Julia Sulina

X-Road Web services migration adapter

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

10 May 2016

Author(s) Title Number of Pages Date	Julia Sulina X-Road Web services migration adapter 47 pages + 4 appendices Tuesday 10th May, 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Development
Instructor(s)	Patrick Ausderau, Lecturer

Web services provide a possibility for information transfer between remote applications of different platforms. Estonian X-Road is a large-scale Web service network of state and private organisations' information systems exchanging data via Simple Object Access Protocol (SOAP) messages. These applications are unprepared for message alteration, caused by renewal of X-Road protocol for cross-border communication.

The concept of the SOAP intermediary is used to develop an adapter converting X-Road messages on their route, instead of adjusting information systems code. Feasibility of the concept is evaluated by a theoretical enquiry and its implementation approbation. This thesis is a report of the lessons learned while analysing, developing and testing the intermediary.

The results show that the adapter is able to accurately recognize a need for transformation and convert messages at a satisfactory speed. The intermediary proved to be usable with various configurations, which makes it suitable for addressing not only the described X-Road service migration, but also further changes in messaging. Acknowledging that, additional testing of the application on a production-like environment should be performed.

Keywords	Web service, SOAP intermediary, X-Road, XML parsing, WSDL comparison

Contents

Abbreviations

API	Application Programming Interface.
CA	Central Authority.
DOM	Document Object Model.
DTD	Document Type Definition.
EU	European Union.
HDD	Hard Disk Drive.
HTML	HyperText Markup Language.
HTTP	Hypertext Transfer Protocol.
HTTPS	HyperText Transfer Protocol Secure.
IDE	Integrated Development Environment.
IP	Internet Protocol.
JAR	Java Archive.
JAX-WS	Java API for XML-Based Web Services.
JAXB	Java Architecture for XML Binding.
JAXP	Java API for XML Processing.
JDK	Java Development Kit.
RIA	Republic of Estonia Information System Authority.
RIHA	Administration System for the State Information System.
SAX	Simple API for XML.
SIS	State Information System.
SMTP	Simple Mail Transfer Protocol.
SOA	Service-Oriented Architecture.
SOAP	Simple Object Access Protocol.
SSD	Solid-State Drive.
STAR	Social services and benefits data register.
StAX	Streaming API for XML.
TCP	Transmission Control Protocol.
UML	Unified Modeling Language.
URL	Uniform Resource Locator.
W3C	World Wide Web Consortium.
WSDL	Web Services Description Language.
XML	eXtensible Markup Language.
XML-RPC	XML Remote Procedure Call.
XSL	Extensible Stylesheet Language.
XSLT	Extensible Stylesheet Language Transformations.

Glossary

Servlet	”A servlet is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients” [1].
Web service	”A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.” [2].
X-Road	X-Road is a name for data exchange layer of information systems and digital registers [3].
XML Schema	XML Schema is a description of XML document type, indicating constraints on its structure and content [4].

1 Introduction

Web services provide a possibility for data exchange to applications of different platforms. Estonian X-Road is an interesting example of a large-scale Web service network of state and private organisations, which exchange information though Simple Object Access Protocol (SOAP) messages sent at a secured communication layer. Its infrastructure enables preserving sensitive data in scattered data stores, with access permitted only to certified members.

According to the current schedule[1], in October 2016, a major change will take place in the X-Road communication protocol with an aim of extending data transmission across state borders within the European Union. This change means that all current members should adjust the production and consumption of X-Road messages.

Taking part in the development of information systems engaged in X-Road communication due to work assignments at Tieto Estonia, caused interest towards the migration process and its implementation possibilities. This thesis aims to test an idea of addressing protocol change outside the member's software. The goal is to create a SOAP intermediary application, which can translate X-Road messages to a new format without altering the code of existing information systems.

[1]02.03.2016

2 Theoretical background

2.1 Simple Object Access Protocol (SOAP) Web services

Web services are ”a new breed of Web application, and they are self-contained, self-describing, modular applications that can be published, located and invoked across the Web”[2] [5, p.219]. Web services[3] create an opportunity for developing applications in the Service-Oriented Architecture (SOA) paradigm, by presenting software functionality in a platform independent manner [6, p.499]. SOA is an computer software design / integration pattern in which application remotely provide services to other components using defined communications protocol [7]. Such applications are able to request the needed information ad hoc [8, p.7] reusing other software parts which provide the requested data [7].

Simple Object Access Protocol (SOAP) is one of the protocols which can be used for information transmission of Web services and it has become common for such message interchange due to its simplicity, extensibility and interoperability [9, p.55; 10, p.387]. SOAP is a standardized protocol[4], representing messages shared between applications in eXtensible Markup Language (XML) format [8, p.7]. Hypertext Transfer Protocol (HTTP) is common for SOAP messages transport[5], where a request[6] is sent over HTTP POST and answered by an HTTP response [8, p.36].

SOAP Web services are described by Web Services Description Language (WSDL) – such a representation makes it easier to create, maintain and consume services by providing a predefined structure of its components [8, p.79]. WSDL is a regularly program-

[2]IBM definition.

[3]W3Schools definition:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. [2]

[4]Standardized by World Wide Web Consortium (W3C) available at https://www.w3.org/TR/soap12/

[5]Other protocols like HyperText Transfer Protocol Secure (HTTPS) and Simple Mail Transfer Protocol (SMTP) can also be used.

[6]Excluding WSDL requests.

matically generated XML document, describing Web service and consisting of:

- *definitions* element: is a WSDL preamble or container, defines target namespace and xmlns namespaces providing XML Schema URLs;
- *types* element: contains all abstract data types definitions belonging to messages imported from XML Schema or defined in WSDL;
- *message* and *part* elements: describe messages that will be exchanged as a collection of named parts (data values) of a particular type;
- *operation* element: presents operation (method) translation to messages[7], introducing messages used on request and response[8];
- *portType* element: defines the service interface;
- *binding* element: provides concrete implementation of a method and operation binding defining protocols;
- *port* element: declares location of the service: service name, port, address. [11, pp.111-120; 8, p.80]

Web service applications act in three main roles: service provider – implements and publishes Web service, service registry – stores information about it, and service requester[9] – discovers and invokes functionality [5, p.220; 12, pp.7-8].

Web services can be implemented using static composition; in this case, the aggregation of services takes place at design time, or dynamic composition, which allows determining and replacing service components during runtime. Static composition is suitable when service functionality rarely changes. Dynamic composition in its turn can accommodate frequent changes; however it requires support of automatic discovery of altered service components. [5, p.221]

Snell et al [8, pp.79-80] states that dynamic discovery can be implemented in applications and may help to adjust a consumer to the changed service structure, though as WSDL descriptions versioning is not supported it is likely for requesting application to encounter problems on major change of service [8, pp.79-80]. Sheng et al [5, p.221] also sees complications in developing dynamic composition services originating from the need of all partners being able to meet the requirements of flexibility [5, p.221]. Snell et al [8, pp.79-80] proposes that WSDL in production should be immutable.

[7]Operation may contain of one (Single-Message Exchange pattern) or more (Multiple-Message Exchanges pattern) message exchanges [8, p.91].

[8]Operation input, output and fault messages.

[9]In the rest of this thesis, the terms service requester and service consumer will be used interchangeably.

SOAP messages are contained in an *envelope*, defining namespace information of message elements and consisting of a *header* (optional), composed of blocks with information about message processing, and a *body* (mandatory), incorporating actual data of the message [8, p.17; 10, pp.390-391].

SOAP serialization is a process of transformation of application data types to XML-based string format [10, p.391]. Deserialization or SOAP parsing is the opposite process of converting the incoming message to application objects [10, p.392; 13, p.90]. Such a conversion of messages is illustrated by figure 1: each message in case of regular request-response communication encounters the serialization and deserialization process twice.

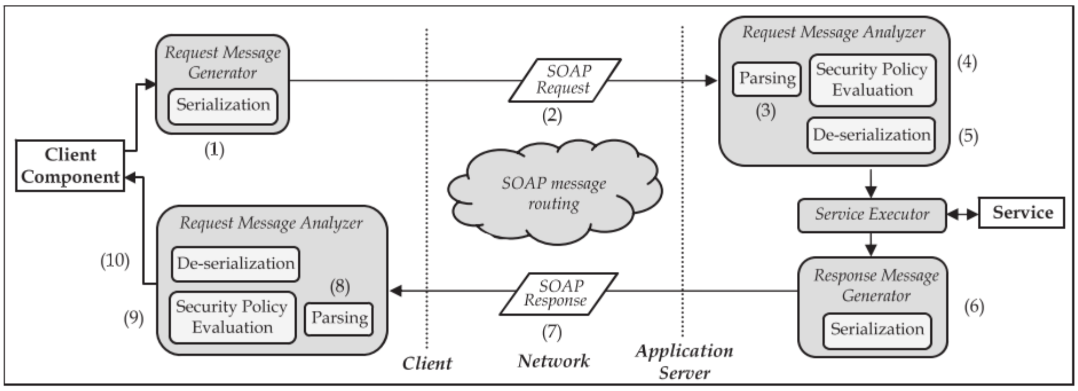

Figure 1: SOAP service call processing. Copied from Tekli et al (2012) [10, p.389].

Li et al [14, p.67] states that Web services meet a number of important purposes, nevertheless not being faultless. The serialization of outgoing messages or overall message processing becomes a performance issue for SOAP services in high-performance requiring applications [9, p.55; 15; 14, p.67]. Besides the verbosity of the protocol increases network traffic [15, p.1; 10, p.387]. Li et al [14, p.67] notes that a problem of message processing performance arises remarkably if the SOAP engine is built in Java. Web services performance is suffering especially because of time-consuming operations with XML [14, p.67]. Abbas et al [16, p.81] claims that XML documents become larger than the same information in other formats because of the repetition of tags.

Studies introduce various ways of dealing with the confronted issues. Abbas et al [16, p.81] propose a dynamic clustering-based aggregation model for XML web messages to increase performance. Both Tekli et al [10, p.391] and Abu-Ghazaleh et al [9, p.56] utilize storing processed messages and using them as templates for transformation of new similar messages, saving time with differential serialization. Other measures include adaptive cache, predeserialized templates [14, p.67], proxy cache [17, p.1614] and measuring similarity using XML trees [13, p.88].

2.2 X-Road Web services

X-Road is an information systems' communication layer – technical and organizational environment – that allows to organize secure and veritable Internet-based data exchange between digital data stores and information systems (public and private) [3]. X-Road Estonia was founded in 2001 to avoid storing sensitive data in centralized database [18]. Information system included into X-Road communication can be any legal entity whose membership application has been approved by Central Authority (CA) [18]. Core of X-Road is made of state institutions registers as seen in figure 2. Top-consumed services by its members also belong to state institutions. For example, in 2015, the services of the Estonian Tax and Customs Board and Population Register were used the most [19].

