SOAP message, because it still conforms to the official SOAP specification. It is allowed to add your own elements. This `<Continue>` element indicates to the receiving web service that a security context is requested and contains a `<Nonce>` element, an empty `<Session/>` element and a `<Nr/>` element set to "1". The `<Nonce>` element contains a by the client web service (*cws*) randomly generated number. This nonce has to be added, because the *cws* has to identify the serving web service (*sws*) as well: the *sws* returns this nonce digitally signed. There are two situations to distinguish from: an non-encrypted SOAP communication within context and an encrypted one within context. The web services for the SEP may only have the need to communicate in such a way that messages are authenticated and that the integrity has been preserved. In that situation, the obfuscation of information by the encryption of elements is not necessary.

5.2.1 Non-encrypted content within context

Web services might need to communicate in a secure way where confidentiality is not very important. Important is however that they are correctly authenticated to each other and that no intruder can intervein. Therefore it is necessary to setup a non-encrypted session. The setup of a non-encrypted session starts as following. The client web service (*cws*) sends a SOAP envelope with a digital signature using its own private key and a `<Continue>` element with its subelements as the last element in the body of the SOAP message. Upon receiving this message, the serving web service (*sws*) determines if it agrees with the way communication will take place. It determines if it agrees using the same canonicalisation method, the signature method, the transform algorithm and the digest method. If it does, it remembers these methods and adopts them. In return the *sws* sends a SOAP envelope back with a digital signature of the body. However, the `<Continue>` element is different. It contains three subelements, the `<Nonce>` element it received, the `<Session>` element filled with a session id and the `<Nr>` element. This session id is a concatenation of a timestamp and a randomly generated number to ensure uniqueness. Note that this number id must be a secure random number, meaning it cannot be predicted. Because the `<Nonce>` element is included in the signature, the *cws* knows it is communicating with the intended *sws*. The *sws* is the web service that has to generate this number, otherwise if the *cws* would generate a session id, the *sws* has to maintain a list of sessions it already had to ensure uniqueness of the session. The `<Nr>` element indicates how many SOAP message the each web service has sent within this session, *not* the total number of messages sent in this session. By examining this number, each web service can determine, upon receiving a

SOAP envelope, whether or not it is the right one (in order) and that no envelopes have been lost in transit. The following example of a SOAP envelope illustrates the response the *sws* could give:

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
   <S:Header>
      <ds:Signature>
        ... specifying which element, key, algorithm and methods are
        ... used and of course the signature value.
      </ds:Signature>
   </S:Header>
   <S:Body>
      <eg:Operation xmlns:eg="http://www.exempli.gratia.com/">
          ... whatever the normal operation may be ...
      </eg:Operation>
      <S:Continue>
         <S:Identity>sws</S:Identity>
         <S:Nonce>2394</S:Nonce>
         <S:Session>2A8GH35-a98J5V-Kjg7J</S:Session>
         <S:Nr_sws>1</S:Nr_sws>
      </S:Continue>
   </S:Body>
</S:Envelope>
```

Figure 5.1: An example of the response message sent to setup a session.

5.2.2 Non-encrypted session setup

Now it is clear how the SOAP message are constructed, the entire session is a sequence of three SOAP messages sent back and forth. The purpose of this session is to ensure that the two web services are indeed communicating with each other and that no adversary can compromise security. The following diagram is a schematic representation of the SOAP messages that are exchanged between web services α and β. It illustrates which SOAP messages are sent and in which order. In the session protocol α initiates the session in figure 5.2.

After a complete run of this protocol, both α and β agree upon the session id and they know how many messages each web service has sent (now and in the future). Using this session setup they can call each others operations and exchange information securely, meaning no adversary can come in between them, at least in theory. In practice there might always be implementation errors or other factors involved that can compromise security. However, for a verification of this protocol, see section 5.3. Any web service can end a session by sending a SOAP envelope back in which a `<SessionEnd/>` element is the last element of the `<Continue>` element.

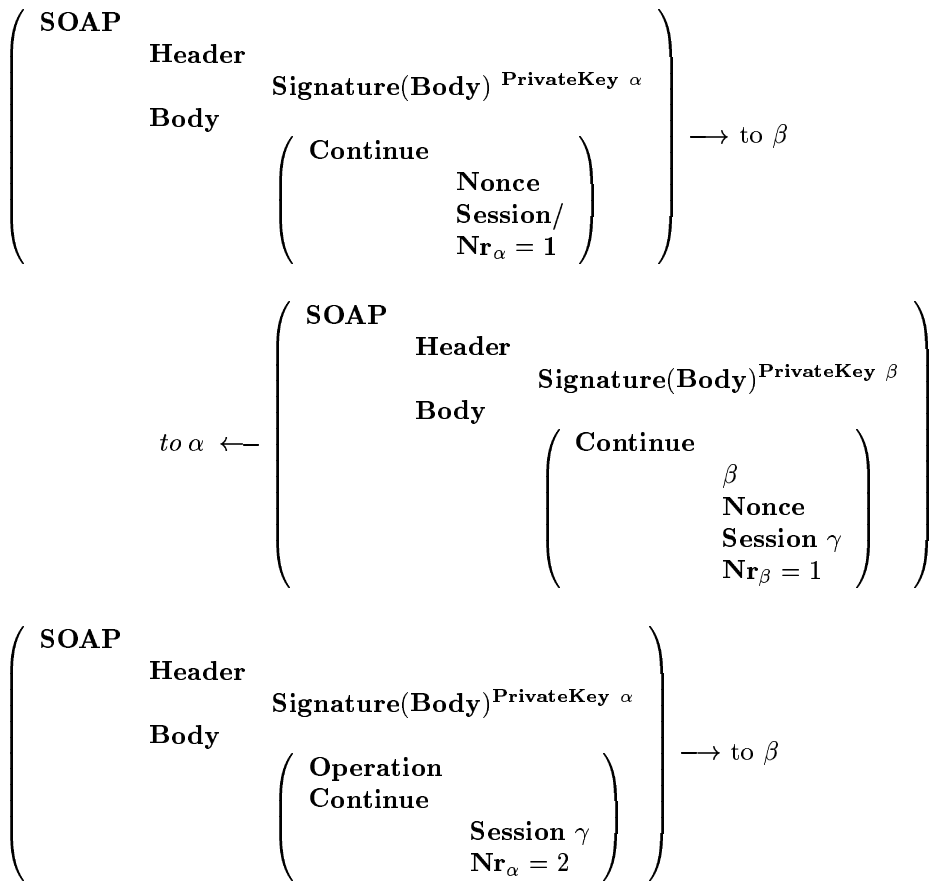

Figure 5.2: The session protocol to setup an non-encrypted session.

5.2.3 Encrypted content within context

Since the web services will have a notion of a security context by remembering the handshake and session id, they can establish a session key as well to encrypt the content of their SOAP envelopes. First, consider the example in figure 5.3 which is a typical response from the sws during the setup of a session. The signature is omitted for brevity.

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
            xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext"
            xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
            xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
   <S:Header>
      <wsse:Security>
         <ds:Signature> .... </ds:Signature>
         <xenc:ReferenceList>
            <xenc:DataReference URI="#body"/>
         </xenc:ReferenceList>
      </wsse:Security>
   </S:Header>
   <S:Body>
      <eg:Operation xmlns:eg="http://www.exempli.gratia.com/">
         ... whatever the normal operation may be ...
      </eg:Operation>
      <S:Continue>
         <S:Identity>sws</S:Identity>
         <S:Nonce>345243</S:Nonce>
         <S:Session>2A8GH35-a98J5V-Kjg7J</S:Session>
         <S:Nr_sws>1</S:Nr_sws>
         <S:Encryption>
         <S:SessionKey>
                <EncryptedKey Id='2A8GH35-a98J5V-Kjg7J'
                          xmlns='http://www.w3.org/2001/04/xmlenc#'>
                   <EncryptionMethod
                          Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
                   <CipherData><CipherValue>xyzabc</CipherValue></CipherData>
                </EncryptedKey>
             </S:SessionKey>
         </S:Encryption>
      <S:/Continue>
   </S:Body>
</S:Envelope>
```

Figure 5.3: An example of a response with encrypted content

If the *sws* agreed upon the security methods, it generates a session key to be used with a symmetric cipher (for example 3DES or AES) and encrypts this session key using the public key of the *cws*. This encrypted key is added to the subelement `<Encryption>` of the `<Continue>` element, inside the `<EncryptedKey>` element which is specified in [14]. The first element of the `<Continue>` element is the identity of the sws. The second element is the `<Nonce>` element as explained before, followed by the `<Session>` element and the `<Nr>` elements. The `<Encryption>` element contains the session key which the *sws* used to encrypt the body of the SOAP envelope. The session key is put in a `<EncryptedKey>` element with an "Id" attribute linking the session key to the session. Further-

more, the encryption method and the key that can be used to obtain the session key from its encrypted form are included in the `<EncryptedKey>` element.

