FFI-rapport 2011/02235

SOA Pilot 2011 – service Infrastructure

Ketil Lund, Frank T. Johnsen, Trude H. Bloebaum and Espen Skjervold

Norwegian Defence Research Establishment (FFI)

3 February 2012

FFI-rapport 2011/02235

1176

P: ISBN 978-82-464-2074-5
E: ISBN 978-82-464-2075-2

Keywords

Tjenesteorientert arkitektur

Nettbasert Forsvar

Eksperiment

Web services

Approved by

Rolf Rasmussen	Project Manager
Anders Eggen	Director

English summary

The SOA[1] pilot was demonstrated for the Norwegian Defense at FFI in June 2011. Our goal was to give a practical demonstration of some of the benefits of using a service-oriented architecture. In the pilot, we included a number of existing military operational systems and service-enabled these by using so-called wrappers. All these systems were then connected in a common information infrastructure, offering their information through services.

In this report, we present the pilot and how it was carried out from a technical (SOA-oriented) point of view. We describe how the functionality of the systems were service-enabled, how the SOA infrastructure was realized, and our recommendations and lessons learned from the pilot.

[1] Service-Oriented Architecture

Sammendrag

SOA[2]-piloten ble vist for Forsvaret på FFI i juni 2011. Målet vårt var å gi en praktisk demonstrasjon av noen av fordelene med å bruke en tjenesteorientert arkitektur. I piloten tok vi utgangspunkt i et antall eksisterende militære operative systemer, og ga dem tjenestegrensesnitt ved hjelp av såkalte ”wrappere”. Alle disse systemene ble så koplet sammen i en felles informasjonsinfrastruktur, og informasjonen fra dem gjort tilgjengelig i form av tjenester.

I denne rapporten presenterer vi piloten og gjennomføringen av den fra et teknisk (SOA-orientert) synspunkt. Vi beskriver hvordan funksjonaliteten i systemene ble gjort tilgjengelige som tjenester, hvordan SOA infrastrukturen ble realisert samt våre anbefalinger og lærdommer fra gjennomføringen av piloten.

[2] Service-Oriented Architecture – tjenesteorientert arkitektur

Contents

1 Introduction

The main motivation for the SOA Pilot work has been to show the possibilities offered through service orientation, and to gain experience in exposing functionality from operational systems as Web services.

In order to realize network enabled capabilities, interoperability is a main concern. Interconnecting heterogeneous systems, both legacy and new systems, implies a transition from stove-pipe systems with little focus on information sharing, to systems with standardized interfaces and data formats. As a consequence, the service-oriented architecture (SOA) paradigm has become a key factor and an important initiative both in NATO and nationally. SOA helps making processes interoperable, and enables a much easier exchange of information. The foundation for a common situational awareness is increased information sharing, and this requires a strategy for making information visible, available, accessible, and understandable. For an introduction to SOA (the paradigm) and Web services (the technology), see [1] Sections 1 and 3.

The goal of the SOA Pilot has been to expose functionality from existing and new systems and applications as services. This is achieved by

- Integrating and connecting different applications and services
- Avoiding tailored (proprietary) point-to-point solutions
- Making ad-hoc users able to access relevant information without needing system-specific solutions
- Having a distributed, loosely coupled SOA-enabled system
- Composition of new services by combining existing services
- Hiding physical location, communication protocols and implementation details from consumer and provider

To ensure interoperability, it has been an important goal to ensure that the services adhere to the standards defined in the NATO Core Enterprise Services [28]. These are defined as "*technical services that facilitate other service and data providers to deliver content and value to end users. They can be thought of as the enablers used by other services […] They are independent of business process and context, and are ubiquitous.*" We implemented our services according to the recommendations in [28] and we also successfully verified compliance with the recommendations through our interconnection with the SOA infrastructure at the NATO C3 Agency (NC3A).

This report covers the SOA-related technical parts of the pilot, and it is organized as follows. In Section 2 we give a brief introduction to the scenario and we describe the infrastructure and information flow. We also show the general principle for wrapping operational systems.

Next, in Section 3 we give a brief description of the different operational systems used in the pilot, and how they were Web services-enabled. In Section 4 we present the WS-Notification standard and how it was used in the SOA Pilot. Then, in Section 5 we describe the WS-Discovery standard, which was used as the service discovery mechanism throughout the pilot. We present the Viewer used in the pilot in Section 6. This viewer was designed specifically for the pilot, and played an important role in demonstrating the benefits of SOA.