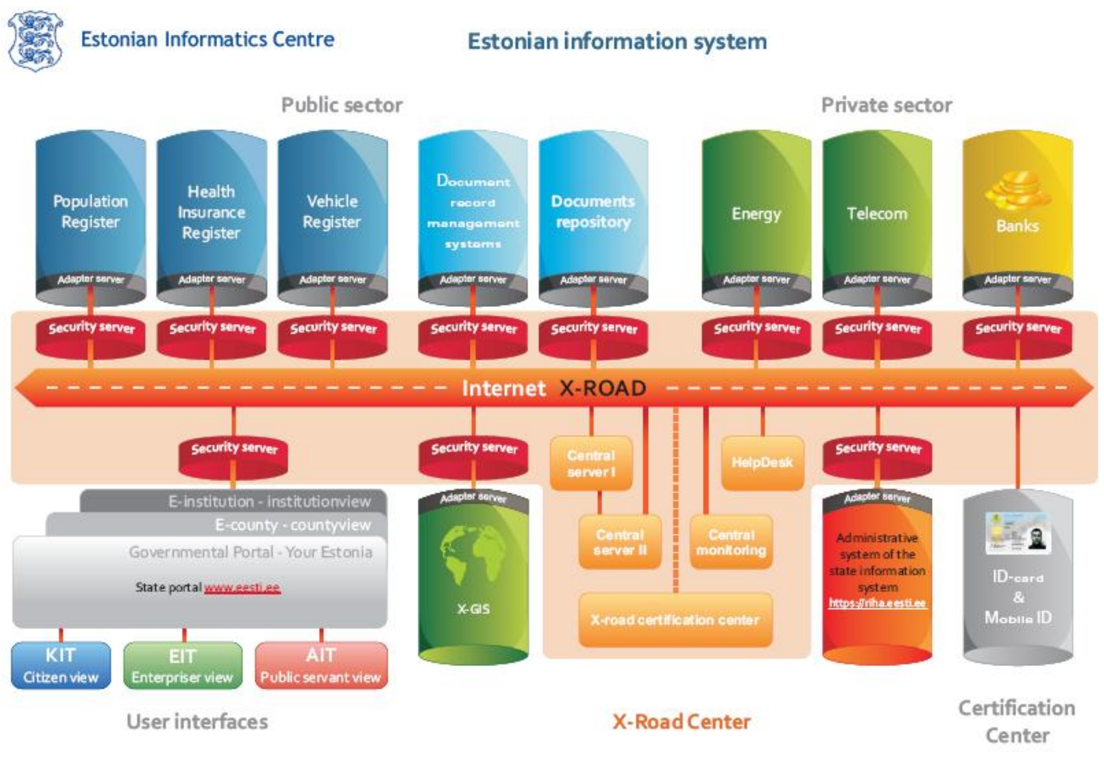

Figure 2: Estonian information system. Copied from Estonian informatics Centre (2011) [20].

During the year 2016, Estonia moves to a new version[10] of X-Road, which means changes in the infrastructure and messaging protocol [21]. Version renewal aims to enable communication with countries within the European Union (EU) using European X-Road. From October 2016 only the new message protocol[11] will be supported, when support for the previously used versions of the protocol[12] ends [21]. As of December 2015, according to the Administration System for the State Information System (RIHA), X-Road is used by 1232 registered organizations and 320 information systems both service providers and consumers [19].

Intercommunication between services in X-Road is based on SOAP and XML Remote Procedure Call (XML-RPC) messages [22] with a new version migrating to only SOAP 1.1 protocol [23]. Using of a standardized protocol enables information transmission between systems written using various technologies through X-Road [3]. Presently, information systems are responsible for composing SOAP requests and sending those to security server, where messages are transformed to be sent to service providers. Service providers respond with data incorporated in SOAP response message [22, p.16]. This message route is illustrated by figure 3. The X-Road message request route leads from the service consumer thought the adapter server (optional), security server on consumer side, X-Road / Internet, security server on provider side and adapter server (optional), to service provider. Consequently, while migrating to the new X-Road messaging version all systems engaged in X-Road communication need to change structures of messages sent and be prepared to receive messages of new composition. Information systems are not ready to use a new protocol without additional changes, because they are designed using static composition [Raivo Tali, Software Architect, 15.02.2016, personal communication].

The main differences between old and new protocols are the language and composition of envelope header attributes, certificates and digital signing of messages in security server, and unified names of messages for request and response [24]. Besides message name changes due to unification, translation of message parts may occur for some services. Additionally, asynchronous calls to Web services, used to prevent blocking other application activity [25], are no longer supported [24].

[10]X-Road version 5 will be changed to 6.
[11]X-Road message protocol version 4.
[12]X-Road message protocol versions 2, 3 and 3.1.

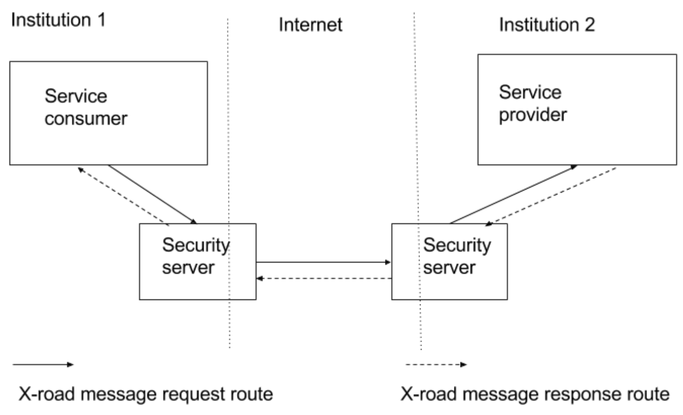

Figure 3: X-Road message route. Adapted from Ministry of Economic Affairs and Communications, State information system department (2007) [22, p.15].

The code in listing 1 shows the old and listing 2 new version of the header blocks of a message on the example of a datastore request fetched from RIHA[13]. The comparison of listings 1 and 2 demonstrates the following changes in the example message header: the `protocolVersion` block is added to new version header, the `id` field's namespace is changed, the `userId` is translated from the Estonian field name `isikukood` and its namespace is altered, the `nimi` (name in Estonian) element is removed, and finally, `andmekogu` and `asutus` simple type elements are substituted with `service` and `client` complex types; besides, child elements of the complex types are introduced.

```
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
    XMLSchema" xmlns:soapenv="http://schemas.xmlsoap.org/soap
    /envelope/" xmlns:xxx="http://***/xxx">
    <soapenv:Header>
        <xxx:id xsi:type="xsd:string">?</xxx:id>
        <xxx:nimi xsi:type="xsd:string">?</xxx:nimi>
        <xxx:isikukood xsi:type="xsd:string">?</xxx:isikukood>
        <xxx:andmekogu xsi:type="xsd:string">?</xxx:andmekogu>
        <xxx:asutus xsi:type="xsd:string">?</xxx:asutus>
    </soapenv:Header>
    ...
</soapenv:Envelope>
```

Listing 1: X-Road message header version 3. Example generated from RIHA WSDL [26].

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/
    soap/envelope/" xmlns:xro="http://x-road.eu/xsd/xroad.xsd
    " xmlns:iden="http://x-road.eu/xsd/identifiers" xmlns:
    prod="http://producer.x-road.eu">
```

[13]"xxx" marks a namespace of the datastore.