5.2.4 Encrypted session setup

The session setup with establishing a session key, is similar to the non-encrypted one. The following diagram is a schematic representation of the SOAP messages being exchanged:

$$\left(\begin{array}{l}\textbf{SOAP}\\ \quad\textbf{Header}\\ \quad\quad\textbf{Signature(Body)}^{\textbf{PrivateKey }\alpha}\\ \quad\textbf{Body}\\ \quad\quad\left(\begin{array}{l}\textbf{Continue}\\ \quad\quad\quad\textbf{Nonce}\\ \quad\quad\quad\textbf{Session/}\\ \quad\quad\quad\textbf{Nr}_\alpha = \textbf{1}\\ \quad\quad\quad\textbf{Encryption/}\end{array}\right)\end{array}\right) \longrightarrow \text{to } \beta$$

$$\textit{to}\ \alpha \longleftarrow \left(\begin{array}{l}\textbf{SOAP}\\ \quad\textbf{Header}\\ \quad\quad\textbf{Signature(Body)}^{\textbf{PrivateKey }\beta}\\ \quad\textbf{Body}\\ \quad\quad\left(\begin{array}{l}\textbf{Continue}\\ \quad\quad\quad\beta\\ \quad\quad\quad\textbf{Nonce}\\ \quad\quad\quad\textbf{Session }\gamma\\ \quad\quad\quad\textbf{Nr}_\beta = 1\\ \quad\quad\quad\textbf{Encryption}\\ \quad\quad\quad\quad\{\textbf{SessionKey}\}^{\textbf{PublicKey }\alpha}\end{array}\right)\end{array}\right)$$

$$\left(\begin{array}{l}\textbf{SOAP}\\ \quad\textbf{Header}\\ \quad\quad\textbf{Signature(Body)}^{\textbf{PrivateKey }\alpha}\\ \quad\textbf{Body}\\ \quad\quad\left(\begin{array}{l}\{\textbf{Operation}\}^{\textbf{SessionKey}}\\ \textbf{Continue}\\ \quad\quad\quad\textbf{Session }\gamma\\ \quad\quad\quad\textbf{Nr}_\alpha = 2\end{array}\right)\end{array}\right) \longrightarrow \text{to } \beta$$

session established

$$\textbf{Nr}_\beta + 1 \quad \overset{*}{\longleftarrow}\overset{*}{\longrightarrow} \quad \textbf{Nr}_\alpha + 1$$

This works as follows: the *cws* first sends a SOAP message with a digital signature. The `<Continue>` element contains a `<Nonce>` element and empty `<Session>` and `<Nr>` elements. In this case another element, the `<Encryption>` element, is added that indicates it wants to communicate with encrypted content and to establish a session key. Upon receiving this SOAP message, the serving web service (*sws*) determines if it agrees with the way communication will take place. It determines if it agrees using the same canonicalisation method, the

signature method, the transform algorithm and the digest method. If it does, it remembers these methods and adopts them. If the *sws* does not agree on these methods, it sends a SOAP message back using its own preferences. For simplicity reasons, the *cws* must then conform with the preferences set by the *sws*, otherwise the session is aborted[2]. The session key is encoded and sent back to the *cws*.

5.3 Analysing SOAP session with Casper

It is not uncommon for security protocols to have flaws so that an intruder can attack it. The purpose of the session protocol as explained in previous sections is to authenticate two web services to each other and to establish a shared secret session key with which they can encrypt their messages to ensure confidentiality. Also, it must not be possible for an adversal intermediary to change anything to the messages in transit without the web services detecting it. To make sure such a protocol is valid, it must be verified. Casper is a Compiler for the Analysis of Security Protocols, developed by the Oxford University Computing Laboratory, see [19]. It is possible to use Casper to model a security protocol and to verify it. Casper converts the modelled protocol into the process algebra CSP [20]. For the language CSP there is model checker FDR [21] that checks certain security properties. These security properties can be statements such as "authenticated correctly" or "shared session key is secret". If there is a reachable state (while executing the protocol) where such a statement is not true, FDR will find a trace. If there is a trace, then there is an attack upon on the protocol. The setup of a session can be modelled as follows. This first example was the first attempt of finding a working protocol. For brevity, only the encrypted version where the session also establishes a session key is displayed below. After modelling it in Casper, the resulting Casper script looked like this:

$$
\begin{aligned}
\mathbf{Msg}\,\alpha_1\ A \to B\ : \mathbf{Header}, \{\mathbf{Body}, \mathbf{Continue}, \mathbf{Nonce}, \mathbf{Session}, \mathbf{Nr}_\alpha(1),\\
\mathbf{Encryption}\}^{\mathbf{Signed}\{\mathbf{PrivateKey}\ \alpha\}}\\
\mathbf{Msg}\,\beta_1\ B \to A\ : \mathbf{Header}, \{\mathbf{Body}, \mathbf{Continue}, \mathbf{Nonce}, \mathbf{Session}(\gamma), \mathbf{Nr}_\beta(1),\\
\mathbf{Nr}_\beta(1), \{\mathbf{SessionKey}^\gamma\}^{\mathbf{PublicKey}\ \alpha}\}^{\mathbf{Signed\ PrivateKey}\ \beta}\\
\mathbf{Msg}\,\alpha_2\ A \to B\ : \mathbf{Header}, \{\mathbf{Body}, \{\mathbf{operation}\}^{\mathbf{SessionKey}^\gamma}, \mathbf{Continue}, \mathbf{Nr}_\alpha(2),\\
\mathbf{Session}(\gamma)\}^{\mathbf{Signed\ PrivateKey}\ \alpha}
\end{aligned}
$$

The assertions that had to be verified were, whether or not web service α and β were correctly authenticated to each other and whether or not they agreed upon the session id and the session key. The session key also had to be a shared secret between the two web services. Unfortunately, running the protocol through Casper resulted in an attack upon the protocol itself, where the intruder acts as if it was *alpha*. This is a very common attack to a security protocol. The following trace consists of three actors where "I" stands for the intruder and the

[2]This is designed this way to avoid a long handshake algorithm

subscript to the intruder means that it is posing as another actor:

$$
\begin{array}{lll}
\mathbf{Msg}\,\alpha_1 & A \longrightarrow I & : \mathbf{Header}, \{\mathbf{Body}, \mathbf{Continue}, \mathbf{Nonce}, \mathbf{Encryption}\}^{\mathbf{Signed\ PrivateKey}\ \alpha} \\
\mathbf{Msg}\,\alpha_1 & I_A \longrightarrow B & : \mathbf{Header}, \{\mathbf{Body}, \mathbf{Continue}, \mathbf{Nonce}, \mathbf{Encryption}\}^{\mathbf{Signed\ PrivateKey}\ \alpha} \\
\mathbf{Msg}\,\beta_1 & B \longrightarrow I_A & : \mathbf{Header}, \{\mathbf{Body}, \mathbf{Continue}, \mathbf{Nonce}, \mathbf{Session}(\gamma), \mathbf{Nr}_\beta(1), \\
 & & \quad \{\mathbf{SessionKey}^\gamma\}^{\mathbf{PublicKey}\ \alpha}\}^{\mathbf{Signed\ PrivateKey}\ \beta} \\
\mathbf{Msg}\,\beta_1 & I \longrightarrow A & : \mathbf{Header}, \{\mathbf{Body}, \mathbf{Continue}, \mathbf{Nonce}, \mathbf{Session}(\gamma), \mathbf{Nr}_\beta(1), \\
 & & \quad \{\mathbf{SessionKey}^\gamma\}^{\mathbf{PublicKey}\ \alpha}\}^{\mathbf{Signed\ PrivateKey}\ \beta} \\
\mathbf{Msg}\,\alpha_2 & A \longrightarrow I & : \mathbf{Header}, \{\mathbf{Body}, \{\mathbf{operation}\}^{\mathbf{SessionKey}^\gamma}, \mathbf{Continue}, \\
 & & \quad \mathbf{Session}(\gamma)\}^{\mathbf{Signed\ PrivateKey}\ \alpha} \\
\mathbf{Msg}\,\alpha_2 & I_A \longrightarrow B & : \mathbf{Header}, \{\mathbf{Body}, \{\mathbf{operation}\}^{\mathbf{SessionKey}^\gamma}, \mathbf{Continue}, \\
 & & \quad \mathbf{Session}(\gamma)\}^{\mathbf{Signed\ PrivateKey}\ \alpha}
\end{array}
$$

