The University of Kansas

Technical Report

Service Oriented Architecture for Monitoring Cargo in Motion Along Trusted Corridors

M. Kuehnhausen, D. T. Fokum, V. S. Frost,
D. DePardo, A. N. Oguna, L. S. Searl, E. Komp,
M. Zeets, D. D. Deavours, J. B. Evans,
and G. J. Minden

ITTC-FY2010-TR-41420-13

July 2009

Project Sponsor:
Oak Ridge National Laboratory

Copyright © 2009:
The University of Kansas
2335 Irving Hill Road, Lawrence, KS 66045-7559
All rights reserved.

Abstract

This thesis describes a system called the *Transportation Security SensorNet* that can be used to perform extensive cargo monitoring. It is built as a *Service Oriented Architecture* (SOA) using open *web service* specifications and *Open Geospatial Consortium* (OGC) standards. This allows for compatibility, interoperability and integration with other *web services* and *Geographical Information Systems* (GIS).

The two main capabilities that the *Transportation Security SensorNet* provides are remote sensor management and alarm notification. The architecture and the design of its components are described throughout this thesis. Furthermore, the specifications used and the fundamental ideas behind a *Service Oriented Architecture* are explained in detail.

The system was evaluated in real world scenarios and performed as specified. The alarm notification performance throughout the system, from the initial detection at the *Sensor Node* service to the *Alarm Reporting* service, is on average 2.1 seconds. Location inquiries took 4.4 seconds on average. Note that the majority of the time, around 85% for most of the messages sent, is spent on the transmission of the message while the rest is used on processing inside the *web services*.

Finally the lessons learned are discussed as well as directions for future enhancements to the *Transportation Security SensorNet*, in particular to security, complex management and asynchronous communication.

Table of Contents

Abstract **i**

Table of Contents **ii**

List of Figures **vi**

List of Listings **viii**

List of Tables **ix**

1 Introduction **1**

2 Statement of Problem **3**

2.1 Proprietary Solutions 3
2.2 Variety of Open Standards 6
2.3 Service Oriented Architecture 9
2.4 Summary 12

3 Background **13**

3.1 Extensible Markup Language 13
3.1.1 Overview 14
3.1.2 Descriptive power 19
3.1.3 Ease of transformation 19
3.1.4 Information storage and retrieval 21
3.1.5 Flexible transmission 22
3.2 Open Geospatial Consortium 22
3.2.1 Sensor Web Enablement (SWE) 24
3.2.2 Geography Markup Language (GML) 26
3.2.3 Catalogue Service for Web (CSW) 28
3.2.4 Observations & Measurements (O&M) 29

3.2.5 Sensor Observation Service (SOS) . . . 31
3.2.6 Sensor Alert Service (SAS) . . . 33

4 Service Oriented Architecture **36**
4.1 Representational State Transfer (REST) . . . 39
4.1.1 Traditional Definition . . . 40
4.1.2 Current Use . . . 40
4.1.3 Further Development . . . 42
4.2 SOAP . . . 42
4.2.1 Message format . . . 43
4.2.2 Faults . . . 44
4.2.3 Further development . . . 45
4.3 Web Service Specifications . . . 46
4.3.1 WS-Addressing . . . 46
4.3.2 WS-Eventing . . . 47
4.3.3 WS-Security . . . 48
4.4 Service Directory . . . 51
4.5 Web Services Description Language (WSDL) . . . 53
4.5.1 Description . . . 55
4.5.2 Types . . . 56
4.5.3 Interface . . . 57
4.5.4 Binding . . . 57
4.5.5 Service . . . 58
4.6 Message Exchange Patterns . . . 58
4.6.1 In-Only . . . 59
4.6.2 Robust In-Only . . . 60
4.6.3 In-Out . . . 60
4.6.4 In-Optional-Out . . . 61
4.6.5 Out-Only . . . 61
4.6.6 Robust Out-Only . . . 62
4.6.7 Out-In . . . 62
4.6.8 Out-Optional-In . . . 62

5 Related Work **64**
5.1 Microsoft - An Introduction to Web Service Architecture . . . 64
5.2 Adobe - Service Oriented Architecture . . . 66
5.2.1 Request-Response via Service Registry (or Directory) . . . 67
5.2.2 Subscribe-Push . . . 67

5.2.3 Probe and Match 68
5.3 Open Sensor Web Architecture 69
5.4 Globus - Open Grid Services Architecture 71
5.5 Service Architectures for Distributed Geoprocessing 74
5.6 Web Services Orchestration 77
5.7 Summary 78

6 Design & Architecture **80**
6.1 Overview 80
6.1.1 Service Oriented Architecture 80
6.1.2 Services 88
6.1.3 Clients 89
6.1.4 Modules 90
6.1.5 Subscriptions 92
6.1.6 Synchronous and asynchronous communication 92
6.2 TSSN Common Namespace 94
6.3 Mobile Rail Network 97
6.3.1 Sensor Node 97
6.3.2 Alarm Processor 102
6.4 Virtual Network Operation Center 104
6.4.1 Sensor Management 105
6.4.2 Alarm Processor 109
6.4.3 Alarm Reporting 111
6.5 Trade Data Exchange 119
6.5.1 Trade Data Exchange Service 119
6.6 Open Geospatial Consortium Specifications 122

7 Implementation Results **123**
7.1 Logging Module 123
7.2 Log Parser 124
7.2.1 Abstraction Layer Model 124
7.2.2 Message Types 125
7.3 Visualization 127
7.4 Performance and Statistics 128
7.4.1 Road Tests with Trucks 128
7.4.2 Short Haul Rail Trial 130

8 Conclusion **134**

8.1 Current Implementation 134

8.2 Future work 135

8.3 Acknowledgment 137

References **138**

List of Figures

3.1 OGC standardization framework as described in [74] 23
3.2 Sensor Web Concept from [10] 25
3.3 Catalogue Service reference model architecture from [63] 28
3.4 Observation process as described in [20] 29
3.5 SOS data publishing process as described in [60] 32
3.6 SOS data consumption process as described in [60] 32
3.7 SAS advertising process described in [78] 33
3.8 SAS notification process described in [78] 34

4.1 Service overview 36
4.2 Traditional web applications and AJAX from Garrett [35] 41
4.3 SOAP message format 44
4.4 WSDL 2.0 overview 54
4.5 In-Only message exchange pattern 59
4.6 Robust In-Only message exchange pattern 60
4.7 In-Out message exchange pattern 60
4.8 In-Optional-Out message exchange pattern 61
4.9 Out-Only message exchange pattern 61
4.10 Robust Out-only message exchange pattern 62
4.11 Out-In message exchange pattern 62
4.12 Out-Optional-In message exchange pattern 63

5.1 Request-Response via Service Registry (or Directory) message exchange pattern from [65] 67
5.2 Subscribe-Push message exchange pattern from [65] 68
5.3 Probe and Match message exchange pattern from [65] 68
5.4 NOSA layer overview from [19] 69
5.5 Globus Toolkit overview from `http://www.globus.org/toolkit/about.html` 72

5.6 Forest fire application from [34] 75
5.7 Forest fire web services architecture from [34] 75
5.8 Web orchestration framework from [47] 78

6.1 Service message overview 81
6.2 Service cloud 82
6.3 Axis2 extensibility from [16] 84
6.4 Axis2 modules from [16] 85
6.5 Service composition 87
6.6 Mobile Rail Network message overview 97
6.7 Mobile Rail Network Sensor Node 98
6.8 Mobile Rail Network Alarm Processor 102
6.9 Virtual Network Operation Center message overview 105
6.10 Virtual Network Operation Center Sensor Management 106
6.11 Virtual Network Operation Center Alarm Processor 109
6.12 Esper architecture from [27] 109
6.13 Virtual Network Operation Center Alarm Reporting 111
6.14 Trade Data Exchange message overview 119
6.15 Trade Data Exchange Service 120

7.1 SOAP message (left) to Log parser classes (right) comparison 124
7.2 Two transmit-receive pairs (red and green) 126
7.3 A message couple (red) 126
7.4 Log file and service interaction visualization 127
7.5 Request performance from [31] 131
7.6 Network transmission and processing performance from [31] 131
7.7 System alarm notification performance from [31] 132

List of Code Listings

3.1 Simple XML book description 14
3.2 Library of books 15
3.3 Library of books using attributes 15
3.4 Extended library of books 16
3.5 Element book format 16
3.6 Element book format with type (elementBook.xsd) 17
3.7 Attribute book format with type (attributeBook.xsd) 18
3.8 Library schema (library.xsd) 18
3.9 Library stylesheet (library.xsl) 20
3.10 Library of books in HTML (library.html) 20
4.1 SOAP message format example 44
4.2 SOAP Fault message example 44
4.3 WSDL Description example 55
4.4 WSDL Types example 56
4.5 WSDL Interface example 57
4.6 WSDL Binding example 57
4.7 WSDL Service example 58

List of Tables

3.1 Example XPath expressions 21
3.2 Collection types from [20] 30

6.1 Sensor Node StartMonitorSensors operation 99
6.2 Sensor Node StopMonitorSensors operation 99
6.3 Sensor Node setSensors operation 99
6.4 Sensor Node AddSeals operation 100
6.5 Sensor Node getLocation operation 100
6.6 Sensor Node GetCapabilities operation 101
6.7 Sensor Node GetObservation operation 101
6.8 Alarm Processor Alert operation 103
6.9 Alarm Processor SensorNodeEvent operation 103
6.10 Alarm Processor SetMonitoringState operation 104
6.11 Sensor Management startMonitoring operation 106
6.12 Sensor Management stopMonitoring operation 107
6.13 Sensor Management getLocation operation 107
6.14 Sensor Management setAlarmSecure operation 108
6.15 Sensor Management setAlarmProcessorMonitoringState operation . . . 108
6.16 Alarm Processor MRN_Alarm operation 110
6.17 Alarm Reporting addSmsProvider operation 112
6.18 Alarm Reporting updateSmsProvider operation 113
6.19 Alarm Reporting removeSmsProvider operation 113
6.20 Alarm Reporting removeSmsProviderById operation 113
6.21 Alarm Reporting getAllSmsProviders operation 114
6.22 Alarm Reporting addContact operation 114
6.23 Alarm Reporting updateContact operation 115
6.24 Alarm Reporting removeContact operation 115
6.25 Alarm Reporting removeContactById operation 115
6.26 Alarm Reporting getAllContacts operation 116

6.27 Alarm Reporting addAlarmContactMapping operation 116
6.28 Alarm Reporting updateAlarmContactMapping operation 117
6.29 Alarm Reporting removeAlarmContactMapping operation 117
6.30 Alarm Reporting removeAlarmContactMappingById operation 117
6.31 Alarm Reporting getAllAlarmContactMappings operation 118
6.32 Alarm Reporting NOC_Alarm operation 118
6.33 Alarm Reporting getAllAlarms operation 118
6.34 TradeDataExchange ShipmentQuery operation 121
6.35 TradeDataExchange ValidatedAlarm operation 121

7.1 XPath expressions for *WS-Addressing* 125

Chapter 1

Introduction

Cargo theft and tampering are common problems in the transportation industry. According to Wolfe [85] the "FBI estimates cargo theft in the U.S. to be $18 billion" and the Department of Transportation "estimated that the annual cargo loss in the U.S. might be $20 billion to $60 billion". Wolfe [85] also gives good reason to believe that the actual number may be even higher than $100 billion because of two reasons. First it is assumed that about 60 percent of all thefts go unreported and second the indirect costs associated with a loss are said to be three to five times the direct costs.

With the advances in technology, this problem has evolved into a cat-and-mouse game where thieves constantly try to outsmart the newest cutting edge security systems.

In terms of securing cargo, there are usually two aspects: first ensuring the physical safety of the cargo and second monitoring and tracking it. The latter especially has become of more interest as of late because many shipments cross national borders and cargo may be handled by a multitude of carriers. All of this leads to a huge demand for tracking and monitoring systems by the cargo owners, carriers, insurance companies, customs and many others.

In this thesis, a framework is introduced which builds on open standards and software components to allow "monitoring cargo in motion along trusted corridors". The focus lies on the use of a *Service Oriented Architecture* and *Geographical Information System*

specifications in order to allow an industry wide adoption of this open framework.

A formal description of the problem to be analyzed can be found in chapter 2. In particular, it discusses the problems of proprietary systems, the advantages of open standards and the approach of using a *Service Oriented Architecture* in the transportation industry.

Chapter 3 gives an in-depth introduction to the *Extensible Markup Language* that is used as the foundation of the framework. Furthermore the specifications provided by the *Open Geospatial Consortium* that define the elements and interfaces for *Geographical Information Systems* are described.

The formal representation of the framework is a *Service Oriented Architecture* which is described in chapter 4 along with the components that it uses.

Chapter 5 refers to related work and focuses on the topics that either deal with the *Service Oriented Architecture* or the *Open Geospatial Consortium* specifications.

The main part of this thesis that details the design and architecture of the framework can be found in chapter 6. It explains the individual components as well as the software parts and specifications that are used in the implementation.

Chapter 7 gives test and performance results and describes the tools that have been developed for that particular purpose.

The thesis concludes with chapter 8 that also provides an outlook for future work.

Chapter 2

Statement of Problem

In order to address the problem of cargo theft, the *Transportation Security SensorNet* project has been created. Its goal is to promote the use of open standards and specifications in combination with web services to provide cargo monitoring capabilities.

The main question is the following:

> "How can a *Service Oriented Architecture*, open standards and specifications be used to overcome the problems of proprietary systems that are currently in place and provide a reusable framework that can be implemented across the entire transportation industry?"

The three main aspects of this question are discussed next.

2.1 Proprietary Solutions

Current commercial systems in the transportation industry are often proprietary. This is because a lot of effort is spent on research and development in order to create what is called *intellectual property*. The assumption is then that as long as the competitors do not have access to the system and its protocols that *intellectual property* is safe and provides a competitive advantage. Another common "benefit" of keeping the

systems closed is the perceived additional security since in order to successfully attack the system its implementation and protocols have to be *reverse engineered*.

The problem with this is that these advantages are often one-sided and favor vendors. Once a proprietary system has been implemented it has to be maintained. What happens if a customer that uses the system invested a lot of money into a its infrastructure and the training of its employees and the company that provides the system releases a new version of it which of course costs money again. The customer has several choices:

Upgrade Throughout the literature this is often considered the most expensive option because of the cost for the upgrade to the new version and the additional training to the employees that has to be provided. The benefits of upgrading are the use of new technology, potential gains in efficiency through new features and the latest bug fixes.

Do Not Upgrade By many regarded as the most cost efficient solution, choosing not to upgrade compromises new features and updates for the ability to save costs. An approach that is taken by some companies is the so-called *skip a version* technique. This allows companies to plan better as internal processes and systems often have to interoperate and need to remain compatible to each other.

Change Vendor In this situation, the new version of the system that is provided by company A does not provide the necessary features or is simply too expensive. Furthermore, a different company B offers a similar product with more features or for less money. The move to the new system is now dependent on the following things: How big are the estimated savings and what are the direct and indirect costs of the transition? It often happens that after careful consideration the costs outweigh the estimated gains and the customer goes back to considering whether or not to simply upgrade. If a transition is made, the process could be time consuming and turn out to be more complicated than expected.

Picture this extreme case as well. What happens if the vendor goes out of business? All of the sudden, the short-term goal is to maintain support for the system and to keep it running while in the long-term to look for a suitable replacement and be forced to transition. Even if this case does not happen the dependency on the vendor can be crucial. If the system has errors or a particular enhancement is desperately needed, the vendor decides what to do about it. For big companies that are major customers this may not be such a big problem because they often get preferential treatment. But for small and medium businesses the wait might be too long and lose them customers and revenue.

The main point here is that many customers are locked into proprietary solutions that are incompatible with similar solutions offered by competitors. In a 2003 survey by the Delphi Group [36] it was found that 52% of developers and 42% of consumers see standards enabling the "approval of projects otherwise threatened by concerns over proprietary system lock-in". Furthermore, an overwhelming 71% of developers and 65% of consumers feel that the use of open standards "increases the value of existing and future investments in information systems".

The problem of non-interoperability with regard to geospatial processing is the topic of a paper by Reichardt [75]. Because *Geographical Information Systems* are often immensely complex, companies that invest heavily into this area often only support their product. As described in the sample scenario, this leads to a lack of coordination among entities such as the *Federal Emergency Management Agency* (FEMA), the *National Transportation Safety Board* (NTSB) and the *Environmental Protection Agency* (EPA) because of the inability to share vital information which is the key to fast decision making and data analysis

2.2 Variety of Open Standards

The idea of open standards and specifications is to define so-called *interfaces* and *protocols* that can be used as references for the implementation of a system. There are many standards committees and industry groups that aim to define them, most often focused on a particular area. Some of the most well-known ones include the *World Wide Web consortium* (W3C), the *Organization for the Advancement of Structured Information Standards* (OASIS), the *International Telecommunication Union* (ITU) and the *International Organization for Standardization* (ISO).

The main principles that govern the development of standards are usually the same across all organizations. The following is an overview according to ISO:

Consensus All parties that are affected by the proposed standard get the chance to voice their opinions. This includes initial ideas and continues with feedback and comments during the standardization process.

Industrywide The idea is to develop global standards that can be used worldwide by entire industries.

Voluntary The standardization process is driven by the people that are interested in it and that see its future benefits across a particular industry. It is often based on so-called *best practices* that are already commonly in use.

The importance of open standards is emphasized in a paper by McKee [56]. It provides the evolution and success of the Internet as the "perfect example" for the use of open standards. In particular it explains that since the Internet is based upon communication and communication means "transmitting or exchanging through a *common system* of symbols, signs or behavior", the process of standardization can basically be seen as "agreeing on a *common system*". The other parts of the paper are focused on

how so-called *openness* can help *Geographical Information Systems* (GIS) but many of the points mentioned apply to open standards in general.

In particular the following aspects are associated with open standards:

Compatibility This includes the ability to share data across vendors and systems in a uniform and non-proprietary form. It allows processes to use essentially the same data in order to perform their specific task without the need of costly conversions or interpretation errors. Most *common* formats are also backward compatible which means that no particular version of the system is needed to interpret the data. Only a certain subset of functionality might be provided when using in older versions though. Another advantage of open formats is the fact that even if a particular version of a format is completely outdated and only used in legacy systems, its specification is still accessible to everyone. Hence systems can still be designed to use the format.

Freedom of Choice A major problem of proprietary solutions that was described earlier was the so-called *vendor lock*. Once a customer implements a proprietary system and builds its infrastructure around it, choices in the future are limited. Open standards by definition are vendor independent. Furthermore many of them support a broad variety of implementation scenarios. These implementations often are not even limited to a particular platform, operation system or programming language. This is especially true for most of the web standards.

Interoperability Through the use of clearly defined interfaces, standards dramatically enhance interoperability. The standards that define interface specifications do not provide a specific implementation but provide references to *best practices* and *implementation patterns* instead. Companies choose what kind of system implementation they prefer. This allows them to make use of existing infrastructure and capabilities that might otherwise have to be changed when using a proprietary system. The uniform access to functionality and data enables companies to connect a multitude of systems

and make more use of them. Also, in case one part of the system has to be replaced, another one that simply provides the same interface can take its place. This allows great flexibility in terms of the overall system design.

Leverage For companies the standardization of concepts, frameworks and common approaches provides a number of benefits. Since research and development can be extremely cost intensive, companies want to make sure there is a guaranteed *return on investment* for them. Open standards do not necessarily lead to increased revenue but they do provide insurance to the companies that they are on the "right" track and what they implement is actually used industrywide. This is very important because customers are aware that when they purchase a system from company A that uses a proprietary or non-standard implementation they might become a victim of *vendor lock*. Acquiring a system that is build on open standards allows them to choose the best and most cost effective solution from a variety of independent implementations. Another advantage is that once different implementations by the main vendors have been established, there is room for custom solutions by smaller vendors, often in the form of extensions or plugins.

Open Source The biggest benefit of using open standards is that fact it leads to innovation. This is because everybody can contribute, suggest enhancements, outline *best practices* and address mistakes. In terms of software this approach is often referred to as *open source*.

However, there are several problems that can be associated with non-proprietary systems. Implementations are based upon the interpretation of the standards which may differ significantly. Furthermore, some implementations only support a subset of the original specification, are slower than the reference implementation or use incompatible sub systems.

2.3 Service Oriented Architecture

The concept of information processing and sharing across various applications using so-called *web services* is the main focus of this thesis. The basic idea is to define components of a system as *services* and users as *clients* that can retrieve data from them. Note that interaction between *services* is done using so-called *embedded clients*. The *services* take care of things such as information processing, data analysis and storage. With all business logic embedded into *services* and interaction between them clearly defined using open standards an infrastructure is built that is called the *Service Oriented Architecture* (SOA).

The Internet allows the following two things that are relevant to geospatial processing: a common means of communication and the ability for efficient information sharing. There exist many standards on how to transmit, receive, encode and decode data. SOA builds on top of them to provide new specifications that enable the design, implementation and use of *web services*. Through these *web services* companies, government agencies and others have the ability to share and process information in a uniform manner which cuts costs, time and resources and improves efficiency. More information on the *Service Oriented Architecture* can be found in chapter 4.

Now why is the SOA such an "enabler"? What is possible now that was not possible before? According to Irmen [44] *automation* and *efficient communication* with partners are the two most important things in *supply chain management* which represents the core of the transportation industry. Let us take a look at how the *Service Oriented Architecture* addresses both of them in regard to the individual topics outlined in the paper.

Automation A vital part in transportation is the so-called *screening* process. Companies that transport goods must ensure safety and therefore check all parties involved in the trade. An important aspect of this is the use of a so-called *denied trade list*

which lists items and companies that are not allowed to import or export into specific countries. With the reduction in manual labor and transition to a *web services* based system that automatically performs these checks, efficiency could be greatly increased.

A closely related topic is *accountability*. Who is responsible if something goes wrong during the trade process? Since goods are often handled by many different parties, it must be possible to monitor the location of cargo and handovers tightly. This is especially important in cases of tampering or even theft of the cargo.

Furthermore, agencies and customs more and more require *electronic trade information* instead of paper documents in order to track trade. Because of different formats and legacy applications that are often unable to provide this information in its entirety, additional resources have to be allocated in order to remain compliant with current practices. *Web services* and open standards can overcome this problem with uniform interfaces and common data formats.

Having the ability to *monitor* the location not just for perishable goods but also for high value goods is of great importance in the transport chain. Current processes should be able to automatically route cargo based on its needs and cost effectiveness.

Irmen [44] also points out that "the lack of integration between products causes users to deal with multiple systems having disparate data and non-uniform input and output" and calls for the use of a single platform. Using the *Service Oriented Architecture* this "call" becomes less necessary because it is platform independent and at the same time able to provide integration of multiple systems and standardized data formats.

Efficient Communication Building a virtual network among the parties involved in the trade process establishes efficient means of communication. It allows the *coordination* between otherwise disparate entities that is essential to provide cost effective and reliable shipping of cargo. The Internet provides the communication layer but it is the standards of *web services* that enable the integration of different systems.

Irmen [44] mentions the so-called *Software-as-a-Service* (SaaS) approach which allows software to run on a per-use basis without the costs of complex hardware infrastructure. This works very well with SOA as the interfaces defined by those services are often *web services* interfaces that are essentially part of SOA.

Security within the transportation industry plays a big role because trade data is to be kept confidential at all times and only distributed on a need-to-known basis. This puts an additional burden on the parties that are involved, as the parties must exchange data confidentially at each point of interaction. If open standards are used for this, *security* is implemented based on interfaces and policies that are easy to manage.

In order to manage the transportation chain in its entirety, a *global view* is often needed. This is problematic since individual parties often only deal with their respective neighbors. Using open standards and the *Service Oriented Architecture* approach each party could provide an uniform information interface that is accessible to other parties in the chain. This allows consistent reporting, monitoring and analysis at each step during the shipping process.

The reporting part especially has gained more attention over the past years as the focus has shifted towards more ethical and socially responsible business practices. *Accountability* coincides with this *social visibility* and therefore improvements in monitoring cargo not only lead to increased revenue on the business side but better public relations as well.

Overall the paper by Irmen [44] gives excellent reasons for open systems in terms of accountability, coordination, scalability and cost. It outlines important aspects that need to be taken into consideration when designing an architecture such as the *Transportation Security SensorNet.*

2.4 Summary

The following chapters describe how open specifications for *Geographical Information Systems* in combination with *web services* can be used to address the problems of proprietary systems that were outlined in section 2.1. In the *Transportation Security SensorNet* (TSSN) this is achieved by using a variety of open standards primarily because of the aforementioned *interoperability* and *freedom of choice* (see section 2.2). The use of a *Service Oriented Architecture* for the TSSN allows the creation of the applications needed for efficient and cost effective transportation chains (see section 2.3).

Chapter 3

Background

3.1 Extensible Markup Language

The *Extensible Markup Language* (XML) is a specification by the *World Wide Web Consortium* (W3C) that is used to describe data in a highly flexible but also concise way. It serves as the basis for most of the specifications that are referenced in this thesis.

As described by Sperberg-McQueen et al. [81] one of the main goals of the specification is interoperability and support for a multitude of applications. This is emphasized by the fact that XML should be human-readable and easy to process by computers. XML can be used to describe, filter and format data while providing storage functionality as well.

In the *Transportation Security SensorNet* it is utilized in a variety of ways. The *web services* and the *Open Geospatial Consortium* standards define their interfaces and data elements using XML. SOAP, as described in section 4.2, is a XML message format that is used as the basis for the transmission of data in the framework. Furthermore, many configuration files for the web services and clients in the *Transportation Security SensorNet* are in XML. The use of the *Extensible Markup Language* is one of the main reasons for the flexibility and reusability of the framework

3.1.1 Overview

In the following sections some basic principles of XML are introduced. Let us start by describing a simple book using XML.

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <book>
3   <!-- english title -->
4   <title>Hamlet</title>
5   <!-- author name -->
6   <author>William Shakespeare</author>
7 </book>
```

Listing 3.1 Simple XML book description

The first line is the XML declaration. It specifies that the described document uses version 1.0 of the XML specification and UTF-8 encoding. Line two starts with the definition of a book that contains a title (line four) and an author (line six). Note that line three and five are comments that are not part of the actual data but can be used to further describe it to humans. This example shows that XML can be as descriptive to humans as it is to computers.

Looking at the XML we can see multiple things. The element *book* has a so-called *start-tag* (line two) and an *end-tag* (line seven). Information about the specific book is kept in between these tags. As for the title and author information the actual data is also contained within their *start-tag* and *end-tags*. This demonstrates one basic type that is used most frequently in XML, an *element*. An *element* consists of a *start-tag* and an *end-tag* with either content or other *elements* in between. Note that there are also so-called *empty-element-tags* that look like `<empty-element/>`. They contain no further content or *elements*.

One of the requirements of using XML in applications is that one needs to define one specific *root element*. Therefore if we wanted to define more books let us put them into a library *root element*.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<library>
  <book>
    <title>Hamlet</title>
    <author>William Shakespeare</author>
  </book>
  <book>
    <title>Great Expectations</title>
    <author>Charles Dickens</author>
  </book>
  ...
</library>
```

Listing 3.2 Library of books

XML is flexible enough to use different descriptions of essentially the same data. The following example represents the same library using *attributes* for title and author information instead of *elements*. *Attributes* are basically *name-value pairs* that contain information about the *element* that they are a part of.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<library>
  <book title="Hamlet" author="William Shakespeare" />
  <book title="Great Expectations" author="Charles Dickens"
      />
  ...
</library>
```

Listing 3.3 Library of books using attributes

The “problem” with this is that if one application uses *elements* and the other application uses *attributes* to describe books in their libraries they seem incompatible. In order to solve this we need to make sure that each description is uniquely identifiable. This can be done declaring so-called *namespaces* as described by Bray et al. [13]. The idea is to attach a specific *Uniform Resource Identifier* (URI) (see Berners-Lee et al. [6]) to the document or element definitions. For example, this would result in `<a:book xmlns:a="http://www.sample.com/elementBook">` for listing 3.2 and `<b:book xmlns:b="http://www.sample.com/attributeBook">` for listing 3.3. Using

these *namespaces* we have the ability to mix different descriptions in a single document.

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <library xmlns="http://www.sample.com/library">
3   <a:book xmlns:a="http://www.sample.com/elementBook">
4     <a:title>Hamlet</a:title>
5     <a:author>William Shakespeare</a:author>
6   </a:book>
7   <b:book xmlns:b="http://www.sample.com/attributeBook"
        title="Great Expectations" author="Charles Dickens" />
8   ...
9 </library>
```

Listing 3.4 Extended library of books

We can also use namespaces to uniquely identify document descriptions. The default description in listing 3.4 is `<library xmlns="http://www.sample.com/library">` and more specific descriptions are in place for each book.

So what do these descriptions actually look like? They are written in XML as well and called *XML Schema Definitions* (XSD). An overview is provided by Fallside and Walmsley [28] and the exact structure by Mendelsohn et al. [57]. While there are other standards in place for describing XML documents, *XML schemas* are the most common.