```
    <soapenv:Header>
        <xro:protocolVersion>?</xro:protocolVersion>
        <xro:id>?</xro:id>
        <xro:userId>?</xro:userId>
        <xro:service iden:objectType="SERVICE">
            <iden:xRoadInstance>?</iden:xRoadInstance>
            <iden:memberClass>?</iden:memberClass>
            <iden:memberCode>?</iden:memberCode>
            <!--Optional:-->
            <iden:subsystemCode>?</iden:subsystemCode>
            <iden:serviceCode>?</iden:serviceCode>
            <!--Optional:-->
            <iden:serviceVersion>?</iden:serviceVersion>
        </xro:service>
        <xro:client iden:objectType="?">
            <iden:xRoadInstance>?</iden:xRoadInstance>
            <iden:memberClass>?</iden:memberClass>
            <iden:memberCode>?</iden:memberCode>
            <!--Optional:-->
            <iden:subsystemCode>?</iden:subsystemCode>
        </xro:client>
    </soapenv:Header>
    ...
</soapenv:Envelope>
```

Listing 2: X-road message header version 4. Gathered from Annuk et al (2015) [23].

Systems use data from X-Road in various situations; there are three main types of requests:

- real time requests from the information system to fetch data for a form fields;
- real time requests from user to get information from a specific service;
- scheduled requests run by the system to update data in the database with a large set of entries, usually made outside of peek hours. [27]

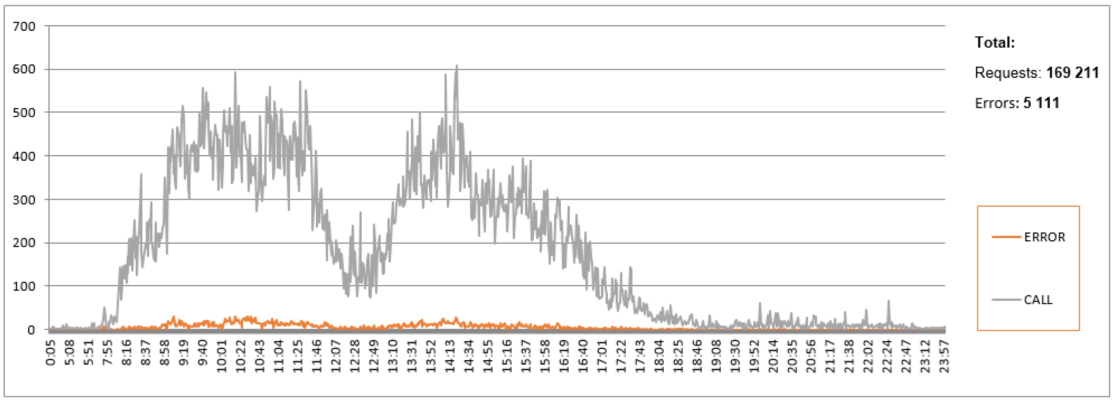

Figure 4: All requests from SIS in December 2014 on a time axis. Copied from Sulina (2015) [27].

Requests to X-Road services are not uniformly divided; the main peeks are during the state institution working hours [27]. These tendencies are visible in figure 4 from an analysis of the logs for State Information System (SIS)[14].

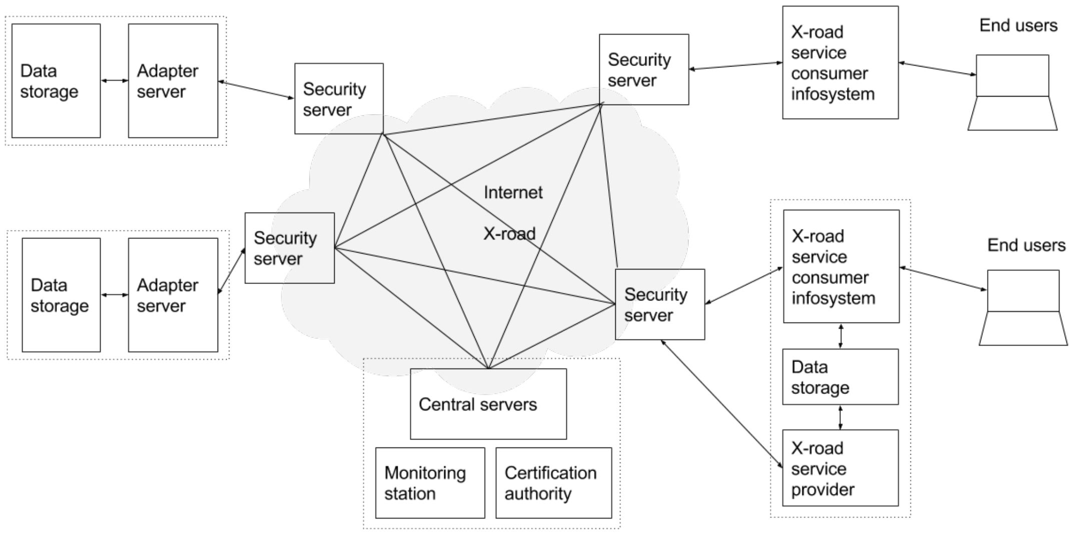

Figure 5: X-Road infrastructure. Adapted from Ministry of Economic Affairs and Communications, State information system department (2007) [22, p.16].

X-Road infrastructure includes security servers – servers with special software, central servers, certification authority[15] and monitoring station [3; 28, §9] as represented in figure 5. X-Road security is provided by security server. According to the Estonian X-Road regulation [28, §11(2)], the service provider information system related components (adapter server, security server) should be placed on the information system intranet. The security server thus transforms messages for crossing security domains. This implementation is conceptually similar to the one proposed by Jeckle et al [29], who describes a possibility of providing security with a single point securing and signing proxy, instead of modifying applications. The security server performs the following tasks: encrypts and decrypts messages, produces logs, controls rights of service requester, prevents unauthorized access of X-Road member service and mediates incoming messages to destination [28, §7].

The adapter-server is optional software that converts a request received from the security server into a form understandable by the information system / data store platform. This server also receives and converts a response for the request to the form required by X-Road, and then mediates it to security server for sending it to the request performer

[14]Examples included in this thesis do not contain real information system / Web service partners names, according to the use agreement with a system owner. Information system will be further referred as SIS.
[15]All X-Road members are certified.

member. [28, §11(4)]

2.3 SOAP message processing

SOAP message processing, as discussed in section 2.1, is an issue when targeting high performance software mainly due to XML processing shortcomings. There are cross-language standards [30, p.137] and specific programming language tools for such message handling, an excerpt of those is reviewed in the following sections.

2.3.1 XML parsing methods

XML is parsed to guarantee a well-formed document[16], to verify that its structure complies with Document Type Definition (DTD) or XML Schema, and to access or modify elements or attributes of the document. Every application intends to produce well-formed XML, possessing the first and, optionally, the second reason listed above. In contrast, accessing or modifying elements is not always necessary. [31, p.33] Taking into account that serialization / deserialization of the message aims for a transfer between XML encapsulated values and application objects, which requires values access and reading, thus, SOAP message processing incorporates the third reason for parsing as well.

Parsing approaches:

- Document Object Model (DOM) parsing approach: complete document model is constructed before access or modification can take place.
- Push parsing approach: a document is processed in parts; encountering predefined parts of the document, push parser produces parsing events, which can be handled by an application using callback handlers, e.g. Simple API for XML (SAX).
- Pull parsing approach: event-based parsing, similarly to push approach. Application controls delivery of the parsing events, e.g. Streaming API for XML (StAX). [31, pp.33-37]

DOM was implemented with an aim of making changes in the XML structure and creating new documents [30, p.126]. The result of XML parsing with the DOM Application Programming Interface (API) is a tree representation of the document in application mem-

[16]Document corresponding to the syntax standard.

ory [32].

SAX differently to DOM is used mostly for read-only document processing [30, p.125]. SAX likewise DOM is a common interface implemented for many different XML parsers [33]. Both are cross-language standard [30, p.137], having their own specific implementations in programming languages.

Table 1: XML Parser API Feature Summary. Copied from Oracle Java documentation [34]

Feature	StAX	SAX	DOM
API Type	Pull, streaming	Push, streaming	In memory tree
Ease of Use	High	Medium	High
XPath Capability	No	No	Yes
CPU and Memory Efficiency	Good	Good	Varies
Forward Only	Yes	Yes	No
Read XML	Yes	Yes	Yes
Write XML	Yes	No	Yes
Create, Read, Update, Delete	No	No	Yes

DOM trees provide maximum flexibility for developers, however consume large amounts of memory and higher processor requirements in case of processing long documents. As stream processors like StAX enable removing already processed elements from memory, developer can only work with one document location at a time. However, in certain situations, working without preserving the whole document tree in the memory increases performance. [34] As seen in table 1 summarizing parsers features, one of the important differences between DOM and StAX is an ability of DOM to update XML structure, which is needed when document tags change.

2.3.2 SOAP message transformation in Java

SOAP messages can be handled in Java as other XML files[17], created with dedicated API classes[18] or processed using Java classes binding[19]. In the last case Java classes are usually generated automatically using WSDL, marshalling and unmarshalling is used for serialization and deserialization [35, ch.4].

[17]For example with Java API for XML Processing (JAXP).

[18]For example javax.xml.soap classes.

[19]XML parsing takes place in each of approaches, being evident or hided by abstraction.

JAXP is one possibility for handling any XML data in the applications implemented in Java programming language. JAXP implements StAX standard[20], it enables parsing both using SAX and DOM API or transforming documents using Extensible Stylesheet Language Transformations (XSLT). JAXP provides namespace support, allowing to work with document structures that might otherwise have naming conflicts. [36]

Examples of automatic tools for generating Java classes code from WSDL include: Java Architecture for XML Binding (JAXB) [37], Java API for XML-Based Web Services (JAX-WS) [38], Apache Axis2 code generator [39], Apache CXF [40] and other. These tools mostly generate classes representing XML with ObjectFactory class, for example JAXB, JAX-WS, which enables to create needed messages by providing message inputs. The indicated approach involves that application has an entity calling creation methods. From one side, if operation message changes, new classes can be regenerated dynamically[21]. From the other side, if inputs for a message creation are also altered, the entity calling renewed classes might fault. Thus, dynamically created classes do not guarantee that the code layers communicating with their operations will function, calling service as before the change. That is why in a large applications common practise is to hold a state of Web service in the WSDL file placed within the application [Raivo Tali, Software Architect, 07.06.2015, personal communication]. Such an approach guaranties that in a case of service changes, with no action for the client application update, only changed operation calls will return fault response, and unchanged operations will function normally. In complicated cases, like the alteration of X-Road protocol version described in this work, all the service operations may change and will produce fault response if automatically generated code is not updated.

Certain applications still do not use automatic tools for generation of the SOAP engine – a layer for communication with Web service. XML files are created manually from strings, or message responses are transformed with Extensible Stylesheet Language (XSL) to HyperText Markup Language (HTML) showing those to a client. This increases code interrelation and complicates changes, because message generation might be scattered within application classes.

[20]From version 1.4.
[21]It is possible to provide link to WSDL and update code automatically.

2.4 SOAP interceptor and intermediary

Transformation of messages could be performed inside an application – changing SOAP processing or plugging in an interceptor – or while a message is travelling to a SOAP destination. Two approaches for retrieving the messages moving in SOAP frameworks are: the SOAP intermediary mechanism, operated as a midpoint proxy, and interception provided by SOAP framework vendors [41, pp.209-210].

Interceptors can provide solutions for cross-cutting concerns like logging, auditing, security [42, ch. Interceptors; 43, p.111]. They allow developers initiate tasks outside of the business logic of the application bound to events of method, invoked when specified event occurs upon the intercepted method [42, ch. Interceptors]. An interceptor – software architecture pattern used to resolve issues in numerous application domains, according to Lin et al [41, pp.209-210], can be used in SOAP Web services to develop flexible portable interceptor mechanism. Currently, interceptors are bound to SOAP engines, which decreases their usability outside of those [41, p.210]. An interceptor is placed within an application, meaning that each application should implement it again for similar changes, as software architectures and frameworks differ. When the interceptor processes SOAP messages within SOAP engine at the client and server sites, the intermediary operates independently of the client application and the server site's Web Service [41, p.210,217].

SOAP intermediaries are additional actors on the route of a message between origin and final destination (between requester and provider) [8, p.91; 44]. While travelling through intermediaries a message is usually processed or transformed. Transformation can be applied for crossing trust domains (security)[22], scalability or providing value added services. Active – undertake additional processing that may modify the outbound message in ways not described in the inbound message, and forwarding intermediaries – process as described in the original message, are classified. [44, ch.SOAP Intermediaries; 45; 46, p.267]

Technically, SOAP intermediary can be developed as a Web service[23] or as a proxy[24]. An intermediary as a Web service, could receive a request instead of destination service, but

[22]For example, X-Road security server.
[23]Using, for example, Spring tools for Web services.
[24]Using Proxy Servlet, or API, for example Membrane Service Proxy [47].

for providing a response use an embedded client calling an original destination. When the response is received by the client, transform it and send to the origin.

Forward proxy is an intermediate server placed between the client and the origin server, requiring client special configuration, functioning similarly to a Web service intermediary. A reverse proxy or gateway, by contrast, is transparent to the client. Thus, no special configuration is necessary. The client makes ordinary requests for content, the reverse proxy forwards request and returns the response content the same way as an origin. [48]

The adapter server, already used in X-Road communication [28, §11(4)], or a proxy pattern [17, p.1614] can be successfully used for transformation of SOAP messages as an intermediary. This thesis aims using the intermediary concept to present a software developed for addressing changes in the SOAP messages while renewing X-Road protocol.

3 Methods and materials

3.1 Context

State information systems of Estonia use X-Road Web services substantially as described in section 2.2. Systems software is not prepared to the renewal of X-Road protocol, which causes changes in all exchanged messages. The aim of this thesis is to test if the problem can be addressed without changing the code of each application. The concept of the SOAP intermediary is used to develop a software making transformation to messages outside requesting application on the route to security server. The outcome of thesis project, named X-Road message adapter, provisional position on X-Road message route is depicted by figure 6. The main challenge of the work is to be able to transform messages from one form to another. For such manipulations input data is needed, which can be produced manually or automatically. As manual data generation is often error prompt, a tool[25] for producing adapter input was implemented among with the intermediary.

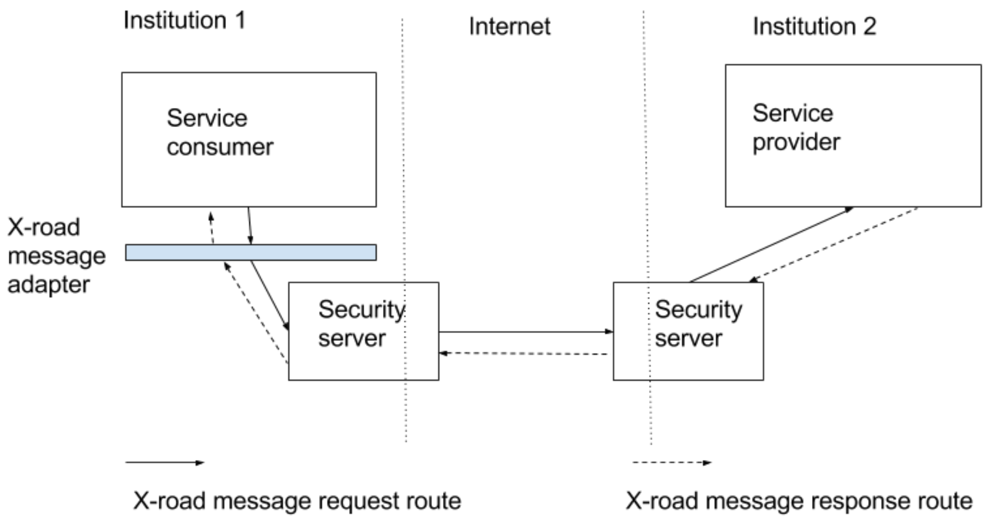

Figure 6: X-Road message adapter inclusion to message route. Adapted from Ministry of Economic Affairs and Communications, State information system department (2007) [22, p.15].

[25]Tool is further referred to as a supporting application.

The thesis project is inspired from work assignments at Tieto Estonia and it uses support provided by the company. All stages of the project, defined in section 3.2 detailed workflow, are carried out by author.

3.2 Methods and project workflow

The research methodology of this thesis similarly to the workflow of the project comprises three stages[26]. Initially, as the outcome of analysis stage, technical possibilities for software implementation[27] and software requirements[28] are determined. Secondly, problems encountered and technical solutions chosen in the development stage are discussed[29]. Finally, produced software is tested according to the requirements, so that a feasibility of proposed intermediary is inspected via a proof of concept[30]. The thesis is a report of the lessons learned [49] while analysing, developing and testing the intermediary and supporting application.

The project is carried out in three main steps: analysis, development and testing. Stages of the thesis workflow and main procedures of each phase are listed below.

1. Planning and analysis:
   - Preliminary analysis of needs: informal discussions with colleagues at Tieto Estonia, media articles and mailing list of changes in X-Road;