The trace found by Casper means that after a complete run of the protocol, α thinks she has established a session with the intruder, however the intruder establishes a session with β and β thinks it has established a session with α. This was just one trace that Casper found during the design of a session. After thorough analysis of the protocol, the correct definition of the protocol in Casper is:

$$
\begin{array}{lll}
\mathbf{Msg}\,\alpha_1 & A \to B & : \mathbf{Header}, \{\mathbf{Body}, \mathbf{Continue}, \mathbf{Nonce}, \mathbf{Session}, \mathbf{Nr}_\alpha(1), \\
 & & \quad \mathbf{Encryption}\}^{\mathbf{Signed}\{\mathbf{PrivateKey}\ \alpha\}} \\
\mathbf{Msg}\,\beta_1 & B \to A & : \mathbf{Header}, \{\mathbf{Body}, \mathbf{Continue}, \mathbf{B}, \mathbf{Nonce}, \mathbf{Session}(\gamma), \mathbf{Nr}_\beta(1), \\
 & & \quad \mathbf{Nr}_\beta(1), \{\mathbf{SessionKey}^\gamma\}^{\mathbf{PublicKey}\ \alpha}\}^{\mathbf{Signed\ PrivateKey}\ \beta} \\
\mathbf{Msg}\,\alpha_2 & A \to B & : \mathbf{Header}, \{\mathbf{Body}, \{\mathbf{operation}\}^{\mathbf{SessionKey}^\gamma}, \mathbf{Continue}, \mathbf{Nr}_\alpha(2), \\
 & & \quad \mathbf{Session}(\gamma)\}^{\mathbf{Signed\ PrivateKey}\ \alpha}
\end{array}
$$

Casper did not find an attack to this protocol. The difference here is that α can determine the identity of β, because in the message is receives back from either β or the intruder, it is stated that the identity of the sender of the second message must be β. The question is though whether or not this is a good representation of a session setup with SOAP. It must be certain that all properties of the SOAP session are modelled in Casper. This means that every aspect of SOAP, relevant to the session setup, must be modelled in Casper. For example, in chapter 3 each specification has a number of security considerations that might influence such a session. For instance, the fact that headers may always be removed by intermediaries may influence the level of security achieved. It is the task of the application to make sure that digital signatures are always verified and abnormalities dealt with.

Chapter 6

Problems with the security extensions

Other then the security considerations of each of the (security) extensions to SOAP used in the design of the system for the SEP, there are other problems with SOAP or with the extensions to it. These problems might compromise security via different ways. This chapter explains the security issues that became relevant during the design of the system and during the research of the setup of a SOAP session. For some of them it is hard to find a solution to the problem, resulting in possible exposure of information.

6.1 Security context

By using the session setup as explained in the previous chapter it is possible to create a session/security context. The benefit of such a session is, that when two web services are communicating with each other, they can have a "conversation" by using SOAP and its extensions as the underlying protocol. The specifications available only provide ways of performing or encoding a certain item or action relevant to security. For example, they allow to encode a security token. This can be any kind of token, like a username or a sessionid. They provide ways of digitally signing (portions of) SOAP messages and the encryption of elements. A complete security solution is never given. Since December 2002 there are two draft specifications that are relevant to the setup of a security context: *Web Services-Trust* (WS-Trust, see [22]) and *Web Services-SecureConversation* (WS-SecureConversation, see [23]). WS-Trust defines extensions that build on WS-Security to request and issue security tokens and to manage trust relationships. However, it only specifies how security tokens can be encoded in XML and how trust can be federated to other services. It only states that "the requestor must prove any required claims to the satisfactory of the security token service" as a method of requesting a security token. The other draft specification, WS-SecureConversation, specifies how the information about a secure conversation or a security context can be encoded in XML. This includes information such as the identifier, when the context was created, when it expires and which keys are used. It does not specify explicitly how such a context can be setup. Especially when web services are exchanging multiple SOAP messages

with for example a WSCL specification or within a business process by using a BPEL specification, such a setup becomes important. Another problem arises because there are a lot of actors contributing to the world of web services. Due to the creation of many specifications by different companies and institutions, maintaining interoperability between web services seems a very difficult task. The setup of a security context between web services from different companies might become quite difficult when they use their own rules and policies.

6.2 Confidentiality issues

The system for the SEP makes use of the XML Encryption specification to ensure the confidentiality of SOAP messages being exchanged between two web services. However, it is not a complete solution to hide the contents of SOAP messages.

6.2.1 SOAP envelope counting

Every SOAP message has a lot of meta-data in the header of a SOAP envelope and in the attributes of the SOAP elements. By carefully analysing the messages being exchanged, either by analysing them in transit as in intermediary (in case of the routing of messages) or by looking at them as they pass by, an adversary may be provided with a lot of information about the conversation. Consider the following example:

$A \longrightarrow B$ "Product information please"
$A \longleftarrow B$ "Here it is, what do you want?"
$A \longrightarrow B$ "How much is this in total?"
$A \longleftarrow B$ "It's expensive of course!!"
$A \longrightarrow B$ "Here is my credit information"
$A \longleftarrow B$ "Thank you for shopping!"

Even with the encryption of the elements or the body of every SOAP message, the service endpoint of any operation is always known (otherwise routing would be impossible). By looking at the number of messages, it could be possible to determine the type of operation being requested. Furthermore, other meta-data in every SOAP message, think of namespace declarations and attributes, might also reveal information. Of course, this is not a problem if it is not relevant whether or not it is known to a third party what type of conversation it is or what the outcome is. As long as the details (such as creditcard numbers) are protected. Which brings up the topic of XML-Encryption granularity.

6.2.2 Message structure (XML-Encryption granularity)

The XML Encryption specification does not define any level of encryption by encrypting elements or their sub-elements. Such a choice is completely arbitrary and left to the application or endpoint. This might also provide information to a third party about the type of conversation being held:

```
<CreditCard Limit="5,000" Currency="GBP">
   <Number>
     <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                    type="http://www.w3.org/2001/04/xmlenc#Content">
        <CipherData>
           <CipherValue>A23B45C56</CipherValue>
        </CipherData>
     </EncryptedData>
   </Number>
   <Expiration>04/04</Expiration>
</CreditCard>
```

It is evident from this example that a creditcard is being used. The only thing that is being protected is the number itself. It might be better to make the granularity of the encryption of the SOAP elements like the following example. However, you might reveal more cipher text for a key which may make it easier to break cryptographically.

```
<EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
               type="http://www.w3.org/2001/04/xmlenc#Content">
   <CipherData>
      <CipherValue>A23B45C56</CipherValue>
   </CipherData>
</EncryptedData>
```

In this example, the entire `<CreditCard>` element is encrypted. When there are more encrypted blocks, it might not be known which element is which.

6.2.3 XML specifications

With web services, it is quite easy to retrieve the layout from a certain SOAP message. There are various methods of doing so. The first one is making use of the XML Schema definitions of the XML documents or XML elements being transferred. Every SOAP request is a request in XML and therefore *usually* has an XML schema definition of the elements that are used in the SOAP message. When an intruder knows this definition and has access to a SOAP message with encrypted content, it can determine via the meta-data which request/conversation is being held. By analysing the XML schemas, the adversary also knows most of the encrypted content.
This gets worse when the WSDL or the WSCL definitions are known. Since web services are designed to let computers communicate at run-time and in a platform independant way and , the syntax and the structure of every operation of web services can be looked up in registries and therefore an adversary knows even in more detail what the content of an encrypted SOAP message is. Consider the following example of a SOAP request, the SOAP elements themselves have been omitted for brevity:

```
<student>
  <first-name>Robert</first-name>
  <last-name>Boezeman</last-name>
  <dateOfBirth>19-12-1979</dateOfBirth>
  <room>453</room>
  <supervisor>andrew</supervisor>
</student>
```

Figure 6.1: An example of a SOAP request.

If this piece of XML is transmitted and the WSDL/WSCL is known, it might be possible for an adversary to perform cryptographic attacks. In this example, there 156 characters of which then only 33 unknown. If this block is completely encrypted, already 78% of the content is known. This is especially a problem when the type of encryption is one of which the length of the encryption block is equal to length of the plaintext. This is the case with algorithm such as DES, Blowfish, IDEA and AES. This problem cannot be solved if it is desirable to have web services that are highly interoperable.

Chapter 7

Conclusion

This chapter discusses the research topics as explained in the introduction. For each topic, it gives a small report of how the research proceeded and what the results are.

- The first part of the research focusses on the communication between the service requestor and the service provider. Also, it focusses on how the proposed security extensions can be used or improved upon to establish secure communication between them. The question here was how the communication between the service requestor and the service provider works and how it can be secured.