Let us describe the first book format.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3    targetNamespace="http://www.sample.com/elementBook"
4    xmlns="http://www.sample.com/elementBook">
5    <xsd:element name="book">
6      <xsd:complexType>
7        <xsd:sequence>
8          <xs:element name="title" type="xsd:string"/>
9          <xs:element name="author" type="xsd:string"/>
10       </xsd:sequence>
11     </xsd:complexType>
12   </xsd:element>
13 </xsd:schema>
```

Listing 3.5 Element book format

We defined an element called *book* that contains two elements called *title* and *author*. Both of them are of type *string* which is a predefined data type. For ease of use and compatibility reasons the specification defines a set of standard data types. The type of *book* is so-called *complex* since it is the parent of other elements. Because this type is defined implicitly it is called *anonymous typing*. If one wanted to reuse the *book* type in some other element definition it makes more sense create a *complex book* type and define an *element* that is of this *type*. The XML schema would then take the following form:

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3    targetNamespace="http://www.sample.com/elementBook"
4    xmlns="http://www.sample.com/elementBook">
5    <xsd:element name="book" type="BookType"/>
6    <xsd:complexType name="BookType">
7      <xsd:sequence>
8        <xs:element name="title" type="xsd:string"/>
9        <xs:element name="author" type="xsd:string"/>
10     </xsd:sequence>
11   </xsd:complexType>
12 </xsd:schema>
```

Listing 3.6 Element book format with type (elementBook.xsd)

Line three defines the so-called *target namespace* of the schema. When the schema is used in a document, elements from it will automatically have this namespace. Line four specifies the *default namespace* for the schema so that elements and types in the schema are able to reference each other. The *sequence* tag at line seven specifies that the elements are to be in order, first *title* and then *author*. Other common options include *all* for random order and *choice* for the exclusive selection of elements.

The second book format could be defined by the following schema:

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3    targetNamespace="http://www.sample.com/attributeBook"
4    xmlns="http://www.sample.com/attributeBook">
5    <xsd:element name="book" type="BookType"/>
6    <xsd:complexType name="BookType">
7      <xsd:attribute name="title" type="xsd:string"/>
8      <xsd:attribute name="author" type="xsd:string"/>
9    </xsd:complexType>
10 </xsd:schema>
```

Listing 3.7 Attribute book format with type (attributeBook.xsd)

The only major difference in listing 3.7 is using an *attribute* instead of an *element* for the *book* information. Since our library should be able to use both descriptions let us define a schema that will allow this.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3    xmlns:a="http://www.sample.com/elementBook"
4    xmlns:b="http://www.sample.com/attributeBook"
5    targetNamespace="http://www.sample.com/library"
6    xmlns="http://www.sample.com/library">
7    <xsd:import namespace="http://www.sample.com/elementBook"
         schemaLocation="elementBook.xsd"/>
8    <xsd:import namespace="http://www.sample.com/
        attributeBook" schemaLocation="attributeBook.xsd"/>
9    <xsd:element name="library">
10     <xsd:complexType>
11       <xsd:choice minOccurs="0" maxOccurs="unbounded">
12         <xs:element ref="a:book"/>
13         <xs:element ref="b:book"/>
14       </xsd:choice>
15     </xsd:complexType>
16   </xsd:element>
17 </xsd:schema>
```

Listing 3.8 Library schema (library.xsd)

The two previously defined schemas are imported in lines seven and eight. Line twelve and thirteen use so-called *references* to these defined elements. In this case we define the number of occurrences of each element explicitly. This is because by default

all elements have a `minOccurs=1` and a `maxOccurs=1`, meaning that they are required but may appear only exactly once. Hence, the library consists of books either in *element* or *attribute* format and the possible number of books ranges from *none* to *infinite*.

The examples that were covered illustrate how XML can be used to describe and store data. But what are the advantages of using XML over other technologies that can essentially do the same? One of the main reasons why the use of XML has grown in recent years is because of the impact of the *Internet*. Applications and data that were previously stored internally, often in proprietary formats, are now made accessible to *remote* locations and users. The need to deal with data in a more open and flexible way became apparent especially for web applications and services. The following sections describe the different ways of how web applications and applications in general can utilize and benefit from XML.

3.1.2 Descriptive power

The description of data using XML enables applications to be very flexible and modular. New fields or attributes of data can be added using schema extensions and applications can choose either to use the extension or the original XML schema definition. Data can even be entirely rearranged using new or modified element and type definitions. This allows different views of the same data which decreases conversion costs and increases reusability and interoperability. In essence the data stays the same, the only thing that changes is its interpretation.

This aspect is essential in a *Service Oriented Architecture* like the *Transportation Security SensorNet* because clients and web services are highly dynamic. Using XML allows the entire framework to be implemented in a flexible, modular and reusable way.

3.1.3 Ease of transformation

Data often needs to be transformed or converted from one format into the other. Since XML only describes the data we can transform it easily into whatever is needed.

For this reason *Extensible Stylesheet Language Transformations* (XSLT) as described by Kay [46] have been introduced. They enable automatic conversion of XML documents using so-called *stylesheets* that are defined in XML. Let us take the initial library in listing 3.2 and transform it into a simple HTML web page.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/
       Transform" version="2.0" >
3    <xsl:template match="/">
4      <html>
5        <body>
6        <h1>Library books</h1>
7        <xsl:for-each select="library/book">
8        <div><xsl:value-of select="title"/> by <xsl:value-of
           select="author"/></div>
9        </xsl:for-each>
10       </body>
11     </html>
12   </xsl:template>
13 </xsl:stylesheet>
```

Listing 3.9 Library stylesheet (library.xsl)

In listing 3.9 a header is specified that displays “Library books”. For each book in the library the title and author are then extracted and put into a relationship sentence. Hence, the resulting output would look like the following:

```
1  <html>
2    <body>
3    <h1>Library books</h1>
4    <div>Hamlet by William Shakespeare</div>
5    <div>Great Expectations by Charles Dickens</div>
6    ...
7    </body>
8  </html>
```

Listing 3.10 Library of books in HTML (library.html)

Note that this is just one of the many possibilities of converting an existing XML document into a different format. Within the *Transportation Security SensorNet* these *Extensible Stylesheet Language Transformations* are used by *Apache Axis2* to create

Java classes from *XML Schema Definitions* and *Web Services Description Language* files so that they can be used by clients and web services (see 6.1.1.2 and section 6.1.1.4).

3.1.4 Information storage and retrieval

Storing data in an XML format makes the data and relations between data more flexible. Databases often face the problem of *sparsity* where when a new column is added to a table all entries must have this new column. XML works in a different way. Additional information fields can be added just to the elements that need them while for all other elements the XML schema would simply define the field as *optional*. This can potentially save a lot of space when compared to storing the same data in traditional databases. In the *Transportation Security SensorNet* this "cost saving" approach is utilized by SOAP during the message transmission (see section 4.2).

In order to retrieve information efficiently from XML several specifications have been designed. Boag et al. [8] describes *XPath* which is a query language specifically designed for XML. It works on the basis of a document tree, the so-called *data model*, that it creates from the original XML. Elements are *nodes* in the tree and attributes so-called *attribute nodes*. Information can then be retrieved using *path expressions*. Table 3.1 shows some examples of the information that we are able to retrieve and the *path expressions* that were used for the library in listing 3.2. *XPath* is used by the *Log Parser* extract information from log files (see section 7.2).

XPath expression	**Result**
library	all books of the library
library/book[1]	first book
//author	all authors
//author/text()	all author names
//book[title="Hamlet"]/author/text()	author name of Hamlet

Table 3.1. Example XPath expressions

Another specification that is used for XML data information retrieval is called *XQuery* which was defined by Siméon et al. [77]. It is more complex and builds on

top of *XPath 2.0*. Immediate result computations and transformations are possible using a so-called *FLOWR* expressions. Where *Xpath* simply extracted information, *XQuery* enables applications and users to directly modify or change the appearance of the information.

3.1.5 Flexible transmission

Since there is a significant overhead associated with conversion, standards have been defined that allow various forms of XML to be transmitted with little or no modification. The simplest form is just to send an XML document from sender to receiver using HTTP which is known as *Representational State Transfer* (REST) (see section 4.1). In that case both parties have the schema information. This is not a lot different than using a binary format since the communication is useless for anybody that does not understand the format. The advantage though would be that there is no conversion from XML into another format necessary. For more advanced scenarios it becomes more feasible to wrap the document that is being transmitted into a standardized transport package or message. The most common way to achieve this for XML is by using *SOAP* which is the case in the *Transportation Security SensorNet* and described in section 4.2.

3.2 Open Geospatial Consortium

The *Open Geospatial Consortium* (OGC) is the de facto authority on open standards for *Geographical Information Systems* (GIS). Its members develop interface specifications for geographical applications. One of the primary goals is interoperability: research and development costs are later diminished by the fact that if one application implements an OGC standard other applications can use it through the predefined interfaces that the standard provides. Furthermore, there is a higher interest in the actual implementation of standards since a majority of the industry agreed upon them. This mitigates one of the main risks that proprietary applications otherwise face, the lack of

user and industry acceptance.

Some of the industry needs cover a wide area of topics whereas others are very specific. For example, there needs to be a standard for dealing with times, locations and their formats which is something that almost all geographical applications face at some point. On the other hand, the format for requesting live feeds from a sensor is of interest only to a smaller group. The OGC tries to cover everything from simple to complex that could enhance the development of spatial information applications and services.

The way it is able to achieve this is by not actually implementing the standards but only providing the framework, the specification and schemas. The usual development framework looks as follows.

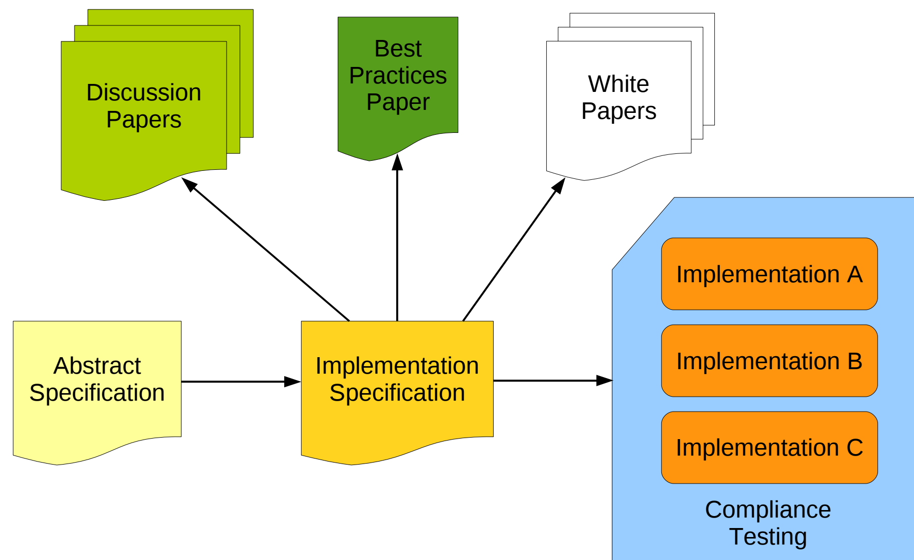

Figure 3.1. OGC standardization framework as described in [74]

First, *abstract specifications* are written that describe the goal and primary concepts of a proposed standard. This is explained in detail by Reed [74]. Second, the *abstract version* of a standard leads to an *implementation specification* which eventually becomes a *standard* after it has been accepted by the OGC members. Third, the industry

in terms of application and service developers implements the *specification* and provides feedback to the consortium. Furthermore the OGC releases *white papers* that provide high-level overviews of the concepts of a standard and a *best practices* paper that describes implementation specific development patterns. So-called *discussion papers* are usually written by developers talking about the technologies and approaches used in their implementations. Finally, the OGC encourages *implementations* to be tested and marked as compliant using their test suites.

An overview of the procedures and the approaches taken are described in the *OGC Reference Model* (ORM) by Percivall et al. [71]. It explains the concepts behind storing geospatial information, referencing locations and times, and creating maps or so-called *geometries* from the available data. The reference model refers to several abstract specifications in order to establish a connection between them and reiterate the goal of developing open interoperable standards. Apart from talking about the approaches behind geospatial information processing, the concepts of *geospatial services* and reusable patterns are introduced.

The *Transportation Security SensorNet* aims to be open and interoperable. It uses the interfaces and elements defined in the specifications of the *Open Geospatial Consortium* and provides concrete implementations, for example the *Sensor Observation Service* and the *Sensor Alert Service.* In terms of web services within a *Service Oriented Architecture* the following standards are of importance.

3.2.1 Sensor Web Enablement (SWE)

One of the main focuses of the OGC in recent years has been the development of concept called *Sensor Web.* In the *Sensor Web Enablement* (SWE) architecture and overview document by Botts et al. [10] it is described as follows:

> "A Sensor Web refers to web accessible sensor networks and archived sensor data that can be discovered and accessed using standard protocols and

application program interfaces (APIs)."

This is best visualized by the concept figure 3.2 from the document.

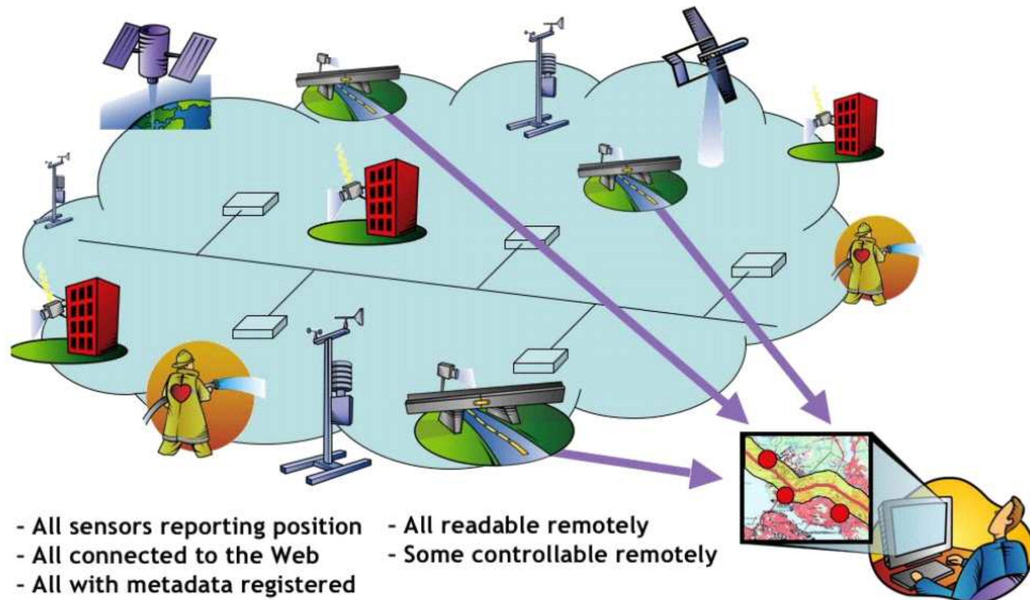

Figure 3.2. Sensor Web Concept from [10]

The idea is to combine various information modeling specifications with the appropriate services that provide the data processing for them. According to Botts et al. [10] the following specifications make up the *Sensor Web*:

- Observations & Measurements (O&M) specifies the representation of sensor measurements.
- Sensor Model Language (SensorML) describes sensors, models and their processes. For instance the discovery of sensors and data preprocessing.
- Transducer Markup Language (TML) specifies the encoding and transport of streaming sensor data in real-time scenarios.
- Sensor Observation Service (SOS) provides interfaces for describing what capabilities a sensor can perform and for retrieving actual observations or measurements.
- Sensor Planning Service (SPS) allows users to query the *sensor web* for a specific need. For example: "monitor the following 5 intersections every minute for excessive traffic for the next week".

- Sensor Alert Service (SAS) provides users with the ability to subscribe to certain sensor events. Like “notify me when the temperature exceeds 100°F”.
- Web Notification Service (WNS) describes message exchange capabilities between clients and services.

This thesis uses the same approach in order to define the *Service Oriented Architecture for Monitoring Cargo in Motion Along Trusted Corridors* called *Transportation Security SensorNet*. However, it has to be noted that there are some differences in the implementation and use of specifications. For instance only a subset of the *Sensor Web* specifications are actually used.

The *Geography Markup Language* (GML), that is only briefly mentioned by Botts et al. [10] in the SWE document, essentially describes some of the main components and elements that are used by most of the specifications in the implementation. Additionally, the *Catalogue Service for Web* (CSW) can provide a so-called *service directory* of available services. The *Sensor Web Enablement* is an initiative from the OGC that aims at the combined growth of theses specifications that will essentially make up the *Sensor Web*. While some of the specifications are agreed standards others like the *Sensor Alert Service* (SAS) are still in draft stage as of summer 2009.

Specifications that are relevant to the *Transportation Security SensorNet* are explained in more detail in the following sections.

3.2.2 Geography Markup Language (GML)

The need for a standard to encode geospatial features in an abstract way that can eventually be mapped onto real world things is elementary. The *Geography Markup Language* (GML) as described by Portele [72, 73] aims at defining most, if not all, features with a geographical background that can be defined. Among the things covered in the specification are *observation models*, *spatial and temporal reference systems*, *geometries* and *units of measure*. It considers a variety of base components that are common between applications and allows for other domain or application specific profiles to be

defined, therefore extending them. Application schemas describe a certain subset of definitions within the standard but might introduce new or extended types that are specific to the application.

The specification is highly hierarchical in the sense that several abstraction layers have been introduced in order to hide complexity. The two base objects that are defined from which all others are derived are *abstract object* and *abstract gml*. Basic types like *features* that model things like roads or rivers add more properties onto the base objects. This extension might be as simple as adding a location name and reference to it.

Things that can be modeled mathematically are part of a so-called *geometry*. This includes *points* which are *primitives*, *lines* and *curves* which are *aggregates* and can lead to more *complex* elements like *polygons* and *surfaces*.

Another big part of the specification is describing temporal constructs like *time instants, periods, intervals, durations* and *calendars*. Coordinate reference systems may be used differently throughout the world therefore definitions for them are included as well. They are used to specify time and location formats for instance. Units of measure are standardized definitions of measures and values of objects. There is also a section in the GML specification called observation which covers mostly simple types of observations. A more in-depth specification covering this is the *Observation & Measurements* (OM) specification (see 3.2.4).

An article by Bardet and Zand [2] gives an excellent example of how data is converted from format called AGS into GML. The main problem that is described is the lack of systematic archiving and exchange of drilling data. Since obtaining this data can be very cost intensive it has become a big issue. Hence, transforming the data into GML allows companies and researchers to take advantage of OGC applications for storage, exchange and visualization of this information. This reduces cost and makes the drilling data more useful. The article represents a case study in the sense that it describes in detail all the steps that were taken to implement the data conversion.

GML is used by many other specifications as the basis for describing geographical

information. In the *Transportation Security SensorNet* it is used by the *Sensor Observation Service* and the *Sensor Alert Service* implementations provided by, among others, the *Sensor Node* at the *Mobile Rail Network* (see section 6.3.1).

3.2.3 Catalogue Service for Web (CSW)

The *Catalogue Service for Web* (CSW) as specified by Nebert et al. [63] describes the "discovery, access, maintenance and organization of catalogues of geospatial information and related resources". It manages resource information for services in the form of metadata.

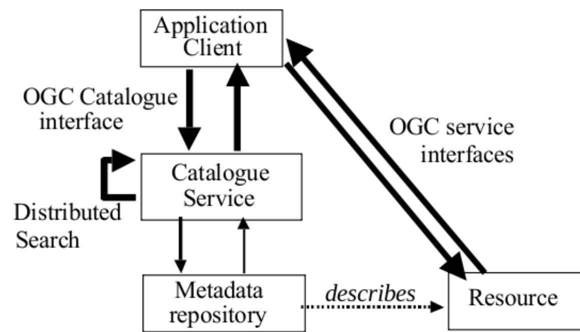

Figure 3.3. Catalogue Service reference model architecture from [63]

Whenever a client requires geospatial information or processing capabilities it queries the *Catalogue Service*. A metadata repository is kept in order to store information such as location, capabilities and schema definitions of services. Information that matches the query is then returned to the client. The client also has the ability to ask for a description of specific metadata elements and use that to get more specific results. The CSW therefore acts as broker between the clients and the services. Once the client has found a suitable service, it looks into the metadata that describes a particular service and uses that information to perform its request.

One of the advantages of this architecture is the ease of use for the client. A lot of services could provide essentially the same functionality. After they have all registered

with the *Catalogue Service* it is up to the client to choose which one to use. If a service is not available the client can simply try a different one. Furthermore it is not necessary for the client to actually know where the services are all the time since the *Catalogue Service* stores this information. This allows for a flexible environment and makes it scalable.

In the *Transportation Security SensorNet* this *service directory* functionality is provided by an implementation of *Universal Description, Discovery and Integration* (UDDI) specification (see section 4.4). Clients and web services in the framework have the option to contact it and retrieve similar information to the one offered by a *Catalogue Service*.

For additional scalability the specification also describes an approach called *distributed search*. Multiple *Catalogue Services* can set up a query topology where each service is responsible for its own metadata but the query is answered collectively. For the schema definitions of the *Catalogue Service for Web* see Nebert et al. [62].

3.2.4 Observations & Measurements (O&M)

Since there exists a variety of different sensors for almost every application, defining a standard that is true to all of them can be quite hard. The goal of the O&M standard as specified by Cox [20, 23] is to build an abstraction layer model that allows users and other services to use whatever granularity they need.

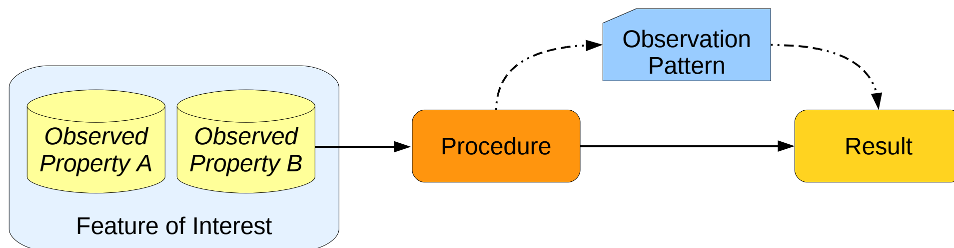

Figure 3.4. Observation process as described in [20]

Whenever an action is performed we basically “observe” a *feature of interest*. What we are interested in is the value of an *observed property* of that feature and in order

to determine this property value we exercise a particular *procedure*. Additionally an *observation pattern* can be useful for estimation and error correction of the observation result. In cases where the result is numeric the term *measurement* is used instead of *observation*. There are other specialized result types ranging from simple to complex. An observation may also be associated with a location. This is quite common.

Depending on the properties of its members, collections of observations can be one of the following types:

Type	Feature	Sampling time	Observed properties
complex	same	same	different
time series	same	different	same
discrete coverage	same	same	elements of a larger feature

Table 3.2. Collection types from [20]

The specification deals with collection types where the *feature of interest* does not change but stays the same. We speak of a *complex* observation when different properties are observed at the same time whenever a sample is taken. In case a particular property is monitored over a certain time period and the property does not change throughout the observation, the collection is called a *time series*. Sometimes the *observed property* we are interested in is made up of many smaller *observed properties*. This scenario describes a *discrete coverage*. An example given by Cox [20] is the observation of temperature values in a particular region where there are multiple sensors in the region but one is only interested in the temperature for the entire region.

Another thing described in the specification is the fact that in many cases the single *observed property* is not actually what is wanted but rather just something indirect. The *sampling of features* concept that deals with this is described by Cox [21, 22]. On the one hand, the *observed property* value could be in need of adjustment or only usable after the application of an algorithm as is often the case with light and temperature values. On the other hand, one value might not be of any importance at all but is just a part of a bigger *sample design*. Sometimes both cases can apply at the same time.

When an observation falls into this category the *sample features* form a particular relation that connects them and a so-called *survey procedure* is defined. This process achieves the desired abstraction where at a higher level the result of this relation looks like just another value since the *sampling of features* works transparently underneath it.

The *Observations & Measurements* (O&M) specification is used by the *Sensor Observation Service* in the *Sensor Node* at the *Mobile Rail Network* (see section 6.3.1). It is used in combination with GML because O&M allows for more complex observations while GML provides a broader field of geographical elements.

3.2.5 Sensor Observation Service (SOS)

The *Sensor Observation Service* (SOS) is described by Na and Priest [60, 61]. It aims to provide the user with observation data in a generic way that allows the use of a variety of different sensors. The two major types mentioned in the specification are *in-situ* and *remote* sensors. The primary goal is to provide access to observations (see 3.2.4). An implementation of this service within the *Transportation Security SensorNet* is provided by the *Sensor Node* (see section 6.3.1).

The service provides so-called *observation offerings* to users and applications. It does this by maintaining a *sensor registry* that contains information such as type, location and other metadata about the sensors that it knows about. This allows clients to perform detailed inquiries about possible observation times, available properties and geographical information of sensors and features.

GML is used to deal with measures and units in the offerings and when referencing observations. Apart from allowing filtering by sensor id the *Sensor Observation Service* is able to filter by spatial, scalar and temporal expressions. The two concepts it uses are called *data publishing* and *data consumption*.

3.2.5.1 Data Publishing

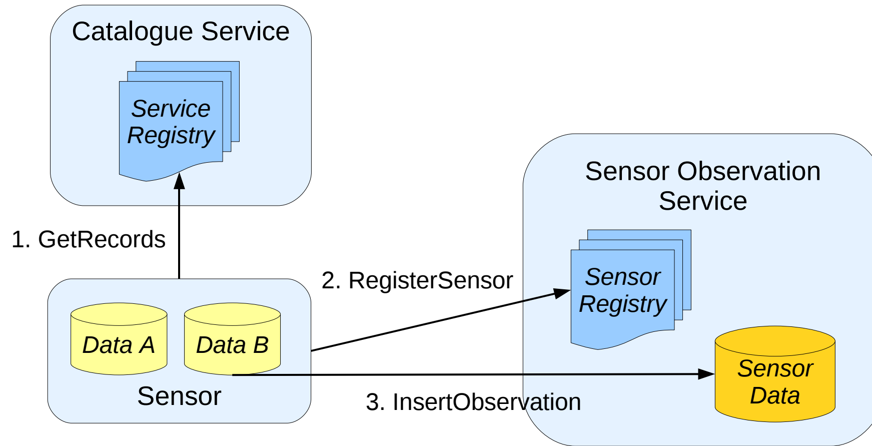

Figure 3.5. SOS data publishing process as described in [60]

The data publisher, usually a sensor, is querying the *Catalogue Service for Web* (CSW) for available *Sensor Observation Services*. After it found a suitable one it registers itself and is then able to publish data. In addition, the new sensor is automatically integrated in new *observation offerings*.

3.2.5.2 Data Consumption

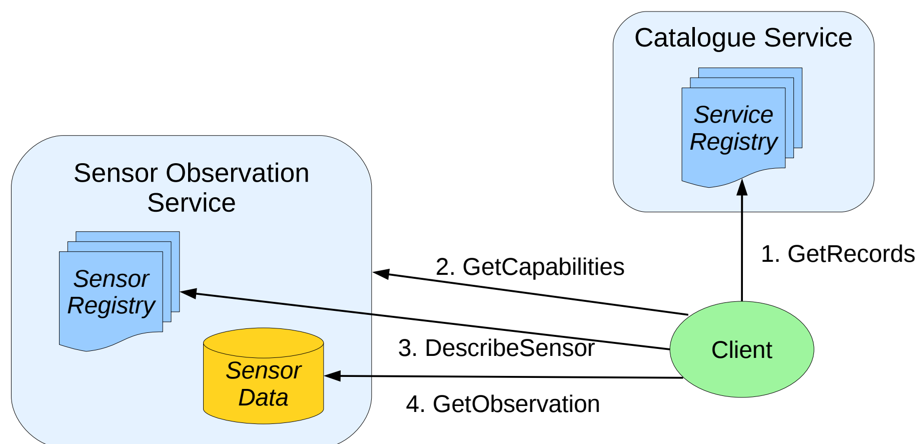

Figure 3.6. SOS data consumption process as described in [60]

The user has identified a need for a particular observation. The *Catalogue Service for Web* then provides *Sensor Observation Services*. Depending on the availability of metadata in the catalogue the user has either already selected a particular sensor or retrieves that information about a sensor from the *observation offerings*. More specific information about a particular sensor can be requested as well. Finally the necessary observations can be retrieved.

3.2.6 Sensor Alert Service (SAS)

In order to allow for an asynchronous alert reporting mechanism to notify users, the *Sensor Alert Service* (SAS) which is a candidate specification by Simonis and Echterhoff [78] has been designed. It proposes an event subscription and notification system that publishes sensor data based on specified criteria. An implementation of this service within the *Transportation Security SensorNet* is provided by the *Sensor Node* (see section 6.3.1).

3.2.6.1 Advertising Process

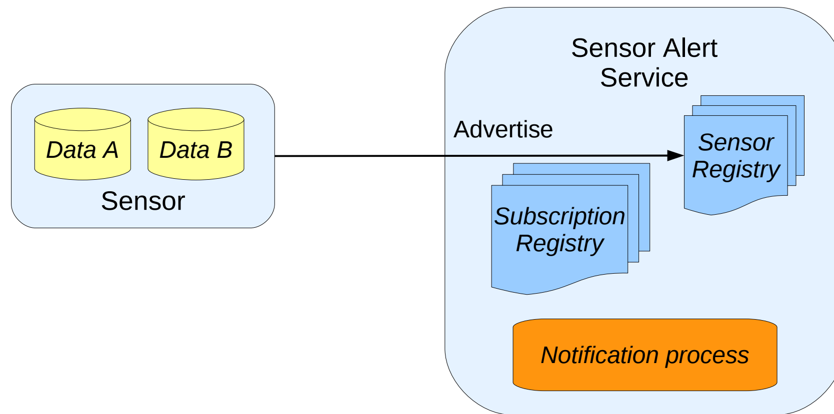

Figure 3.7. SAS advertising process described in [78]

The idea is that sensors advertise their data to the SAS. They then enter into an *advertisement* agreement to publish this data whenever it becomes available.

3.2.6.2 Notification Process

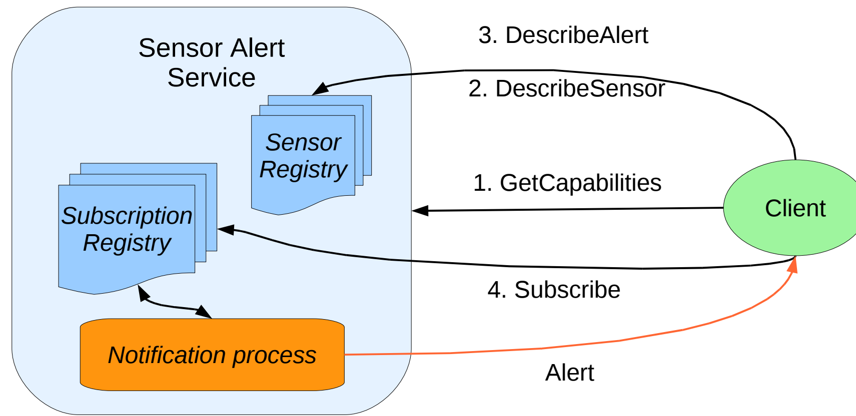

Figure 3.8. SAS notification process described in [78]

For the client, the service provides so-called *subscription offerings*. By choosing a particular offering the client subscribes to the sensor data that is defined by the offering. The SAS may modify or apply algorithms to the original sensor data which is in a way similar to applying an *observation pattern* as described in the O&M specification (see 3.2.4). The offerings are linked to *subscription criteria* that are used internally to match the sensor data that is published by the sensors to the individual clients that subscribed to them. The *Sensor Alert Service* additionally provides the client with means to retrieve all necessary information about the sensor itself and the alert data, especially the format.

The main difference between the *Sensor Observation Service* and the *Sensor Alert Service* is the way query results are provided. If the client is in need of particular sensor data on an ad hoc basis, it asks the *Catalogue Service for Web* for a matching SOS and queries the SOS in order to fulfill this need. The key aspect for the *Transportation Security SensorNet* is that the SOS only deals with providing the sensor data synchronously.

In case an alert system is needed to monitor whenever some sensor data reaches a

critical value the client does not directly act as the one querying for sensor data but rather the SAS. The client simply tells the SAS the necessary criteria for an alert through the means of a subscription. The SAS then monitors incoming sensor data and sends out notifications accordingly. This is done asynchronously without the client having to constantly query for data itself.

Chapter 4

Service Oriented Architecture

The main idea behind *Service Oriented Architecture* is that applications are defined as so-called *web services* which communicate with each other using a set of predefined protocols and standards. In terms of technologies, programming languages and platforms used, these *web services* can be completely independent systems. The key here is that their interfaces are specified using web service standards.

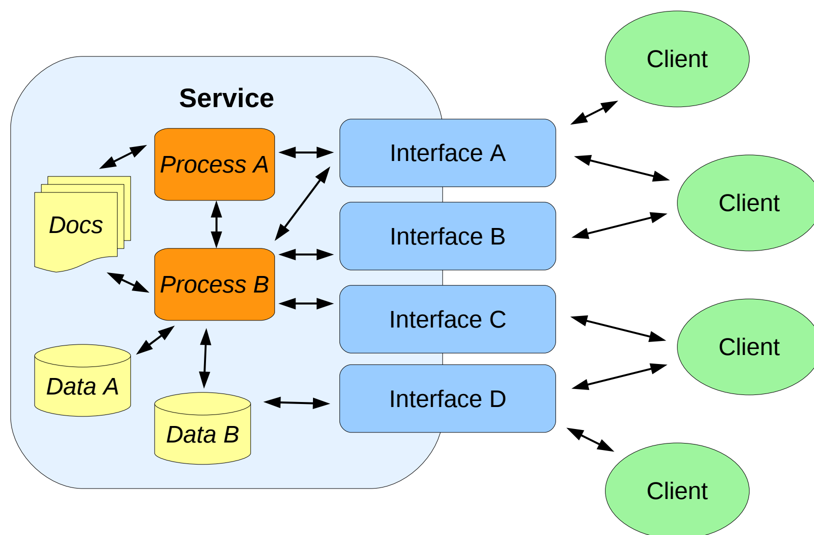

Figure 4.1. Service overview

The book "Service-Oriented Architecture: Concepts, Technology, and Design" by Erl

[26] describes these fundamentals in more detail. In particular the main components that make up a *Service Oriented Architecture* are outlined here.

A *message* represents the data that is required for a so-called *unit of work*. An *operation* covers the logic that processes these *messages*. The grouping of logic that handles related *units of work* is defined as a *service*. Additionally, the book defines a *process* as the business logic that combines several *operations* in order to complete a larger piece of work. Erl not only covers the basic concepts of SOA but also explains how they can be applied in the real world.

The principles of service orientation according to Erl [26] consist of the following:

- *Reusability* of logic, operations and services
- *Contracts* that define the service and information exchange
- *Loose coupling* of relationships with the goal of minimizing dependencies
- *Abstraction* that hides implementation logic of services
- *Composability* of services to form a more complex process
- *Autonomy* of logic within a service
- *Stateless* use of information in a service
- *Discoverability* of services

The SOA approach allows for what is called *loose coupling* between services. It defines each individual service in two ways. First, a service provides a specific functionality that could be for instance data processing or information storage. It is *autonomous* in doing so which means that it only dependents on itself for providing this functionality. Second, each service can be replaced by a different service that has the same interface. This *flexibility* allows the user to choose between services based on cost, performance or availability.

Because the functionality of an entire business process or system often depends on things like cost, availability and quality of a service, so-called *service contracts* can be

defined that allow for the combination of several services into a more complex system that adheres to specific constraints. This is often necessary given the highly dynamic environments of distributed, mobile, grid and peer-to-peer systems.

The *Service Oriented Architecture* is especially useful when dealing with legacy applications. Since the entire application or system can be "hidden" behind *interfaces*, the integration or *encapsulation* of it into current business models requires far less effort. Instead of converting or rewriting a complete application, web service interfaces for it can be defined so that it becomes usable as a web service.

As mentioned before, two of the most important concepts in a *Service Oriented Architecture* are *autonomy* and *flexibility.* In addition, SOA is very cost effective because web services by default are built in a *reusable* way and because of the idea that the most *optimized* service which provides the desired capabilities is chosen. Furthermore SOA is highly scalable since it allows for the easy integration of *broker*, *proxy* and *load balancing* scenarios.

The *statelessness* principle can be seen as a rather soft requirement since there are instances of when a service needs to maintain at least some sense of state. An example would be an "online time series data processor" that looks at a specific time window in order to find patterns. It needs to keep track of the data parts that make up the window and therefore information across multiple messages.

Most of the *Service Oriented Architecture* deployments make use of at least some sort service registry that contains metadata about services and allows them to be *discovered.* The most standardized approach is the use of *Universal Description, Discovery and Integration* (UDDI) (see section 4.4) although a recent investigation by Al-Masri and Mahmoud [1] found that of all the web services that were discovered 72% can be found using web search engines and only 38% are registered in UDDI Business Registries.

Since SOA itself is a concept, several so-called *Web Services* (WS) specifications have been developed that deal with the different aspects of it. One of the most notable standards is *WS-Addressing* (see 4.3.1) which describes how routing information can be

directly attached to messages. Another one is *WS-Security* (see 4.3.3) that provides *end-to-end* message integrity and confidentiality.

The benefits of SOA according to Newcomer and Lomow [64] and their relationship to the *Transportation Security SensorNet* can be summarized as follows:

- *Efficient development* through modularity because services can be implemented independently and solely on the basis of contracts and service descriptions. This allows for tasks and implementations of clients and web services in the TSSN to be split up among team members.
- *More reuse* since it is based on open standards, *loose coupling* and platform independence. The implementation is being made available to everyone and represents an reference example as to how web services can be utilized in sensor networks.
- *Simplified maintenance* in the sense that modifications to the implementation do not necessarily change the service because of abstraction and the fact that clients utilize the service only through interfaces. With the core of the web services in the TSSN being implemented, further development can be focused on specific aspects such as security and enhancements without breaking the current system.
- *Incremental adoption* since legacy applications can be "wrapped" into a service and single applications can be transitioned into the *Service Oriented Architecture* step-by-step. This is of importance to the *Trade Data Exchange* as it needs to acquire cargo and route information from already existing systems (see section 6.5).
- *Graceful evolution* because service interaction is only interface based and services can easily be replaced by faster, cheaper or more complex implementations. With new technology and hardware becoming available parts of the current implementation of the *Transportation Security SensorNet* may be upgraded easier.

4.1 Representational State Transfer (REST)

REST is one of the major steps away from *Remote Procedure Calls* (RPC) and towards scalable and distributed web service architectures. Even though *Service Oriented Architectures* most often make use of the more flexible SOAP and its surrounding *web*

services specifications, as is the case with the *Transportation Security SensorNet*, REST still plays an important role and is widely supported.

4.1.1 Traditional Definition

The *Representational State Transfer* (REST) concept was first introduced by Fielding [30]. It originally describes the following elements:

Data Elements A *resource* represents the main data element. It can be anything like information, data or image. A *resource identifier* is used to uniquely map to a particular *resource*. In order to know what the *resource* actually is, so-called *representations* are defined.

Connectors According to REST, all interactions between a client and server are stateless. This makes it highly scalable since the server does not need to keep state information. Additionally, multiple requests at the server can be handled at the same time. Furthermore, requests can be cached, transferred by intermediaries and reused.

The original definition of request (*in*) and response (*out*) parameters is the following. *In* parameters are control data, resource identifier and an optional representation. *Out* parameters consist of response control data, optional resource metadata and optional representation.

Components The *user agent* defines the source of the request and the *origin server* is used for so-called namespace resolution of the request.

4.1.2 Current Use

The architectural style of REST has been adapted for web services and is called *RESTful*. It is closely tied to HTTP. The idea here is that *resources* are made available through *Uniform Resource Identifiers* (URI). The *representation* in most cases is XML but can also be specified using so-called *Multipurpose Internet Mail Extensions* (MIME)

types. HTTP methods such as POST, GET, PUT and DELETE are used as operations for modifying the resources.

REST can be seen as an "old" standard for web services that is still in use mainly because it is easy to use and highly flexible. It has traditionally been used in environments where the communication parties need to transmit small and "relatively" simple messages. An advantage is that the requirements on bandwidth are usually smaller when using REST compared to other approaches. With the advent of *Asynchronous JavaScript and XML* (AJAX) it has seen an abundance of new application fields. This is mainly due to the fact that AJAX uses the RESTful web service approach to provide asynchronous interaction with a web server.

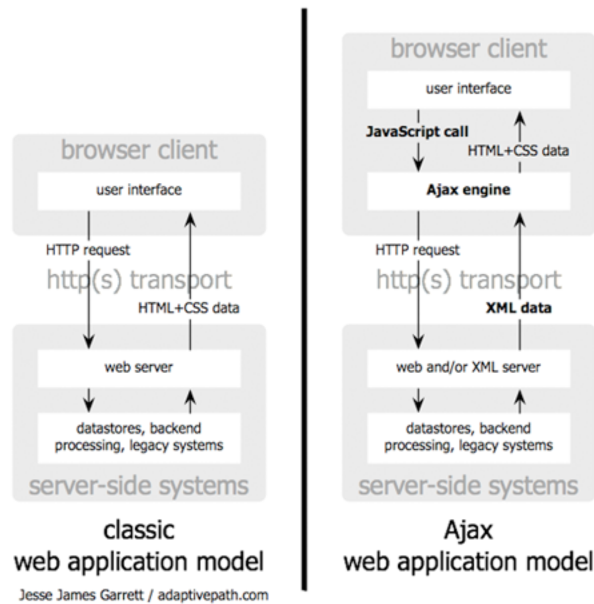

Figure 4.2. Traditional web applications and AJAX from Garrett [35]

Notable examples that use this approach are Google web applications such as *GMail*, *Maps* and *Docs*. Since AJAX is in use by entire industries, a standardization process as described by van Kesteren [83] has been started.

4.1.3 Further Development

Especially with recent developments in HTML5 as defined by Hyatt and Hickson [43] the flexibility of REST allows it to be used in more and more applications. The differences to HTML4 in terms of web application integration are significant. The enhancements described by van Kesteren [82] include *Application Programming Interfaces* (API) for playing video and audio, editing, drag and drop and more. An important addition is the ability for offline storage which allows web applications to replace desktop applications. The specification for this is defined by van Kesteren and Hickson [84]. This was currently only possible through extensions such as *Google Gears*.

All of this development and use of AJAX makes RESTful web services very appealing as they can easily be used from web applications. *Apache Axis2* which is the foundation of the *Transportation Security SensorNet* supports REST for accessing web services. This allows the use of TSSN web services in web applications without the need for additional development effort.

4.2 SOAP

The *Transportation Security SensorNet* makes use of SOAP as the default message exchange protocol. In the following SOAP is explained and a comparison with REST is made, which includes the reasons behind choosing SOAP over REST for the TSSN implementation.

According to Cabrera et al. [14] SOAP, which was formerly called *Simple Object Access Protocol*, provides “a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML”. It is a message standard for web services that aims to provide more flexibility and better interoperability than REST. In a comparison of SOAP to REST by Pautasso et al. [70] it was concluded “to use RESTful services for tactical, ad hoc integration over the Web (à la Mashup) and to prefer [SOAP in combination with] WS-* Web

services in professional enterprise application integration scenarios [, which is the case with the *Transportation Security SensorNet*,] with a longer lifespan and advanced QoS requirements". The reasoning for this, including a detailed description of SOAP, follows.

One of the main differences between SOAP and REST is complexity. SOAP and the so-called *web services* (WS) specifications built around it allow for the most complex scenarios while maintaining a relatively simple basic format. REST on the other hand is usually used in point-to-point communications and the exchange of simple XML. Furthermore, one of the major drawbacks of REST is that it is tied very closely to HTTP transport whereas SOAP is not.

SOAP is independent from platforms and programming languages and allows different transport protocols to be used as so-called *bindings*. According to Nielsen et al. [67] a binding represents a "formal set of rules for carrying a SOAP message within or on top of another protocol (underlying protocol) for the purpose of exchange". This includes describing how the protocol provides the necessary services to transport SOAP messages, how errors are handled and most importantly what *features* are provided by the underlying protocol. Although HTTP remains the most common binding, the extension of binding possibilities was one of the main enhancements to the original SOAP 1.1 specification by Box et al. [11], the other being the more clearly defined use of XML schemas.

SOAP enables extensive end-to-end message routing which is important in dealing with firewalls. The *WS-Addressing* specification (see 4.3.1) describes this in more detail. Another important aspect is security, which is available as *WS-Security* (see 4.3.3) for instance. Overall SOAP is simple in its default form yet very extensible.

4.2.1 Message format

The basic format according to the SOAP 1.2 specification by Nielsen et al. [66] defines an *Envelope* that includes a mandatory *Body* and an optional *Header* as seen in figure 4.3. The *Header* contains control information in the form of so-called *header*

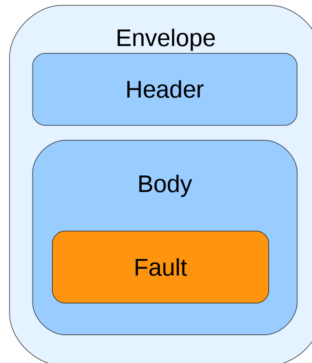

Figure 4.3. SOAP message format

blocks. These *blocks* can be used for routing or to pass processing directives to services. The *Body* is the mandatory *payload* of the message and contains the data that is being transmitted. Listing 4.1 shows the basic format that is used by all SOAP messages:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <soapenv:Envelope
3    xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
4    <soapenv:Header>
5     ...
6    </soapenv:Header>
7    <soapenv:Body>
8     ...
9    </soapenv:Body>
10 </soapenv:Envelope>
```