   - Developing an initial application idea of SOAP intermediary;
   - Defining software requirements;
   - Consideration of possible intermediary architectural implementation alternatives: interceptor, proxy;
   - Analysing possibilities for organizing inputs, review of WSDL comparison tools.
2. Development:
   - Developing the supporting application and intermediary.
3. Testing:
   - Testing the supporting application using test data;
   - Testing the intermediary using test and information system data;

[26] Analogous research approach was used, for example, by Lin et al [41].
[27] Technical possibilities for software implementation are described in section 2.
[28] Software requirements are outlined by section 3.3.
[29] See section 5 for details.
[30] Analysed by sections 4 and 5.

- Comparison of the testing results with requirements.

Stages of the project are not to be considered carried out strictly sequentially. Since a need for additional analysis occasionally arise on the development stage, besides, methods of the application are tested simultaneously by unit tests.

3.3 Software requirements

This section includes functional and non-functional[31] requirements defined for developed applications in order to provide criteria for the thesis project product evaluation. The requirements also serve as a basis for software design and testing [51, p.30].

3.3.1 Functional requirements

Supporting application should:

R1. generate properties files displaying changes in provided WSDLs readable to the intermediary application.

Intermediary should:

R1. receive SOAP request messages from a configured application;

R2. be able to recognize request messages needing transformation;

R3. transform request messages in the old format according to the changes specified in properties files;

R4. forward all received request messages to a destination (through security server), not depending on the transformation need and success;

R5. receive SOAP response messages sent back from the destination;

R6. be able to recognize response messages needing transformation;

R7. transform response messages in the new format back to old if the request was previously transformed, according to the changes specified in properties files;

R8. forward all received response messages to the configured application, not depending on the transformation need and success.

[31]According to Bode et al [50, p.1] non-functional requirements are quality properties of the software.

3.3.2 Nonfunctional requirements

Nonfunctional requirements for intermediary application:

- Performance requirements: Speed of processing a message inside the intermediary should be under 1 second. Multiple message processing should be supported.
- Usability and configurability requirement: The application enables set up on a server and custom configuration.
- Reliability requirements: Fault tolerance – communication is not broken if some message encounters faults in processing. Minimum impact on the normal operations of Web services [41].
- Extensibility: Further changes in Web services can be addressed by changing configuration.
- Portability: The application enables transfer to other systems.
- Logging of faults and processed data.

3.4 Development environment

Software described in this thesis was developed using Java 1.8 programming language. Tools and APIs used:

- Eclipse Integrated Development Environment (IDE);
- Apache Maven[32];
- JUnit 4.12[33];
- Java API for Membrane Service Proxy 4.1;
- JAXP API;
- Java API for Membrane SOA Model 1.4 [54].

3.5 Test arrangements

Preliminary, key methods of both the intermediary and supporting applications are tested with JUnit tests present in source code[34]. Subsequently described tests are performed

[32]Apache Maven is a software tool for building and managing any Java-based project, also referred to as Maven [52].

[33]JUnit is a framework for writing repeatable unit tests [53].

[34]Accessible from https://github.com/julia-sulina/b-project/tree/master/intermediary/src/test and https://github.com/julia-sulina/b-project/tree/master/wsdlproperies/src/test

running applications in the developing environment and using additional tools. Test results are analysed by comparing testing tools' and applications' logs with expected results. Testing environment details for both the intermediary and supporting application approbation are described in appendix 3.

As requirements of the intermediary application include a possibility of setting it up on a server, initially, an opportunity of server testing was considered. This alternative rendered unreasonable, because X-Road services would be inaccessible for the test server without a security server. At the time of application testing, X-Road has not yet migrated to a new version. For this reason, there were hardly any security servers or test services ready to receive translated calls. Besides, even if technically feasible, testing actions would involve special permissions from X-Road members / authorities. As a result, in the scope of this project, was decided to test the intermediary without using network on a local computer. At the same time, recognising that test results will differ from ones potentially produced on a server, e.g. missing network latency. However, quite fair feedback about the application functionality can still be achieved.

Three different test arrangements are made for testing the intermediary as described below:

1. Accuracy testing – testing the intermediary using test messages[35] when SOAP requester and provider are both simulated with SoapUI 5.2.1[36] with an aim of approbation of the transformation algorithms. Test configuration for SoapUI and intermediary are presented in appendix 4 (SoapUI test) and source code.
2. Performance testing with Grinder application[37] for simulating high load on the intermediary system, using SoapUI service mocking for answering requests. Grinder configuration properties, test scripts and requests can be found in appendix 4 (Grinder test) and source code.
3. Testing the intermediary with a real information system application generating requests, SoapUI tool responding, to access X-Road message processing. Test arrangements configuration is included in source code[38].

Approbation of the supporting application using test data is carried out by running it with

[35]Test message example is displayed in listing 9.
[36]SoapUI is a testing solution available from https://www.soapui.org/
[37]Grinder is a load testing framework available from http://grinder.sourceforge.net/
[38]Available from https://github.com/julia-sulina/b-project-private/tree/master/test/intermediary/application_test

certain input WSDLs and comparing an output with expected result. Input data is initially fetched from RIHA[39]. As WSDLs of new X-Road version have not yet been published as of March 2016, the application is tested on initial shortened WSDLs and WSDLs transformed according to the version changes documentation [23]. Supporting application is expected to be used on a local computer, that is why tests are run in the development environment. For testing, the application is configured to fetch files available on local computer[40]. Input files used for testing are available from source code[41]. During a first test input of one pair of WSDLs is provided, when on a second test the application is run with two pair of WSDL files. Such tests evaluate application functioning with different amount of files input.

[39]https://riha.eesti.ee/riha/main/

[40]Configuration file accessable from https://github.com/julia-sulina/b-project/tree/master/test/wsdlproperties/configuration

[41]https://github.com/julia-sulina/b-project/tree/master/test/wsdlproperties

4 Results

The thesis project resulted in producing software for intercepting X-Road messages, implemented in two separate applications. Main application or SOAP intermediary is meant to be placed on a server within message route and perform actual transformation. Supporting application is to be run before configuring the intermediary for providing input data. Main features and design of each application are briefly described in following sections.

SOAP intermediary and supporting application were tested independently. Intermediary tests were performed to measure application accuracy, performance and ability to process data from an information system. So that different configuration setups were tried out during the tests. In contrast, the supporting application was only tested for capability of expected data production, as performance is not crucial quality for this software purpose.

4.1 Description of SOAP intermediary

SOAP intermediary[42] software is designed to be run as regular Java Application, requires Java Development Kit (JDK) 8, configuration file[43] and translation input data in properties files, specified in section 4.2. The intermediary contains of two main functional components: proxy setup and translation handling, as depicted by figure 7.

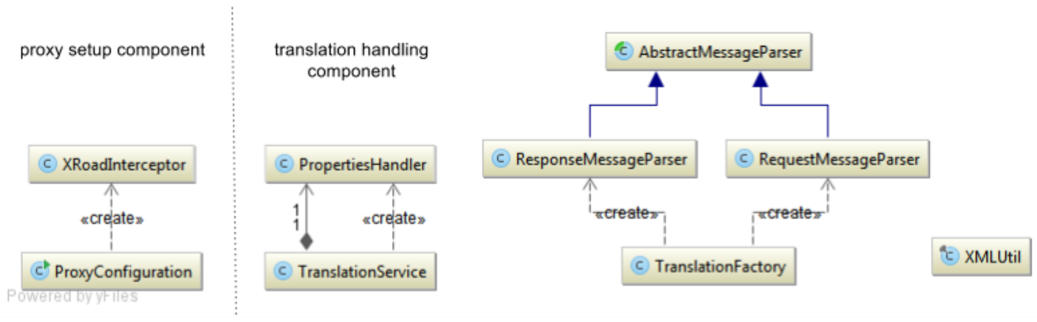

Figure 7: Components and main classes of intermediary application. Generated using Intellij Idea native Unified Modeling Language (UML) diagram tool.

[42]Documentation of SOAP intermediary Java project available from http://users.metropolia.fi/~juliasu/intermediary/doc/, source code accessible from https://github.com/julia-sulina/b-project/tree/master/intermediary

[43]Configuration file default name – xroadProxy.properties, contents described in appendix 1.

4.1.1 Proxy setup component

The proxy setup component is composed from `ProxyConfiguration` and `XRoadInterceptor` classes, see figure 7. Classes implement service proxy and interceptor using Java API for Membrane Service Proxy 4.1[44].

Regular Membrane Service Proxy is a reverse HTTP proxy [55], which setup contains:

- rule matching – selection of service proxy by matching filter criteria to incoming HTTP request.
- dispatching – setting exchange destination for the selected service proxy target.
- user feature – execution of interceptors configured for proxy by user.
- HTTP client – forwarding the request to the exchange destination. [56]

The intermediary is a service proxy configured in a way that rule matching filter will forward suitable incoming requests to it. Rule matching filter criteria are: incoming Transmission Control Protocol (TCP) port, local Internet Protocol (IP) address, hostname – the HTTP request Host header, and request path, matched by a path prefix or a Java regular expression [57]. These criteria as well as dispatching target are configurable for designed intermediary, using properties file attached in appendix 1. The only statically predefined parameter is a request type configured to HTTP POST, because the intermediary is designed to handle only SOAP messages.

`ProxyConfiguration` class loads a configuration[45] from properties file and setups Membrane service proxy[46]. Custom `XRoadInterceptor` is added to the proxy as a user feature, thus all proxy requests will be processed before forwarding. An interceptor receives a message of request and response encapsulated in `Exchange`[47] object. To recognize SOAP from other accidental messages the interceptor uses API's `SoapOperationExtractor`[48] class. Interceptor is in charge of capturing proxied messages, evaluating the need and way of translation, calling translation handling component for translation performing[49], as well as forwarding messages back to the proxy. Messages are then dispatched to the destination.

[44] API jar file is added to the classpath of a project.

[45] Configuration for rule matching includes hostname, path and port to listen, and dispatching – target host and port.

[46] `com.predic8.membrane.core.rules.ServiceProxy`

[47] `com.predic8.membrane.core.exchange.Exchange`

[48] `com.predic8.membrane.core.interceptor.soap.SoapOperationExtractor`

[49] See section 4.1.3 for details.

4.1.2 Translation handling component

The translation handling component includes translation service and factory, parsers, properties and XML handling utility classes, additionally, supportive classes as a custom exception and constants. `TranslationService` class responds for performing message translation. On construction it loads a translation component configuration[50]. According to the configuration, properties files handlers[51] are created. When translation is requested from the translation service, it turns to `TranslationFactory` class, which produces a new instance of request[52] or response parser[53] according to the current need, this way factory pattern design is used. Response and request parser share several fields and methods by inheriting `AbstractMessageParser` class, this relation is visible in figure 7. Parsers implement method translate, which executes translation routine according to a suitable algorithm.

Translation is performed with a help of JAXP API DOM parsing, which enables to update message structures dynamically. `XMLUtil` class is adapted from Apache Software Foundation to simplify work of XML objects building. Custom exception class[54] is introduced and exception handling is used to improve software reliability. When possible, translations are loaded straight from properties handlers to document structure, in complex cases, e.g. request header translation, a list of objects for transformation[55] is employed.

4.1.3 Translation algorithms

The translation need is determined as illustrated in figure 8. Firstly, the service controls message operation name presence in a list of operations that should not be translated (ignore list). On condition that operation name is found in this list, message translation is skipped and a forwarding case algorithm used. Otherwise, the next step control is executed. Secondly, the service searches message contents for a new tag name set in proxy configurations (appendix 1) to decide on message belonging to the old or new version format. On this step, an action of the program depends on the message type being

[50]Configuration contents are listed in appendix 1.
[51]`PropertiesHandler` class instances.
[52]`RequestMessageParser` class.
[53]`ResponseMessageParser` class.
[54]`MessageParsingException`
[55]`MessageTag` class instances.

request or response. If the message presently processed is a request and the tag is found in its contents, the message will be regarded suitable for the forwarding case algorithm, else the message will be handed to the next control. If the message is a response, the program acts in the opposite way. Thirdly, the list of operations needing translation for body tags is examined. In a case that this list contains operation of current message, the complex case algorithm is applied, if not, then the simple case algorithm is used for transformation.

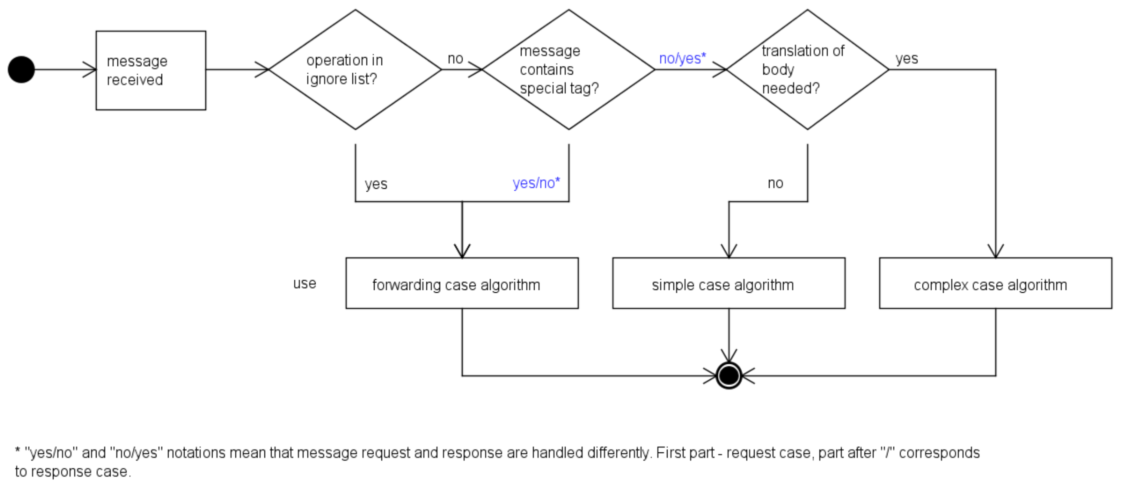
message received
operation in ignore list?
no
message contains special tag?
no/yes*
translation of body needed?
yes
yes
yes/no*
no
use
forwarding case algorithm
simple case algorithm
complex case algorithm
* "yes/no" and "no/yes" notations mean that message request and response are handled differently. First part - request case, part after "/" corresponds to response case.

Figure 8: Translation algorithm determination

Forwarding case algorithm: nothing needs to be translated.

1. Request / response procedures:
   - forward.

Simple case algorithm: only message header elements require translation.

1. Request procedure:
   - save message header to `Exchange` object property for use in response parsing;
   - translate header:
     - load prepared default header from provided file or cache;