  During the research it became evident that a lot of work is going on the world of web services. Big IT companies such as Microsoft and IBM are spending many resources on the development of standards, also relating to security. However, it seems to be the case that there are more standards being developed than actual web services being built. Most of the specifications under development are not used or have not been implemented yet. Therefore, it was sometimes hard to implement. The communication between the service requestor and the service provider is not very hard to setup. However, when using extensions to SOAP, it turns out that the implementations that implement the extensions are in a very early stage of development as well. This is especially difficult when building a network of secure web services that use the (proposed) security extensions. However, it was possible to implement the setup of a SOAP session as explained in chapter 5.3.

- The second part of the research done involved a thorough analysis of SOAP and the way it binds to the transport layer protocols. An important part of the research was to analyse SOAP and to determine if web services and SOAP can be used to design a system for the SEP to connect their computing systems that they use for their administration. The system designed and built for the SEP is discussed in detail in chapter 4. The question here was whether or not it was possible to design a secure network of web services. What are the requirements of that system and how must it be designed?

  The design of the system resulted in a schematic view of the computer

systems used by the SEP of the OUCL. UML-models were made to give an insight in the architecture of the network of web services and of each class. After discussions with the involved actors, it became evident that the design met the requirements of the desired system. However, because of the proportions of the system, it was impossible to implement all parts of it. Instead, the choice was made to implement the portions of the design with which the suitability of the design could be proved. Although the system is not fully implemented, the work done is a very good starting point to finish the implementation.

- Finally, the proposed (security) extensions to SOAP and how they can be used to setup a session/security context within SOAP were a big part of the research done for the SEP. The existing standards do not have a notion of a session/security context at the time of this writing.

  In theory it is possible to setup a secure SOAP session. This can be done by using the proposed security extensions to SOAP, which are still under development. The research involved the analysis of those standards and ways to use and develop them to create such a session, by using parts of those standards. The result of the research is a definition of a security protocol that has been analysed using Casper. More information on how a security context can be setup can be found in chapter 5. The analysis of the session setup proves that an adversary can not break the protocol by the interception, alteration or the (re-)sending of messages. However, implementation faults and the insecurity of the servers running the web services, faults in the security considerations of each specification or if any of the assumptions made in chapter 4.6 are not upheld, the level of security might be compromised. Furthermore, security is most of all a social problem as well as a technical one.

Remaining work

First, there is still some implementation work to be done. As explained, there are parts of the design document that have not been implemented. There are also beta versions or newer implementations of for example the SOAP D-Sig specification that are improvements upon the already implemented one.
Furthermore, there are a lot of specifications still under development that add extensions to SOAP and try to tackle the security issues mentioned in this thesis from another way. There is a lot of work to be done still in the area of securing web services and it might be very well possible that there are already specifications under development that try to setup a SOAP session and try to achieve message level security with web services. The stated solution in this thesis is just one of the ways to do this.

Appendix A

Glossary

cws - client (of a) web service. This term is used in this thesis to indicate the initiator of a SOAP request/conversation.

BPEL - Business Process Execution Language. This language is designed to model a business process consisting of various web services into one.

OUCL - Oxford University Computing Laboratory. This is the part of the University of Oxford where research is being performed in many of the areas of computing science.

SOAP - Simple Object Access Protocol. The protocol used by web services nowadays. It is a simple lightweight protocol designed to let computer systems communicate in a platform independant, highly interoperable way.

SEP - Software Engineering Programme. At the Oxford University Computing Laboratory there is programme called the software engineering programme which is responsible for everything related to software engineering.

sws - serving web service. This term is used in this thesis to indicate the serving web service that receives the first SOAP message from a client of a web service.

UDDI - Universal Description Discovery and Integration - This is a standard for discovering Web Service Description Language definitions of web services to dynamically discover and invoke them. This standard is used in many registries on the Internet.

W3C - World Wide Web Consortium. This institution is responsible for maintaining standards on the Internet. They are responsible for the development of many protocols and standards such as XML, SOAP and many others.

WSCL - Web Services Conversation Language. This language is designed to describe conversations between web services so that they can exchange multiple SOAP messages. It is possible to describe entire XML documents that can be exchanged between them.

WSDL - Web Services Description Language. This language is designed to describe web services and the way their operations can be invoked dynamically.

XML - eXtensible Markup Language. This language is being developed by the World Wide Web Consortium and is a simple, very flexible text format to structure data and information.

Appendix B

Secure SOAP sequence

There are three different classes in the diagram on page 67. The *ComlabWebService* is the parent class of all the web services in the design of the network of web services. It is an abstract class meaning that each web service extends it. Explaining the diagram:

1 The client web service makes a normal SOAP request (an envelope) as each web service does. This function is abstract in the class *Web service*. This means that every web service extending the class must implement the function *CreateSOAPEnvelope()*.

2 The client web service instantiates a SOAPDigitalSigner class which is responsible for digitally signing a SOAP message and to add a `<Security>` element with a `<Signature>` element.

3 The SOAPDigitalSigner is provided with a pointer to the right SOAPEnvelope.

4 The KeyPair to make the signature is declared.

5 The SOAPDigitalSigner is given the command to make the signature. It makes this signature according to the processing rules specified in [12].

6 The SOAPEnvelope is sent to the other web service.

7 The receiving web service instantiates an Validator class. This class is responsible for validating digital SOAP signature. It also does this according to the processing rules in [12].

8 The web service gives the Validator the correct SOAPEnvelope.

9 The right KeyPair to validate the signature is specified to the Validator.

10 The session is set, *s* is aware of the session and the security context.

11 The *validate()* function is called to give the command to validate the signature.

12 The serving web services handles the SOAP request normally. This function is abstract, meaning that every web service that extends the class *ComlabWebService* must implement the function *HandleSOAPEnvelope()*.

13-17 These steps are analogous to steps 1 through 5.

18 The SOAPEnvelope is sent back.

19-23 These steps are analogous to steps 7 through 11.

x,y Each web service instantiates a new SOAPDigitalSigner and a Validator per session. After the session is completed, they are destroyed.

This SOAP messaging diagram does not depict the encryption of the body of a SOAP message for brevity and readibility of the diagram. However, there is another class for that, the *EncryptedBody* class (see page 34). It can easily be understood how this class should be deployed: by letting every web service instantiate this class if needed and before sending a SOAPEnvelope, let this class encrypt the body first. Upon receiving a SOAP envelope with an encrypted body, a web service calls the *EncryptedBody* class to decrypt it.

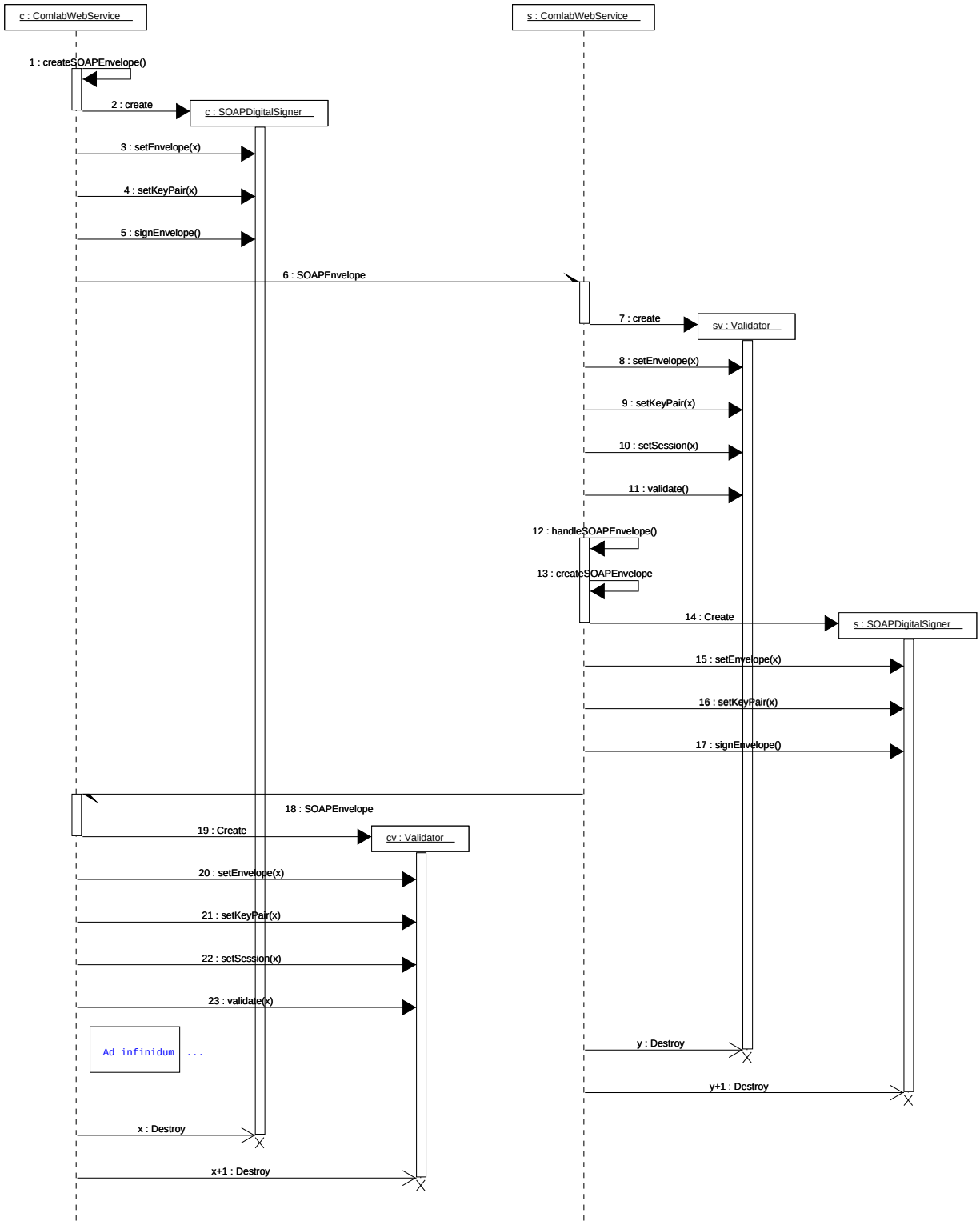

Figure B.1: Sequence diagram of how a session is setup.