In Section 7 we discuss how the SOA principles can be used in an environment where networks with different classification levels are connected through a data diode. Then, in Section 8 we present the lessons learned from our work with the pilot, as well as recommendation based on what we learned. Finally, in Section 9 we conclude the report.

2 Overview

2.1 Scenario

The scenario takes place in an expeditionary operation located in the Lillestrøm area. It was decided to use the vicinity of FFI since the pilot included real movements and tracking within the operational area, as well as need for radio coverage. All this was greatly simplified by having it located close to FFI.

The following types of actors participate in the scenario:

- 2 Ground Teams
- Tactical Unit with HQ
- Tactical Command HQ
- Task Unit (QRF)
- Fighter aircraft
- Surface Vessel
- UAV
- Intel provider
- National HQ on home ground

This report does not go into details on the functions and roles of these actors. Instead, our focus is on the information exchange requirements between the different actors/units, and how using a service-oriented architecture can contribute to meeting these requirements. For a description of the actual scenario, we refer to [25].

Assumptions
Some common assumptions with regard to the scenario and architectures were made:

- Some actors are given functionality that is not implemented yet in the respective platform.
- The same radio (and waveform) is utilized within the coalition.
- Satellite communication is available to the Ground Team when standing still.

- Basic security is handled on the IP level.
- The Ground Teams and the Task Unit have connectivity using the SOA infrastructure and are able to publish services directly on the infrastructure without doing an internal information exchange between FACNAV and NORCCIS II before publishing services.
- For demonstration purposes the services published from each actor are expected to be able to use the communication medium available in the actor's location.
- The fighter aircraft is assumed to have the ability to exchange and utilize services with Link 16 information through the SOA infrastructure.
- The Surface Vessel can act as a relay for the fighter for distribution of services.
- Surveillance from the UAV is considered as an intelligence actor in this context.

Note that these are assumptions, and not necessarily implemented in the pilot. For instance, we did not actually IP level security or satellite communications. Nor did we use planes or vessels, these were only simulated. We did use cars for ground movement, though.

2.2 SOA enabling of infrastructure and systems

The physical infrastructure of the SOA Pilot consisted of a wired LAN located in our lab, in addition to a VPN connection to NC3A, and two different radio networks (Kongsberg WM600 and MRR). On the NC3A side, the physical infrastructure was constituted by a wired LAN.

Except for the WM600 UHF radios, all this is standard hardware being used in the military today. Correspondingly, all the operational systems included in the pilot, except for NORMANS and the sensor network, are systems being used operationally. With this as a starting point, we then built a SOA on top of the existing infrastructure, and made the operational systems available as services.

In our work we use the following definition of SOA:
SOA is an architectural style for making resources available in a way that they may be found and utilized by parties who don't need to be aware of them in advance.

The way resources are made available and discoverable is as services. A service is described using a formal specification (for instance, a WSDL, if Web Services are used to realize SOA), and the focus is on transmission formats. This means that the client is independent of the service, as opposed to what is the case when the client is based on an Application Programming Interface (API) and reuse of code. As a consequence the service becomes autonomous, meaning that the runtime environment of the service can be changed without this affecting the clients.

Following the SOA principles we get a loose coupling between the clients and the services, with respect to both time (enabling asynchronous communication) and place (location of both client and service can be changed without need for reconfiguration). The clients discover services as they appear and invoke or subscribe to these (provided these are known service types). This is illustrated in Figure 2.1.

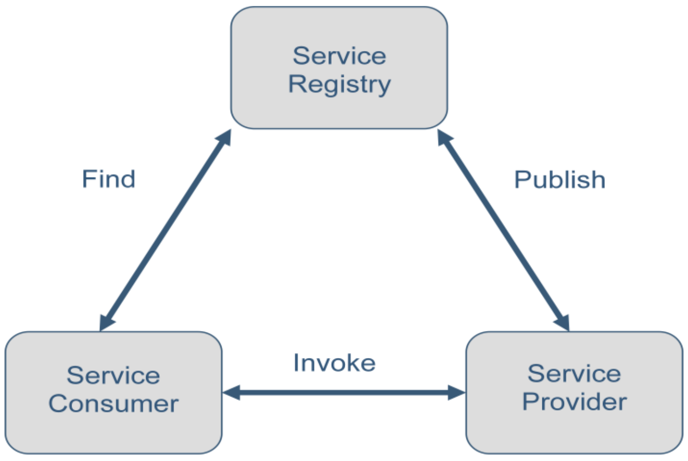


Figure 2.1 The SOA triangle

For message transport, SOAP over HTTP/TCP was used, and most information was disseminated using publish/subscribe (WS-Notification, see Section 4). An overview of the physical infrastructure is shown in *Figure 2.2*. As can be seen from this figure, the majority of the operational systems were connected directly to the wired LAN.