     - obtain values from message header tags;
     - find corresponding values for new header elements and insert them to default header.
   - compose a new message, merging new header with an old body;
   - forward.
2. Response procedure:
   - restore header:
     - get saved message header from `Exchange` object property;

    - replace header.
- forward.

Complex case algorithm: message header and body should be translated both ways

1. Request procedure:
    - save message header (like in the simple case);
    - translate header (like in the simple case);
    - compose a new message (like in the simple case);
    - translate body replacing tag names with new names in the merged message;
    - forward.
2. Response procedure:
    - restore header (like in the simple case);
    - translate body (opposite to the request);
    - forward.

4.2 Description of the supporting application

Supporting application[56] generates properties files for SOAP intermediary comparing different versions WSDLs of services. The program is run as a regular Java Application, requires Java JDK 8, configuration file[57] and WSDL files for comparison.

Each WSDL is converted to Java objects using API for Membrane SOA Model WSDL-Parser class[58], which generates hierarchical model structure containing all elements of WSDL[59]. WSDLs are paired and their structures are compared for changes by generic class. Firstly, objects that were not changed are found and removed from further analyses. Unchanged objects are defined as objects which have the same type, name and position in the hierarchy of both WSDLs. Secondly, remaining objects are matched in pairs using following criteria:

- Positions in the hierarchy match – elements that are children of the same parent are compared, e.g. parts of the same message.
- Equal child objects present in the different parent nodes, e.g. in a case of message name change, same parts containing in two messages enable to pair them.

[56]Documentation of supporting application Java project available from http://users.metropolia.fi/~juliasu/wsdlproperties/doc/, source code accessible from https://github.com/julia-sulina/b-project/tree/master/wsdlproperies

[57]Configuration file default name – wsdl.properties, contents – appendix 2.

[58]Available from http://www.membrane-soa.org/soa-model-doc/1.4/java-api

[59]See list in section 2.1.

- Same node attributes values – two objects on the same level of hierarchy tree, being children of the same object can be paired according to containing same attributes e.g. type.
- No potential pairing node is present – if a node is present in the hierarchy of one document and missing in the hierarchy of other, the node is regarded as deleted or added, depending on the position in new or old WSDLs, i.e. paired with nothing (null)[60].

Some WSDL elements, however, cannot be paired according to any of named criteria. For this reason user can provide properties list of known translations, which will be consulted during the pairing process[61].

When changed objects are found, a list of translations containing their descriptions is composed. From this list, properties files are generated. While generation, a position of the element is also marked. Property entry structure is following: `position.old_name = new_name_with_namespace`. Position for header entries is marked as `header` and for body elements, operation name is used.

Functionality of application classes can be summarized as following:

- `ComparationService` – initiates comparison process;
- `Comparator` – handles comparison routine;
- `FilesHandler` – reads WSDLs and outputs property files;
- `PairGenerator` – pairs any elements of the same type according to criteria;
- `Tuple` – generic class representing elements pairs;
- `ChangedItem` – contains position, old and new name of an element;
- `Mapper` – maps from Tuple to ChangedItem.

Generic classes, `Function` and `Predicate` interfaces are used in order to reuse functionality for different objects e.g. `PairGenerator` class. Lists are processed using streams and lambda expressions.

[60] In case that compulsory node is added user should provide its value, optional nodes are skipped.

[61] Reconsidering as an example headers of listings 1 and 2 user might provide following translations for the tags: `isikukood=userId`, `andmekogu=service`, `asutus=client`, see section 2.2 for detailed analysis of these headers change.

4.3 Test results of SOAP intermediary

4.3.1 Accuracy testing

SoapUI tool tests were run to try all translation algorithms[62] functioning accuracy. Description and results of these tests are reported in table 2. Test 1 and 2 were designed to evaluate the forwarding algorithm, while test 3 for the simple case algorithm and test 4 for the complex case algorithm. Description of test shows expected results, actual results were gathered from SoapUI tool and intermediary application logs[63].

Table 2: Intermediary application tests using SoapUI tool

	Test	Description	Result	Errors
1	New message test	New protocol message sent to intermediary, should be analysed and forwarded without processing.	Message forwarded in both directions.	-
2	Ignore list test	Old protocol message sent to intermediary with operation name, marked in ignore list. Nevertheless, operation uses old protocol, intermediary should forward it without processing.	Message forwarded in both directions.	-
3	Header translation test	Old protocol message sent to intermediary, message header should be translated.	Header translated in both directions.	-
4	Header and body translation test	Old protocol message sent to intermediary, operation name marked in configuration as message requiring body translation. Message header and body should be translated.	Header and body translated in both directions.	-

Table 2 shows that test results corresponded to expected outcome. Besides, all four tests run were finished without errors.

[62]See section 4.1.3.

[63]Available from source code https://github.com/julia-sulina/b-project/tree/master/test/intermediary/soapui_test

4.3.2 Performance testing

Grinder tool tests results presented in table 3 show the intermediary application performance while multiple requests simultaneous processing. For three tests, the tool was configured to send same requests from 10 processes with 10 threads each, so that 100 messages were sent to the application almost simultaneously. For the next tests, processes amount was doubled.

Mean time in table 3 shows an average from sending a request to receiving response by Grinder tool. Between those events, message first of all travelled through the intermediary application, being translated or not according to the test setup. Then, reached SoapUI tool, which mocked a Web service and immediately sent a response back to the requester. The response again was received in the intermediary application, which processed and forwarded it back to Grinder.

Table 3: Tests using Gringer tool

Test	Successful Tests	Errors	Mean Time, ms	Mean Time Standard Deviation, ms	Mean Response Length, bytes
Test 1	100	0	66,4	47,9	6140
Test 2	100	0	110	74,4	5750
Test 3	100	0	229	209	5710
Test 1	200	0	133	133	6140
Test 2	200	0	186	171	5750
Test 3	200	0	391	302	5700

The test 1 results show the application performance while forwarding requests without transformation. While test 2, messages header was transformed in both directions. In test 3 message header and body were transformed. During the tests there was no network latency or other slowing conditions e.g. long request processing in a target Web service. Test results show that the application can handle several simultaneous requests without fault. Logs of Grinder, SoapUI tool and intermediary application show that all messages during the tests were processed in expected way[64]. In real life conditions the application

[64]Available from source code https://github.com/julia-sulina/b-project/tree/master/test/intermediary/grinder_test

may work slower due to bigger properties file contents searching, but probably would not have such large load as in the tests[65]. From the test results can be observed, that larger load makes operations longer. However, presented data does not allow to make statistical analysis of load influence on mean time variation, it still displays a difference between the tests. Test 1 mean time grows linearly i.e. time doubles with doubled messages amount, while Test 2 and 3, taking more time with initial load, increment less with doubled amount of messages.

4.3.3 Information system data testing

The intermediary application was configured to receive requests of a state information system to test real life X-Road message processing. The test setup was following: SoapUI was configured to imitate a Web service and the information system sent requests. The intermediary was tested on 9 different requests[66]. All requests were expected to undergo header translation.

On the first test, from 9 requests 4 failed translation and 5 were successfully translated. Failed requests were forwarded to service without alteration. An error source was easily discovered. The intermediary application expected a message containing envelope, header and body tags with `soapenv` namespace e.g. `<soapenv:Envelope>`. The information system sent out requests with tags namespace `ns2` e.g. `<ns2:Envelope>`.

After errors discovery in the first test, the intermediary application was fixed by adding a method, which controls and unifies namespaces of the main SOAP tags. Only then the intermediary performs actual translation operations. After this fix, same 9 operation requests tests were run. Resulting in one `NullPointerException`[67] and one fault[68] from SoapUI tool, which cause is unknown. Headers of all other operation requests were translated.

Limitation of presented results is that the intermediary is not tested with a real service provider, only with the consumer information system. SoapUI tool is not performing same

[65]At the busiest time information system produce 30-40 requests in a minute [27].

[66]Not all possible requests of the system were used, ones easily performed from user interface were preferred.

[67]Exception was fixed after the test.

[68]Message was not recognized by SoapUI tool, but appeared well-formed.

operations on messages like X-Road service. The first test showed that the application was initially not fully adjusted to real life data. Before being able to run it in the production environment, the application should be tested on a server with X-Road service provider.

4.4 Test results of the supporting application

Supporting application was tested by an input of one WSDL pair and two WSDL files pairs. When the first test was run for determining changed elements pairing capability of the application, then another aimed approbation of multiple inputs. A limited amount of tests does not guarantee supporting application functioning faultlessly in all possible cases. Nevertheless run tests showed software ability to prepare properties files usable by the intermediary in both tests. To conclude, the supporting application produced expected results during the tests[69].

[69]Configuration and results of the tests available in source code https://github.com/julia-sulina/b-project/tree/master/test/wsdlproperties

5 Discussion

5.1 Technical decisions

Technical questions arose at each stage of the project. Initially, several principal decisions were unavoidable. For example, a source from which changes in Web services could be determined needed to be selected. As X-Road services, like other SOAP services, use WSDLs, it was relevant to work with those for change analysis. The other possibility was to map alterations from SOAP messages, but in such case, the messages would need to be additionally generated from WSDL with some tool, and only then be compared. WSDLs are not usually versioned [8, pp.79-80], but in the case of X-Road, WSDLs are stored in one service (RIHA). Besides, old versions are commonly documented in the development of the requester or presented within an application that makes them available.

When WSDLs are fetched, the next issue will be faced: how changes could be discovered, collected and represented in a systematic way. Review of WSDLs comparison tools showed that there are hardly any ready-made tool for performing such processing. One instrument found, was the Membrane SOA Model comparison tool as a command-line application and Java API[70]. This tool, however, is quite inflexible in output, producing predefined strings, and does not allow extension e.g. providing of known changes. Thus supporting application development idea was unfold. After evaluating different possibilities, I came to the conclusion that WSDL processing should be performed only once, producing results which could be utilised any time. In this way the message translation itself can be made quicker using data from previously prepared changes.

Consequently, the Membrane SOA Model API was only used for deserialization of WSDLs. Several difficulties were encountered when working with WSDL converted to an object, because its hierarchy is a multi-branch tree, and API documentation of tree representation in the object was partly missing. It took several tries and fails to clarify how to extract the needed elements out of the tree for further comparison. Theory about the

[70]Available from http://www.membrane-soa.org/soa-model-doc/1.4/java-api/compare-wsdl-java-api.htm

WSDL structure[71] as described in section 2.1 helped to focus only on the parts needed for messages.

Provided that changes in service occurred, capturing the right translation becomes the next challenge. The structure and hierarchy of the components could serve for that purpose, but not all alterations can be paired in this way. As a result a possibility to provide known translations was introduced. Detailed results for the translations pairing criteria were presented in section 4.2.

Another consideration was, what format of output should be prepared. As translation is a pair of values – old and new – the format of the properties file was considered suitable for the supporting application output. Properties files are commonly used in Java for configuration or localization, and contain entries of the key and value. Since searching in such files is easily performed, they are acceptable as input data as well. A properties file is also readable for the developer[72], who can control if the input was generated fully and maintain it.

The most significant decision in the intermediary application was architectural. Possibilities for implementation of the intermediary tried and evaluated during the thesis project were: Proxy Servlet[73]; Interceptor and Filter patterns, plugged in to SOAP engine or together with separate Servlet; Spring tools for Web services – Spring service and client in one application; Membrane API. Finally, Membrane API was chosen, as it enabled building a reverse proxy with a custom interceptor in several lines of code as seen in listing 3. Furthermore, this API provided additional support classes for SOAP[74]. The reverse proxy was evaluated as the best solution over other choices, because it does not require additional configuration outside it.

[71]See list in section 2.1.
[72]Can be opened as a regular text file.
[73]Possible implementations available from

- https://github.com/mitre/HTTP-Proxy-Servlet
- http://edwardstx.net/2010/06/http-proxy-servlet/
- http://noodle.tigris.org/
- https://sourceforge.net/projects/j2ep/

[74]For example message decoding with Message class getBodyAsStringDecoded() method, which helps to avoid message wrong encoding.