Appendix C

Sequence diagram

In figure C.1 there is a sequence diagram that explains the way the **Client** web service and the **Print** web service communicate. This sequence is just one possible trace of execution, but illustrates all the available functionality for retrieving information about courses and printing documents. To explain the diagram in a step by step manner:

1 The *DoPost()* operation is inherited from the **ComlabWebService** class and handles the way SOAP envelopes are binded to the HTTP protocol. This operation is called by the servlet container that xthe web services are deployed in.

2 Since the **Client** has to interact with the end-user, it has the *ProcessHTML-Request()* to do this.

3 The normal creation of any SOAPEnvelope is done here, this method is inherited from the **ComlabWebService** class.

4 This function retrieves the acronym given by the end-user and adds it to the SOAP envelope, to make the request to the *Print* web service.

5 The envelope is sent.

6 Same as step 1.

7 The method *HandleSOAPEnvelope()* is also inherited from the **Comlab-WebService**. Every web service must implement it, in order to process it.

8 This function of the **Print** web service takes a normal SOAP envelope and adds elements to the body, according to the internal structure of the **Course** class.

9 The function *RetrieveDirList()* retrieves the directory list of a particular course, so that it can be added to the SOAP envelope.

10 The SOAP envelope retrieved from the **Database** web service contains all the information on a particular course. The **Print** web service incorporates this information along with the retrieved directory list in to its own session with the **Client** inside a SOAP envelope.

11-12 Are the same as steps 6 and 7.

13 The special operation in the **Client** to interpret SOAP envelopes with a course specified inside.

14-15 Respectively, the directory list and the number of attendants of a course are retrieved.

16 The retrieved information is displayed back to the end user, in the form of a web page.

17 If the end-user then decides to print any of the documents an X amount of times, the function *printDocument()* is called. This function adds the request to print document to the newly created SOAP envelope (steps 1 to 3).

18 This envelope is then sent to the **Print** web service.

19 When the **Print** web service handled the SOAP envelope, it gives the correct command to the printer.

20 It makes a report envelope of its progress.

21 This envelope is sent back to the **Client** which in its turns processes it and gives the report back to the end-user.

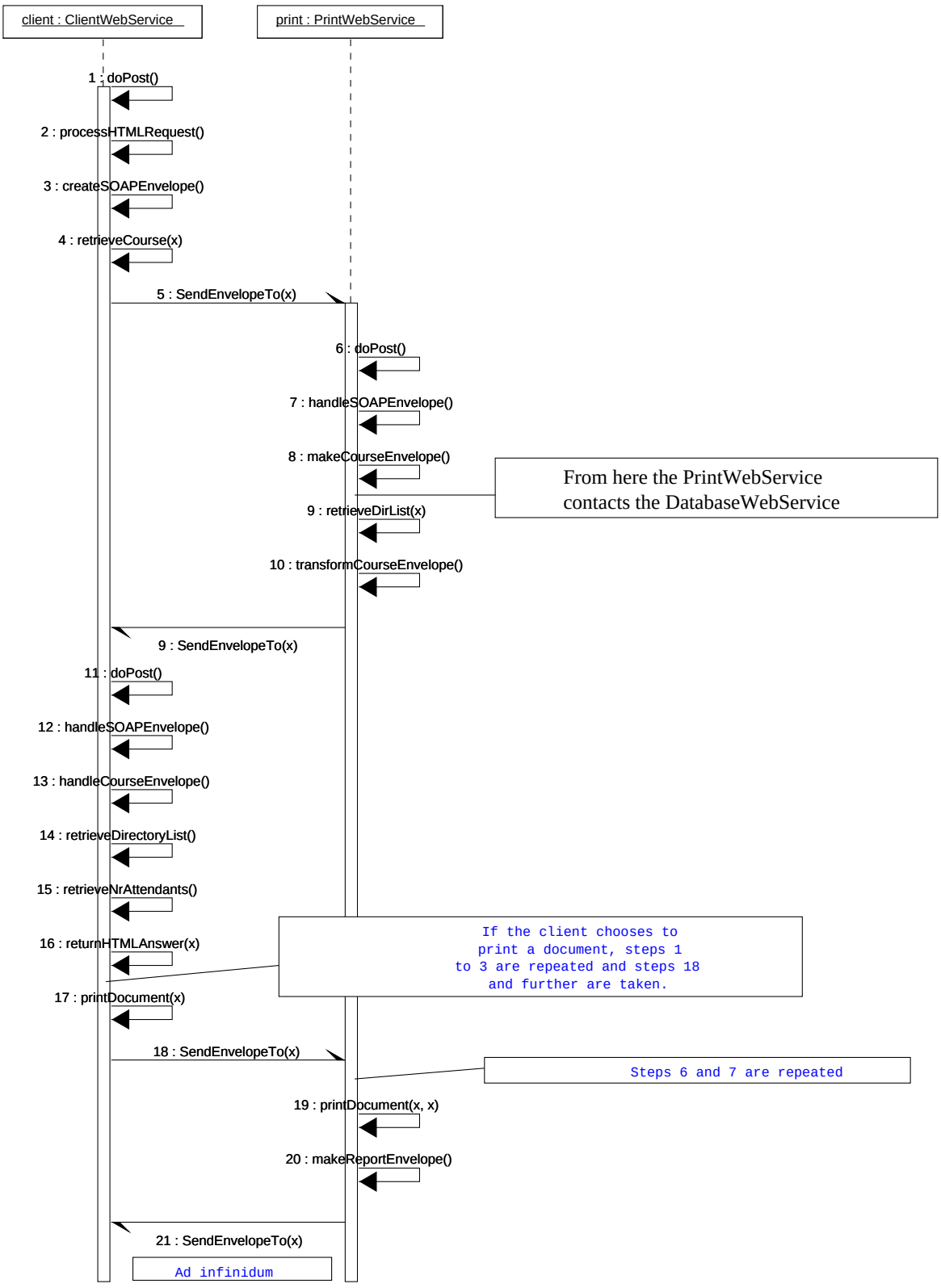

Figure C.1: Client web service - Print web service sequence diagram

Appendix D

Sequence diagram

The diagram in figure D.1 is a sequence diagram that explains the way the **Client** web service and the **Password** web service communicate. To explain the diagram in a step by step manner:

1 The *DoPost()* operation is inherited from the **ComlabWebService** class and handles the way SOAP envelopes are binded to the HTTP protocol. This operation is called by the servlet container that the web services are deployed in.

2 Since the **Client** has to interact with the end-user, it has the *ProcessHTML-Request()* to do this.

3 The normal creation of any SOAPEnvelope is done here, this method is inherited from the **ComlabWebService** class. The acronym of the course of which the password file must be retrieved is added to the envelope.

4 The envelope is sent.

5 Same as step 1.

6 The method *HandleSOAPEnvelope()* is also inherited from the **Comlab-WebService**. Every web service must implement it, in order to process it.

7 This function of the **Print** web service takes a normal SOAP envelope and adds elements to the body, according to the internal structure of the **PasswordFile** class.

8 The **Database** web service filled the envelope with the password file of the course. The function *handlePwdfileEnvelope()* retrieves the password file from the SOAP envelope and stores it in memory.

9 The function *readExistingPwdfile()* reads the currently deployed password file from the right directory on the Solaris server and stores it in memory.

10 After both password files have been retrieved and stored in memory, this operation merges them, according to the steps mentioned in section 4.5.5.