Listing 4.1 SOAP message format example

4.2.2 Faults

Apart from the basic message format, the specification also describes the *Fault* format that is common for all messages containing error information.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <soapenv:Envelope
3    xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
4    <soapenv:Header>
5     ...
```

```
6     </soapenv:Header>
7     <soapenv:Body>
8       <soapenv:Fault>
9         <soapenv:Code>
10          <soapenv:Value>soapenv:Receiver</soapenv:Value>
11        </soapenv:Code>
12        <soapenv:Reason>
13          <soapenv:Text xml:lang="en-US">Transport error: 404
                Error</soapenv:Text>
14        </soapenv:Reason>
15        <soapenv:Detail/>
16      </soapenv:Fault>
17    </soapenv:Body>
18  </soapenv:Envelope>
```

Listing 4.2 SOAP Fault message example

The *Fault* consists of three parts. The *Code* part classifies the error into a predefined set dealing with version mismatches, so-called *mustUnderstand* header blocks, data encoding, and sender and receiver issues. The *Reason* allows the *Fault* to be described in terms of an error message and supports multiple languages. The *Details* part may contain application specific information.

4.2.3 Further development

The SOAP 1.2 Primer by Lafon and Mitra [48] includes references to several enhancements of the standard. The main reason for this is the potential for performance problems and the need for binary data transport in SOAP.

The *XML-binary Optimized Packaging* (XOP) specification by Mendelsohn et al. [58] defines the use of *MIME Multipart/Related* messages provided by Levinson [51] to avoid encoding overhead that occurs when binary data is used directly within the SOAP message. XOP extracts the binary content and uses URIs to reference it in the so-called *extended part* of the message. An abstract specification that uses this idea is the *Message Transmission Optimization Mechanism* (MTOM) by Nottingham et al. [68].

Another extension of this is *Resource Representation SOAP Header Block* (RRSHB) as described by Gudgin et al. [37] that allows for caching of data elements using so-called *Representation header blocks*. They contain resources that are referenced in the SOAP *Body* which might be hard to retrieve or simply referenced multiple times. Instead of having to reacquire them over and over again, a service may choose to use the cached objects which speeds up the overall processing time.

4.3 Web Service Specifications

The web services in the *Transportation Security SensorNet* make use of *web service specifications* in order to address topics such as addressing, event notification and security in a uniform and standardized way. The specifications that are relevant to the TSSN are described in the following sections while their implementations are addressed in chapter 6.

4.3.1 WS-Addressing

The *WS-Addressing* core specification by Gudgin et al. [39] and its SOAP binding by Gudgin et al. [38] defines how message propagation can be achieved using the SOAP message format. Usually the transport of messages is handled by the underlying transport protocol but there are several advantages of storing this transport information as part of the header in the actual SOAP message. For example, it allows the routing of messages across different protocols and management of individual flows and processes within web services.

WS-Addressing uses so-called *EndPointReferences* which are a collection of a specific address, reference parameters and associated metadata that further describe its policies and capabilities.

Addressing Header The header fields defined by the specification are the following:

- *To* which represents the destination of the message

- *From* contains the source, a so-called *EndPointReference*
- *ReplyTo* specifies that in case of a response, a message is supposed to be sent to this *EndPointReference*, which might be different from the *From* field
- *FaultTo* defines the *EndPointReference* for the fault message in the case of an error
- *Action* identifies the purpose of the message, in particular the web service operation, and is the only required field
- *MessageID* uniquely identifies every message
- *RelatesTo* references the *MessageID* of the request message in request-response message exchanges; the relationship can also be specified explicitly by defining a so-called *RelationShipType*

4.3.2 WS-Eventing

In order to allow for subscriptions to web services, the *WS-Eventing* specification has been defined by Box et al. [12]. It describes the process of establishing subscriptions as well as how the subsequent publications are delivered to the subscribers. The specification relies on *WS-Addressing* for the routing of messages. The two main components of a subscription in this specification are the *Subscribe* and the *SubscribeResponse* message. After subscriptions have been created, publications will be sent out accordingly.

Subscribe The client that wants to subscribe to a particular web service needs to define the following:

- The *Action* field of the *WS-Addressing* header is set to `http://schemas.xmlsoap.org/ws/2004/08/eventing/Subscribe`
- *ReplyTo* is the *EndPointReference* that receives the response to this subscription request
- A *MessageID* that uniquely distinguishes multiple requests from the same source
- *EndTo* defines an *EndPointReference* that is used when the subscription ends unexpectedly

- *Delivery* contains the *EndPointReferences* that are to receive the publications
- An *Expires* field that defines the expiration time of the subscription
- *Filter* that by default defines an *XPath* expression as the *Dialect*, but could be any form of expression that is applied to potential publications in order to filter them

SubscribeResponse The response to a subscription request is generated by the so-called *subscription manager*. It sends back a message with these fields:

- The *Action* field of the *WS-Addressing* header is set to `http://schemas.xmlsoap.org/ws/2004/08/eventing/SubscribeResponse`
- *RelatesTo* specifies the subscription request that this is a response to
- *SubscriptionManager* that contains its own *Address* and the unique *Identifier* for the subscription
- An *Expires* field that defines the expiration time of the subscription

The *WS-Eventing* specification also offers message constructs for the renewal, status retrieval and unsubscribing of subscriptions. Additionally a so-called *subscription end* message is automatically generated by the service that publishes information in order to notify subscribers of errors or other reasons for it being unable to continue the subscription.

It has to be noted that without additional specifications like *WS-ReliableMessaging* the delivery of publications is based purely on best effort and is not guaranteed.

4.3.3 WS-Security

The *WS-Security* specification as described by Lawrence et al. [49] deals with the many features needed to achieve so-called *end-to-end* message security. This provides security throughout message routing and overcomes the limitations of so-called *point-to-point transport layer security* such as *HTTPS*. Furthermore, the specification aims to

provide support for a variety security token formats, trust domains, signature formats and encryption technologies.

The two main aspects of security are the following:

Confidentiality This means that the information contained in a message is only *available* or *visible* to entities that are authorized. *Encryption* provides this *confidentiality* for messages.

Integrity The *integrity* of a message is maintained if it has not been modified on the way from one entity to another. Applying a *signature* enables the receiver to check if the message has been altered during the transmission.

These aspects among others are defined as part of the SOAP message. Most of the security provided by *WS-Security* is specified in header blocks of the SOAP header. The following represent its important parts:

Tokens The specification supports various types of security tokens directly:

- *User Name Tokens* for username and password pairs
- *Binary Security Tokens* which essentially are *X.509 certificates* or *Kerberos tickets*
- *XML Tokens* described by the *Security Assertion Markup Language* (SAML) or *Extensible Rights Markup Language* (XrML)
- *Encrypted Data Tokens* in which case the token itself is encrypted as well

A different way of specifying these tokens is to reference them. This is useful because at times the security tokens are specified in a different part or even completely outside of the SOAP message. The *WS-Security* specification defines the following most commonly used:

- *Security Token References* which can be used to wrap around non-standard implementations

- *Direct References* for using a *URI* as a reference point
- *Key Identifiers* that uniquely identify security tokens
- *Embedded References* which directly include tokens instead of pointing to them

Signatures In order to ensure the integrity of messages so-called *signatures* can be applied by the sender. The receiver is then able to check the validity of the message using this *signature*. Important properties that can be conveyed in the SOAP header using *WS-Security* are:

- *Signed Info* that defines the algorithms to be used for so-called *namespace transformations* and proper ordering of signature and encryption elements (for example, sign an encrypted message or encrypt a signed message)
- *Signature Value* containing the actual digital signature
- *Key Info* that defines the type of the signature used

The specification also allows for various forms of so-called *Signature Confirmations* to be sent out as responses to the initial messages. They can provide additional security in certain scenarios.

Encryption *WS-Security* provides great flexibility when it comes to the actual encryption of the message. It supports *header*, *body* as well as *individual block* encryption. The reason it is able to do this lies in the fact that it makes use of the following two constructs:

- *Reference List* that points to the *Encrypted Data* elements which, since they are completely independent of each other, enables different encryption techniques and keys to be used
- *Encrypted Key* which allows symmetric keys to be embedded in the message and is used for encrypting the SOAP *header*

Security Timestamps Most of the time, security policies need to make sure that it is possible to change previously distributed keys and force the ones that are not to be used anymore to *expire*. For this purpose *WS-Security* supports so-called *Security Timestamps* that can be attached to the message. Two fields are defined:

- *Created* describes the time when the message was serialized for transmission
- *Expires* defines the point in time when the security applied to this message is no longer considered valid

It has to be noted that *WS-Security* does not provide any methods for time synchronization which may potentially limit the effectiveness of *Security Timestamps* in certain scenarios.

A white paper by Chanliau [15] extends the definition of security to areas such as secure message delivery, metadata and trust management. It references the web service specifications that have been introduced to deal with these aspects of security in more detail.

4.4 Service Directory

Because web services by default are *loosely coupled* there has to be a way of for them to establish connectivity with each other. In general there are two different approaches for doing this. First, let a service A directly know about the presence and address of a service B that it seeks to contact. This can cause a variety of problems as all the addresses have to be managed manually which leads to scalability issues. Second, define a so-called service registry that keeps track of available services and acts as a mediator between clients and services.

The latter approach has been realized using the *Universal Description, Discovery and Integration* (UDDI) specification as described by Bellwood et al. [5] and is being used in the *Transportation Security SensorNet*. UDDI provides a XML based service registry and directory that provides the following:

- *Information* on web services and their categorizations, so-called *metadata*
- *Discovery* of web services based on specific criteria
- *Connection information* such as required security aspects, provided transports and operation parameters that describes in detail how to connect to a service
- *Alternatives* in case of a failure of one service

A paper by Bellwood [4] describes the main focus areas of version 3 of the UDDI specification:

Multi-registry Environments In order to allow for the logical separation of service registries, UDDI supports so-called *root* registries that act as parents to *affiliates*. Furthermore the replication of registries is supported. Whenever a web service publishes information to a registry it is able to either provide a key as a "suggestion" or have the registry automatically assign a new unique key to the information.

The UDDI also provides means for transferring the custody and ownership from one so-called *business entity* to another. This is an important aspect when it comes to handling cargo in the transportation industry. The *Transportation Security SensorNet* is able to provide this functionality by using an implementation of the UDDI.

Subscriptions Apart from the basic search interface that the UDDI provides, the specification describes two different subscription models:

- *Active* subscriptions check whether or not specified criteria of the previously defined subscriptions match current entries in the registry. This is done synchronously, meaning only when a request has been issued.
- *Passive* subscriptions allow for the registry to store so-called *asynchronous callbacks* for subscriptions. The registry checks against its entries on its own and independently of the initial subscriber. Whenever it finds a match it sends out a notification.

The *Transportation Security SensorNet* provides support for *active* subscriptions transparently to clients and web services . Web services automatically register with the UDDI when they are started. Clients are then able to use them by just specifying the type of service that they need. An according web service is then automatically handed to them using an underlying *active* subscription to the UDDI.

Policies The UDDI supports a complex policy abstraction model which main components are:

- *Rules* that define actions for when a set of particular conditions is met
- *Decisions* which comprise of a set of rules
- *Information access and control* that defines what kind of functionality can be provided with regard to inquiries, publications, subscriptions and others.

Policies are also used to enforce security although the specification acknowledges that only the *integrity* part of it is defined. This is partly due to the fact that the UDDI is supposed to be a public registry and lookup directory. For this particular purpose, the focus is more on the reliability of entries which can be ensured using *signatures*.

Advanced policy management that is able to restrict access to web services and even single operations as well as encrypted message exchanges are especially important when it comes to the scalability and production deployment of the *Transportation Security SensorNet*. Within the TSSN policy information as of summer 2009 is not yet in the UDDI but kept directly in the clients and web services.

4.5 Web Services Description Language (WSDL)

In order to allow services to interact and collaborate they need to share information about interfaces, operations, parameters, data elements and means of contact with each other. This has been addressed by the *Web Services Description Language* (WSDL). The most widely used and supported version is WSDL 1.1 as described by Christensen

et al. [17] but the newer version 2.0 provides a cleaner and more extensible specification. According to Liu [54] the main improvements include the following:

- Renaming of some elements to express their intentions in more detail (*definitions* to *description*, *port type* to *interface*, *ports* to *endpoints*)
- Reorganizing the messages constructs that were previously disparate (definition is now part of *types*)
- Operations contain messages in a particular *Message Exchange Patterns*
- Introduction of more *Message Exchange Patterns*, see section 4.6
- Allows for *interface* inheritance

Overall WSDL 2.0 is a clear evolution and in many ways a lot cleaner but also far less supported than WSDL 1.1. The *Transportation Security SensorNet* uses WSDL 2.0 as it aims to provide an open framework that is extensible in the future. Figure 4.4 provides an overview of the core components of WSDL 2.0.

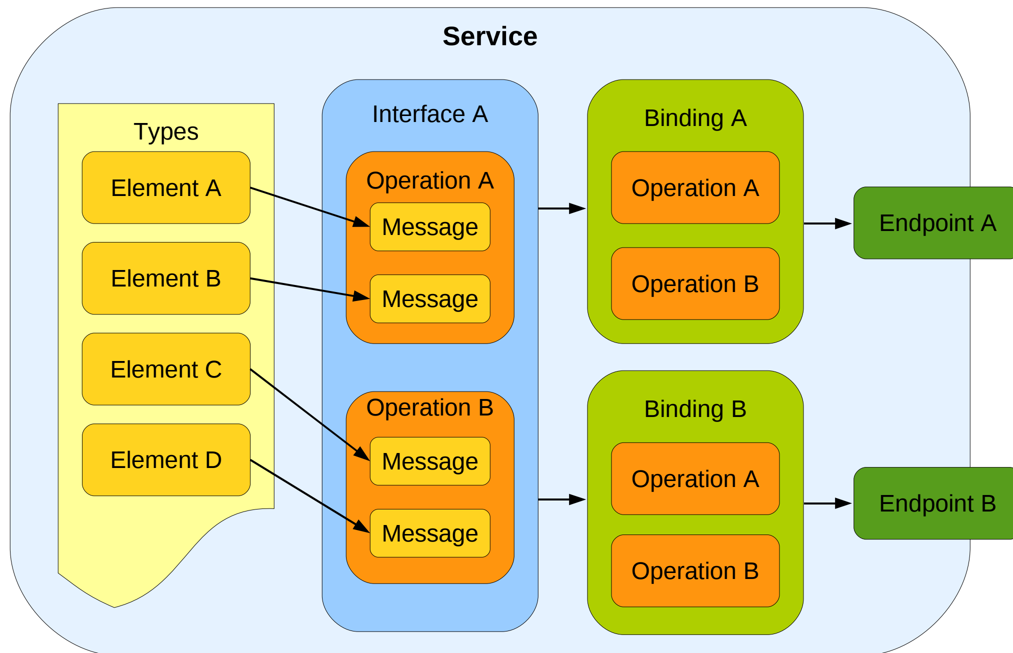

Figure 4.4. WSDL 2.0 overview

Elements that are being used by the service are defined in the *types* section. They essentially make up the *messages* of an *operation*. A group of operations then defines a so-called *interface*. A *binding* specifies the transport format for these *interfaces*. Finally the network addresses for the *bindings* are exposed as *endpoints*. Hence, a *service* can be seen as a group of *endpoints* that allow clients to use the functionality provided by the service through clearly defined *interfaces* and specified transport formats.

Interfaces from other services may be included using `<include schemaLocation="..." />` in which a location pointing to a valid WSDL file must be specified. The import namespace must be the the same as the one for the WSDL that it is included into. In order to be able to use different namespaces while still maintaining modularity, WSDL files can also be imported using `<import namespace="..." schemaLocation="..." />` and specifying a target namespace. Both of these directives are modeled after XML Schema *includes* and *imports* by Bray et al. [13].

The following is a more detailed description of the *Core Language* part of the WSDL 2.0 specification by Moreau et al. [59]. Another introduction to the main components is provided in the *Primer* by Booth and Liu [9]

4.5.1 Description

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <description
3   xmlns="http://www.w3.org/ns/wsdl"
4   xmlns:a="http://www.sample.com/elementBook"
5   xmlns:tns="http://www.sample.com/library"
6   xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
7   targetNamespace="http://www.sample.com/library">
8   ...
9 </description>
```

Listing 4.3 WSDL Description example

The *description* acts as the root for a WSDL 2.0 document that contains all other elements. It takes care of defining the target namespace and aliases for namespaces. In

the example the default namespace is set to WSDL which specifies that the document is a WSDL document. The `xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"` references the SOAP binding for WSDL. The other namespaces that are mentioned refer to the library example which was introduced in section 3.1.1.