```java
public static void main(String[] args) throws Exception {
    loadConfiguration();

    String hostname = confHostname;
    String method = POST;
    String path = confPath;
    int listenPort = confListenPort;

    ServiceProxyKey key = new ServiceProxyKey(hostname,
       method, path, listenPort);

    String targetHost = confTargetHost;
    int targetPort = confTargetPort;
    ServiceProxy proxy = new ServiceProxy(key, targetHost,
       targetPort);

    XRoadInterceptor interceptor = new XRoadInterceptor();
    proxy.getInterceptors().add(interceptor);
    HttpRouter router = new HttpRouter();
    router.add(proxy);
    router.init();
  }
```

Listing 3: ProxyConfiguration class main method

Not everything worked out completely at the start of the Membrane project API usage. Documentation for this Java API is scarce, because commonly the project is run as a standalone application configured with XML, without custom classes. Initially, it was unclear if the whole Membrane project was needed and how to make its methods available from outside. One of the obstacles was that the documented version API's artifact was missing from the Maven Central Repository. Therefore it could not be included as a regular Maven dependency to the project. Other possibilities were tried, e.g. including Java Archive (JAR) files manually or trying to run a Web archive of the project, until it was discovered that some newer versions were available for Maven. At the same time possibilities of intermediary application deployment were evaluated and a regular Java Application was chosen.

Choosing the way of working with X-Road messages while performing translation was another decision milestone of the project. Generation of Java classes from WSDL, using automatic tools discussed previously[75], as regularly when building Web services e.g. with Spring, was perceived as a promising opportunity. The initial idea was to generate classes and an object factory from old and new WSDLs. When translating, an operation

[75] See section 2.3.2.

(method) has to be chosen according to the current message. However, an issue with such a design is that even if the operation name stays the same[76] and helps to match old and new operation classes, the class fields would not be corresponding whenever message elements change. The problem arises when mapping one object values to another, which requires implementation of special mappers, in the worst case for every translation. Finally, the JAXP DOM parser was preferred for work with SOAP messages, as it provides a lot of flexibility for processing. Messages are not that large in X-Road to become a performance issue.

The next problem faced was to evaluate if a message needs translation or not. The message translation decision need was presented in figure 8. The first question is how to recognize if the encountered message is constructed according to the new or old version of the protocol. Luckily, only the new version of the header contains tag `<protocolVersion>` with value 4, which makes the decision straightforward. Theoretically, all the messages of the old version need translation of the header; however only some need translation of the body elements. In such a case there should be different approaches for translations, as shown in transformation algorithms in section 4.1.3. One properties file has the names of operations which need body tags translation. There is also an ignore list for messages that should not be translated. The ignore list is included with consideration of the transition process. There can be a time when some of the service providers are still able to consume only old protocol messages, and others already migrate. Then the ignore list can be updated according to timely needs.

Handling of adding and deleting message elements is likewise problematic. If a new message has the acquired fields, which are not compulsory, those may be skipped, on condition that the field value cannot be generated or provided by the user. In case that a new request has additional compulsory fields, there is a problem. A new field value should be provided by the user and the program needs to signalize it, because in the opposite case the generated request will fail. If the old message has fields no more needed, those should be deleted. Whenever there are changes in the field names, the translator should handle them.

Some translations may add complications. An example of such a situation is header trans-

[76]If operation name changed, additional matching logic need to be implemented.

lation discussed previously in this work in section 2. As can be seen by comparing listings 1 and 2, the simple type element was changed to a complex type. The difficulty is providing values of complex type for all compulsory children fields. The solution used in this work is that a special values properties file should be composed[77]. For example, if a tag `asutus` displayed in listing 4 should be transformed to a tag `client` with sub-tags `xRoadInstance` , `memberClass`, `memberCode` as shown in listing 5, the value list configuration similar to listing 6 would be employed. Entries with a structure `old_value.new_name = new_value` are used in cases when the new value depends on the old one, and structure `default.new_name = new_value`, when translated messages will share the new values[78]. This file cannot be created fully automatically, only a template will be generated, as values are not known from WSDL and can be different for many messages.

```
<xxx:asutus xsi:type="xsd:string">123456</xxx:asutus>
```
Listing 4: Example of transformation input element.

```
    <xro:client iden:objectType="MEMBER">
         <iden:xRoadInstance>EE</iden:xRoadInstance>
         <iden:memberClass>BUSINESS</iden:memberClass>
         <iden:memberCode>123456</iden:memberCode>
     </xro:client>
```
Listing 5: Example of transformation expected output element.