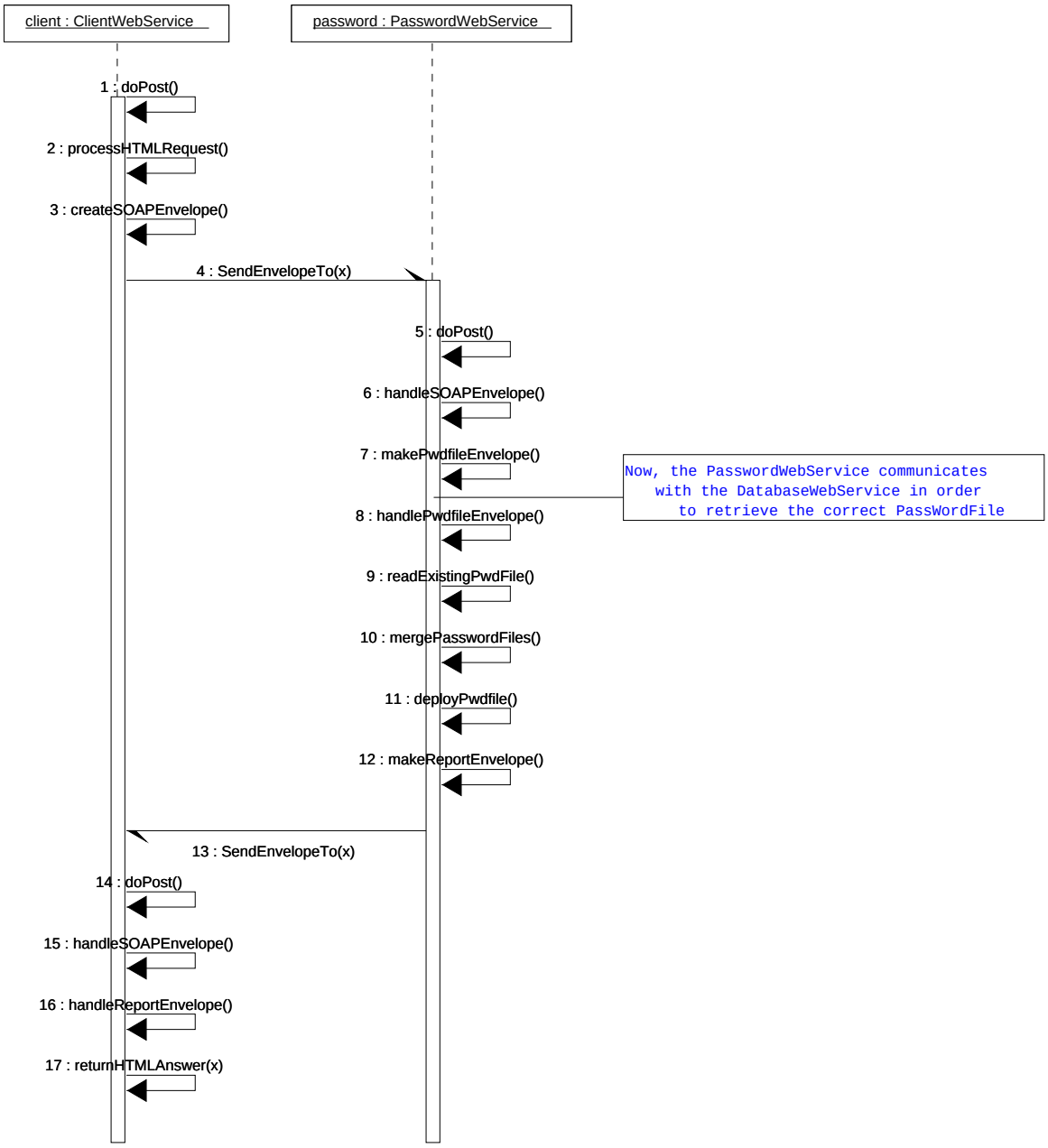

Figure D.1: Client web service - Password web service sequence diagram

11 Once the password files have been merged into a new one, it can be deployed. This is also done according to the rules mentioned in section 4.5.5.

12 To provide the **Client** web service with feedback, report envelope is created.

13 The envelope is then sent back to the [Client]

14-15 These are the same as steps 5 and 6.

16 The report envelope is processed to give feedback to the end-user

17 The end-user is provided with feedback.

Appendix E

Sequence diagram

The diagram in figure E.1 is a sequence diagram that explains the way the **Print** web service and the **Database** web service communicate. The **Print** web service does not start this communication chain by itself, it is a an elaboration of the diagram in appendix C. To explain the diagram in a step by step manner:

1 After the **Client** web service requested information on a certain course, by providing the Print web service with the corresponding acronym, it creates a SOAP envelope with the internal structure as the **Course** class. This is done by the operation *makeCourseEnvelope()*.

2 The envelope is sent.

3 The *DoPost()* operation is inherited from the **ComlabWebService** class and handles the way SOAP envelopes are binded to the HTTP protocol. This operation is called by the servlet container that the web services are deployed in.

4 The method *HandleSOAPEnvelope()* is also inherited from the **ComlabWebService**. Every web service must implement it, in order to process it.

5 The **Database** web service stores the envelope. The function *handleCourseEnvelope()* retrieves the acronym, the course and its structure from the SOAP envelope and stores it in memory.

6 A connection to the SSTL-database is established.

7 The operation *retrieveCourse()* makes the right query that is executed on the database. When it has created the query, it calls the function *execQuery()* which performs the call to the database. The output of the query is stored in the memory of the **Database** web service.

8 The envelope is then filled with the retrieved course information.

9 The envelope is sent back to the **Password** web service.

10-11 These stept are the same as stept 3 and 4.

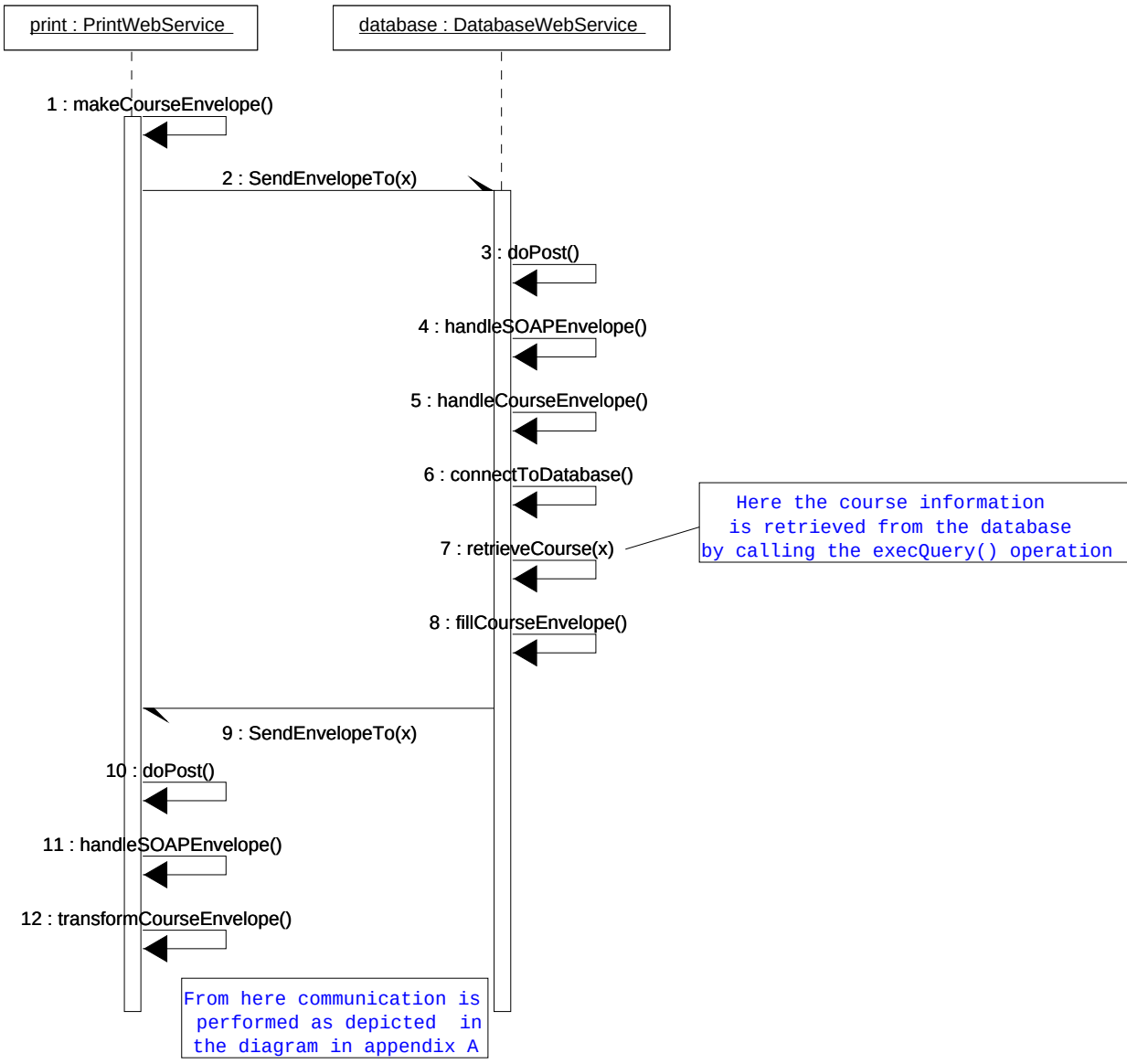

Figure E.1: Print web service - Database web service sequence diagram

12 The received envelope containing the course information is processed by the **Print** web service.

After this, the **Print** web service proceeds as depicted in the sequence diagram in appendix C.