4.5.2 Types

```
1  <types>
2    <xsd:import
3      namespace="http://www.sample.com/elementBook"
4      schemaLocation="elementBook.xsd"/>
5    <xsd:schema
6      targetNamespace="http://www.sample.com/library">
7      <xsd:element name="bookList">
8        <xsd:complexType>
9          <xs:element ref="a:book" minOccurs="0" maxOccurs="
               unbounded"/>
10       </xsd:complexType>
11     </xsd:element>
12     <xsd:element name="user" type="xsd:string">
13     <xsd:element name="error" type="xsd:string">
14   </xsd:schema>
15 </types>
```

Listing 4.4 WSDL Types example

XML schema elements for the service are defined in the *types* part of the WSDL. Additionally schema *includes* and *imports* are supported. The elements can then be referenced by messages later on. The code in listing 4.4 imports the *book* element from the library example which is used in the *bookList* describing a list of *books*. Additionally elements called *user* and *error* are defined in the same library namespace. Since *user*, *error*, *book* and *bookList* are fully described by the WSDL, they can now be used by both the service and the client. The service might have known about them already but by using WSDL it makes them available to clients and other services in a standardized way.

4.5.3 Interface

```
1 <interface name="LoanInterface">
2   <fault name="UserIsUnknown" element="tns:error"/>
3   <operation name="getBooks" pattern="http://www.w3.org/ns/
        wsdl/in-out">
4     <input messageLabel="Request" element="tns:user"/>
5     <output messageLabel="Response" element="tns:library"/>
6     <outfault ref="tns:UserIsUnknown">
7   </operation>
8 </interface>
```

Listing 4.5 WSDL Interface example

Since version 2.0, WSDL allows for multiple *interfaces* to be defined and supports inheritance between them. An *interface* includes a group of *operations* that consist of *messages*. The *operations* must be associated with a *Message Exchange Pattern* (MEP). For more information see section 4.6. According to the MEP that is used, *input* and *output* messages are specified. They reference elements from the *types* part of the WSDL. Note that since the MEP is *In-Out* in which a *fault* would replace the response in case of an error, an *outfault* is specified. In the example an *operation* is defined that allows a user to retrieve a list of the books that were loaned.

4.5.4 Binding

```
1 <binding name="LibrarySOAPBinding"
2   interface="tns:LoanInterface"
3   type="http://www.w3.org/ns/wsdl/soap"
4   wsoap:version="1.2"
5   wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/
        HTTP/">
6   <fault ref="tns:UserIsUnknown" />
7   <operation ref="tns:getBooks"
8     wsoap:action="tns:getBooks" />
9 </binding>
```

Listing 4.6 WSDL Binding example

Each *binding* is able to reference the *interfaces* that were previously described in the WSDL. It associates them with a specific format and protocol that is then used to transmit messages. A *binding* can also be defined on a *operation* or even *message* level. This however is not as commonly used. The *binding* that is specified in listing 4.6 associates the *LoanInterface* with SOAP 1.2. According to the SOAP binding part of the WSDL specification by Orchard et al. [69] the *type* attribute is used to define SOAP whereas the version and the protocol (SOAP 1.2 over HTTP) are specified using the SOAP namespace. Note that for the *operation* in the example a so-called *SOAP action* is set which allows SOAP messages received by the service to be pointed to the according web service *operation*.

4.5.5 Service

```
1 <service name="LibraryService"
2   interface="tns:LoanInterface">
3   <wsdl2:endpoint name="LibrarySOAPEndpoint"
4     binding="tns:LibrarySOAPBinding"
5     address="http://www.sample.com/library/soap" />
6 </service>
```

Listing 4.7 WSDL Service example

The last part in a WSDL document is providing an *endpoint* that specifies a network address at which the service can be reached. The same *interface* could potentially have several different *bindings*. For each of them an *endpoint* has to be defined in order to be able to use them. Hence, a service essentially exposes the defined *interfaces* and their *bindings*.

4.6 Message Exchange Patterns

In order to manage the most complex communication scenarios so-called *Message Exchange Patterns* (MEP) have been defined. They are specified for each *operation* in the WSDL document (see section 4.5.3). The basic patterns are explained in detail in

the following sections.

The *Message Exchange Patterns* are in large part based on so-called *fault propagation rules* which specify what happens in case of an error. SOAP uses them to clearly define how error messages are sent from clients to services and in between services. This allows both parties to be aware of their error handling responsibilities. The following *fault propagation rules* are defined:

Fault Replaces Message Whenever an error occurs, the message that was supposed to be sent is replaced by a fault.

Message Triggers Fault In case of an error a fault is sent back to the sender of the message. The message itself is not replaced though.

No Faults No fault is created at any time. If something goes wrong only the party that encounters the error knows about it, nobody else.

A combination of these *fault propagation rules* and the messages that are exchanged between client and service make up the *Message Exchange Patterns*. Note that whenever two services exchange messages, one is always acting as the client. Hence the MEPs depict only client-service interactions.

In the Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts by Orchard et al. [69] the following *Message Exchange Patterns* are defined:

4.6.1 In-Only

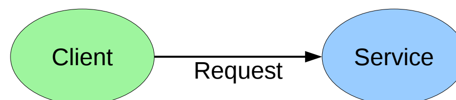

Figure 4.5. In-Only message exchange pattern

Messages in this pattern are one way only. It is defined by `http://www.w3.org/ns/wsdl/in-only`. No *Faults* are sent. This can be seen as a fire-and-forget approach.

4.6.2 Robust In-Only

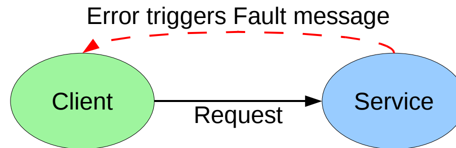

Figure 4.6. Robust In-Only message exchange pattern

This message pattern is identified by `http://www.w3.org/ns/wsdl/robust-in-only` and extends *In-Only* in the sense that it creates *Faults* when errors occur.

4.6.3 In-Out

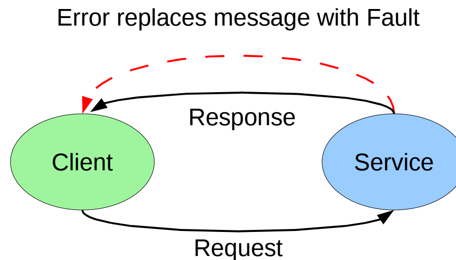

Figure 4.7. In-Out message exchange pattern

The most common *Message Exchange Pattern* is defined by `http://www.w3.org/ns/wsdl/in-out`. It specifies a request-response model where in the case of an error a *Fault* replaces the response message. Services often act as data or application providers where clients issue their requests and the service responds with either the requested data or the result of the processing that it provided.

Additional MEPs have been defined by Lewis [53]:

4.6.4 In-Optional-Out

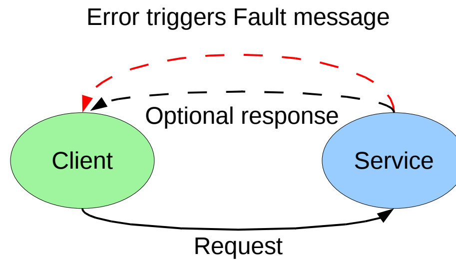

Figure 4.8. In-Optional-Out message exchange pattern

The pattern identified by `http://www.w3.org/ns/wsdl/in-opt-out` makes the response of an In-Out message exchange optional. It can be used for control messages where responses are often status messages and the assumption is that only errors are of importance in which case a *Fault* is generated.

4.6.5 Out-Only

Figure 4.9. Out-Only message exchange pattern

`http://www.w3.org/ns/wsdl/out-only` defines a *Message Exchange Pattern* that is mostly used in asynchronous communication environments and subscriptions. It is assumed that the client registered or subscribed with the service and that the service sends *notifications* back to the client at a later time. This version does not send out *Faults*.

4.6.6 Robust Out-Only

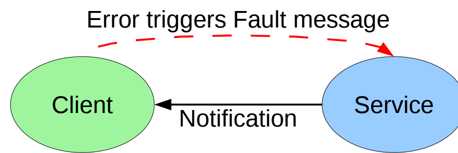

Figure 4.10. Robust Out-only message exchange pattern

In a similar fashion to *Out-Only* this pattern which is defined by `http://www.w3.org/ns/wsdl/robust-out-only` sends out messages to a client. The difference is that in case of an error it creates a *Fault*.

4.6.7 Out-In

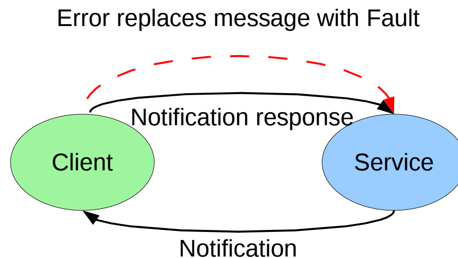

Figure 4.11. Out-In message exchange pattern

Being the reverse of the *In-Out* pattern `http://www.w3.org/ns/wsdl/out-in` describes a request-response communication that is initiated by the service. In subscription scenarios for instance the response can be seen as an acknowledgment that the notification has been received by the client. A *Fault* replaces the *notification response* in case of an error.

4.6.8 Out-Optional-In

An extension of the basic *Out-In* message exchange the `http://www.w3.org/ns/wsdl/out-opt-in` pattern provides in a sense a *selective acknowledgment* of the *notifi-*

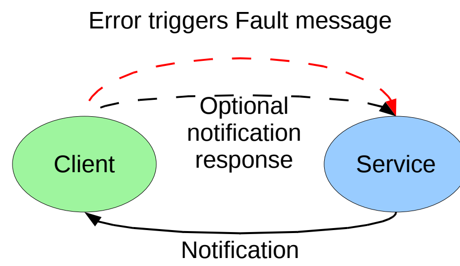

Figure 4.12. Out-Optional-In message exchange pattern

cation that was sent out. It allows for robustness by being able to send *Faults*.

Chapter 5

Related Work

In the following sections related work that is relevant to various aspects of the *Transportation Security SensorNet* such as *Service Oriented Architecture*, web services, communication models, the *Open Geospatial Consortium* specifications and sensor networks is analyzed.

5.1 Microsoft - An Introduction to Web Service Architecture

The paper by Cabrera et al. [14] about web service architectures gives an excellent introduction to what eventually evolved into the *Service Oriented Architecture*. The key ideas described are the following:

Message only approach The only thing that is exchanged between services are messages. This principle avoids potential problems that could occur when functionality embedded in different components becomes too intertwined. It also ensures flexibility and interoperability between services. The services and messages are defined in *Web Service Description Language* (WSDL) and then transported using SOAP. How the messages are sent from one service to the other is specified is so-called *Message Exchange Patterns* (MEP). Additional properties like security or reliability are standardized in the *Web Service* (WS) specifications.

Flexible protocol stack In order to provide support for a variety of systems, SOA needs a protocol layering model that ranges from general purpose to highly specific. The modular architecture of SOAP describes a protocol that consists of “building blocks”. This ensures two things. First, you only pay for what you actually use and second, it can be complemented or extended at any time.

Autonomy of services As described before, services aim to embed their functionality and be independent from each other. The extensibility of SOAP allows for the so-called *evolution* of a web service, also known as versioning. The *mustUnderstand* annotation can be provided to signal that the recipient of a message needs to know how to handle the SOAP header specifics. In order to maintain this autonomy and at the same time allow complex business models to be used, services must form *trust relationships* with the services that they use. The reason for this is that essentially there is no apparent difference between two services that provide the same interface. Businesses must know that they can trust their data to be handled confidentially by the service that they choose. Without this trust paradigm there are many potential security concerns. Another point mentioned is the move from a centralized system to a more federated approach using SOA which is able to deal better with the entire message exchange model.

Managed transparency In order to be flexible enough to support different programming languages and platforms, *Service Oriented Architectures* use a service abstraction layer model. The implementation and internal processes of a service are completely hidden from its client. The only thing visible are the so-called *interfaces* that are provided. Every service in SOA is described using the *Web Service Description Language* (WSDL). The WSDL file of a service defines its capabilities and provides a standard for the interoperability of clients and services.

Protocol-based integration The interaction between services should be restricted to the communication using a predefined protocol only. This allows for applications to be self-contained and independent of their implementation language and system. As described before it provides this by using abstraction layering through interfaces and the use of metadata. The *Service Oriented Architecture* follows the “nothing is shared” approach. This *autonomy* is the reason why it can provide the aforementioned *flexibility*.

Cabrera et al. [14] outline concepts that led to the implementation of *Service Oriented Architectures* and development of the *web services* specifications that surround them and are used by the TSSN. A lot of the main approaches have been standardized in various committees and organizations by now but were only in the early stages when this paper first came out.

5.2 Adobe - Service Oriented Architecture

An Adobe technical paper by Nickul et al. [65] outlines general architecture approaches that can be taken when transitioning business processes to the *Service Oriented Architecture*. It mentions a widely used technology called the *Enterprise Service Bus* (ESB) that provides a standardized means of communication for all services that connect to it. For the *Transportation Security SensorNet* this is of importance when it comes to asynchronous communication as the *Java Message Service* (JMS) uses queues that are on the ESB for message exchanges (see section 6.1.6 and 8.2).

In the example that is provided, three business processes all have some sort of login, authentication, name and address management. The problem that occurs most often in scenarios like this is how to synchronize states across all three processes. Using SOA this common task is bundled into a service that all three processes connected to the ESB can use which improves efficiency and greatly decreases required maintenance.

In addition to the basic *Request-Response*, several other *message exchange patterns* that go beyond the standardized ones (see section 4.6) are described:

5.2.1 Request-Response via Service Registry (or Directory)

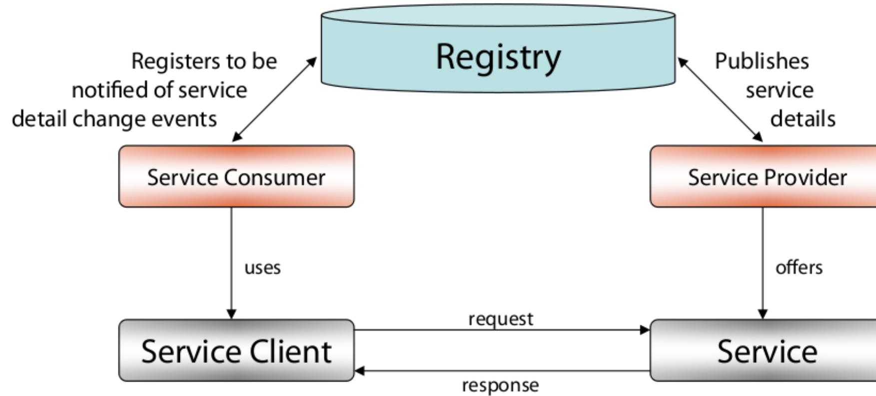

Figure 5.1. Request-Response via Service Registry (or Directory) message exchange pattern from [65]

A so-called *registry* keeps track of service metadata. The *service provider* is responsible for updating it whenever a change occurs and the *service consumer* subscribes to the *registry* for any of these changes. The metadata that is provided is then used to configure a *service client*. Hence, the client can issue *requests* and receive *responses*.

The *Transportation Security SensorNet* essentially uses a very similar approach with the UDDI. Web services automatically register with the UDDI when they are started and clients are able to use specific services by looking them up in the UDDI.

5.2.2 Subscribe-Push

The *service consumer* uses the client to subscribe to specific *events* as shown in figure 5.2. Whenever the service encounters one of these *events* it pushes *notifications* back to the client or other endpoints that were defined in the subscription. This approach is conceptually similar to what is described by the *WS-Eventing* specification (see 4.3.2).

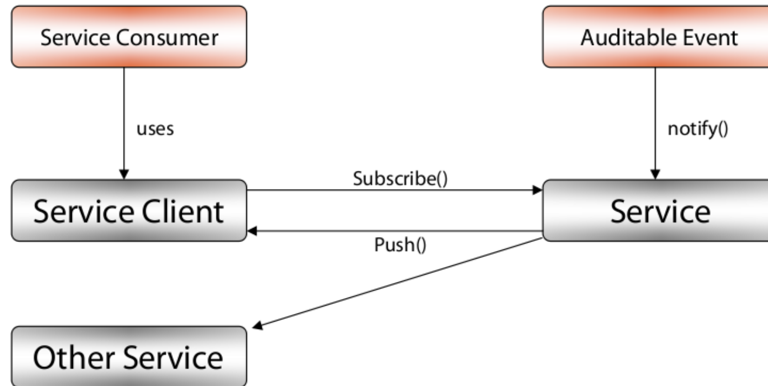

Figure 5.2. Subscribe-Push message exchange pattern from [65]

5.2.3 Probe and Match

When there is no *service registry* available, a client has to discover usable services on its own. By using multicast or broadcast messages it *probes* until suitable services respond with a *match*. A hybrid approach could use the registry for a candidate set of services to *probe*. This pattern does not scale very well because it is highly dependent on the available bandwidth.

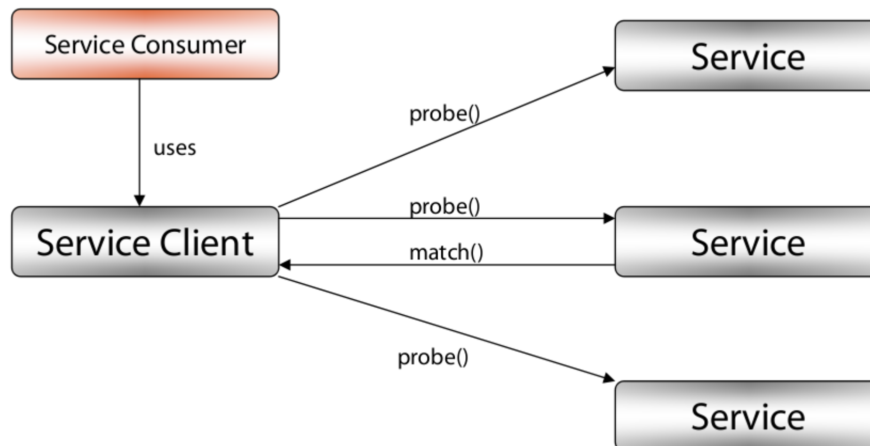

Figure 5.3. Probe and Match message exchange pattern from [65]

5.3 Open Sensor Web Architecture

An approach to implement the proposed standards of the *Sensor Web Enablement* that are described in section 3.2.1 is outlined by Chu et al. [19]. A more detailed definition of the system and its core services is provided in the thesis by Chu [18]. The system is called *NICTA Open Sensor Web Architecture* (NOSA) and is focusing on the combination of sensor networks and distributed computing technologies. For this purpose the following four layer model is defined:

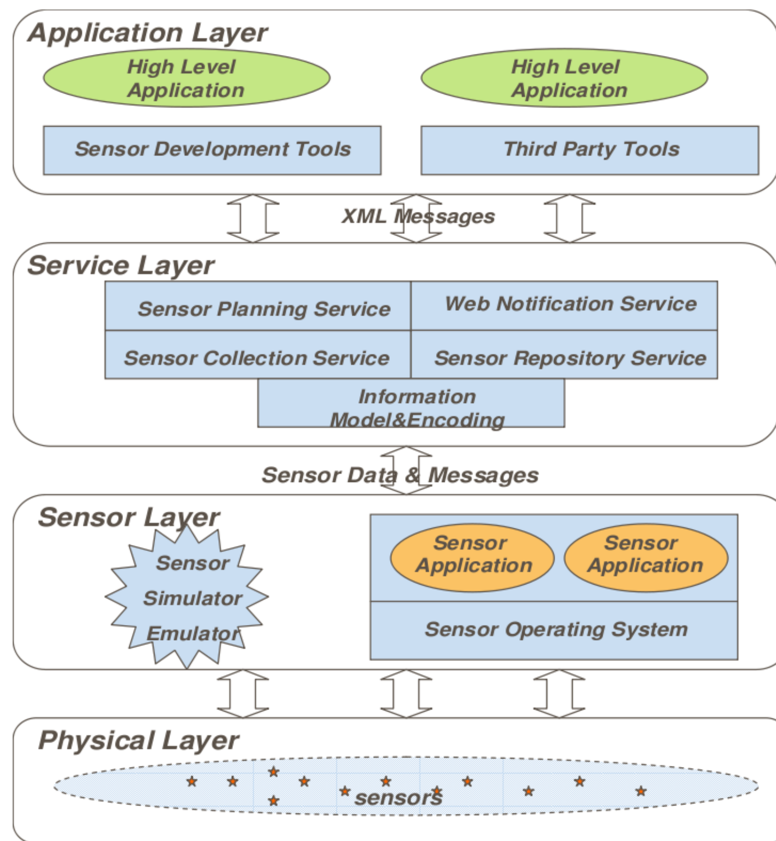

Figure 5.4. NOSA layer overview from [19]

Physical layer The sensors can be contacted using standardized means such as ZigBee and other IEEE 802.15 protocols. They can also interact with each other.

Sensor layer This layer provides the main sensor applications that are built on top of the *Sensor Operating System*. This operating system is called *TinyOS* (see Levis et al. [52]) and is widely used in low power sensor environments. It deals with the control, monitoring and retrieving of data from the sensors in the *physical layer*. The sensor layer acts as the basis for services that make use of this data.

Service layer Web services that are compliant to the ones defined in the *Sensor Web Enablement* are part of this layer. They provide a uniform and standardized way of dealing with sensors and the data that they gather.

Application layer Applications that want to interact with the underlying service infrastructure are provided with development and third party tools that to make use of the open standards web service interfaces.

The *Transportation Security SensorNet* uses a similar approach but has some significant differences. The goal of both implementations is to integrate a sensor network into a web services architecture using open standards. NOSA uses a sensor application that is tightly integrated into the *Sensor Operating System* and then provides sensor data and control to web services in a non-standard format. TSSN on the other hand implements sensor management and monitoring functionality inside a single service, the *Sensor Node* (see section 6.3.1) and allows different sensors to be "plugged in". This allows other services to use standard web service interfaces and SOAP messages in order to access sensors.

Furthermore, the web services used by NOSA are implemented manually according to the *Open Geospatial Consortium* specifications which causes them to be limited as not everything that is specified is also implemented. In contrast, the TSSN uses automatic code generation (see section 6.1.1.4) that enables it to use all OGC specifications. Since their elements and interfaces are generated the only thing that has to be implemented is functionality. This approach significantly reduces development efforts.

5.4 Globus - Open Grid Services Architecture

Globus is an architecture that is based on grid computing. It focuses on providing capabilities as services in a grid environment using standard interfaces and protocols. An initial paper by Foster et al. [32] gives an overview of the architecture and design decisions. In particular, Globus supports "local and remote transparency with respect to service location and invocation" and "protocol negotiation for network flows across organizational boundaries". Its service approach is similar to the *Service Oriented Architecture* that is used by the *Transportation Security SensorNet*. Additionally, security concepts that work inside a grid are applicable to SOA and vice versa.

Services Functionality in the Globus defined architecture can be achieved using so-called *grid services* which utilize standard interfaces in order to provide the following:

- *Discovery* of capabilities and the services using standardized *naming* conventions
- *Lifetime management* which includes *dynamic service instance creation* and *concurrency control* of data and processes
- *Notification* of clients and subscribers in case of events
- *Manageability* of service relationships and maintenance
- *Upgradability* in terms of versioning to ensure compatibility between services
- *Authorization* to enforce access control

Protocols The two important aspects regarding protocols that Globus deals with are:

- *Reliable service invocation* ensures that the exchange of messages which is the core of service interaction is reliable. This allows for the means of communication necessary in a grid computing environment.
- *Authentication* addresses the need to verify the identity of clients and services in the grid

The current architecture of Globus as shown in figure 5.5 is still based on the same principles that were initially described by Foster et al. [32]. The combination of custom components and web services components provides an architecture for security, data management, execution management, information services and a common runtime in a grid environment. In the following, the approaches taken are described in detail.

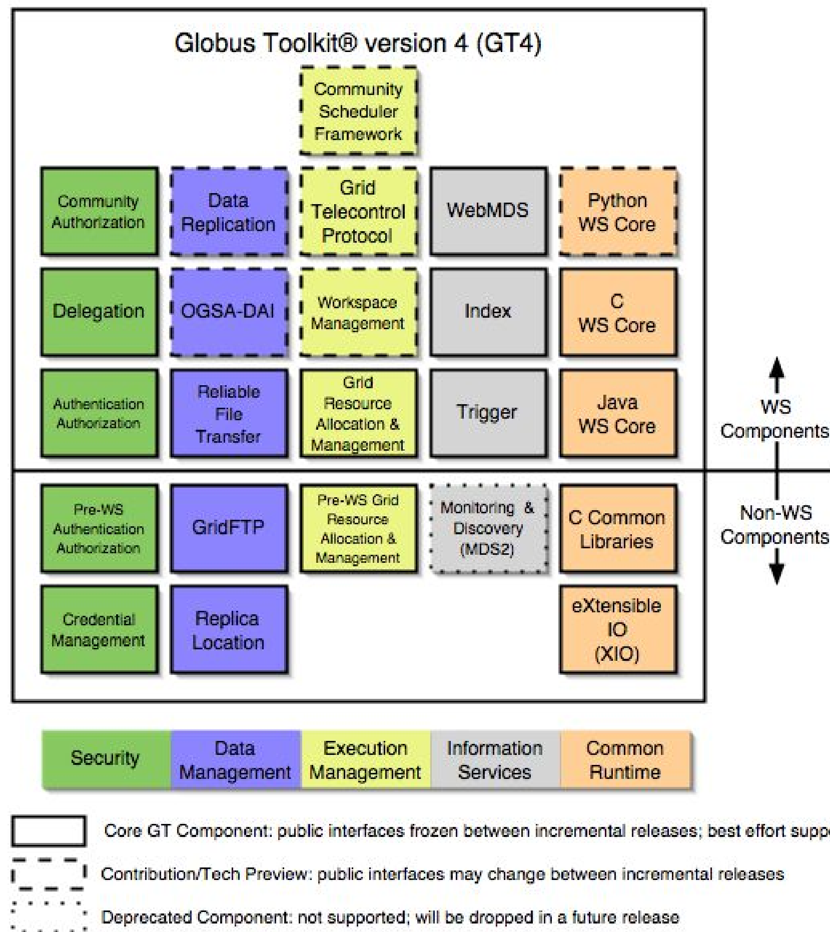

Figure 5.5. Globus Toolkit overview from `http://www.globus.org/toolkit/about.html`

Service model All entities are represented as services that provide standard interfaces over which their capabilities are accessible. Invocation of a particular functionality and the interaction between services is performed using message exchanges. These

grid services utilize *web services* specifications for their interfaces and implementations. Since a service in Globus is both, dynamic and stateful, it is assigned a so-called *grid service handle* (GSH) to uniquely identify it. In order to support the *upgradability* concept, a particular version of the service is identified by a *grid service reference* (GSR).