```
default.client.objectType=MEMBER
default.client.xRoadInstance=EE
123456.client.memberClass=BUSINESS
123456.client.memberCode=123456
```
Listing 6: Example of transformation value list configuration.

Searching in properties file, if the same tag name should be translated for one operation, and not for another may become a complication. That is the reason why the position of the tag as the operation name or header is included in the property entry. For instance, searching translation with values for the header is performed in two stages. First, translation is looked for by searching value of `header.old_name` in translation properties file; if such a value is not found, the item of header will not require translation. In case that an element requires translation, the values list will be examined, searching sequentially for `new_name`, `old_value.new_name`, `default.new_name` if none of these are found, the value of element stays the same; otherwise it gets the first found value. Body translations are searched by `operation_name.old_name`, and if not found, they will not be translated.

[77]Structure of an entry in this file is `old_value.new_name = new_value` or `default.new_name = new_value`.

[78]For example all messages from Estonia will share `xRoadInstance` tag with value `EE`.

The problem with SOAP message translation is that when a request is translated in one way, the response should be translated in the opposite way. Particularly, the task involves dealing with bidirectional translation. Firstly, an idea was to implement a powerful parser, which would handle transformation in both directions. However, after analysing the needed transformations, it was clear that converting the request and response being similar do not follow the same algorithm routine[79]. Under those circumstances the factory pattern and abstract parser were developed as specified in section 4.1.2.

The header bidirectional translation was solved by preserving the old header in the `Exchange` object[80], and reusing it in the translating response. Another assessed possibility was to add a header of request to `HashMap` with id. On response, search the header by id, restore it and remove the found object from the `HashMap`. However, `Exchange` object was preferred, because it enabled a simple property setting for each message exchange, making searching unnecessary. For the body part, no better solution than adding both ways translations to the properties file was found.

Reusing the header was inspired from the template parsing examples used in several studies discussed previously [9; 10; 14]. Also the message cache [17] was evaluated as a possibility to make transformations faster saving message responses. This means that encountering the same request, the application is able to answer it without communicating to the service provider from the previously saved data, similarly to the proxy cache solution of Kungas et al [17, p.1614]. However, as the same X-Road requests rarely repeat, this possibility will hardly add any processing speed. The producer-consumer pattern was also examined as an opportunity for speed enhancement, but was considered unsuitable for the Membrane API singleton interceptor bean.

Adding serialisation / deserialisation overhead to SOAP processing is clearly one of the weaknesses of the project. This problem in software implementation originates from slow XML parsing operations, as discussed in several studies [14; 16]. Limited resources for carried out the project did not allow to take many measures for performance increasing. One measure was to prefer a document structure copying over reading it from an XML file. The original document was held in a map. Withal, if reading from a file is more costly operation than copying the DOM structure in all cases, is unproven. Another issue with

[79]Translation algorithms are fully described in section 4.1.3.
[80]Membrane API `com.predic8.membrane.core.exchange.Exchange` class.

XML parsing is that the JAXP DOM builder is not thread-safe. Multi-threading issues were solved with using the XML utilities component while DOM building[81]. Even if the application adds extra time to message processing, this might be a less costly solution than changing each application. Change of an existing information system application requires resources for development and testing, as alteration in the SOAP engine can indirectly influence other parts of the application.

During the work on this project some problems appeared more complicated than they actually were, because of my lack of experience. That is why some code produced was later erased. For example, when it became obvious that namespaces could be handled automatically with JAXP [36], like `Node node = (Node)document.importNode(messageBody , true);`, the manual adding of elements namespaces was already written.

During this project I developed my skills and knowledge about handling XML and properties files, proxy configuration, interceptors and other project related fields. For example, I learned that properties can be loaded using properties object or resource bundle. The difference here is that when searching for property in a resource bundle exception will be thrown if the property is not present, this does not happen if to look for such value using properties object.

5.2 Accordance to requirements

A supporting application is developed in accordance to the functional requirement, as it produces properties files, which can be used as an input for an intermediary application. A limitation of the supporting application is incapability of producing values for new tags which need to be provided by the user.

The functional requirements of the intermediary application are also satisfied[82]. Results of accuracy testing showed that intermediary software received requests from the configured application (R1), was able to recognize requests requiring translation (R2) and transform request messages according to the changes specified (R3). The first test of in-

[81]Adapted from Apache Software Foundation https://svn.apache.org/repos/asf/shindig/branches/1.0.x/java/common/src/main/java/org/apache/shindig/common/xml/XmlUtil.java

[82]Intermediary functional requirements, as defined in section 3.3, are referred in this paragraph by R with requirement number.

formation system data showed that even when translation of message failed, the request and response were still forwarded to the destination as in successful accuracy tests (R4 and R8). Accuracy test logs confirm that a response message was also received by the intermediary (R5), a need for processing a response was judged as anticipated (R6) and transformation was performed according to configurations in properties files (R7).

Non-functional requirements for the intermediary application were largely matched according to the scope of this project. Performance tests showed a satisfactory speed of message processing by intermediary software and an ability to process multiple requests arriving simultaneously. It can be concluded that the intermediary application showed satisfactory performance and met the performing requirements, taking into account the previously mentioned limitations of local computer testing[83].

The application set up on the server was not tested, being left out from the scope of the project due to technical reasons described in section 3.5. But configuration of the intermediary proved to be customizable, as it was adjustable for different tests: processing data input from testing tools and from the information system. The indicated quality also affirms application portability. To summarise, achievement of usability and configurability requirements was not fully verified by test results.

Reliability requirements turned out to be met, as the software did not interrupt message processing on translation fail, illustrated by information system data first test, and functioned quite fast, as concluded from performance testing. Input data changeability, while the testing tool and information system tests input data provided for intermediary was different, can indirectly prove that further changes in Web services can be addressed by the special configuration of intermediary application. Extensibility of the application is verified by its portability and meets the narrow requirement set in the methods section 3.2. The intermediary application logged faults and processed data as seen from intermediary log files present for each test[84].

To conclude, the results show that the developed X-Road intermediary is able to recognize and translate SOAP messages accurately. While processing simultaneous requests

[83]Limitations described in section 4.3.3.

[84]Files named *log_intermediary.txt* accessible from https://github.com/julia-sulina/b-project/tree/master/test/intermediary.

performs satisfactorily fast, is capable of handling X-Road message data from information system and can be configured to work with various applications. Therefore the developed intermediary is a feasible solution concept for X-Road protocol switching, acknowledging that additional testing on the server setup and with information system data should be performed.

5.3 Improvements

The first test of the intermediary with information system data failed in some cases, and therefore the intermediary application was fixed by including the method for unification of SOAP main tags (envelope, header and body). The method implementation includes string processing, which might not be the best solution and can be further improved. The intermediary should be extensively tested and other possible faults not yet discovered be fixed.

The developed intermediary enables configuration using properties files. However, with the current software solution, to be able to reconfigure the application it must be restarted. There could be a scheduled operation or some other dynamic way to update the list of properties without the application restart.

One possibility to improve the intermediary performance, not currently implemented, is to store the translated header in the cache. For this, the incoming header should be normalized[85] and stored together with its translation. When the same header is encountered again, it will be just fetched from cache[86] without performing translation manipulations. This may be very efficient as headers repeat much more often in X-Road communication than the messages body[87].

When using a foreign API, the threat of unexpected bugs is possible. Both the intermediary and supporting application included such APIs. However rewriting functionality already available from open source does not guarantee bug amount reduction and is questionable improvement, decreasing only application dependence. As the intermedi-

[85]Normalization increases significantly the probability that SOAP messages with the same content have the same textual representation as well [17, p.1615].

[86]Similar solution to Kungas et al [17, p.1615] cache could be used, however message still would be forwarded to data provider.

[87]Header in X-Road message shows service requester and provider.

ary has many architectural alternatives, improvement could be done by determining the best available of them by testing all the different APIs.

A limitation of the supporting application is that it produces a quite strictly predefined structure of the properties files, which makes it narrowly usable. An improvement could be to develop a more flexible comparator for WSDL files.

6 Conclusions

Thesis project originated from an interest in the Web service migrating process while changing the protocol of Estonian X-Road communication. This change involves the need for adjusting production and consumption of service messages by current X-Road members. The project was grounded on the concept of the SOAP intermediary and aimed to test possibilities of addressing service change outside of the members' software. The goal was to create an application, which could translate X-Road messages into a new format without altering the code of existing information systems.

The work resulted in developing intermediary software for converting SOAP messages and an auxiliary tool generating input data for transformation. The results show that the implemented software is able to recognize the need for transformation and to convert messages accurately. It performs satisfactorily fast while processing simultaneous requests and is capable of handling X-Road information system message data. Therefore the developed intermediary is a feasible solution for X-Road protocol switching. The application proved to be usable with various configurations, which makes it suitable for addressing not only current migration, but also further changes in messages. Acknowledging that its additional testing on the server setup with the information system service provider should be performed. Absence of such tests is a limitation of the project.

Weakness of the message converting by intermediary is adding extra time to SOAP data exchange, even if transformation is performed quite fast. Nevertheless, the intermediary might be a less costly solution than changing each X-Road application. Alteration of the existing information systems requires resources for development and testing, as a change in the SOAP engine can indirectly influence other parts of the application. The intermediary, on the contrary, once configured works with messages independently from the X-Road member architecture and platform.

Another possible solution for avoiding problems of Web service alteration this thesis aimed to address is to develop systems with a highly adaptive dynamic SOAP engine of automated service changes discovery. Challenges in such a design are implementing unchangeable interface for interacting with other layers of software and engine reliability, so that any alteration in SOAP service would not influence functioning of the remaining application. Practical implementation of such an engine interface might be a question for further study.

References

1 Oracle. Servlet (Java EE 6). docs.oracle.com; 2011. Available from: https://docs.oracle.com/javaee/6/api/javax/servlet/Servlet.html [cited 2016-03-20].

2 W3C. Web Services Architecture. W3C Working Group Note; 2004. Available from: https://www.w3.org/TR/ws-arch/ [cited 2016-02-27].

3 Undusk M. Mis on X-tee? Riigi Infosüsteemide Arenduskeskus; 2008. (in Estonian). Available from: https://www.ria.ee/public/RIHA/RIHA_teabep_ev_14.11.08/riha-xtee-jaotusmaterjal.pdf [cited 2016-02-15].

4 W3C. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. www.w3.org; 2012. Available from: https://www.w3.org/TR/xmlschema11-1/ [cited 2016-03-15].

5 Sheng QZ, Qiao X, Vasilakos AV, Szabo C, Bourne S, Xu X. Web services composition: A decade's overview. Information Sciences. 2014 oct;280:218–238. Available from: http://www.sciencedirect.com/science/article/pii/S0020025514005428.

6 Yuxuan M, Ralph D. Research on the runtime behavior of composite SOAP web services under transient loads. Proceedings - International Conference on Computer Science and Software Engineering, CSSE 2008. 2008;3:499–504.

7 Microsoft. Chapter 1: Service Oriented Architecture (SOA). msdn.microsoft.com; 2016. Available from: https://msdn.microsoft.com/en-us/library/bb833022.aspx.

8 Snell J, Tidwell D, Kulchenko P. Programming Web Services with SOAP. OŔeilly Media; 2002.

9 Abu-Ghazaleh N, Lewis MJ. Differential deserialization for optimized SOAP performance. In: Proceedings of the ACM/IEEE 2005 Supercomputing Conference, SC'05; 2005. p. 55–64.

10 Tekli JM, Damiani E, Chbeir R, Gianini G. SOAP Processing Performance and Enhancement. IEEE Transactions on Services Computing. 2012 jan;5(3):387–403. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5719596.

11 Zimmermann O, Tomlinson M, Peuser S. Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects. Springer Science & Business Media; 2005.

12 Kreger H. Web Services Conceptual Architecture (WSCA 1.0); 2001. Available from: http://www.cs.uoi.gr/~pvassil/downloads/WebServices/Tutorials/WebServicesConceptualArchitecture.pdf.

13 Phan KA, Tari Z, Bertok P. Similarity-Based SOAP Multicast Protocol to Reduce Bandwith and Latency in Web Services. IEEE Transactions on Services Computing. 2008 apr;1(2):88–103. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4689465.

14 Li L, Niu C, Chen N, Wei J, Huang T. High Performance Approach for Server Side SOAP Processing. International Journal of Web Services Research. 2009;6(2):66–93.

15 Tekli J, Damiani E, Chbeir R. XML-Based Multicasting to Improve Web Service Scalability. International Journal of Web Services Research. 2012;9(1):1.

16 Abbas AM, Bakar AA, Ahmad MZ. Fast dynamic clustering SOAP messages based compression and aggregation model for enhanced performance of Web services. Journal of Network and Computer Applications. 2014 may;41:80–88. Available from: http://www.sciencedirect.com/science/article/pii/S1084804513002154.

17 Küngas P, Dumas M. Configurable SOAP Proxy Cache for Data Provisioning Web Services. In: Conference SAC'11 The 2011 ACM Symposium on Applied Computing TaiChung, Taiwan — March 21 - 24, 2011; 2011. p. 1614–1621.

18 Veldre A. Introduction to X-Road. Estonian Information System Authority; 2016. Available from: https://www.ria.ee/en/introduction-to-xroad-part1.html [cited 2016-02-18].

19 Republic of Estonia Information System Authority. Statistics about the X-Road. Republic of Estonia Information System Authority; 2016. Available from: http://x-road.eu/xtee-stats/ [cited 2016-02-18].

20 Estonian Informatics Centre. Riigi infosüsteemi illustreeriv skeem 2011. Riigi Infosüsteemide Arenduskeskus; 2011. Available from: https://www.ria.ee/public/publikatsioonid/x-road.jpg [cited 2016-02-15].

21 Peterkop T. X-tee versiooni 6 juurutamise ajakava. Käskkiri. Riigi Infosüsteemi Amet; 2015. (in Estonian).

22 Majandus- ja Kommunikatsiooniministeerium Riigi infosüsteemide osakond. Riigi IT arhitektuur; 2007. (in Estonian).

23 Annuk S, Nõgisto I, Kromonov I, Freudenthal M. X-Road: Message Protocol v4.0. Technical Specification.; 2015. Available from: https://www.ria.ee/public/x_tee/pr-mess_x-road_message_protocol_v4.0_4.0.8_Y-743-11.pdf [cited 2016-02-08].

24 Republic of Estonia Information System Authority. Lühiülevaade peamistest erinevustest X-tee versioonide 5 ja 6 vahel; 2015. (in Estonian). Available from: https://www.ria.ee/public/x_tee/X-tee_v5_ja_v6_erisused.pdf [cited 2016-03-01].

25 Zdun U, Voelter M, Kircher M. Design and implementation of an asynchronous invocation framework for web services. International Journal of Web Services Research. 2004;1(3):42–62.

26 RIHA - Riigi infosüsteemi haldussüsteem, Teenused. RIHA;. (in Estonian). Available from: https://riha.eesti.ee/riha/main [cited 2016-03-09].

27 Sulina J. Log Analysis of Information System. Document prepared at Tieto Estonia on the request of information system owner, used on the permission of the sides.; 2015. (in Estonian).

28 Vabariigi Valitsus. Infosüsteemide andmevahetuskiht. Riigi Teataja; 2008. (in Estonian). Available from: https://www.riigiteataja.ee/akt/119012011015?leiaKehtiv [cited 2016-03-05].

29 Jeckle M. Seamlessly Securing Web Services by a Signing Proxy. International Journal of Web Services Research. 2004;1(3):88–100.

30 McLaughlin B. Java & XML. O'Reilly Media, Inc.; 2001.

31 Vohra A. Pro XML Development with Java Technology. Apress; 2007.

32 Ashdown L, Greenberg J, Melnick J. Oracle XML Developer's Kit. Programmer's Guide. XML Parsing for Java. Oracle; 2014. Available from: https://docs.oracle.com/cd/B28359_01/appdev.111/b28394/adx_j_parser.htm#CCHCCEHA [cited 2016-02-27].

33 SAX. Quickstart. www.saxproject.org; n.d. Available from: http://www.saxproject.org/quickstart.html [cited 2016-03-07].

34 Oracle. Why StAX? Java Documentation. docs.oracle.com; 2015. Available from: https://docs.oracle.com/javase/tutorial/jaxp/stax/why.html [cited 2016-03-07].

35 Kalin M. Java Web Services: Up and Running. O'Reilly Media, Inc.; 2013.

36 Sun Microsystems Inc. Java API for XML Processing (JAXP) Tutorial. www.oracle.com; n.d. Available from: http://www.oracle.com/technetwork/java/intro-140052.html [cited 2016-03-08].

37 Ort E, Mehta B. Java Architecture for XML Binding (JAXB). www.oracle.com; 2003. Available from: http://www.oracle.com/technetwork/articles/javase/index-140168.html#binsch [cited 2016-03-05].

38 Soldano A, Schlebusch N. JAX-WS User Guide - JBoss Web Services - Project Documentation Editor. docs.jboss.org; 2014. Available from: https://docs.jboss.org/author/display/JBWS/JAX-WS+User+Guide [cited 2016-03-05].

39 The Apache Software Foundation. Apache Axis2 – Code Generator Wizard Guide for Eclipse Plug-in. axis.apache.org; 2016. Available from: https://axis.apache.org/axis2/java/core/tools/eclipse/wsdl2java-plugin.html [cited 2016-03-05].

40 The Apache Software Foundation. Apache CXF - WSDL to Java. cxf.apache.org; n.d. Available from: http://cxf.apache.org/docs/wsdl-to-java.html [cited 2016-03-05].

41 Lin CC, Fang CL, Liang D. A portable interceptor mechanism for SOAP frameworks. Computer Standards & Interfaces. 2013 nov;36(1):209–218. Available from: http://linkinghub.elsevier.com/retrieve/pii/S0920548913000226.

42 Juneau J. Introducing Java EE 7: A Look at What's New. Apress; 2013.

43 Yener M, Theedom A. Professional Java EE Design Patterns. Wiley; 2014.

44 Graham BS, Simeonov S, Boubez T, Davis D, Daniels G, Nakamura Y, et al. Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI. Pearson Education; 2002.

45 W3C. Relaying SOAP Messages. www.w3.org;. Available from: http://www.w3.org/2000/xp/Group/2/02/27-SOAPIntermediaries.html [cited 2016-01-06].

46 Nagappan R, Skoczylas R, Sriganesh RP. Developing Java Web Services: Architecting and Developing Secure Web Services Using Java. John Wiley & Sons; 2003. Available from: https://books.google.com/books?id=-41_zzAk4hgC&pgis=1.

47 predic8 GmbH. Membrane Service Proxy Documentation. www.membrane-soa.org; 2015. Available from: http://www.membrane-soa.org/service-proxy-doc/4.1/ [cited 2016-03-09].

48 The Apache Software Foundation. Apache Module mod_proxy - Apache HTTP Server Version 2.4. httpd.apache.org; 2016. Available from: https://httpd.apache.org/docs/current/mod/mod_proxy.html [cited 2016-03-09].

49 Bjørnson FO, Dingsøyr T. Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used. Information and Software Technology. 2008;50(11):1055–1068.

50 Bode S, Riebisch M. Tracing the Implementation of Non-Functional Requirements. In: Non-Functional Properties in Service Oriented Architecture: Requirements, Models and Methods. IGI Global; 2011. p. 424. Available from: https://books.google.com/books?id=BD_oa5Ki4bgC&pgis=1.

51 IEEE Std. IEEE Recommended Practice for Software Requirements Specifications. IEEE; 1998. Available from: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=720574.

52 The Apache Software Foundation. Maven – Welcome to Apache Maven. maven.apache.org; 2016. Available from: https://maven.apache.org/ [cited 2016-03-20].

53 JUnit. JUnit - About. junit.org; 2016. Available from: http://junit.org/ [cited 2016-03-20].

54 predic8 GmbH. Membrane SOA Model Documentation. www.membrane-soa.org; n.d. Available from: http://www.membrane-soa.org/soa-model/api-doc/ [cited 2016-03-09].

55 predic8 GmbH. Open Source Reverse Proxy for SOAP & REST - Membrane. http://www.membrane-soa.org/; 2015. Available from: http://www.membrane-soa.org/service-proxy/ [cited 2016-03-15].

56 Bayer T, Kessler F. Membrane Service Proxy wiki. github.com; 2014. Available from: https://github.com/membrane/service-proxy/wiki [cited 2016-02-26].

57 predic8 GmbH. Membrane SOA Router - Rule Matching. www.membrane-soa.org; 2015. Available from: http://www.membrane-soa.org/service-proxy-doc/4.1/rule-matching.htm [cited 2016-02-26].

58 The Grinder. Script Gallery. http://grinder.sourceforge.net/; n.d. Available from: http://grinder.sourceforge.net/g3/script-gallery.html [cited 2016-03-18].

1 Intermediary configuration