Appendix F

Sequence diagram

The diagram in figure F.1 is a sequence diagram that explains the way the **Password** web service and the **Database** web service communicate. The **Password** web service does not start this communication chain by itself, it is a an elaboration of the diagram in appendix D. To explain the diagram in a step by step manner:

1 After the **Client** web service requested a password file transfer by providing the **Password** web service with the acronym of the course, it creates a SOAP envelope of which the structure corresponds to the structure of the **PasswordFile** class. This is done by calling the operation *MakePwdfileEnvelope()*.

2 The envelope is sent.

3 The *DoPost()* operation is inherited from the **ComlabWebService** class and handles the way SOAP envelopes are binded to the HTTP protocol. This operation is called by the servlet container that the web services are deployed in.

4 The method *HandleSOAPEnvelope()* is also inherited from the **ComlabWebService**. Every web service must implement it, in order to process it.

5 The **Database** web service stores the envelope with the password file of the course. The function *handlePwdfileEnvelope()* retrieves the password file from the SOAP envelope and stores it in memory.

6 A connection to the SSTL-database is established.

7 The operation *retrievePasswordFile()* makes the right query that is executed on the database. When it has created the query, it calls the function *execQuery()* which performs the call to the database. The output of the query is stored in the memory of the **Database** web service.

8 The envelope is then filled with the contents of the retrieved password file.

9 The envelope is sent back to the **Password** web service.

10-11 These stept are the same as stept 3 and 4.

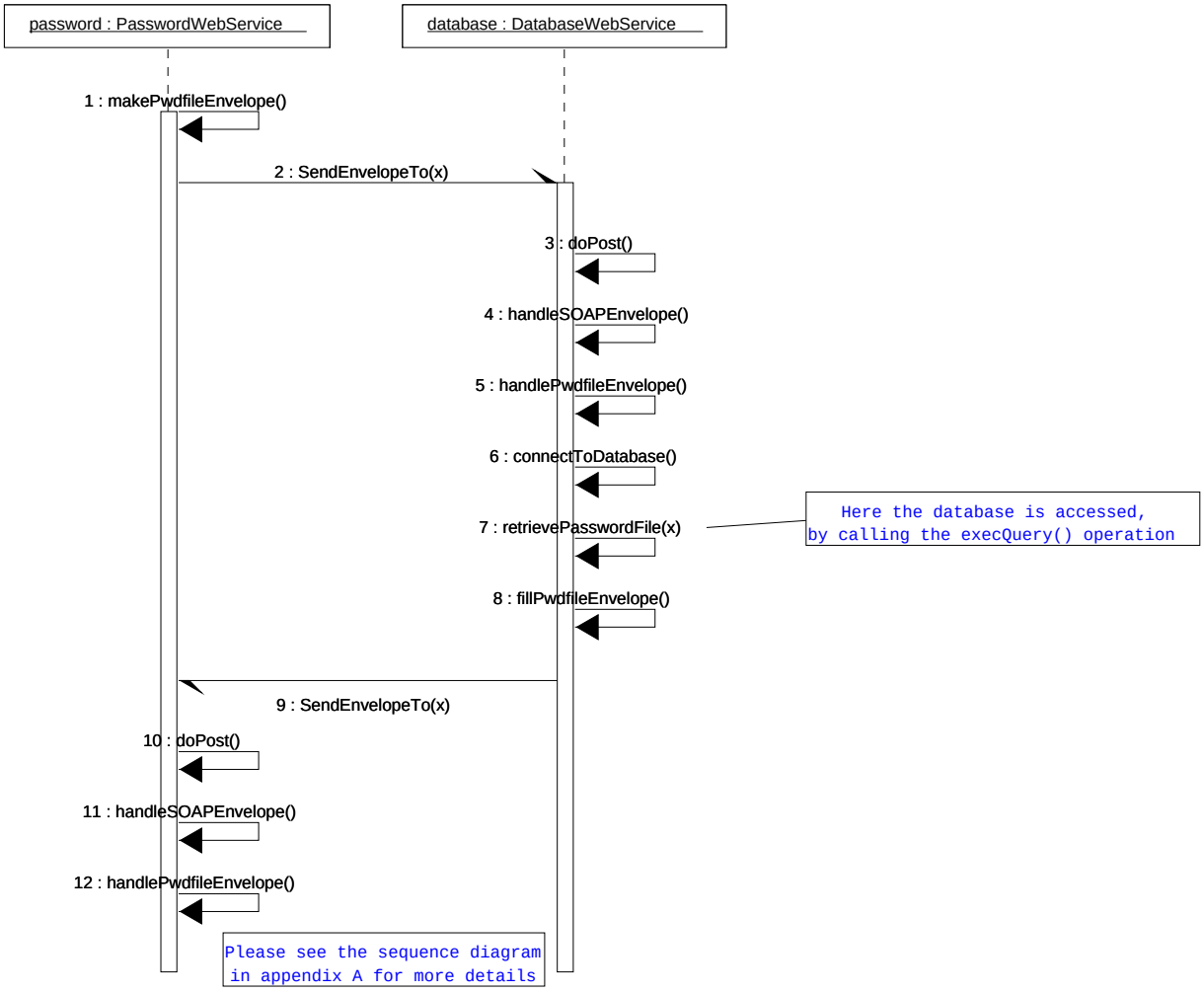

Figure F.1: Password web service - Database web service sequence diagram

12 The received envelope containing the password file is processed.

After this, the **Password** web service proceeds as depicted in the sequence diagram in appendix D.

Appendix G

Sequence diagram

The sequence diagram in figure G.1 illustrates how the **ComlabWebService** is updated by the SWS.

1 When the **SecurityWebService** is started, it initialises the access list, which holds all the lists of the web services that have access to each other. It does this by calling the function *initialiseAccessList()*. This is done only once when the SWS is started the first time. This list is then stored in memory. The SWS assumes that the list is accurate upon reading, meaning that no web service needs to be updated.

2 If the administrator of the network of web services changes one of the lists on file, the administrator calls the function *reviewAccess()*. The SWS then compares the access list stored in memory with the file **accesslist.xml** stored in the same directory as the SWS is deployed in. If any changes are made, the SWS proceeds with updating the corresponding web service(s).

3 It creates a new SOAP envelope first.

4 The SWS then adds elements to the body of the SOAP envelope according to the internal structure of the **AccessList** class appropriate to the web service that needs its list updated.

5 The envelope is sent.

6 The *DoPost()* operation is inherited from the **ComlabWebService** class and handles the way SOAP envelopes are binded to the HTTP protocol. This operation is called by the servlet container that the web services are deployed in.

7 The method *HandleSOAPEnvelope()* is also inherited from the **ComlabWebService**. Every web service must implement it, in order to process it.

8 Every web service that extends the **ComlabWebService** automatically has the operation *HandleAccessListEnvelope()* which is responsible for updating the key repository that it has.