Factories Services in the grid that are able to create new service instances are called *factories*. Whenever a new service is created, it is automatically assigned a new *grid service handle*.

Service lifetime management Globus allows task specific services to be instantiated. These so-called *transient* services perform a predefined task and terminate upon its completion. It is also possible to associate a particular lifetime with a service. Note that services that need more time in order to complete their task may request a *lifetime extension*. An important aspect regarding the lifetime management is time synchronization across all services. In order to achieve this, Globus uses the *Network Time Protocol* (NTP).

Handles and references A so-called *HandleMap* is used to map *grid service handles* to specific *grid service references*. This is necessary since *grid service references* have a defined lifetime and may expire. The *HandleMap* ensures that it only returns valid *grid service references* and not ones that are already terminated. This among other things also allows detailed access control all the way down to the operation level. For this to work, every service needs to register with a so-called *home HandleMap*. The *grid service handle* is constructed in a way that it automatically references this *home HandleMap* to ensure scalability.

Service data and service discovery Every *grid service* is associated with so-called *service data* which in Globus is a collection of XML documents that describe the capabilities of the service. By default each service provides this data using the mandatory

FindServiceData interface. The overall system contains a *registry* that contains references to each individual service. It provides a *Registry interface* that is used to register *grid service handles*. Since the availability of services can change, the *registry* has to adapt. In order to deal with these dynamics in the grid environment, registrations must be refreshed otherwise they expire after a specified time.

Notification Globus provides an asynchronous notification system that is based on subscriptions. A client acts as a so-called *NotificationSink* that issues a request for particular events to the so-called *NotificationSource*. In the case of events, notifications are then pushed from the *source* to the *sink*.

Change management *Web services* interfaces in the grid environment are uniquely named in order to provide manageability. Whenever a significant portion of the interface or implementation is changed, a new unique name must be provided.

In contrast to the *Transportation Security SensorNet*, Globus makes use of web service specifications in some of its components but also provides custom implementations and interfaces as for service discovery and notifications. The TSSN uses web services specifications and *Open Geospatial Consortium* standards almost exclusively which ensures standards compliance and compatibility. For service discovery the UDDI (see section 4.4) is used and for notifications *WS-Eventing* (see section 4.3.2).

5.5 Service Architectures for Distributed Geoprocessing

A research article by Friis-Christensen et al. [34] deals with the integration of *Open Geospatial Consortium* specifications. It outlines the implementation of an application that analyzes the impact of forest fires using web services. The purpose of the application is to assess the damage inflicted by fires based on land cover data for a particular area. The previous solution looked like figure 5.6.

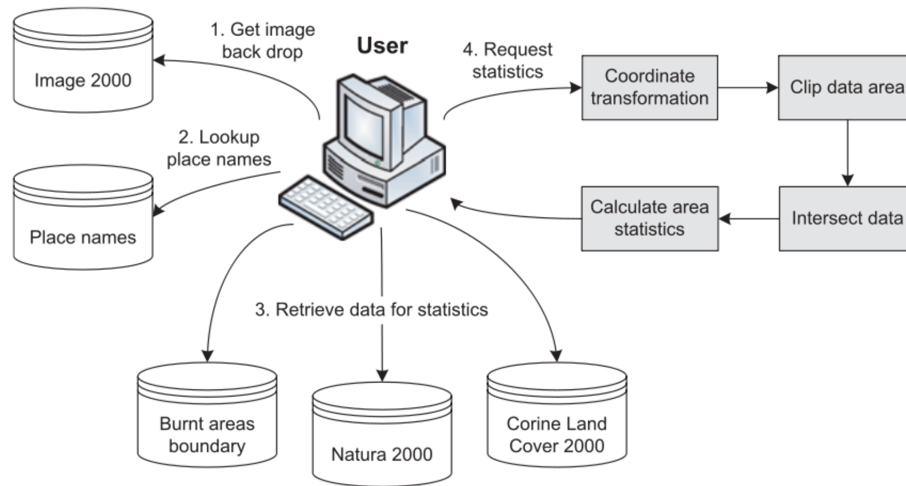

Figure 5.6. Forest fire application from [34]

Friis-Christensen et al. [34] discuss advantages and disadvantages of their improved, web services based implementation and outline potential solutions for problems that they discovered.

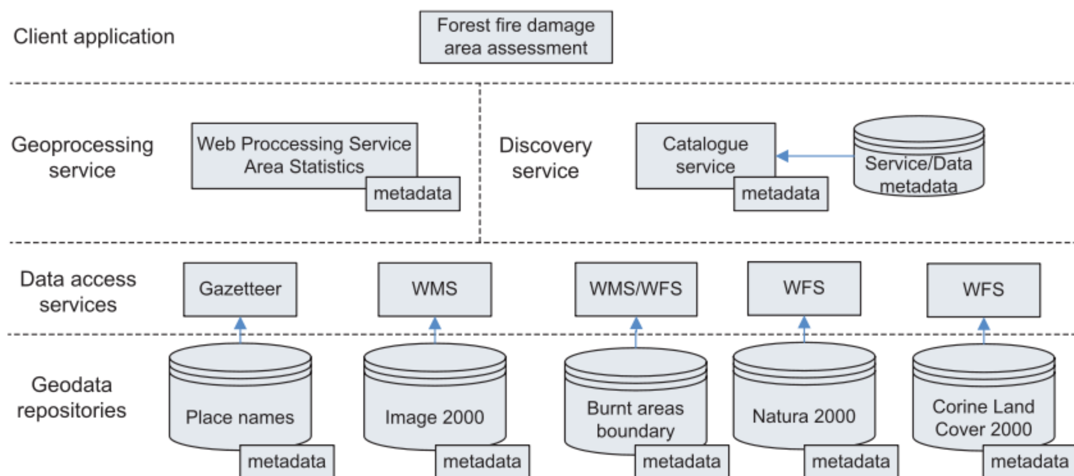

Figure 5.7. Forest fire web services architecture from [34]

Architecture The main focus is the transition from a client application to a flexible web services architecture using *Open Geospatial Consortium* specifications. As shown

in figure 5.7 the components include multiple data sources that are made available through data access services like the *Web Map Service* and the *Web Feature Service*. A geoprocessing service performs the analysis of the data and provides it to a client. Furthermore a discovery service serves as the registry for all services and their metadata.

The general process is described as follows:

1. Retrieve a map
2. Select a time and area of interest
3. Search for data source *masks* that deal with burnt areas
4. Search for target data *masks* that serve as the basis for the assessment of fire damage
5. Execute the process which retrieves the masked features, performs calculations and returns the desired statistics
6. Display statistics

Statistics Service This is the implementation of a *Web Processing Service* (WPS) according to the OGC specifications. Apart from the general *getCapabilities* interface, a *describeProcess* interface is defined which is used to explain how data is handled within a particular process and what functionality the process provides. The *execute* operation is used to start the specified process with previously defined filters, so-called *masks*, as the parameters. During the processing, the statistics service uses these masks to collect *features* from the data sources.

Mapping and Feature Services These services provide the relevant data such as satellite imagery and statistics either in its entirety or through the application of specified *masks*.

Catalogue The catalogue serves as a service registry and allows searching for services and features based on *title*, *bounding box* and *time of interest*.

Client In the implementation that is described in the paper, the client application is browser based. It uses a combination of client (AJAX) and server (JSP) based technology to display maps and the calculated fire damage statistics

The prototype implemented uses synchronous communication in between services. The problem in this case is that the actual processing can take quite a long time. In the future the authors want to transition to an asynchronous communication model that is similar to the OGC *Web Notification Service*.

In addition, it is pointed out that even though standardized interfaces allow for a combination of services which provides flexibility, the transport of high volumes of data is often not feasible in geoprocessing scenarios which can lead to highly specialized but not very reusable services.

The implementation described by Friis-Christensen et al. [34] is interesting in the sense that it exclusively uses specifications from the *Open Geospatial Consortium* which makes it compatible to other *Geographical Information Systems*. The *Transportation Security SensorNet* aims to be OGC compliant as well but includes specifications that deal with sensor networks such as the *Sensor Observation Service* and the *Sensor Alert Service*, something that this forest fire web service architecture does not even address.

5.6 Web Services Orchestration

A paper that specifically deals with the problem of reusability of services and so-called "next generation challenges" was written by Kiehle et al. [47]. The idea here is to increase transparency and reusability by splitting processes into smaller more reusable processes and utilizing a work flow management system called *Web Services Orchestration*. This is especially important for the integration of the *Transportation Security SensorNet* into systems used in the transportation industry. Its modular design and architecture allow single components to be reused and and information flows to be created.

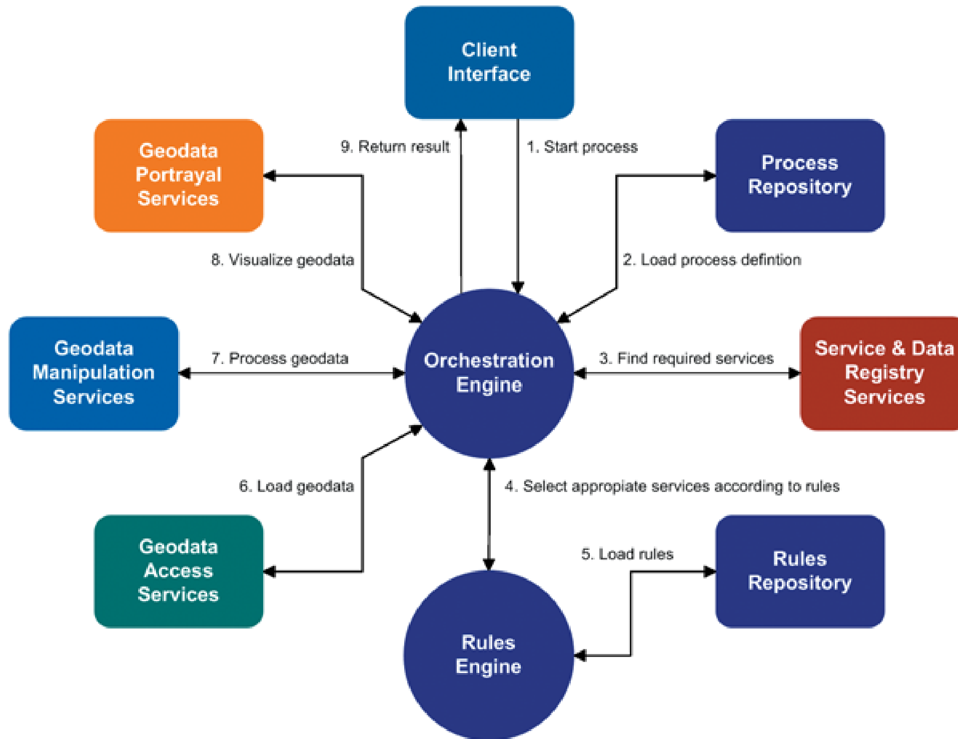

Figure 5.8. Web orchestration framework from [47]

The *Web Processing Service* specification describes how services can be arranged and combined into so-called *service chains* that form a process. Two alternatives are commonly used in order to achieve this. A *Web Processing Service* can be setup to combine and "encapsulate" other individual web services and therefore provide the desired abstraction. However, the best way to define work flows is using the so-called *Business Process Execution Language* (BPEL). BPEL enables complex *service chains* as shown in figure 5.8 to be defined without the need for custom and potentially not reusable *Web Processing Services* that just "encapsulate" services.

5.7 Summary

The related work addresses the following key technologies that play an important part in the *Transportation Security SensorNet*:

Service Oriented Architecture The development of the *Service Oriented Architecture* and its web services specifications has come a long way but is still far from over. Even though specifications exist, organizations and businesses often implement components that are similar to the specification but not compliant. As discussed before, this is the case for service discovery and notifications in Globus. Two common reasons behind this are the following. First, the specification may be available but there are hardly any reference implementations that can be used. Second, extensions to the specification that are necessary for a particular implementation or in a specific environment such as the grid are not covered by the standard.

Open Geospatial Consortium The specifications by the *Open Geospatial Consortium* are often complex and there is significant development effort necessary to implement the elements, interfaces and functionality they define. Automatic code generation as described section 6.1.1.4 and used by the *Transportation Security SensorNet* can facilitate their implementations but is not used very often.

Sensor Networks The implications on communication models that sensor networks have, in particular asynchronous message exchanges, are often ignored in web service architectures. As seen in NOSA, the focus is on the implementation of a subset of OGC standards for a particular sensor network, but the link to an overall *Service Oriented Architecture* seems to be missing.

It is evident that current systems seem to lack the combination of SOA, OGC specifications and sensor networks. The *Transportation Security SensorNet* combines all these technologies and bridges the gap between implementations that just deal with SOA and OGC specifications and systems that use OGC standards in sensor networks.

Chapter 6

Design & Architecture

6.1 Overview

This chapter describes the architecture of the *Transportation Security SensorNet* (TSSN). It provides an in-depth discussion of design aspects and the implementation.

6.1.1 Service Oriented Architecture

> "Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains." MacKenzie et al. [55]

Building a "Service Oriented Architecture for Monitoring Cargo in Motion Along Trusted Corridors" makes sense. According to a study by the Delphi Group [36], companies that collaborate usually request compliance for the following standards: XML 74%, J2EE (Java) 44% and SOAP 35%. The architecture used for the implementation of the *Transportation Security SensorNet* utilizes all three technologies by separating functionality into *web services*. This allows for high flexibility and is very cost effective (see chapter 4).

Haas et al. [40] early on proposed various *models* for web service architectures. The *Message Oriented Model* focuses on message relations and how they are processed. An

approach that centers around resources and ownership is the so-called *Resource Oriented Model*. The *Policy Oriented Model* defines constraints and focuses on security and quality of service. Ideas from all these models have been combined with the *Service Oriented Model* into what has become the *Service Oriented Architecture*. Of the proposed models it has been the most widely implemented.

A book that provides an excellent overview of Java and *web services* is written by Kalin [45]. Note that the *Service Oriented Architecture* by definition is programming language and platform independent. It is built on the basis of requests and responses and the independence of so-called *web services*. The choice to use Java for the implementation was made because the *Transportation Security SensorNet* is built on top of previous research on the *Ambient Computing Environment for SOA* by Searl [76] which is written in Java.

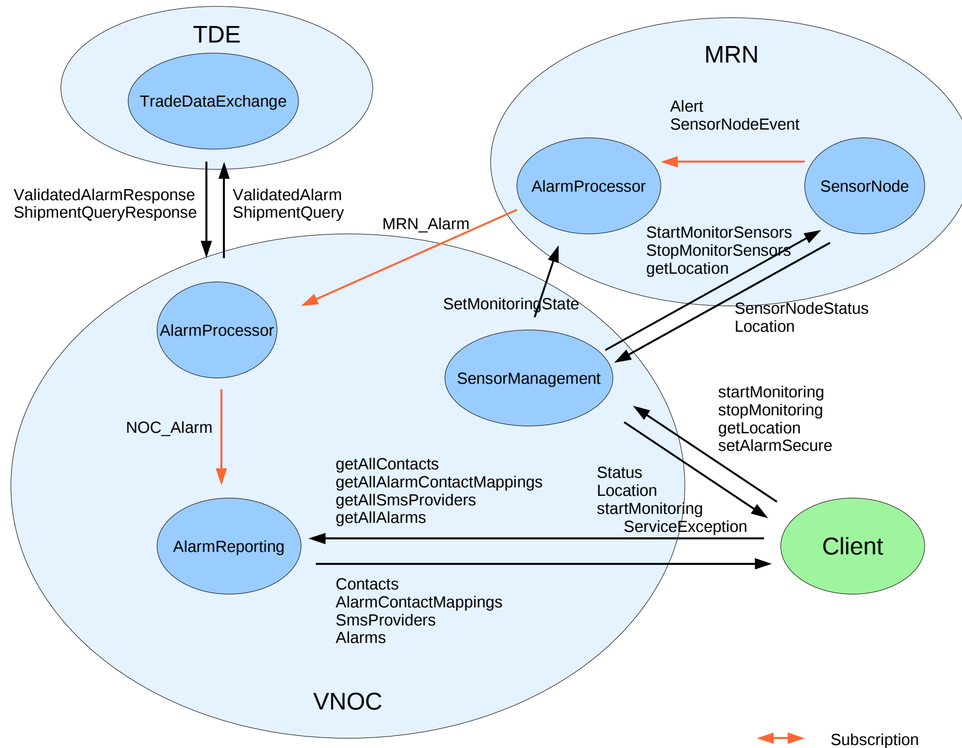

Figure 6.1. Service message overview

The main components of the *Transportation Security SensorNet* are sensor management and alarm notifications. An overview of the services and relevant message exchanges is shown in figure 6.1.

The so-called *Trade Data Exchange* (TDE) (see section 6.5) provides shipment, route, logistics and relevant cargo information. It is managed externally and used by the system only through its specified interface. The *Virtual Network Operation Center* (VNOC) (see section 6.4) is responsible for the processing of sensor data and alarms. One of the major capabilities that it provides is alarm notification. The *Mobile Rail Network* (MRN) (see section 6.3) deals with the actual management of sensors. *Web services* at the *Mobile Rail Network* capture sensor data from the sensors and "preprocess" that data. A detailed description of each individual service is provided later in this chapter.

The architecture consists of web services that are separated into so-called *service clouds*. These *service clouds* represent the different geographically distributed locations (e.g. Overland Park, KS; Lawrence, KS and on a moving train) where services are deployed and are shown in figure 6.2.

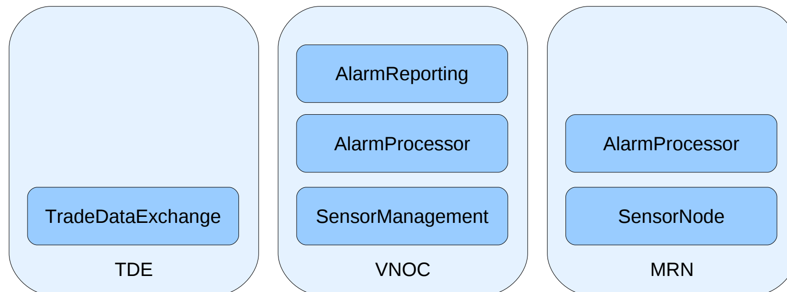

Figure 6.2. Service cloud

The *web services* are developed according to the *web service* specifications and the standards provided by the *Open Geospatial Consortium*. This means that they aim to be standards compliant. Since the OGC specifications are at times very complex, the *Geography Markup Language* for example defines over 1000 elements, the basis for

the framework was implemented using custom interface definitions first and adding the OGC ones later. This enabled fast prototyping and testing of the system.

An analysis of geospatial problems and their potential solutions is done by de Smith et al. [24]. Among other things it is pointed out that using standards, in particular the specifications provided by the *Open Geospatial Consortium*, greatly increases interoperability and allows for the development of distributed systems that are more flexible than commonly used *Geographic Information Systems*.

The following sections explain in-depth the approaches and technologies used in the implementation of the *Transportation Security SensorNet* that represents a "Service Oriented Architecture for Monitoring Cargo in Motion Along Trusted Corridors".

6.1.1.1 Ambient Computing Environment for SOA

The infrastructure described by Searl [76] called *Ambient Computing Environment for SOA* forms the basis of the implementation of the *Transportation Security SensorNet*. It provides a complete SOAP stack using *Apache Axis2* and a variety of other useful programs that assist in the development of a *Service Oriented Architecture*.

The *Ambient Computing Environment for SOA* [76] deals with multiple ownerships and federations that provide *web services*. In particular it covers the following aspects:

- *Service Discovery* across different federations
- *Authentication* of clients and services
- *Authorization* of clients and services
- *Subscriptions*

The implementation of the capabilities provided is based on *Apache Axis2* and the *web service* specifications. It is explained in detail in the following sections.

6.1.1.2 Apache Axis2

Apache Axis2 is a software stack that allows the development and running of *web services* and clients. Its architecture as described by Chinthaka [16] consists of the following main components:

AXIs Object Model (AXIOM) AXIOM is an XML object model that aims for high performance while requiring low amounts of memory. The idea behind it is the application of a so-called *pull parser*. This allows objects to be built from XML only up to the information that is needed by the user while the rest of it is *deferred*.

The advantage of this is that the memory that an object requires is significantly reduced. Furthermore, since the entire object model does not have to be constructed before information can be retrieved, which is the case in the *DOM parser*, this approach also increases performance.

Extensible Messaging Engine As can be seen in figure 6.3, Axis2 provides a very modular architecture that allows for a variety of different implementations of *web services* as long as they adhere to certain specifications.

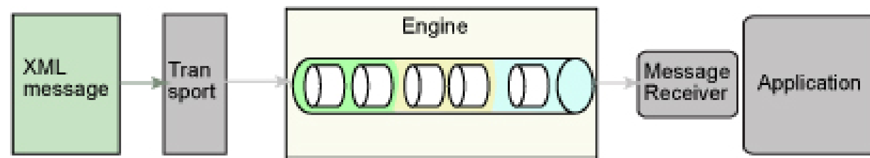

Figure 6.3. Axis2 extensibility from [16]

A variety of transports such as HTTP, SMTP, JMS and TCP can be used for message exchanges. Inside the *engine* each message goes through so-called *phases* that are part of the *piping model* which is used to implement *Message Exchange Patterns* (see section 4.6). Inside these *phases* messages can be modified, filtered or processed. The advantage of doing this inside a *phase* is that it applies to all messages. This allows for service independent processing implementations. The *message receiver* will then be re-

sponsible for handing over the actual message to the service implementation accordingly. They also take care of *synchronous* and *asynchronous* message communication.

Context Model Axis2 provides a hierarchical context model that distinguishes between the following levels:

- *Configuration* of Axis2
- *Service Group* which is a collection of *services*
- *Service* which contains several *operations*
- *Operation* that consists of *messages*
- *Message* that is sent or received

These contexts are important in the implementation of *web service* specifications such as *WS-Security* and *WS-Policy*. It means that these specifications can be applied on a level basis which provides great flexibility.

Pluggable Modules In order to provide even more flexibility and to make the implementation of *web service* specifications easier to use, Axis2 provides so-called *modules*:

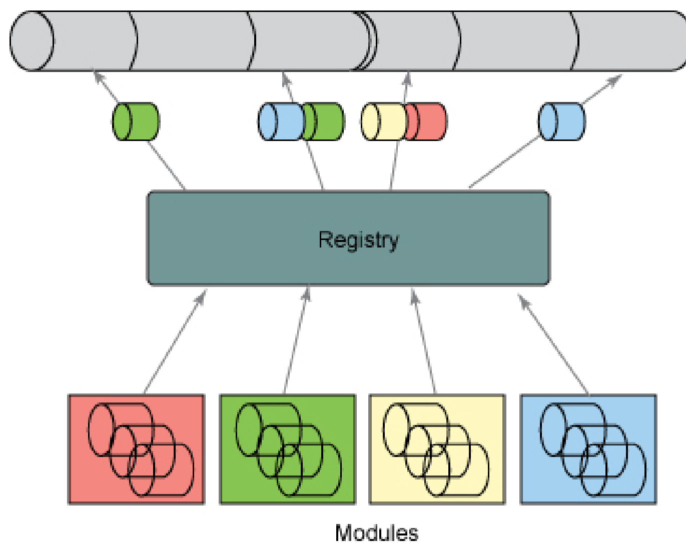

Figure 6.4. Axis2 modules from [16]

These allow an implementation of message processing that is common and useful for many *web services* to be shared. *Modules* can also be *engaged* or *disengaged* on the following levels:

- *System* which means that every service makes use of the module such as *WS-Addressing*
- *Service* which useful for *WS-Eventing*
- *Operation* that for example allows fine grained security using *WS-Security*

More information about the modules that are used in the *Transportation Security SensorNet* see section 6.1.4.

Data Binding Since a majority of data processing, element definitions and interface specifications are in XML, Axis2 provides a variety of so-called *data binding frameworks* such as XMLBeans [33], Java Architecture for XML Binding (JAXB) [29] and JiBX [80]. In addition, the *Axis2 Data Binding* (ADB) can be used, which due to its tight integration with Axis2 is highly performant. For instance, every object contains a so-called *factory* that is able to transform XML into the specific object and vice versa.

As part of this thesis further development was done by the author on this *data binding* to support a full range of *Open Geospatial Consortium* specifications such as the *Sensor Observation Service*, *Sensor Alert Service* and most notably the *Geography Markup Language*.

Several changes to the initial version of Axis2 were made in order to either fix bugs or support more functionality. In particular the build structure was adapted to work better with the *Transportation Security SensorNet* development. It makes extensive use of *Apache Ant* for the automatic generation of elements from their respective XML schema definitions, the compilation of Java classes and the deployment of *web services* and clients

6.1.1.3 SOAP

Service Oriented Architectures make use of SOAP as a flexible message format. The *Transportation Security SensorNet* does the same since *web service* specifications can easily be integrated and applied to SOAP messages.

An in-depth discussion of SOAP can be found in section 4.2.

6.1.1.4 WSDL

All services in the *Transportation Security SensorNet* are defined using the *Web Services Description Language* (WSDL) version 2.0. An in-depth introduction is provided in section 4.5. This section explains how the combination of WSDL files and XML schemas make up the foundation of a web service.

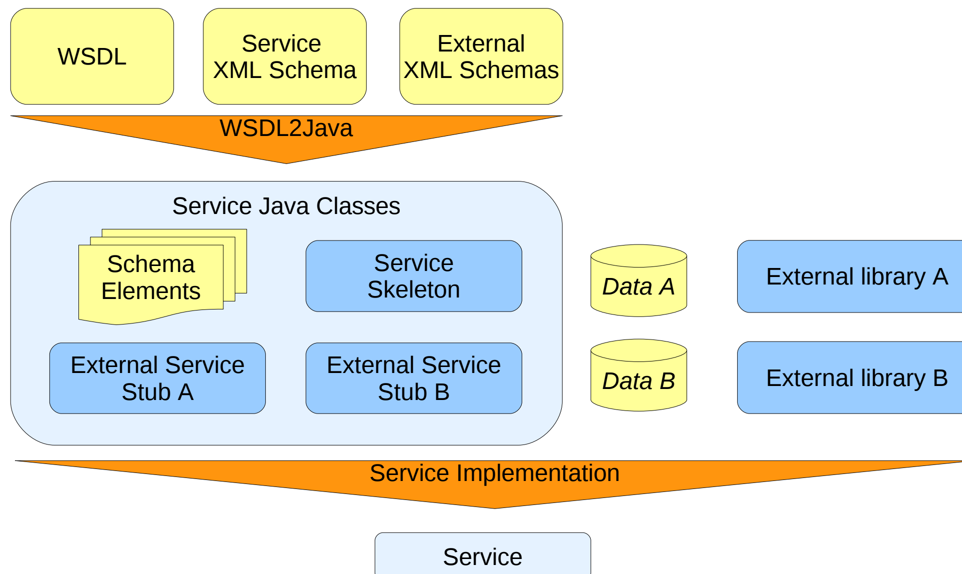

Figure 6.5. Service composition

Utilizing the automatic code generator of Axis2 called *WSDL2Java*, all elements defined in the XML schemas are available as Java classes. Furthermore a *skeleton* is created that contains the operations of the web service as methods. Interaction with

other services is achieved using their respective *stubs* which provide methods for each of its defined operations. They allow clients to perform requests directly using Java. This is because Axis2 provides the entire SOAP stack from the message format to the parsing into elements all the way up to the invocation of a method that represents a service operation.

The composition of the generated parts, data and external libraries then forms the actual service implementation.

6.1.2 Services

The services that are implemented in the *Transportation Security SensorNet* make use of a variety of components. For long term information storage, a MySQL database is used. A so-called *object-relational mapping* tool called *Hibernate* [41] enables objects to be stored and retrieved transparently without the need of complicated database interactions.

Esper [27] provides complex event and alarm processing and is used at the *Virtual Network Operation Center*. The *Alarm Processor* at the *Mobile Rail Network* currently uses a less complex approach.

The *Sensor Node* is responsible for the actual communication with the sensors. It makes use of the so-called *Hi-G-Tek* (HGT) [42] protocol and a serial connection library for Java called *RXTX*.

Each component and its particular use is explained in the later sections when each individual service is described. At a high level, one of the main aspects when dealing with web services is the definition of whether they are *stateless* or *stateful*:

6.1.2.1 Stateless

By default web services are meant to be *stateless*. This is because most message exchanges are completely independent of each other. Web services usually offer calculations, information or capabilities that only require the service to perform a specific

action and give a response. This is part of the *autonomy* approach of web services (see chapter 4).

Even in the case where a web services provides data, the service is still considered *stateless* since the retrieval of the data at any given time is not dependent on the internal state of the service but only on the underlying data. If the data changes there is no state change in the web service and it still provides the same functionality.

6.1.2.2 Stateful

The need for *stateful* web services has been identified for the *Transportation Security SensorNet* because there are certain limitations in just using *stateless* web services. Given a so-called *online* data processor that analyzes sensor data; using a *stateless* web service, it is impossible to react to trends and complex events because the service is limited to single data objects that it receives.