```
#proxy configuration
#compulsory configuration field, sets hostname proxy listens
     to, default value "*" - all
hostname=

#compulsory configuration field, sets path proxy listens to,
     default value ".*" - all
path=

#compulsory configuration field, sets port number proxy
    listens to, no default value
listenport=

#compulsory configuration field, sets proxy target hostname,
     no default value
targethost=

#compulsory configuration field, sets proxy target port
    number, no default value
targetport=

#translations configuration
#compulsory configuration field, sets a tag name with
    namespace of the message, which is present only in
    messages not needing translations
newTagNameWithNameSpace = xro:protocolVersion

#optional configuration field, sets name of a property file
    containing list of messages which need body translation
operationPropertiesFileName =

#optional configuration field, sets name of a property file
    containing list of messages which never need to be
    translated
ignorelistPropertiesFileName =

#compulsory configuration field, sets name of a property
    file containing list of tag translations
translationPropertiesFileName =

#compulsory configuration field, sets name of a property
    file containing new values for translated tags
valuesPropertiesFileName =

#compulsory configuration field, sets path to properties
    files
propertiesFilePath =

#compulsory configuration field, sets path to xml files
xmlFilePath =
```

Listing 7: Intermediary configuratiuon, xroadProxy.properties file.

Listing 7 shows intermediary configuration properties available from source code[88]. In case that compulsory properties are not set, application will throw configuration exception or set a default value for property. Configuration include proxy settings, such as host-names, path, ports and translation settings, such as properties files' names. Listing 2 shows XML file[89] used as template for header in intermediary configuration.

[88]https://github.com/julia-sulina/b-project/blob/master/intermediary/src/main/resources/xroadProxy.properties

[89]Available from source code https://github.com/julia-sulina/b-project/blob/master/intermediary/src/main/resources/header.xml

2 Supporting application configuration

```
# old and new version of WSDL should have same name
# compulsory configuration field, folder containing old
    version WSDL files
wsdl.old.folder=
# compulsory configuration field, folder containing new
    version WSDL files
wsdl.new.folder=
# optional configuration field, folder containing for
    generated files, default value project root
wsdl.properties.output.folder=
# optional configuration field, file containing known
    translations
wsdl.known.name.differences.properties.file=
```

Listing 8: Supporting application configuratiuon, wsdl.properties file.

Supporting application configuration[90] displayed in listing 8 requires providing folder paths to WSDLs, where WSDL file names should match to be paired. Such configuration enables process multiple input file pairs at once.

[90]Available from https://github.com/julia-sulina/b-project/blob/master/wsdlproperies/src/main/resources/wsdl.properties

3 Test environment

Testing of the applications was performed on local computer without using network, computer system properties are indicated in table 4. Development environment, in which tried application was run, is described on page 18. Testing tools are specified in the test arrangements section on page 19.

Table 4: System properties

Component	Description
Operating System Name	Microsoft Windows 7 Enterprise
System Type	64-bit Operating System
Processor	Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz, 3201 MHz, 4 Cores
Installed Physical Memory (RAM)	8.00 GB(7.89 GB usable)
Total Virtual Memory	15.8 GB
Hard Disk Drive (HDD) Size	465.27 GB
Solid-State Drive (SSD) Size	112.80 GB

4 Intermediary test arrangements

SoapUI test

SoapUI tool projects are accessible from source code as XML files, which can be imported to software for test reproduction[91]. Each of intermediary application tests required SOAP message test data, which content varied according to the test purpose. Exact test messages used for performed tests are located in source code together with other test-specific configurations[92]. Listing 9 displays example structure of the message used.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
   XMLSchema" xmlns:soapenv="http://schemas.xmlsoap.org/soap
   /envelope/" xmlns:xxx="http://***">
   <soapenv:Header>
      <xxx:id xsi:type="xsd:string">?</xxx:id>
      <xxx:nimi xsi:type="xsd:string">?</xxx:nimi>
      <xxx:isikukood xsi:type="xsd:string">?</xxx:isikukood>
      <xxx:andmekogu xsi:type="xsd:string">?</xxx:andmekogu>
      <xxx:asutus xsi:type="xsd:string">?</xxx:asutus>
   </soapenv:Header>
   <soapenv:Body>
      <xxx:REQUEST_NAME soapenv:encodingStyle="http://
         schemas.xmlsoap.org/soap/encoding/">
         <keha xsi:type="xxx:REQUEST">
            <!--You may enter the following 3 items in any
               order-->
            <isikukood xsi:type="xxx:IsikukoodType">?</
               isikukood>
            <algus_kuup xsi:type="xsd:date">?</algus_kuup>
            <lopp_kuup xsi:type="xsd:date">?</lopp_kuup>
         </keha>
      </xxx:REQUEST_NAME>
   </soapenv:Body>
</soapenv:Envelope>
```

Listing 9: Intermediary test message example.

[91] https://github.com/julia-sulina/b-project/tree/master/test/intermediary/soapui_test
[92] https://github.com/julia-sulina/b-project/tree/master/test/intermediary

Grinder test

Grinder tool detail configurations are downloadable from source code[93], example script run for testing is presented in listing 10 and configuration file in listing 11.

```
from net.grinder.script.Grinder import grinder
from net.grinder.script import Test
from net.grinder.plugin.http import HTTPRequest

test1 = Test(1, "Request resource")
request1 = HTTPRequest()

test1.record(request1)

class TestRunner:
    def __call__(self):
    msgStr = ''

        # read the SOAP message from the XML template file
    msg = open("message.xml","r")
    msgStr = msg.read()
    msg.close()
    result = request1.POST("http://localhost:4000/", msgStr)

        # result is a HTTPClient.HTTPResult. We get the
            message body
        # using the getText() method.
    writeToFile(result.text)

# Utility method that writes the given string to a uniquely
    named file.
def writeToFile(text):
    filename = "%s-page-%d.xml" % (grinder.processName,
       grinder.runNumber)

    file = open(filename, "w")
    print >> file, text
    file.close()
```

Listing 10: Intermediary test script. Adapted from Gringer examples [58].

```
grinder.processes=10
grinder.script=script-test1.py
grinder.threads=10
grinder.runs=1
grinder.logDirectory=logs
```

Listing 11: Grinder tool configuration (grinder.properties file) to run a test of 100 requests.

[93]https://github.com/julia-sulina/b-project/tree/master/test/intermediary/grinder_test