9 When this function has updated the access list a new SOAP envelope is created.

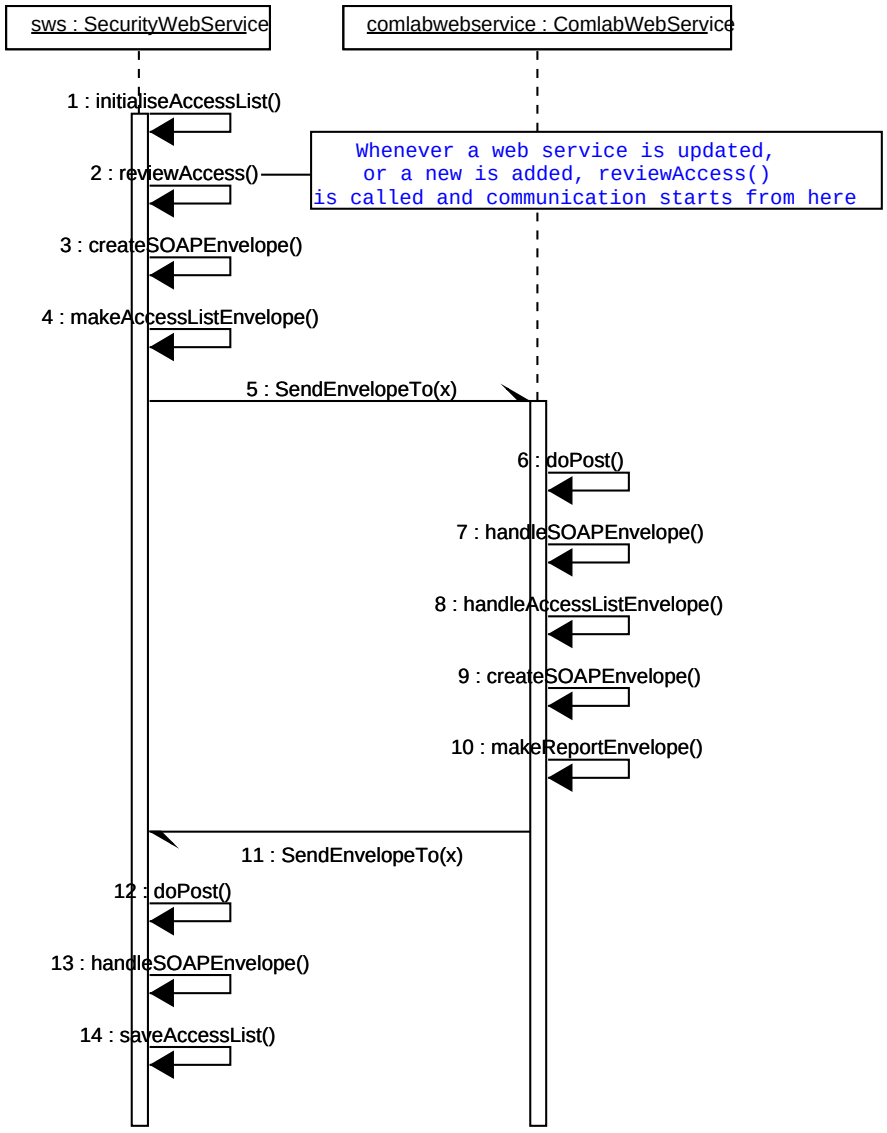

Figure G.1: Security web service sequence diagram

10 A report is created indicating success or failure.

11 The report SOAP envelope is sent back to the SWS.

12-13 These are the same steps as 6 and 7.

14 Once the SWS has confirmation that the access list of the web service has been updated, it saves its list into memory and the administrator is notified of the succes or failure.

Appendix H

SOAP example

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
            xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
   <S:Header>
      <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
         <ds:Signature>
            <ds:SignedInfo>
               <ds:CanonicalizationMethod Algorithm= "http://www.w3.org/2001/10/xml-exc-c14n#"/>
               <ds:SignatureMethod Algorithm= "http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
               <ds:Reference URI="#msgbody">
                  <Transform Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
                  </Transforms>
                  <ds:DigestMethod Algorithm= "http://www.w3.org/2000/09/xmldsig#sha1"/>
                  <ds:DigestValue>LyLsF0Pi4wPU...</ds:DigestValue>
               </ds:Reference>
            </ds:SignedInfo>
            <ds:SignatureValue>KiGF9JK7G4Tyu...</ds:SignatureValue>
               <ds:KeyInfo>
                  <wsse:SecurityTokenReference>
                     <wsse:Reference URI="[reference to sender's key]"/>
                  </wsse:SecurityTokenReference>
               </ds:KeyInfo>
         </ds:Signature>
      </wsse:Security>
   </S:Header>
   <S:Body Id="msgbody">
       <Course>
          <name>XML processing</name>
          <id>xml</id>
          <attendantList>
              <attendant>John Doe</attendant>
              <attendant>Jane Doe</attendant>
              ...
          </attendantList>
          <date>31-01-2003</date>
          <acronym>xml</acronym>
          <documentList>
              <document>lecturesnotes.pdf</document>
              <document>errata.ps</document>
              ...
          </documentList>
       </Course>
   </S:Body>
</S:Envelope>
```

This example illustrates how a the internal structure of a SOAP envelope matches the internal structure of the class it represents. For each class there is a standard SOAP envelope.

Bibliography

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML).* http://www.w3.org/TR/REC-xml, October 2000.

[2] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 1.2.* http://www.w3.org/TR/wsdl12, March 2003.

[3] Carlos C. Tapang. *Web Services Description Language (WSDL) Explained.* http://msdn.microsoft.com/library/en-us/dnwebsrv/html/wsdlexplained.asp, July 2001.

[4] Tom Bellwoo, Luc Clment, David Ehnebuske, Andrew Hately, Maryann Hondo, Yin Leng Husband, Karsten Januszewski, Sam Lee, Barbara McKee, Joel Munter, and Claus von Riegen. *UDDI Version 3.0.* http://uddi.org/pubs/uddi_v3.htm, July 2002.

[5] Satoshi Hada. *SOAP security extensions: digital signature.* http://www-106.ibm.com/developerworks/webservices/library/ws-soapsec/, October 2002.

[6] Robert-Jan Boezeman. *Requirements and Risks analysis for the secure deployment of web services.* http://www.boezeman.net/ boezeman/reqrisk.pdf, November 2002.

[7] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. *SOAP Version 1.2 Part 1: Messaging Framework.* http://www.w3.org/TR/soap12-part1, June 2002.

[8] Aaron Skonnard. *Routing SOAP Messages with Web Services Enhancements 1.0.* http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/routsoapwse.asp, January 2003.

[9] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Venkatesh Chopella, Kannan Govindarajan, Alan Karp, Harumi Kuno, Mike Lemon, Gregory Pogossiants, Shamik Sharma, and Scott Williams. *Web Services Conversation Language (WSCL) Version 1.0.* http://www.w3.org/TR/wscl10, March 2003.

[10] John J. Barton, Satish Thatte, and Henrik Frystyk Nielsen. *SOAP Messages with Attachments.* http://www.w3.org/TR/SOAP-attachments, December 2000.

[11] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. *Simple Object Access Protocol (SOAP) 1.1.* http://www.w3.org/TR/SOAP, May 08 2000.

[12] Mark Bartel, John Boyer, Barb Fox, and Ed Simon. *XML-Signature Syntax and Processing.* http://www.w3.org/TR/2000/CR-xmldsig-core-20001031/, October 31 2000. XML-Signature W3C Candidate Recommendation.

[13] Bob Atkinson and Giovanni Della-Libera. *Web Services Security (WS-Security).* http://www-106.ibm.com/developerworks/library/ws-secure.pdf, 05 April 2002.

[14] Takeshi Imamura, Blair Dillaway, and Ed Simon. *XML Encryption Syntax and Processing.* http://www.w3.org/TR/xmlenc-core/, December 2002. W3C Recommendation.

[15] Takeshi Imamura and Hiroshi Maruyama. *Decryption Transform for XML Signature.* http://www.w3.org/TR/xmlenc-decrypt, December 2002. W3C Recommendation.

[16] Ivar Jacobson Grady Booch, James Rumbaugh. *The Unified Modelling Language User Guide, the ultimate tutorial of the UML from the original designers.* Addison-Wesley, August 2000.

[17] Giovanni Della-Libera, Brendan Dixon, and Joel Farrell. *Securing Web Services World: A proposed architecture and roadmap.* http://msdn.microsoft.com/library/en-us/dnwssecur/html/securitywhitepaper.asp, April 7 2002.

[18] Steve Graham, Simeon Simeonov, Toufic Boubez, Doug Davis, Glen Daniels, Yuichi Nakamura, and Ryo Neyama. *Building web services with Java.* Sams Publishing, December 2001. Making sense of XML and SOAP and WSDL and UDDI.

[19] Gavin Lowe, Philipa Broadfoot, and Mei Lin Hui. *Casper, a compiler for the Analysis of Security Protocols.* http://web.comlab.ox.ac.uk/oucl/work/gavin.lowe/Security/Casper/, December 2001.

[20] C.A.R. Hoare. *Communicating Sequential Processes*, 1985.

[21] Formal systems (Europe) Ltd. *Failures-Divergence Refinement - FDR 2 User Manual.* http://www.formal.demon.co.uk/FDR2.html, 1997.

[22] Giovanni Della-Libera, Brendan Dixon, Praerit Garg, and Phillip Hallam-Baker. *Web Service Trust Language (WS-Trust).* http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-trust.asp, December 2002.

[23] Giovanni Della-Libera, Brendan Dixon, Praerit Garg, and Satoshi Hada. *Web Service Secure Conversation (WS-SecureConversation).* http://www-106.ibm.com/developerworks/library/ws-secon, December 2002.