Let us say that a web service is monitoring whether *seals* that lock cargo containers are broken and is supposed send out warning messages whenever they are. The service has limited capacity in terms of storing historic data but should still be able to intelligently determine if a sensor reading that shows that a seal is broken is just a misreading or a real threat. This is only possible if the service keeps track of previous states. In contrast, a *stateless* service would only be able to react to the current reading and is forced to make decisions based on this single piece of data.

Another example is the *Alarm Processor* service (see section 6.3.2) at the *Mobile Rail Network* that is used in the *Transportation Security SensorNet* implementation. It classifies sensor data from containers either as *information* or *security* depending on whether one is currently allowed to open the container or not.

6.1.3 Clients

Clients are able to make use of the operations provided by the *web services*. They usually utilize the same modules as the service. This means that in theory all *web*

services could have clients. Since a lot of the services in the *Transportation Security SensorNet* interact independently from users, the number of clients that are available to users is actually smaller.

One of the aspects of clients in the *Transportation Security SensorNet* is the management of the sensors. The *Sensor Management* service (see section 6.4.1) provides this among other things like retrieving the location of a particular *Sensor Node*.

Another aspect is the management of alarm notifications. For this purpose the *Alarm Reporting* service (see figure 6.13) defines various management operations for clients.

In order to facilitate the use of those clients, a so-called *Command Center Graphical User Interface* was implemented that works just like a desktop application. This is in addition to the command line interface that every client provides using the *Apache Commons Command Line Interface* (CLI) library.

6.1.4 Modules

Axis2 provides the possibility to "plug in" so-called *modules* that add functionality or change the way a service behaves. This allows a specific capability to be shared among different services without having to implement it in each of them. In general, the web service specifications that are used in Axis2 are implemented as modules. For more information see section 6.1.1.2.

6.1.4.1 Ping

In order to check the status of a particular service Axis2 provides a module that adds an operation called *pingService* to a service. This can be used to check the status of either a specific operation or all operations that the service defines. The client part that actually uses this operation was not part of Axis2 and had to be implemented by the author.

6.1.4.2 Logging

Especially for debugging purposes and performance evaluations, it is of great benefit to be able to see the raw SOAP messages that are sent and received. The so-called *logging* module that was implemented provides this functionality. In particular the following information is captured:

- *Time* when the message was sent or received
- *Service* which is used
- *Operation* that is being executed
- *Direction* of the message, which can be either incoming or outgoing. Note that there are special directions that deal with incoming and outgoing faults.
- *From* address of the message
- *Reply to* address that may differ from the *From* address
- *To* address of the message
- *Schema element* that is being "transported" as part of the operation containing the request parameters or the response elements
- *Size* of the message in bytes
- *Message* which represents the entire SOAP message in a readable form

In terms of analyzing the *Transportation Security SensorNet* and its performance the *logging* module was engaged in all services. More information on the findings can be found in chapter 7.

6.1.4.3 Addressing

An implementation of the *WS-Addressing* specification as described in section 4.3.1 comes as part of the *addressing* module in the Axis2 core. It fully supports all components of the standard and its *ReplyTo* and *RelatesTo* fields are used among other things to allow for *asynchronous* communication (see section 6.1.6) in the TSSN.

6.1.4.4 Savan

The *Savan* module enables web services and clients in Axis2 to make use of various forms of subscription mechanisms as defined by the *WS-Eventing* specification (see section 4.3.2).

6.1.4.5 Rampart

In order to provide security according to the *WS-Security* specification (see section 4.3.3) for the TSSN the *Rampart* module was developed by Axis2. It makes extensive use of the *WS-SecurityPolicy* standard described by Lawrence et al. [50].

6.1.5 Subscriptions

Subscriptions are a fundamental part of the overall architecture of the *Transportation Security SensorNet*. They are used by the *Alarm Processor* at the *Virtual Network Operation Center* as well as in the *Mobile Rail Network*. These web services, that act as information publishers, utilize the *Savan* module to provide the operations defined in *WS-Eventing*.

6.1.6 Synchronous and asynchronous communication

By default Axis2 uses request-response in a *synchronous* manner. This means that the client has to wait and is therefore *blocking* until it receives the response from the service. In certain scenarios, for instance when the service needs a large amount of processing time, the client can experience timeouts. Furthermore, in the *Transportation Security SensorNet* where the *Mobile Rail Network* is only intermittently connected to the *Virtual Network Operation Center*, *synchronous* communication shows its limitations.

A better option is to make the communication between services *asynchronous*. This resolves timeout issues and deals with connections that are only temporary. The follow-

ing aspects need to be taken into consideration when using *asynchronous* communication:

6.1.6.1 Client

The client needs to make changes in regard to the how the request is sent out. Axis2 provides a low-level *non-blocking client API* and additional methods in the service stubs that allow callbacks to be registered. These so-called *AxisCallbacks* need to implement two methods, one that is being invoked whenever the response arrives and the other to define what happens in case of an error.

6.1.6.2 Transport Level

Depending on the transport protocol that is being used, Axis2 supports the following approaches.

- *One-way* uses one channel for the request and another one for the response such as the *Simple Mail Transfer Protocol* (SMTP)
- *Two-way* allows the same channel to be used for the request and the response, for example HTTP

For asynchronous communication to work the two-way approach was modified through the Axis2 *client API* which provides the option of using a *separate listener*. This tells the service that it is supposed to use a new channel for the response. In order to correlate request and response messages Axis2 makes use of the *WS-Addressing* specification, in particular the *RelatesTo* field.

6.1.6.3 Service

The final piece of asynchronous communication is to make the service processing asynchronous as well. This is done by specifying so-called *asynchronous message receivers* in the services configuration in addition to the *synchronous* ones.

Axis2 then uses the *ReplyTo* field of the *WS-Addressing* header in the client as a sign to send an immediate *acknowledge* of the request back to it. Furthermore it processes the request in a new thread and sends the response out when it is done, allowing the communication to be performed in asynchronous manner completely.

There exist various forms of transport protocols that are suitable for *asynchronous* communication. Axis2 by default supports HTTP, SMTP, JMS and TCP as transports but other transports can easily be defined and plugged in. The *Java Message Service* (JMS), for instance, makes use of so-called *queues* which allow clients and services to store on them and retrieve messages in a flexible manner. This is essential for satellite communication which is part of the next stage of the implementation of the *Transportation Security SensorNet*.

6.2 TSSN Common Namespace

Elements are often shared among a variety of services. Since defining the same element over and over again is neither a scalable nor maintainable approach, it makes sense to specify a common namespace for them and let the web services that want to use them, include them. In the *Transportation Security SensorNet* these shared elements are part of the so-called *TSSN Common* namespace.

In particular the following elements and types are defined:

Simple Types

A *TrainID_t* represents a unique assembly unit of engines and rail cars.

The *SensorNodeID_t* uniquely identifies a *Sensor Node*.

A *HGT_SealID_t* is a combination of four characters and eight numbers that is used to identify a *Hi-G-Tek* tag or sensor.

Location

The *LocationBean* is used to store GPS location information. It consist of:

- *longitude*
- *latitude*
- *quality* of the so-called *GPS fix*

The so-called *quality* can be one of the following predefined ones:

- *none*, no position information available
- *old*, more than 1 minute without a valid position
- *poor*, last position information less than 60 seconds old and *GPS fix* is bad
- *fair*, last position information less than 40 seconds old and *GPS fix* is okay
- *good*, last position information less than 20 seconds old and *GPS fix* is okay
- *great*, last position information less than 10 seconds old and *GPS fix* is good

Messages

A *Status* is used widely as a return message and indicates the success or failure of an operation. It has the following fields:

- *status* that is defined as a boolean and signals *success* or *failure*
- *message* which contains information on the success or failure

A *Failure* that represents the occurrence of an exception is made up of a simple *message*.

Alarms

The *AlarmSeverity* which can be either one of the following:

- *Information* that someone might be interested in
- *Maintainence* related
- *Security* breach of a seal
- *Hazard* that needs to be investigated

The *AlarmType* can be one of these:

- *Message* that contains no other inherent meaning
- *SensorLimitReached* that is propagated when an observed property value exceeds certain limitations
- *SensorLost* which means that the specified sensor cannot be reached
- *SensorFound* which informs of an established connection to a particular sensor
- *Exception* that has occurred in a service

The one element that is most commonly used for the alarm notifications is the *MRN_AlarmBean* because it contains all the valuable information of an alarm.

- *SourceNode* that identifies the *Sensor Node*
- *TrainId* that identifies the associated train
- *TimeStamp* when the alarm occurred
- *Type* of the alarm, an *AlarmType*
- *Severity* of the alarm, an *AlarmSeverity*
- *Message* that contains the alarm data or information
- *Location* of the alarm, a *LocationBean*

Other commonly used or shared elements such as the *ExceptionReport* are part of the *web service* specifications and are described separately when explaining each service individually in the following sections.

6.3 Mobile Rail Network

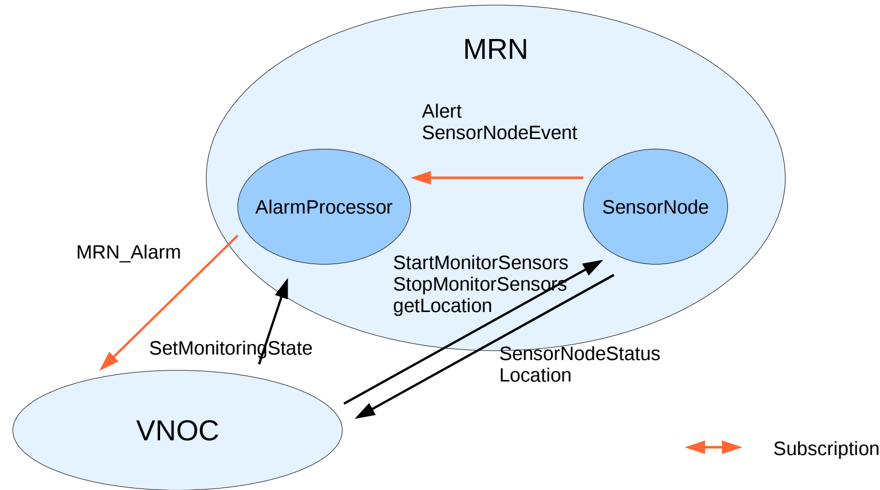

Figure 6.6. Mobile Rail Network message overview

The *Mobile Rail Network* is a collection of services that is located on a train or in a rail yard. Its services provide the abilities to manage sensors, monitor them and propagate sensor alerts to the *Virtual Network Operation Center*. This section describes them in detail.

6.3.1 Sensor Node

The *Sensor Node* contains the actual sensor monitoring and management application and its components are shown in figure 6.7. It provides several abstraction layers that allow various forms of sensors to be used. The current implementation makes use of so-called *Hi-G-Tek* (HGT) sensors. Interaction with these sensors is performed using a so-called *Automatic Vehicle Location* (AVL) reader. The *Sensor Node* implements the functionality that allows higher level management of the sensors and the data that they provide through the use of a *sensor registry*, the *sensor data* storage and *sensor data processing*

Attaching a GPS sensor to the *Sensor Node* allows sensor events to be tagged with

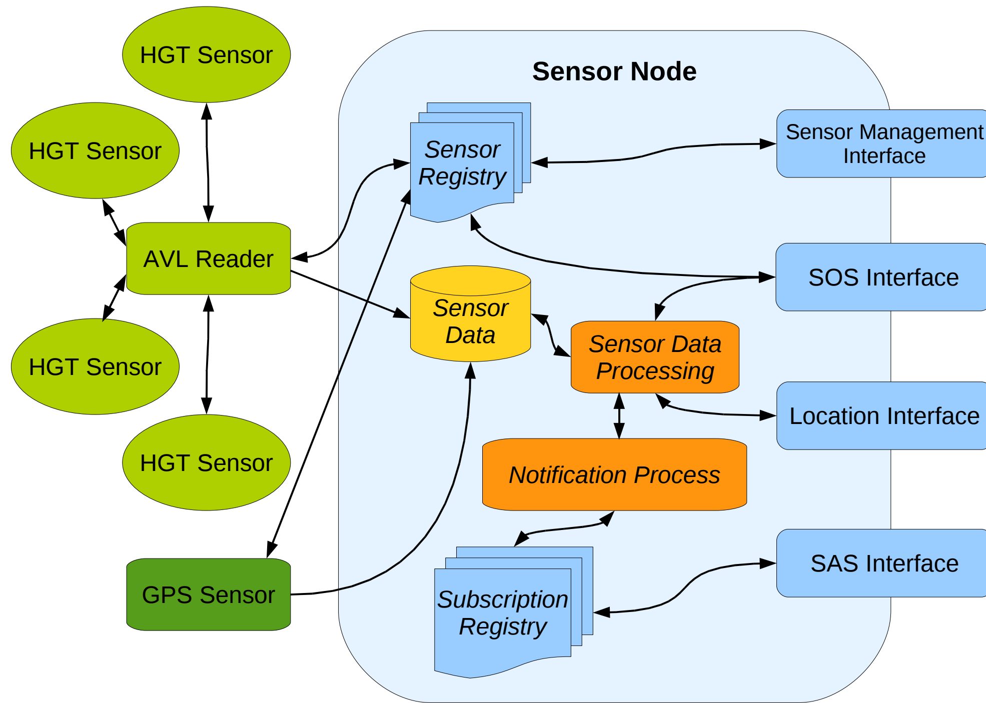

Figure 6.7. Mobile Rail Network Sensor Node

the specific location that they appeared at. The core functionality of the *Sensor Observation Service* that allows the service to offer its capabilities and observations is implemented. Furthermore, a *subscription registry* is available for alert notifications. The next sections explain the implementation details of these capabilities.

6.3.1.1 Sensor control

The following operations provide the ability to manage the underlying sensor infrastructure that is part of the *Sensor Node*.

StartMonitorSensors

The *StartMonitorSensors* operation described in table 6.1 starts the monitoring application. The *Sensor Node* then watches the status of the specified sensors identified by the *sensorIds* using the AVL reader via the HGT protocol. Note that even though

Message Exchange Pattern	In-Out
Parameters	*trainId* *sensorIds*
Response	*SensorNodeStatus*
Fault	*ows:ExceptionReport*

Table 6.1. Sensor Node StartMonitorSensors operation

the *Sensor Node* may be aware of additional sensors, it only captures events generated by the monitored sensors. The *trainId* specifies the train that the sensor node and the sensors are associated with.

StopMonitorSensors

Message Exchange Pattern	In-Out
Parameters	*none*
Response	*SensorNodeStatus*
Fault	*ows:ExceptionReport*

Table 6.2. Sensor Node StopMonitorSensors operation

The sensor monitoring application is stopped and the sensors are released by the *StopMonitorSensors* operation (table 6.2).

setSensors

Message Exchange Pattern	In-Out
Parameters	*none*
Response	*SensorNodeStatus*
Fault	*ows:ExceptionReport*

Table 6.3. Sensor Node setSensors operation

The HGT sensors that are used allow for a so-called *sleep mode*. Since they need to be “awake” in order to receive commands from the monitoring application, the *setSensors* operation described in table 6.3 sends so-called *set* signals to the sensors.

AddSeals

Message Exchange Pattern	In-Out
Parameters	*sensorIds*
Response	*SensorNodeStatus*
Fault	*ows:ExceptionReport*

Table 6.4. Sensor Node AddSeals operation

It is possible to tell the monitoring application to monitor additional sensors, which in case of HGT sensors are called seals, that are specified by the *sensorIds* using the *AddSeals* operation (table 6.4).

6.3.1.2 Location retrieval

Clients can also inquire about the current location of the *Sensor Node* when a GPS sensor has been attached.

getLocation

Message Exchange Pattern	In-Out
Parameters	*none*
Response	*Location*
Fault	*ows:ExceptionReport*

Table 6.5. Sensor Node getLocation operation

The *getLocation* operation described in table 6.5 provides a location query interface to the user. It retrieves the current location of the sensor node. Since a GPS sensor is usually attached to the *sensor node* directly, its location information is retrieved and not the one of a particular sensor.

6.3.1.3 OGC specifications

In order to provide standardized support for utilizing the functionality, the *Sensor Node* uses *WS-Eventing* to allow subscriptions to alerts that is similar to the *Sensor*

Alert Service (see section 3.2.6) and provides the following operations of the *Sensor Observation Service* (see section 3.2.5):

GetCapabilities

Message Exchange Pattern	In-Out
Parameters	*sos:GetCapabilities*
Response	*sos:Capabilities*
Fault	*ows:ExceptionReport*

Table 6.6. Sensor Node GetCapabilities operation

In accordance with the *Sensor Observation Service* specification, the *GetCapabilities* operation described in table 6.6 enables users to retrieve information about the sensors and the data they provide, the so-called *offerings*. The *Capabilities* element returned by this implementation also contains a list of sensor ids that are currently monitored.

GetObservation

Message Exchange Pattern	In-Out
Parameters	*sos:GetObservation*
Response	*om:Observation*

Table 6.7. Sensor Node GetObservation operation

The *GetObservation* operation (table 6.7) is a simplified version of the *Sensor Observation Service* equivalent and is used to retrieve current or historical sensor data from a sensor which identified by a sensor id that is part of the *GetObservation* parameter.

The provided *Observation* is a reduced version of the *Observation* in the *Observations & Measurements* specification and provides the time, format and the measurement of the sensor data observed.

The *Sensor Node* provides its functionality through the operations that were described. They allow sensor management, provide location information and OGC compliant interfaces.

6.3.2 Alarm Processor

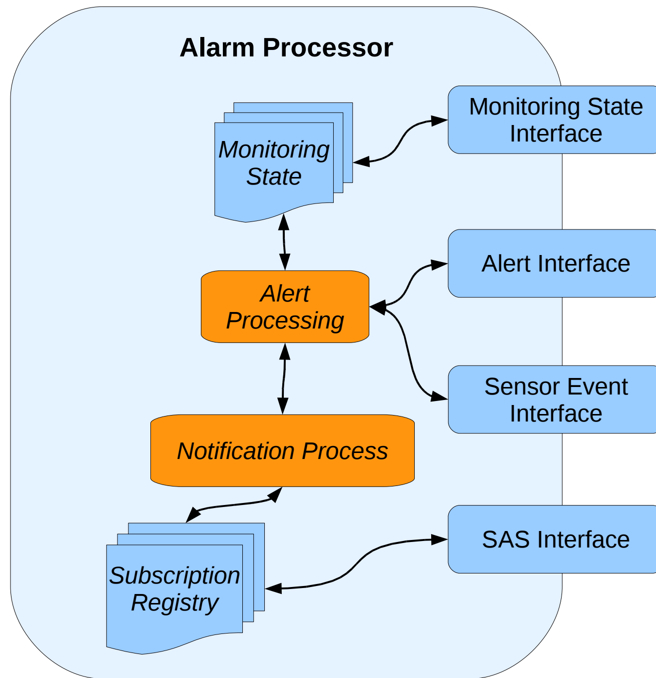

Figure 6.8. Mobile Rail Network Alarm Processor

The *Alarm Processor* on the *Mobile Rail Network* performs an initial filtering of sensor events generated by the *Sensor Node*. It subscribes to of all events of the *Sensor Node*, providing interfaces for generic sensor events as well as sensor alerts. Alerts reported to the *Alarm Processor* include potential alarms that the *Sensor Node* reports, GPS acquisitions and losses, and status messages of the monitoring application such as when it is started and stopped. In case the data is not as complex as an alert, the *event* element provides a simple structure with a timestamp and a data field.

The *Alarm Processor* classifies alerts into either *information* or *security* alarms depending on its current monitoring state. It is also responsible for deciding whether or not to forward the alarm to the *Virtual Network Operation Center* for further processing and possible transmission to the decision maker. Its implementation details are discussed next.

6.3.2.1 Notifications

The following operations are defined in reference to the *Sensor Alert Service* (see section 3.2.6) for receiving notifications from the *Sensor Node*.

Alert

Message Exchange Pattern	In-Only
Parameters	*sas:Alert*

Table 6.8. Alarm Processor Alert operation

The *Alert* operation described in table 6.8 represents a simplified version of its *Sensor Alert Service* equivalent. It contains fields for storing all the necessary information about a sensor node alert. In particular:

- *SensorID* of the particular sensor causing the alert
- *TimeStamp* of the alert
- *NodeId* of the *Mobile Rail Network*
- *TrainId* that identifies the current train association
- *AlertData* which contains the raw alert information
- *Latitude* of the alert location
- *Longitude* of the alert location
- *PosQuality* that specifies the quality of the GPS signal when the location was retrieved

SensorNodeEvent

Message Exchange Pattern	In-Only
Parameters	*SensorNodeEvent*

Table 6.9. Alarm Processor SensorNodeEvent operation

Simple events can occur as well and are reported using the *SensorNodeEvent* operation (table 6.9). They contain these two fields:

- *TimeStamp* of the event
- *EventData* which contains the raw alert information

6.3.2.2 Monitoring State

The *Alarm Processor* can be configured using the following operation:

SetMonitoringState

Message Exchange Pattern	Robust-In-Only
Parameters	*monitoringState*
Fault	*Failure*

Table 6.10. Alarm Processor SetMonitoringState operation

The *SetMonitoringState* operation described in table 6.10 specifies the current monitoring state of the *Alarm Processor*. It can be used to enable or disable security. When it is enabled, seal breaks are reported using a *security* notification instead of basic *information* message.

The *Alarm Processor* uses the described operations for handling alerts and events that it receives from the *Sensor Node*. In addition, it provides functionality to specify its monitoring state, in particular to switch between *information* and *security* mode.

6.4 Virtual Network Operation Center

The *Virtual Network Operation Center* as shown in figure 6.9 represents the management facility of the TSSN and consists of services that receive and process alerts received from *Mobile Rail Networks*. It works with the *Trade Data Exchange* to associate shipment and trade information with a particular alert. Furthermore, the *Alarm Reporting* service provides clients with the ability to be notified upon specific events. The processes that are involved in performing these tasks are the topic of this section.

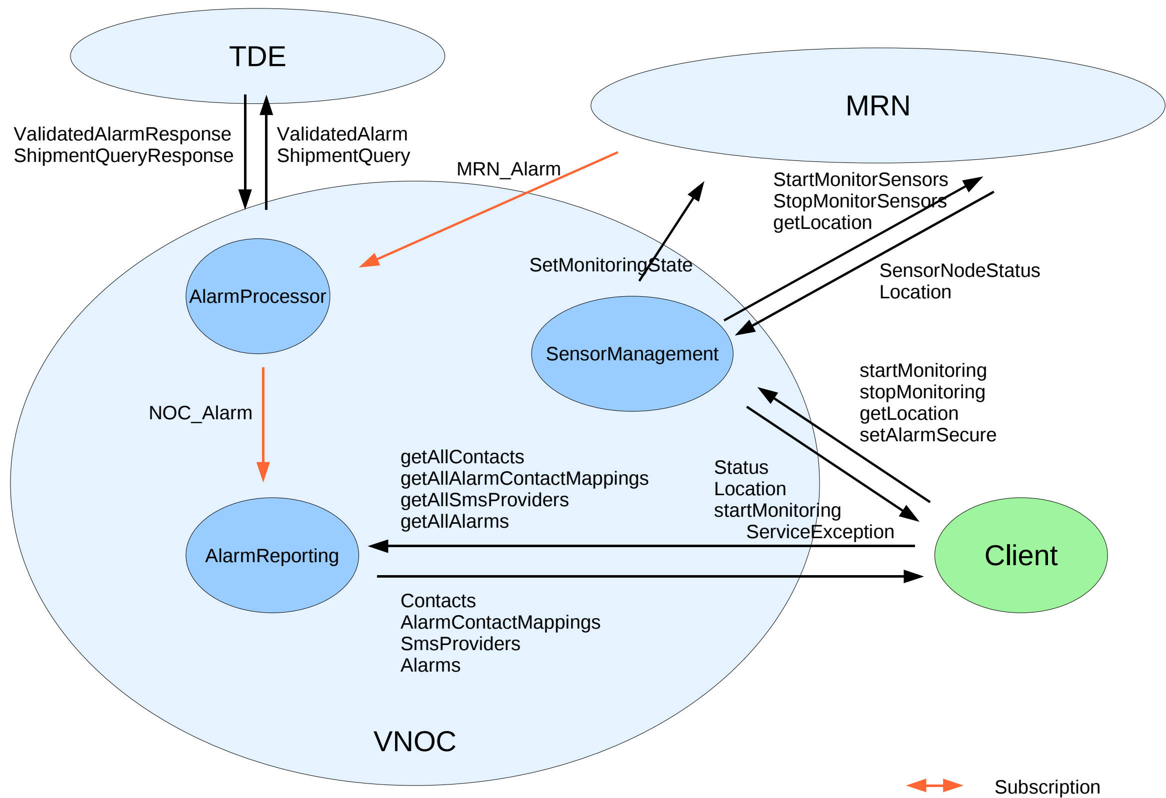

Figure 6.9. Virtual Network Operation Center message overview

6.4.1 Sensor Management

The *Sensor Management* service (figure 6.10) is responsible for controlling sensors and alarm reporting. It provides methods for starting and stopping sensor monitoring. Additionally the monitoring state which defines how alerts are interpreted and processed can be specified. The *Sensor Management* service essentially relays these "control" messages to the according *Mobile Rail Network*. Another functionality that is provided is the ability to query for a specific MRN's location. The implementation details of the interfaces that it provides to clients are described in the following.

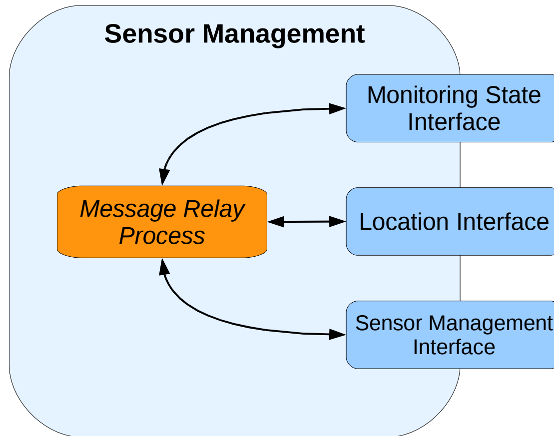

Figure 6.10. Virtual Network Operation Center Sensor Management

6.4.1.1 Sensor control

The following operations enable remote sensor management of the *Mobile Rail Networks.*

startMonitoring

Message Exchange Pattern	In-Out
Parameters	*collectorId* *trainId* *tagId* *sensorId*
Response	*tssn:Status*
Fault	*ows:ExceptionReport*

Table 6.11. Sensor Management startMonitoring operation

The *startMonitoring* operation described in table 6.11 tells the MRN that is specified by the *collectorId* to start monitoring sensors. The *collectorId* is the identifier of an individual sensor node. Furthermore the *trainId* provides the *Sensor Node* with the information of which train it is coupled to. This can be used later on to refine alarm processing and more importantly container handovers between trains. The *tagId* and *sensorId* are used in a parent-child sensor relationship. In this case a tag as the parent

would monitor the associated sensor as a child while the the sensor node only interacts with the tags. In case of sensor events the reporting chain would be sensor → tag → sensor node.

stopMonitoring

Message Exchange Pattern	In-Out
Parameters	*collectorId*
Response	*tssn:Status*
Fault	*ows:ExceptionReport*

Table 6.12. Sensor Management stopMonitoring operation

The *stopMonitoring* operation (table 6.12) is the opposite of the *startMonitoring* operation. It tells the specified sensor node to stop monitoring all sensors.

6.4.1.2 Location retrieval

Clients can inquire about the location of a particular *Sensor Node.*

getLocation

Message Exchange Pattern	In-Out
Parameters	*collectorId*
Response	*tssn:LocationBean*
Fault	*ows:ExceptionReport*

Table 6.13. Sensor Management getLocation operation

The *getLocation* operation described in (table 6.13) provides a location query interface to the user. It retrieves the current location of the specified *Sensor Node.*

6.4.1.3 Monitoring state

Alarm Processors at the *Mobile Rail Networks* can be configured using the following operations:

setAlarmSecure

Message Exchange Pattern	In-Out
Parameters	*collectorId* *secure*
Response	*tssn:Status*
Fault	*ows:ExceptionReport*

Table 6.14. Sensor Management setAlarmSecure operation

The specified MRN *Alarm Processor* can be contacted using the *setAlarmSecure* operation (table 6.14) in order to enable or disable security in its monitoring state. When the security state is enabled, seal breaks are reported using a *security* notification instead of basic *information* message.

setAlarmProcessorMonitoringState

Message Exchange Pattern	In-Out
Parameters	*collectorId* *monitoringState*
Response	*tssn:Status*
Fault	*ows:ExceptionReport*

Table 6.15. Sensor Management setAlarmProcessorMonitoringState operation

The *setAlarmProcessorMonitoringState* operation described in table 6.15 provides a more flexible configuration interface to the *Alarm Processor* on the MRN. Settings are specified in a descriptive and extensible monitoring state bean which could hold additional state information such as time frames for monitoring sensors or GPS location zones in which to automatically switch into security state. This state bean is used for instance by the *setAlarmSecure* operation.

The operations described allow the *Sensor Management* service to control *Sensor Nodes* and their monitoring state. Additionally, it is able to retrieve the location of *Sensor Nodes*.

6.4.2 Alarm Processor

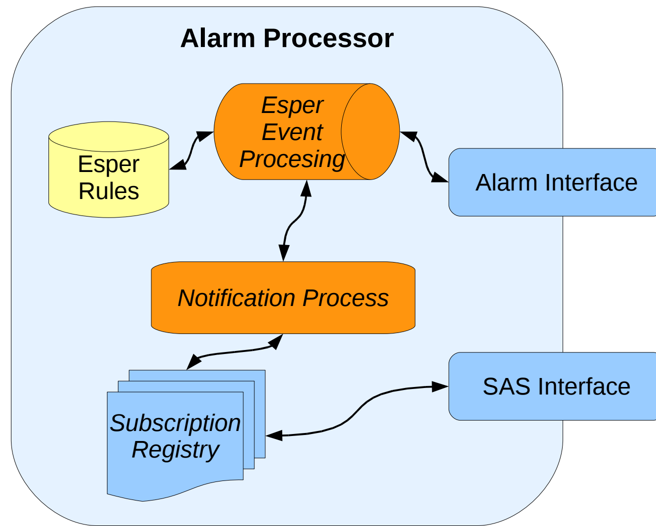

Figure 6.11. Virtual Network Operation Center Alarm Processor

In contrast to the "basic" processing that is performed by the *Alarm Processor* at the *Mobile Rail Network*, the *Alarm Processor* as shown in figure 6.11 at the VNOC has more resources such as the associated shipment and trade information available which is provided by the *Trade Data Exchange* and can therefore process alarms in a more complex way. This advanced filtering and processing is done using a complex event processing system called *Esper* developed by Bernhardt and Vasseur [7].

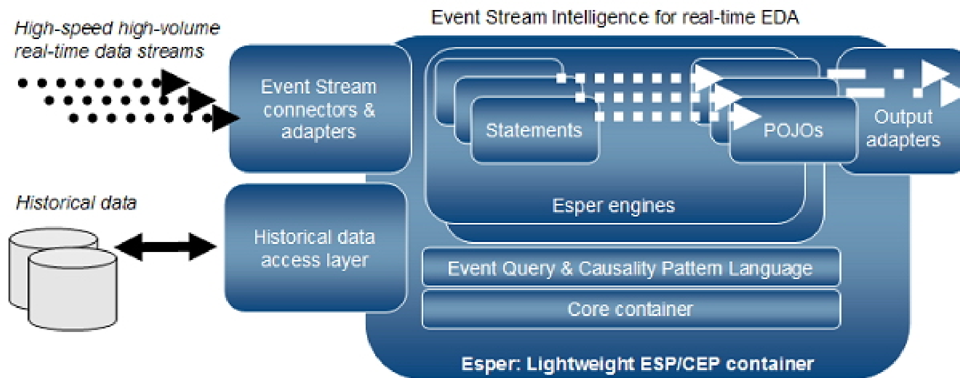

Figure 6.12. Esper architecture from [27]

Esper works on the basis of *sliding windows* in which events that are close together on the time axis are analyzed and correlated. It also supports using historical data from a variety of sources. An efficient query and filtering language called *Event Processing Language* allows for the most complex scenarios to be implemented. In the TSSN it is used for instance to filter out alarms for which shipment information could not be retrieved from the TDE and mark them as *security* notifications.

6.4.2.1 Notifications

The *Alarm Processor* receives alarm notifications from the *Mobile Rail Network* using the following operation:

MRN_Alarm

Message Exchange Pattern	In-Only
Parameters	*mrnpub:MRN_Alarm*

Table 6.16. Alarm Processor MRN_Alarm operation

The *MRN_Alarm* operation described in table 6.16 is used as a notification interface for alarms from the *Alarm Processor* on the MRN. The *Alarm Processor* service subscribes to alarms from its counterpart on the *Mobile Rail Network*. The alarms are of type *tssn:MRN_AlarmBean* (see section 6.2).

Upon receiving an alarm, shipment data is retrieved from the *Trade Data Exchange* and attached to the original alarm. *Esper* then processes the alarm and passes it on to the *Alarm Reporting* service.

The *Alarm Processor* at the VNOC primarily provides functionality for the *Mobile Rail Network* to deliver alert notifications. It uses *Esper* to perform complex event processing, taking into consideration alert data and information from the TDE, and to forward alarms to the *Alarm Reporting* service.

6.4.3 Alarm Reporting

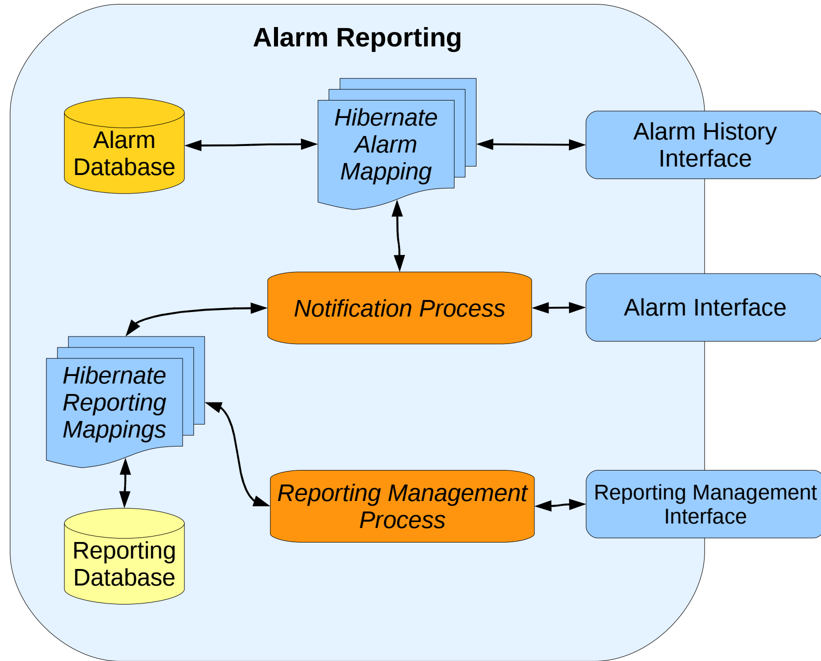

Figure 6.13. Virtual Network Operation Center Alarm Reporting

The *Alarm Reporting* service deals with the following two aspects. First, it stores alarms long term to allow for in-depth reporting and analysis. Second, clients that want to be notified of particular alarms can register with the *Alarm Reporting* service. Whenever alarms occur notifications are sent out to the registered clients via email and/or SMS accordingly.

For long term data storage and to maintain a registry of the client notifications the *Alarm Reporting* service makes use of the *MySQL* database. In order to remain flexible and provide an abstraction layer to the core database functionality a tool called *Hibernate* [41] was utilized. An excellent introduction to the so-called *object-relational* mapping is provided by Bauer and King [3]. The main advantage is that objects referenced in code can easily be *persisted* into a relational database and vice versa. The only thing that needs to be defined is the so-called *mapping*. Once that has been defined

Hibernate takes care of the rest.

Since the objects that are being stored in the database are defined using XML schemas and then automatically compiled into Java objects during the build process, it makes sense to specify the mappings in XML as well. This is done in the *Transportation Security SensorNet*. Another approach that is supported by *Hibernate* is using so-called *annotations* within the Java objects themselves. This is not possible because of the aforementioned build process as the objects would have to be reannotated at every build.

The registry that is used for notifications contains so-called *alarm contact mappings* that specify what kind of alarms a specific contact wants to be notified of. In case the contact wants to receive SMS notifications, a SMS provider has to be specified as well. The implementation details of the interfaces provided are described in the following.

6.4.3.1 SMS Providers

SMS providers have the following fields:

- *id* that uniquely identifies a provider
- *name* of the provider
- *emailSuffix* which is used for the email-based delivery of sms messages

The *emailSuffix* is used to construct an email address that is used to send out the SMS. For example “123456789@sampleProvider.com” where “123456789” is the phone number of the contact and “@sampleProvider.com” the email suffix of the phone provider.

addSmsProvider

Message Exchange Pattern	In-Out
Parameters	*SmsProvider*
Response	*tssn:Status*

Table 6.17. Alarm Reporting addSmsProvider operation

The *addSmsProvider* operation described in table 6.17 adds a new sms provider to the service. Note that the sms provider id is left blank (*null*) intentionally in this case and only the name and the email suffix have to be provided. The *Alarm Reporting* service automatically assigns an id to the new sms provider and stores it.

updateSmsProvider

Message Exchange Pattern	In-Out
Parameters	*SmsProvider*
Response	*tssn:Status*

Table 6.18. Alarm Reporting updateSmsProvider operation

Within the *updateSmsProvider* operation (table 6.18) the sms provider is identified by its id. The service looks for changes made to the sms provider and saves them.

removeSmsProvider

Message Exchange Pattern	In-Out
Parameters	*SmsProvider*
Response	*tssn:Status*

Table 6.19. Alarm Reporting removeSmsProvider operation

The *Alarm Reporting* service identifies sms providers that match the provided name and email suffix with elements in the database and removes them. The *removeSmsProvider* operation described in table 6.19 allows for pattern-based removal of sms providers. It also checks if there are still contacts associated with it.

removeSmsProviderById

Message Exchange Pattern	In-Out
Parameters	*Id*
Response	*tssn:Status*

Table 6.20. Alarm Reporting removeSmsProviderById operation

Since the id uniquely identifies an sms provider it can be removed explicitly using the *removeSmsProviderById* operation (table 6.20). The same check as in the *removeSmsProvider* operation is in place.

getAllSmsProviders

Message Exchange Pattern	In-Out
Parameters	*none*
Response	*SmsProviders*

Table 6.21. Alarm Reporting getAllSmsProviders operation

The *getAllSmsProviders* operation described in table 6.21 provides an interface to retrieve all available sms providers in a list form.

6.4.3.2 Contacts

Contacts have the following fields that contain general information about them:

- *id* that uniquely identifies a contact
- *affiliation* that represents an organization or company
- *name* which usually is first and last name of a person
- *email* address of the contact
- *smsProviderId* reference to the phone provider's *email-to-SMS* service
- *cellPhoneNumber* for SMS notifications

An *email* address or *cellPhoneNumber* must be provided, not necessarily both.

addContact

Message Exchange Pattern	In-Out
Parameters	*Contact*
Response	*tssn:Status*

Table 6.22. Alarm Reporting addContact operation

The *addContact* operation (table 6.22) is similar to the *addSmsProvider* operation in the sense that no id has to be provided for the new contact. The contact is stored in the database with an automatically assigned id.

updateContact

Message Exchange Pattern	In-Out
Parameters	*Contact*
Response	*tssn:Status*

Table 6.23. Alarm Reporting updateContact operation

Within the *updateContact* operation described in table 6.23 the service retrieves the specified *contact* by its id and saves the changes that were made to it.

removeContact

Message Exchange Pattern	In-Out
Parameters	*Contact*
Response	*tssn:Status*

Table 6.24. Alarm Reporting removeContact operation

The *removeContact* operation (table 6.24) removes the specified contact. It also allows for pattern based removal. A check is in place that prevents removal of contacts for which there still exist alarm contact mappings.

removeContactById

Message Exchange Pattern	In-Out
Parameters	*Id*
Response	*tssn:Status*

Table 6.25. Alarm Reporting removeContactById operation

The contact that is identified by the id is removed using the *removeContactById* operation described in table 6.25. The same check as in *removeContact* is in place.

getAllContacts

Message Exchange Pattern	In-Out
Parameters	*none*
Response	*Contacts*

Table 6.26. Alarm Reporting getAllContacts operation

A list of all the defined contacts can be retrieved with the *getAllContacts* operation (table 6.26).

6.4.3.3 Alarm Contact Mappings

Alarm contact mappings have the following fields:

- *id* that uniquely identifies a mapping
- *severity* of the alarm
- *type* of alarm
- *contactId* which references a particular contact
- *method* of notification (email or SMS)

These mappings are used by the *Alarm Reporting* service to determine what kind of notifications each contact receives and which *methods* to use for delivering them.

addAlarmContactMapping

Message Exchange Pattern	In-Out
Parameters	*AlarmContactMapping*
Response	*tssn:Status*

Table 6.27. Alarm Reporting addAlarmContactMapping operation

A new "alarm to contact" mapping is created using the defined entities with the *addAlarmContactMapping* operation (table 6.27).

updateAlarmContactMapping

Message Exchange Pattern	In-Out
Parameters	*AlarmContactMapping*
Response	*tssn:Status*

Table 6.28. Alarm Reporting updateAlarmContactMapping operation

Within the *updateAlarmContactMapping* operation described in table 6.28 the service retrieves the specified *alarm contact mapping* by its id and saves the changes that were made to it.

removeAlarmContactMapping

Message Exchange Pattern	In-Out
Parameters	*AlarmContactMapping*
Response	*tssn:Status*

Table 6.29. Alarm Reporting removeAlarmContactMapping operation

The *removeAlarmContactMapping* operation (table 6.29) removes the specified alarm contact mapping.

removeAlarmContactMappingById

Message Exchange Pattern	In-Out
Parameters	*Id*
Response	*tssn:Status*

Table 6.30. Alarm Reporting removeAlarmContactMappingById operation

The alarm contact mapping that is defined by the id is removed using the *removeAlarmContactMappingById* operation described in table 6.30.

getAllAlarmContactMappings

The service provides a list of all the alarm contact mappings that are in place with the *getAllAlarmContactMappings* operation (table 6.31).

Message Exchange Pattern	In-Out
Parameters	*none*
Response	*AlarmContactMappings*

Table 6.31. Alarm Reporting getAllAlarmContactMappings operation

6.4.3.4 Notifications

The *Alarm Reporting* service receives alarm notifications from the *Alarm Processor* at the *Virtual Network Operation Center* using the following operation:

NOC_Alarm

Message Exchange Pattern	In-Only
Parameters	*nocpub:NOC_Alarm*

Table 6.32. Alarm Reporting NOC_Alarm operation

This operation is used to provide a notification interface primarily for the subscription of alarms from the *Alarm Processor*. The *Alarm Reporting* service subscribes to alarms and provides this operation for its notifications. An alarm here is a combination of the *tssn:MRN_AlarmBean* and shipment and trade information received from the *Trade Data Exchange*.

6.4.3.5 Alarm history

getAllAlarms

Message Exchange Pattern	In-Out
Parameters	*none*
Response	*Alarms*

Table 6.33. Alarm Reporting getAllAlarms operation

A list of all the alarms that the service has received are retrieved using the *getAllAlarms* operation described in table 6.33. The alarms are of type *tssn:MRN_AlarmBean*. Note that the associated shipment data is not stored in the *Alarm Reporting* service as

it is permanently available in the *Trade Data Exchange*.

6.5 Trade Data Exchange

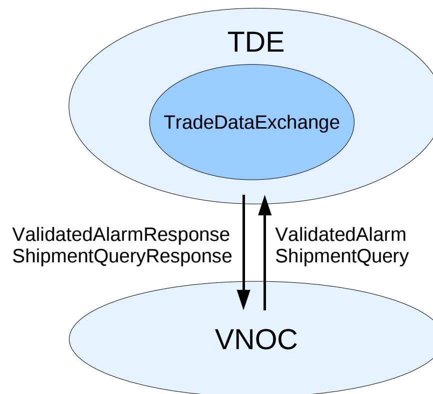

Figure 6.14. Trade Data Exchange message overview

The *Trade Data Exchange* [79], as shown in figure 6.14, in a sense represents a shipment and other trade data information provider. It aims to be a collection of heterogeneous systems that stores and manages the business aspects of a transport of goods. This is due to the fact that there is a variety of different systems implemented by the parties that participate in the transport chain (see section 2.1 and section 2.3). Some provide route information while others manage contracts and shipment data. For the current implementation of the *Transportation Security SensorNet* this "collection" of information and management services is combined into a single service, the *Trade Data Exchange* service.

6.5.1 Trade Data Exchange Service

The *Trade Data Exchange* service (figure 6.15) interacts with the *Alarm Processor* at the *Virtual Network Operation Center*. Upon request it provides shipment and trade information for a specified alarm. It also provides functionality that can be used for long term alarm storage, although in its current implementation fairly limited. Since

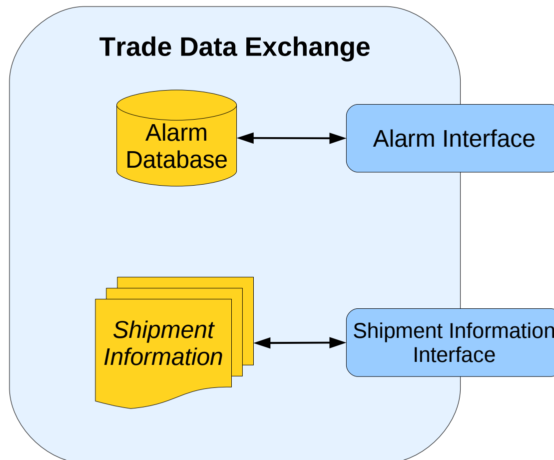

Figure 6.15. Trade Data Exchange Service

the service was designed externally, the elements used are not compatible to the TSSN common elements or any of the other services.

The *alarm data* element used has the following fields:

- *timeOccured* which represents the time when the alarm occurred
- *train_id* that uniquely identifies a train
- *tag_id* that uniquely identifies a tag (in this case a seal)
- *sensor_id* that uniquely identifies a sensor
- *alarm_type* which is either *Door open*, *Door closed*, *Sensor missing* or *Sensor returned*

This element has some shortcomings such as no location information, no alarm data field and limited *alarm types* but is currently used by the TSSN for the lack of a better interface to the shipment information.

6.5.1.1 Information inquiry

The following operation is provided to retrieve shipment and trade information from the *Trade Data Exchange*.

ShipmentQuery

Message Exchange Pattern	In-Out
Parameters	*alarm_data*
Response	*shipment_data*

Table 6.34. TradeDataExchange ShipmentQuery operation

The *ShipmentQuery* operation as described in table 6.34 provides *shipment_data* for the specified *alarm_data*. The shipment data contains the following information:

- *train_id* that uniquely identifies a train
- *equipment_id* that uniquely identifies a rail car; it consists of an *initial* and a *number*
- *car_position* of the container that the sensor is attached to
- *bic_code* that uniquely identifies a so-called *intermodal unit*
- *stcc* which is the *Standard Transportation Commodity Code* of the goods shipped

It has to be noted that no route information is made available through this inquiry.

6.5.1.2 Alarm storage

For long term storage of alarms the next operation is provided:

ValidatedAlarm

Message Exchange Pattern	In-Out
Parameters	*alarm_data*
Response	*status*

Table 6.35. TradeDataExchange ValidatedAlarm operation

Using the *ValidatedAlarm* operation (table 6.35) the *Trade Data Exchange* service receives *alarm_data* and stores it in a database.

6.6 Open Geospatial Consortium Specifications

As described before, the amount of work that is required to fully implement specifications of the *Open Geospatial Consortium* such as the *Sensor Observation Service* and the *Sensor Alert Service* is immense. The focus of the first stage of the implementation of the *Transportation Security SensorNet* is on the sensor management and alarm notification capabilities. However, at the *Mobile Rail Network* the *Sensor Node* provides an implementation for the *Sensor Observation Service* as defined by the OGC. Furthermore, services in the TSSN that utilize subscriptions, in particular the *Alarm Processor*, are able to receive *subscribe* requests and publish *alerts* in a manner that is similar to the *Sensor Alert Service*. The difference to the proposed SAS specification is that the services that subscribe are already aware of the *capabilities*, *sensor* types and *alert* types. Therefore the operations that allow the retrieval of this information, as described in section 3.2.6, need to be implemented in order to be fully compliant.

Chapter 7

Implementation Results

In this chapter tools that were developed and used to monitor the *Transportation Security SensorNet* are described. The *logging* module (section 7.1) plays the most important part as it captures message flows throughout the TSSN. These can then be analyzed using the *log parser* (section 7.2) and visualized by the *Visual SensorNet* tool (section 7.3). Performance measurements that were made throughout a series of trials are the used to evaluate the communication speed, processing times and alarm notifications (see section 7.4) within the TSSN.

7.1 Logging Module

The *logging* module as described in section 6.1.4.2 provides extensive logging capabilities to the *web services* in the *Transportation Security SensorNet.* It was *engaged* during development and testing of the entire system since it logs all messages that are sent and received. In addition, it also writes the raw contents of the SOAP messages into log files.

7.2 Log Parser

The *log parser* enables parsing and most importantly the merging of log files. It transforms the raw SOAP messages back into Java elements that can then be filtered and analyzed.

7.2.1 Abstraction Layer Model

Since SOAP is essentially XML, information from the so-called *log messages* can retrieved using *XPath* [8] *path expressions*. For this purpose the *log parser* provides an object *abstraction layer model* that corresponds to the specific parts in the SOAP message.

An example mapping is shown in figure 7.1. It displays the structure of the original SOAP message (for more information on SOAP see section 4.2) on the left and the equivalent *log parser* objects on the right. Note that the corresponding objects highlighted in yellow are actual classes while the *Header* and *Body* are not abstracted separately.

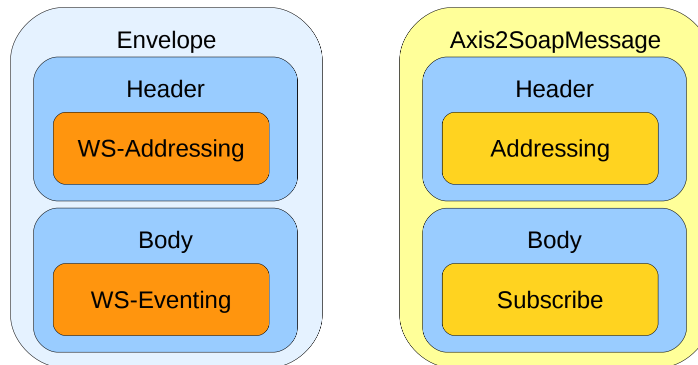

Figure 7.1. SOAP message (left) to Log parser classes (right) comparison

The *log parser* objects would then provide access to their properties using *XPath expressions*. In this case they correspond to their respective *web service* specifications but they could also be defined according to the XML schema definitions of any other

element. For example, for the *WS-Addressing* (see section 4.3.1) equivalent object the *path expressions* in table 7.1 are used:

XPath expression	Method equivalent
//To/text()	getTo()
//ReplyTo/Address/text()	getReplyTo()
//From/Address/text()	getFrom()
//MessageID/text()	getMessageId()
//RelatesTo/text()	getRelatesTo()
//Action/text()	getAction()

Table 7.1. XPath expressions for *WS-Addressing*

This mapping process is easily defined and allows for an in-depth analysis of the messages that are sent and received in the *Transportation Security SensorNet*.

7.2.2 Message Types

Since the *logging* module is enabled on both ends of a message exchange, the *log parser* is able to correlate messages. In order to do this it makes use of the so-called *message id* that is provided by the *WS-Addressing* specification. The following two types of message associations are present in the log files:

Transmit-Receive Pair Whenever a message is sent out by a particular client or service it is captured by the *logging* module. The receiving service logs the message as well but as an incoming message. The content of the message is essentially the same which can also be seen by the fact that they have the same *message id*. The outgoing and the incoming message are combined and form what is called a *transmit-receive pair*.

This allows us to compute the message transfer or so-called *transmit* time which describes how long it takes to transmit the message from one entity to another using the following equations:

$$transmitTime_1 = time_{2.Incoming} - time_{1.Outgoing} \tag{7.1}$$

$$transmitTime_2 = time_{4.Incoming} - time_{3.Outgoing} \tag{7.2}$$

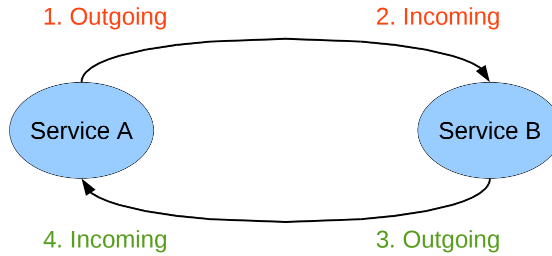

Figure 7.2. Two transmit-receive pairs (red and green)

As shown in figure 7.2 the *log parser* automatically detects the *transmit-receive pairs* and stores them in a particular list for further analysis.

Message Couple The most common *message exchange pattern* as described in section 4.6 is the *In-Out* pattern. It defines request-response based message transfers which the *log parser* calls *message couples*. A single *message couple* consists of two messages, the outgoing request and the outgoing response on the receiving entity, which is shown in figure 7.3. They can be correlated using the *WS-Addressing* specification. The request will carry a *message id* and the response a so-called *relatesTo id* in addition to its own unique *message id*.

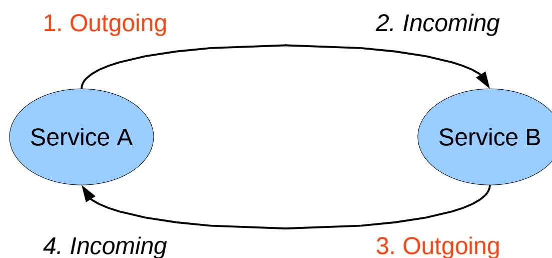

Figure 7.3. A message couple (red)

Note that a *message couple* can also be seen as a combination of two *transmit-receive pairs*. This relationship is extremely useful in computing measures such as *round trip*

and *processing* times:

$$roundTripTime = time_{4.Incoming} - time_{1.Outgoing} \tag{7.3}$$

$$processingTime = time_{3.Outgoing} - time_{2.Incoming} \tag{7.4}$$

The *log parser* provides functionality to associate messages and analyze complete end-to-end message flows. More details on the performance measurements and test results can be found in section 7.4.

7.3 Visualization

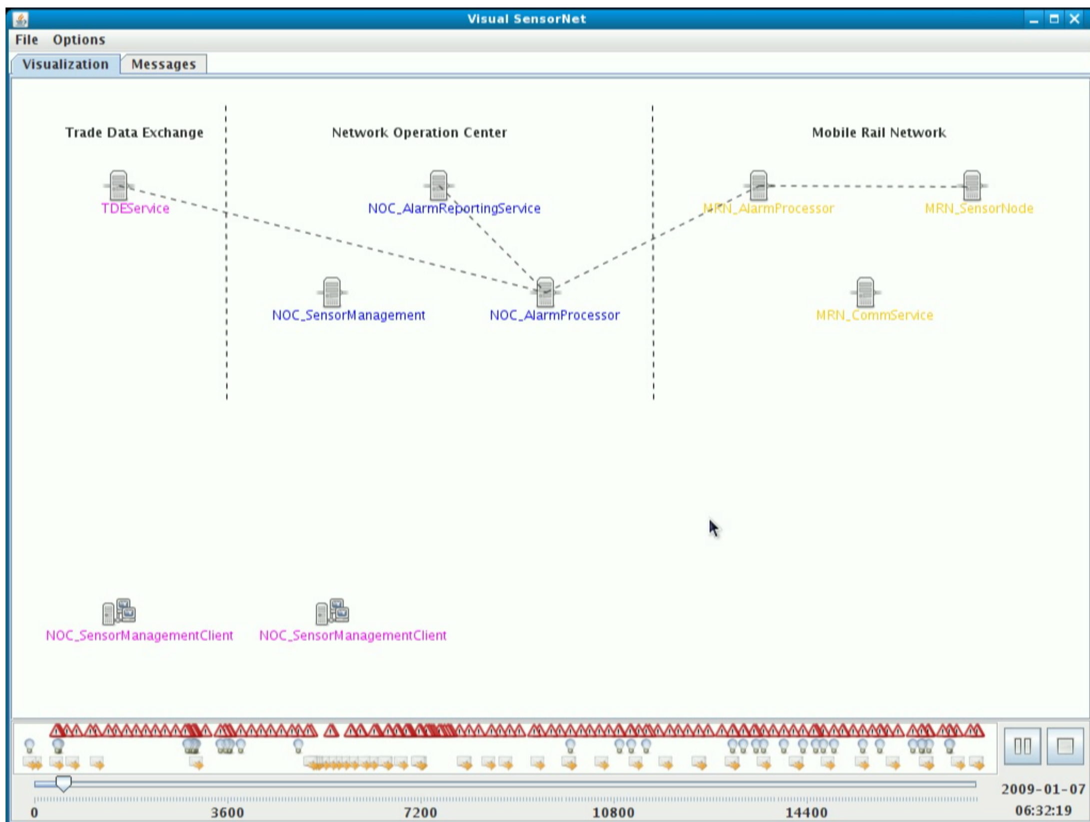

Figure 7.4. Log file and service interaction visualization

In order to be able to understand the message flows better without needing too

much of a technical background, a visualization tool called the *Visual SensorNet* was developed. It makes use of the *log parser* to display services, clients and messages that are present in log files.

The user is able to load and merge log files to create a visualization of services and clients as shown in figure 7.4. The layout of these services is defined according to their membership in a particular *service cloud*. Furthermore, any point in time that is part of the log files can be "jumped to" using the *time line*. It displays significant events in the log files:

- *Alarms*, *alerts* and *sensor node events* with a warning sign
- *Requests* such as location retrieval with a light bulb sign
- *Control messages* such as *start monitoring* with a message sign

The scenario that was captured by the log files can also be played back in portions or in its entirety. Using the *Visual SensorNet* tool, it is therefore possible to analyze service interactions and message flows conveniently.

7.4 Performance and Statistics

An in-depth analysis of the real world scenarios that were performed to test the *Transportation Security SensorNet* is given by Fokum et al. [31]. For the tests the *Trade Data Exchange* was deployed in Overland Park, the *Virtual Network Operation Center* at the *University of Kansas* in Lawrence and the *Mobile Rail Network* either on a truck or on a train. Note that in both cases the communication between the *Mobile Rail Network* and the *Virtual Operation Center* was established using a *GSM modem*. The main findings are as follows:

7.4.1 Road Tests with Trucks

During the tests the overall system had to deal with several issues. The location was not always available due to loss of so-called *GPS fixes*. This caused some alarms to

be reported with an inaccurate or old location. Furthermore, at some point the GSM connection broke down but could be reestablished. Note that no messages were lost in the process though.

In order to test the range of the *AVL Reader*, one of the goals was to find out at what point the reader loses contact to the sensors that it monitors. During the testing this distance was found to be about 400 meters. This was mainly due to significant hardware tuning and enhancements that were made by members of the *SensorNet* project. One of the reasons why range is so important is the fact that in the second stage the *Transportation Security SensorNet* was deployed in the engine of a train and it had to monitor sensors that were positioned on different railcars. In contrast to many other sensor networks where sensors surround a so-called *base station* in a circular manner with the aims of minimizing distance, the rail scenario represents an almost linear sensing approach where the distance to the *base station* increases for each sensor.

Another problem was the significant clock drift on the *Mobile Rail Network* during relatively short tests (about 2 1/2 hours). Unfortunately this makes some time measurements unreliable, in particular those in between the MRN and the VNOC. Note that this is not such a big problem within the *Mobile Rail Network* and *Virtual Network Operation Center service clouds* though, since there is a greater interest in relative times such as the *processing* time of an operation. This problem could partially be solved by letting the *log parser* that was used for the analysis apply a time adjustment parameter. A better and more natural solution to this problem is discussed in section 8.2.

Note that these observations are mostly hardware related. The implementation of the *Transportation Security SensorNet* as described in this thesis worked and was able to provide the sensor management as well as complete end-to-end alarm notification capabilities.

7.4.2 Short Haul Rail Trial

This more advanced scenario was performed by deploying the *Mobile Rail Network* on a locomotive of a train along with sensors attached to containers for it to monitor. The train traveled approximately 35 kilometers during the trip, from a rail *intermodal facility* to a rail yard.

The system faced some of the same issues as during the truck trials such as loss of GPS, GSM and sensor communication. The data that was collected however shows that again the *Transportation Security SensorNet* was able to deal with them and send out alarm notifications reliably. The log files were analyzed using the *log parser* and led to the following:

Message Counts An overview of the message flow is shown in figure 6.1. During the *short haul rail trial* the *Sensor Node* reported 546 alerts to the *Alarm Processor*. After filtering, the details of which are explained in section 6.3.2, 131 alarms were sent to the *Alarm Processor* at the *Virtual Network Operation Center*. For 63 of them, shipment information was queried from the *Trade Data Exchange* and 33 were stored as so-called *validated alarms*. All of the 131 alarms that the *Alarm Processor* received were sent out to *Alarm Reporting* service which notified the according contacts via SMS and email.

There were also 30 inquiries for the location of the *Mobile Rail Network*.

Message Sizes Looking at the communication between the *Virtual Network Operation Center* and the *Mobile Rail Network* one can notice the following pattern. So-called *control messages* such as *startMonitoring* or *getLocation* are always initiated at the *Virtual Network Operation Center*. Since these messages usually transmit only a small functional request, the average message size is around 690 bytes. On the other hand, Alarms are always sent from the *Mobile Rail Network* and contain of a lot of valuable information. Hence the average message size is about 1420 bytes.

Request Performance As shown in figure 7.5, the time it took for messages from the *Virtual Network Operation Center* (*Sensor Management*) to send requests to the *Mobile Rail Network* (either *Sensor Node* or *Alarm Processor*) and receive a response was about 4.4 seconds on average. The fastest request was answered in 0.9 seconds while the slowest took about 11 seconds.

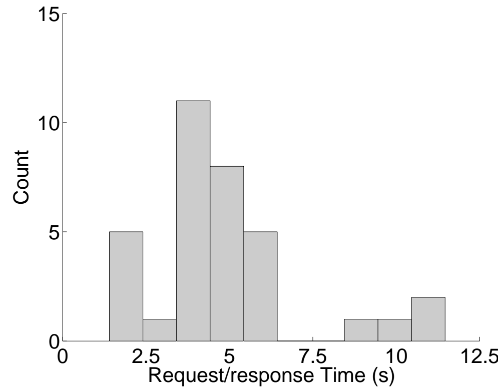

Figure 7.5. Request performance from [31]

Overall these numbers meet the expectations of the transportation industry. Performing a location inquiry given an average train speed of 30 km/h and 60 seconds to retrieve the location, the actual position and the reported one may differ by as much as 500 meters. However, the *Transportation Security SensorNet* provides location information in less than 5 seconds resulting in a maximum difference of just 41.7 meters.

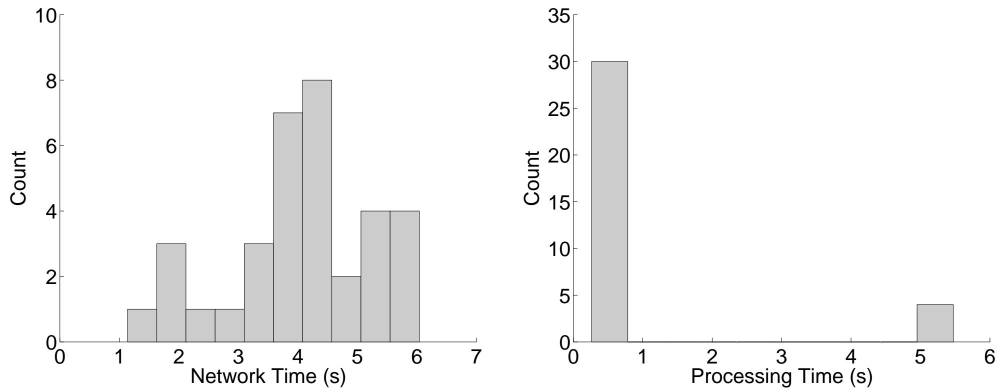

Figure 7.6. Network transmission and processing performance from [31]

The bottleneck here is the message *transmit* time as defined in equation 7.1. As shown in figure 7.6, processing on the *Sensor Node* took only 0.6 seconds on average whereas about 85% of the time is spent on message transmission. This percentage is likely to increase when switching to satellite communication instead of communicating with the GSM modem which was used in the trials.

Alarm Notification Performance Because of the problems with the clock drift, the measured times for messages coming from the *Mobile Rail Network* going to the *Virtual Network Operation Center* are unreliable. However, taking our previous findings about the request performance the time for this particular transmission can be estimated using the average *round trip* and the *processing* times:

$$\overline{transmitTime} = \frac{\frac{1}{n}\sum_{i=1}^{n}(roundTripTime_i - processingTime_i)}{2} \quad (7.5)$$

$$= \frac{4.4\ seconds - 0.6\ seconds}{2} \quad (7.6)$$

$$= 1.9\ seconds \quad (7.7)$$

Given this estimate, we can compute the total time it takes from for an alarm to go through the entire TSSN as shown in figure 7.7.

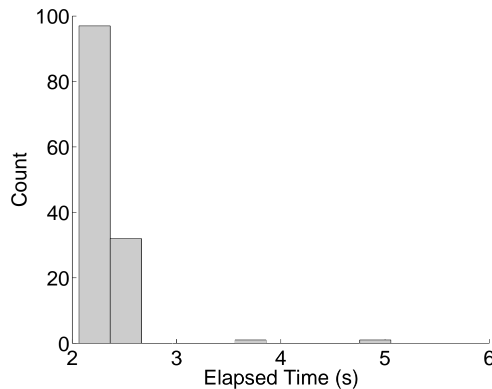

Figure 7.7. System alarm notification performance from [31]

This includes the times from the *Sensor Node* to the *Alarm Processor* at the *Mobile Rail Network*, the approximated transmit time of 1.9 seconds, and the time from the *Alarm Processor* to the *Alarm Reporting* service at the *Virtual Network Operation Center*. On average this yields about 2.1 seconds with the fastest time being just over 1.9 seconds and the slowest around 4.9 seconds.

Both, the road test with trucks and the short haul rail trial can be called successful because they displayed the capabilities of the TSSN, its good performance and that the functionality implemented in the web services worked. In particular, two of its main capabilities, location inquiry and alarm notification were extensively demonstrated. Furthermore, the time it took from registering alerts, propagating them through the *Transportation Security SensorNet* and sending out notifications accordingly is under 5 seconds and significantly smaller than expected for such a complex system.

Chapter 8

Conclusion

8.1 Current Implementation

The implementation of the *Transportation Security SensorNet* using a *Service Oriented Architecture* works. Testing has been completed in a lab environment as well as in the real world and TSSN was evaluated in chapter 7.

The complete system provides a *web services* based sensor management and alarm notification infrastructure that is built using open standards and specifications. Particular functionality within the system has been implemented in *web services* that provide interfaces according to their respective *web service* specifications.

Using standards from the *Open Geospatial Consortium* allows the integration of the system into *Geographic Information Systems.* Although not all the interfaces are fully implemented as of summer 2009, the basic *Sensor Observation Service* and *Sensor Alert Service* are. Other *Open Geospatial Consortium* specifications can be integrated a lot easier now because enhancements to the Axis2 schema compiler have been made by the author (see 6.1.1.2).

WS-Eventing plays an important role in the *Transportation Security SensorNet* as it is essential for the alarm notification chain. The specification that is used by all the clients and services is *WS-Addressing.* Note that HTTP, which represents the underlying

transport layer of most the *web services*, already provides an addressing scheme. This however, is not as useful as it seems because web services may change their *transport layer* and messages sometimes require complex routing. The reasoning behind this and other things have been explained in detail in section 4.3.1.

Overall the *Transportation Security SensorNet* provides a *Service Oriented Architecture for Monitoring Cargo in Motion Along Trusted Corridors* based on the extensible infrastructure of the *Ambient Computing Environment* for SOA. This *web services* based implementation allows for platform and programming language independence and offers compatibility and interoperability.

The integration of *Service Oriented Architecture*, *Open Geospatial Consortium* specifications and sensor networks is complex and difficult. As described in section 5.7, most systems and research focuses either on the combination of SOA and OGC specifications or on OGC standards and sensor networks. However, the *Transportation Security SensorNet* shows that all three areas can be combined and that this combination provides capabilities to the transportation and other industries that have not existed before. In particular, web services in a mobile sensor network environment have always been seen as slow and producing a lot of overhead. The TSSN, as shown by the results in chapter 7, demonstrates that this is not necessarily true.

Furthermore, the *Transportation Security SensorNet* and its *Service Oriented Architecture* allow sensor networks to be utilized in a standardized and open way through web services. Sensor networks and their particular communication models led to the implementation of asynchronous message transports in SOA and are supported by the TSSN.

8.2 Future work

After evaluating the current implementation, several points of improvement were identified.

Clock Synchronization In order to deal with the clock drift issue mentioned in section 7.4, enhancements are currently developed that will allow the time on the *Mobile Rail Network* to be adjusted using a local *Network Time Protocol* server. It is provided the so-called *pulse per second* from a GPS sensor attached to the *Sensor Node*. As a result of this there should hardly be any time synchronization problems left.

Service Discovery Due to several problems in the specific implementation of the UDDI that was used, for the trials most of the services were made aware of the other services through the means of configuration instead of service discovery. Since using a UDDI provides far better scalability, it is an essential piece of future versions of the *Transportation Security SensorNet*

Multiple service clouds During the trials all services were unique which in an operational system this is not the case. There are issues that need to be explored in dealing with multiple versions not only of single *web services* but multiple *Virtual Network Operation Centers* and *Mobile Rail Networks*. This is especially important when it comes to managing policies and subscriptions properly.

Security The current system only provides entry points for the *WS-Security* in terms of the *Rampart* module. There are several issues in the current implementation of the module, especially with regard to attaching policies to *web services* and clients. Further development is underway to implement *WS-Security*.

In between the *Virtual Network Operation Center* and the *Mobile Rail Network* communication is secured by establishing a *Virtual Private Network* (VPN). However, this is not practical using a satellite link because of performance reasons.

Sensors management is done at the *Sensor Node* but as of now there is no support for the secure handover to other *Sensor Nodes*. The remote management systems need to be improved in this area.

Asynchronous Communication The implementation of the *Transportation Security SensorNet* that was used during the trials made use of a "relatively" stable GSM modem connection that provided good performance and coverage. Furthermore, messages were sent in a *synchronous* manner.

In the next stage of development, the communication between the *Virtual Network Operation Center* and the *Mobile Rail Network* is done over a satellite link that is provided by a *communication service*. This means that several topics have to be addressed.

First, the current message sizes should be reduced in order to accommodate for the loss of speed. Possible optimizations have been discussed in section 4.2.3 but compression or conversion into binary formats are options as well.

Second, an enhancement that is currently being pursued and that deals better with message queuing on both ends of the communication is the switch to the *Java Message Service* as the transport. This is discussed by Easton et al. [25]. The *Java Message Service* uses so-called *Enterprise Service Bus* queues in order to send and receive messages. This allows the current implementation to work almost unmodified as the only thing that changes is the choice of transport for a few *web services* to fully support asynchronous communication.

8.3 Acknowledgment

The work for this thesis is supported by the *Office of Naval Research* through Award Number N00014-07-1-1042, *Oak Ridge National Laboratory* (ORNL) via Award Number 4000043403, and the *KU Transportation Research Institute* (KUTRI).

References

[1] Eyhab Al-Masri and Qusay H. Mahmoud. Investigating web services on the world wide web. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 795–804, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-085-2. URL http://doi.acm.org/10.1145/1367497.1367605.

[2] Jean-Pierre Bardet and Amir Zand. Spatial modeling of geotechnical information using gml. *Transactions in GIS*, 13(1):p125 – 165, 20090101. ISSN 13611682. URL http://search.ebscohost.com.www2.lib.ku.edu:2048/login.aspx?direct=true&db=aph&AN=36983054&site=ehost-live.

[3] Christian Bauer and Gavin King. *Hibernate in Action.* Manning, 2005.

[4] Tom Bellwood. Rocket ahead with UDDI V3. IBM article, IBM, November 2002. http://www.ibm.com/developerworks/webservices/library/ws-uddiv3/.

[5] Tom Bellwood, Luc Clement, David Ehnebuske, Andrew Hately, Maryann Hondo, Yin Leng Husband, Karsten Januszewski, Sam Lee, Barbara McKee, Joel Munter, and Claus von Riegen. UDDI Version 3.0. OASIS specification, OASIS, July 2002. http://uddi.org/pubs/uddi-v3.00-published-20020719.htm.

[6] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), January 2005. URL http://www.ietf.org/rfc/rfc3986.txt.

[7] Thomas Bernhardt and Alexandre Vasseur. Event-driven application servers, 2007. URL http://dist.codehaus.org/esper/JavaOne_TS-1911_May_11_2007.pdf.

[8] Scott Boag, Anders Berglund, Don Chamberlin, Jérôme Siméon, Michael Kay, Jonathan Robie, and Mary F. Fernández. XML path language (XPath) 2.0. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xpath20-20070123/.

[9] David Booth and Canyang Kevin Liu. Web services description language (WSDL) version 2.0 part 0: Primer. W3C recommendation, W3C, June 2007. "http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626.

[10] Mike Botts, George Percivall, Carl Reed, and John Davidson. OGC Sensor Web Enablement: Overview And High Level Architecture. OGC white paper,

OGC, December 2007. http://portal.opengeospatial.org/files/?artifact_id=25562.

[11] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. W3C note, W3C, July 2003. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[12] Don Box, Luis Felipe Cabrera, Craig Critchley, Francisco Curbera, Donald Ferguson, Steve Graham, David Hull, Gopal Kakivaya, Amelia Lewis, Brad Lovering, Peter Niblett, David Orchard, Shivajee Samdarshi, Jeffrey Schlimmer, Igor Sedukhin, John Shewchuk, Sanjiva Weerawarana, and David Wortendyke. Web services eventing (ws-eventing). W3C member submission, W3C, March 2006. http://www.w3.org/Submission/2006/SUBM-WS-Eventing-20060315/.

[13] Tim Bray, Richard Tobin, Dave Hollander, and Andrew Layman. Namespaces in XML 1.0 (second edition). W3C recommendation, W3C, August 2006. http://www.w3.org/TR/2006/REC-xml-names-20060816.

[14] Luis Felipe Cabrera, Christopher Kurt, and Don Box. An Introduction to the Web Services Architecture and Its Specifications. Microsoft technical article, Microsoft, October 2004. http://msdn.microsoft.com/en-us/library/ms996441.aspx.

[15] Marc Chanliau. Web Services Security: What's Required To Secure A Service-Oriented Architecture. Oracle white paper, Oracle, October 2006.

[16] Eran Chinthaka. Web services and Axis2 architecture. IBM article, IBM, November 2006. https://www.ibm.com/developerworks/webservices/library/ws-apacheaxis2/.

[17] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. W3C note, W3C, March 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[18] Xingchen Chu. Open sensor web architecture: Core services. Master's thesis, University of Melbourne, Australia, 2005. http://www.gridbus.org/reports/OSWA-core%20services.pdf.

[19] Xingchen Chu, Tom Kobialka, and Rajkumar Buyya. Open sensor web architecture: Core services. In *In Proceedings of the 4th International Conference on Intelligent Sensing and Information Processing*, pages 1–4244. Press, 2006. http://www.gridbus.org/papers/ICISIP2006-SensorWeb.pdf.

[20] Simon Cox. Observations and Measurements - Part 1 - Observation schema. OGC implementation specification, OGC, December 2007. http://portal.opengeospatial.org/files/?artifact_id=22466.

[21] Simon Cox. Observations and Measurements - Part 2 - Sampling Features. OGC implementation specification, OGC, December 2007. http://portal.opengeospatial.org/files/?artifact_id=22467.

[22] Simon Cox. Observations and Measurements - Part 2 - Sampling Features. OGC schema, OGC, . http://schemas.opengis.net/sampling/.

[23] Simon Cox. Observations and Measurements - Part 1 - Observation schema. OGC schema, OGC, . http://schemas.opengis.net/om/.

[24] Michael J de Smith, Michael F Goodchild, and Paul A Longley. *Geospatial Analysis - A Comprehensive Guide to Principles, Techniques and Software Tools*. Matador, 2008. http://www.spatialanalysisonline.com.

[25] Peter Easton, Bhakti Mehta, and Roland Merrick. SOAP over java message service 1.0. W3C working draft, W3C, July 2008. http://www.w3.org/TR/2008/WD-soapjms-20080723.

[26] Thomas Erl. *Service-Oriented Architecture - Concepts, Technology, and Design*. Prentice Hall, 2005.

[27] EsperTech. Esper - Event Stream and Complex Event Processing for Java. URL http://www.espertech.com/.

[28] David C. Fallside and Priscilla Walmsley. XML schema part 0: Primer second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[29] Joe Fialli and Sekhar Vajjhala. Java architecture for xml binding (jaxb) 2.0. Java Specification Request (JSR) 222, October 2005.

[30] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[31] Daniel T. Fokum, Victor S. Frost, Daniel DePardo, Martin Kuehnhausen, Angela N. Oguna, Leon S. Searl, Edward Komp, Matthew Zeets, Joseph B. Evans, and Gary J. Minden. Experiences from a Transportation Security Sensor Network Field Trial. ITTC Tech. Rep. ITTC-FY2009-TR-41420-11, University of Kansas, Lawrence, KS, June 2009.

[32] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, June 2002. http://www.globus.org/alliance/publications/papers/ogsa.pdf.

[33] Apache Software Foundation. XMLBeans, July 2008. URL http://xmlbeans.apache.org/.

[34] Anders Friis-Christensen, Nicole Ostländer, Michael Lutz, and Lars Bernard. Designing service architectures for distributed geoprocessing: Challenges and future directions. *Transactions in GIS*, 11(6):p799 – 818, 20071201. ISSN 13611682. URL http://search.ebscohost.com.www2.lib.ku.edu:2048/login.aspx?direct=true&db=aph&AN=28048261&site=ehost-live.

[35] Jesse James Garrett. Ajax: A new approach to web applications, February 2005. URL http://adaptivepath.com/ideas/essays/archives/000385.php.

[36] Delphi Group. The value of standards. Survey, Delphi Group, Ten Post Office Square, Boston, MA 02109, June 2003. www.ec-gis.org/sdi//ws/costbenefit2006/reference/20030728-standards.pdf.

[37] Martin Gudgin, Yves Lafon, and Anish Karmarkar. Resource representation SOAP header block. W3C recommendation, W3C, January 2005. http://www.w3.org/TR/2005/REC-soap12-rep-20050125/.

[38] Martin Gudgin, Martin Gudgin, Marc Hadley, Tony Rogers, Tony Rogers, and Marc Hadley. Web services addressing 1.0 - SOAP binding. W3C recommendation, W3C, May 2006. http://www.w3.org/TR/2006/REC-ws-addr-soap-20060509.

[39] Martin Gudgin, Marc Hadley, and Tony Rogers. Web services addressing 1.0 - core. W3C recommendation, W3C, May 2006. http://www.w3.org/TR/2006/REC-ws-addr-core-20060509.

[40] Hugo Haas, David Booth, Eric Newcomer, Mike Champion, David Orchard, Christopher Ferris, and Francis McCabe. Web services architecture. W3C note, W3C, February 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[41] Red Hat. Hibernate Reference Documentation 3.3.1. Technical report, September 2008. http://www.hibernate.org/hib_docs/v3/reference/en-US/pdf/hibernate_reference.pdf.

[42] Hi-G-Tek. URL http://www.higtek.com/.

[43] David Hyatt and Ian Hickson. HTML 5. W3C working draft, W3C, February 2009. http://www.w3.org/TR/2009/WD-html5-20090212/.

[44] Melissa Irmen. 10 ways to reduce the cost and risk of global trade management. *Journal of Commerce*, March 2009. http://www.joc.com/node/410216.

[45] Martin Kalin. *Java Web Services: Up and Running*. O'Reilly, February 2009.

[46] Michael Kay. XSL transformations (XSLT) version 2.0. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[47] Christian Kiehle, Klaus Greve, and Christian Heier. Requirements for next generation spatial data infrastructures-standardized web based geoprocessing and web service orchestration. *Transactions in GIS*, 11(6):p819 – 834, 20071201. ISSN

13611682. URL http://search.ebscohost.com.www2.lib.ku.edu:2048/login.aspx?direct=true&db=aph&AN=28048260&site=ehost-live.

[48] Yves Lafon and Nilo Mitra. SOAP version 1.2 part 0: Primer (second edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[49] Kelvin Lawrence, Chris Kaler, Anthony Nadalin, Ronald Monzillo, and Phillip Hallam-Baker. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). OASIS standard, OASIS, February 2006. http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[50] Kelvin Lawrence, Chris Kaler, Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. WS-SecurityPolicy 1.2. OASIS standard, OASIS, July 2007. http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/ws-securitypolicy.pdf.

[51] E. Levinson. The MIME Multipart/Related Content-type. RFC 2387 (Proposed Standard), August 1998. URL http://www.ietf.org/rfc/rfc2387.txt.

[52] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag, 2004. http://www.cs.berkeley.edu/~culler/AIIT/papers/TinyOS/levis06tinyos.pdf.

[53] Amelia A. Lewis. Web services description language (WSDL) version 2.0: Additional MEPs. W3C note, W3C, June 2007. http://www.w3.org/TR/2007/NOTE-wsdl20-additional-meps-20070626.

[54] Canyang Kevin Liu. First Look at WSDL 2.0. SAP article, SAP, January 2005. https://www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/74bae690-0201-0010-71a5-9da49f4a53e2.

[55] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F Brown, Rebekah Metz, and Booz Allen Hamilton. Reference Model for Service Oriented Architecture 1.0. OASIS standard, OASIS, October 2006. http://docs.oasis-open.org/soa-rm/v1.0/.

[56] Lance McKee. The Importance of Going "Open". OGC white paper, OGC, July 2005. http://portal.opengeospatial.org/files/?artifact_id=6211.

[57] Noah Mendelsohn, Murray Maloney, Henry S. Thompson, and David Beech. XML schema part 1: Structures second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[58] Noah Mendelsohn, Hervé Ruellan, Martin Gudgin, and Mark Nottingham. XML-binary optimized packaging. W3C recommendation, W3C, January 2005. http://www.w3.org/TR/2005/REC-xop10-20050125/.

[59] Jean-Jacques Moreau, Sanjiva Weerawarana, Roberto Chinnici, and Arthur Ryman. Web services description language (WSDL) version 2.0 part 1: Core language. W3C recommendation, W3C, June 2007. http://www.w3.org/TR/2007/REC-wsdl20-20070626.

[60] Arthur Na and Mark Priest. Sensor Observation Service. OGC implementation specification, OGC, October 2007. http://portal.opengeospatial.org/files/?artifact_id=26667.

[61] Arthur Na and Mark Priest. Sensor Observation Service. OGC schema, OGC. http://schemas.opengis.net/sos/.

[62] Douglas Nebert, Arliss Whiteside, and Panagiotis (Peter) Vretanos. OpenGIS Catalogue Services Schemas. OGC schema, OGC. http://schemas.opengis.net/csw/.

[63] Douglas Nebert, Arliss Whiteside, and Panagiotis (Peter) Vretanos. OpenGIS Catalogue Services Specification. OGC implementation specification, OGC, February 2007. http://portal.opengeospatial.org/files/?artifact_id=20555.

[64] Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services (Independent Technology Guides)*. Addison-Wesley Professional, December 2004. ISBN 0-321-18086-0. http://portal.acm.org/citation.cfm?id=1044935.

[65] Duane Nickul, Laurel Reitman, James Ward, and Jack Wilber. Service Oriented Architecture (SOA) and Specialized Messaging Patterns. Adobe article, Adobe, December 2007. www.adobe.com/enterprise/pdfs/Services_Oriented_Architecture_from_Adobe.pdf.

[66] Henrik Frystyk Nielsen, Marc Hadley, Anish Karmarkar, Noah Mendelsohn, Yves Lafon, Martin Gudgin, and Jean-Jacques Moreau. SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[67] Henrik Frystyk Nielsen, Anish Karmarkar, Noah Mendelsohn, Martin Gudgin, Yves Lafon, Marc Hadley, and Jean-Jacques Moreau. SOAP version 1.2 part 2: Adjuncts (second edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part2-20070427/.

[68] Mark Nottingham, Hervé Ruellan, Noah Mendelsohn, and Martin Gudgin. SOAP message transmission optimization mechanism. W3C recommendation, W3C, January 2005. http://www.w3.org/TR/2005/REC-soap12-mtom-20050125/.

[69] David Orchard, Hugo Haas, Sanjiva Weerawarana, Amelia A. Lewis, Roberto Chinnici, and Jean-Jacques Moreau. Web services description language (WSDL) version 2.0 part 2: Adjuncts. W3C recommendation, W3C, June 2007. http://www.w3.org/TR/2007/REC-wsdl20-adjuncts-20070626.

[70] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-085-2. URL http://doi.acm.org/10.1145/1367497.1367606.

[71] George Percivall, Carl Reed, Lew Leinenweber, Chris Tucker, and Tina Cary. OGC Reference Model. Technical report, OGC, November 2008. http://portal.opengeospatial.org/files/?artifact_id=31112.

[72] Clemens Portele. OpenGIS Geography Markup Language (GML) Encoding Standard. OGC implementation specification, OGC, August 2007. http://portal.opengeospatial.org/files/?artifact_id=20509.

[73] Clemens Portele. OpenGIS Geography Markup Language (GML) Encoding Standard. OGC schema, OGC. http://schemas.opengis.net/gml/.

[74] Carl Reed. Topic 0: Abstract Specification Overview. OGC abstract specification, OGC, June 2005. http://portal.opengeospatial.org/files/?artifact_id=7560.

[75] Mark Reichardt. The Havoc of Non-Interoperability. OGC white paper, OGC, December 2004. http://portal.opengeospatial.org/files/?artifact_id=5097.

[76] Leon S. Searl. Service Oriented Architecture for Sensor Networks Based on the Ambient Computing Environment. ITTC technical report, ITTC, February 2008. www.ittc.ku.edu/sensornet/trusted_cooridors/papers/41420-07.pdf.

[77] Jérôme Siméon, Don Chamberlin, Daniela Florescu, Scott Boag, Mary F. Fernández, and Jonathan Robie. XQuery 1.0: An XML query language. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/.

[78] Ingo Simonis and Johannes Echterhoff. Sensor Alert Service. OGC candidate implementation specification, OGC, June 2007. http://portal.opengeospatial.org/files/?artifact_id=24780.

[79] KC SmartPort. Trade Data Exchange - Nothing short of a logistics revolution. *Journal of Commerce*, November 2008. URL http://www.joc-digital.com/joc/20081110/?pg=29.

[80] Dennis Sosnoski. JiXB, March 2009. URL http://jibx.sourceforge.net/.

[81] C. M. Sperberg-McQueen, François Yergeau, Eve Maler, Jean Paoli, and Tim Bray. Extensible markup language (XML) 1.0 (fifth edition). W3C proposed edited recommendation, W3C, February 2008. http://www.w3.org/TR/2008/PER-xml-20080205.

[82] Anne van Kesteren. HTML 5 differences from HTML 4. W3C working draft, W3C, June 2008. http://www.w3.org/TR/2008/WD-html5-diff-20080610/.

[83] Anne van Kesteren. The XMLHttpRequest object. a WD in last call, W3C, April 2008. http://www.w3.org/TR/2008/WD-XMLHttpRequest-20080415/.

[84] Anne van Kesteren and Ian Hickson. Offline web applications. W3C note, W3C, May 2008. http://www.w3.org/TR/2008/NOTE-offline-webapps-20080530/.

[85] Michael Wolfe. In this case, bad news is good news. *Journal of Commerce*, July 2004. www.ismasecurity.com/ewcommon/tools/download.aspx?docId=175.