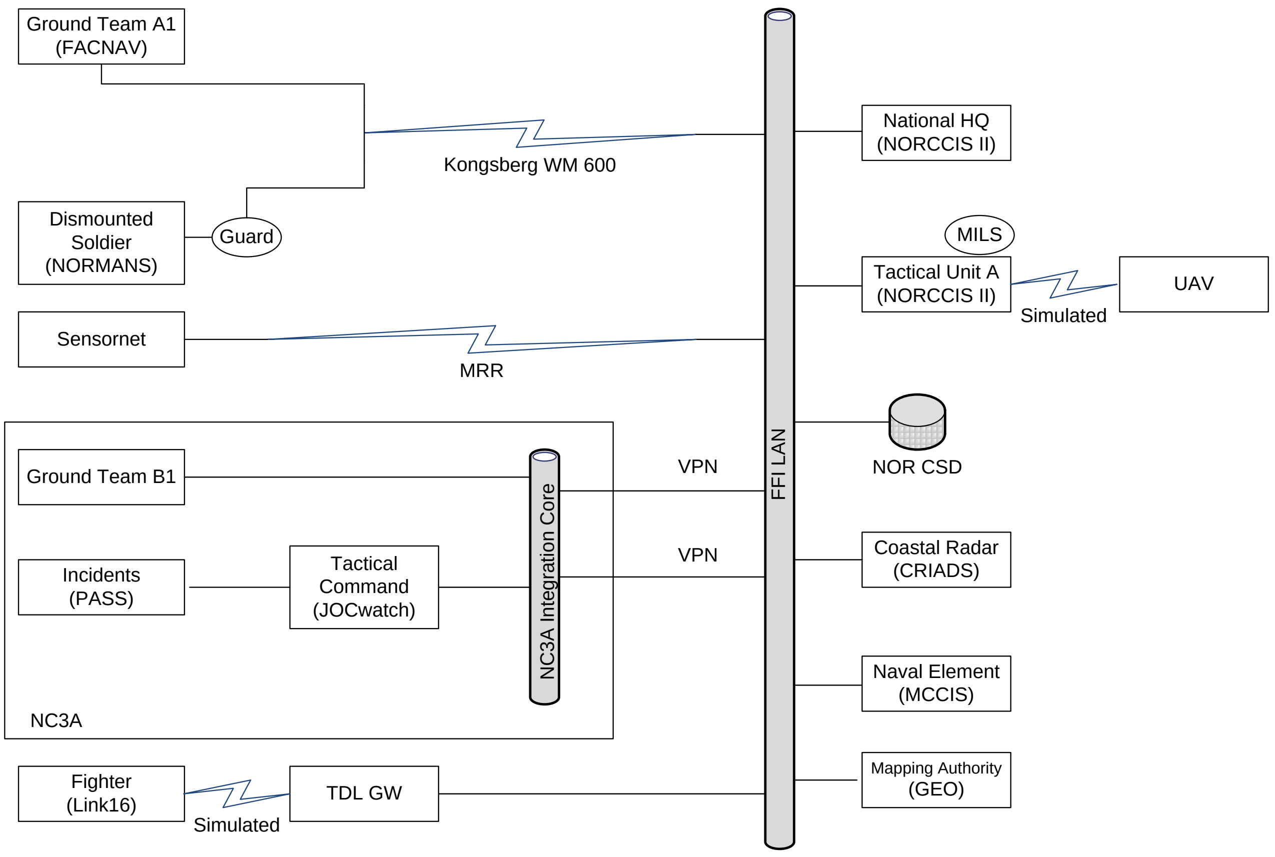


Figure 2.2 Physical infrastructure and operational systems

The different operational systems appeared as services on the network, and could be dynamically discovered and invoked (i.e., subscribed to).

2.3 Wrapping of operational systems

Services in a SOA can be realized in three different ways, as shown in Figure 2.3:

- They can be implemented as services from the start. This is what is normally done when implementing new systems
- Existing (non-SOA) systems can be "wrapped", in the sense that a piece of software is placed in front of the system, presenting services to the outside world, while communicating "natively" with the wrapped system
- Existing services and systems can be combined into new, aggregated services, by using software that coordinates (orchestrates) access to the individual systems and services, and presents the aggregate as a new service.

None of the operational systems used in the SOA Pilot offered the required information as Web services, and we therefore used the wrapper approach on these systems. In addition, our Viewer (see Section 6) has the ability to offer the information it receives from other services as a new service, thus representing a type of composite service.

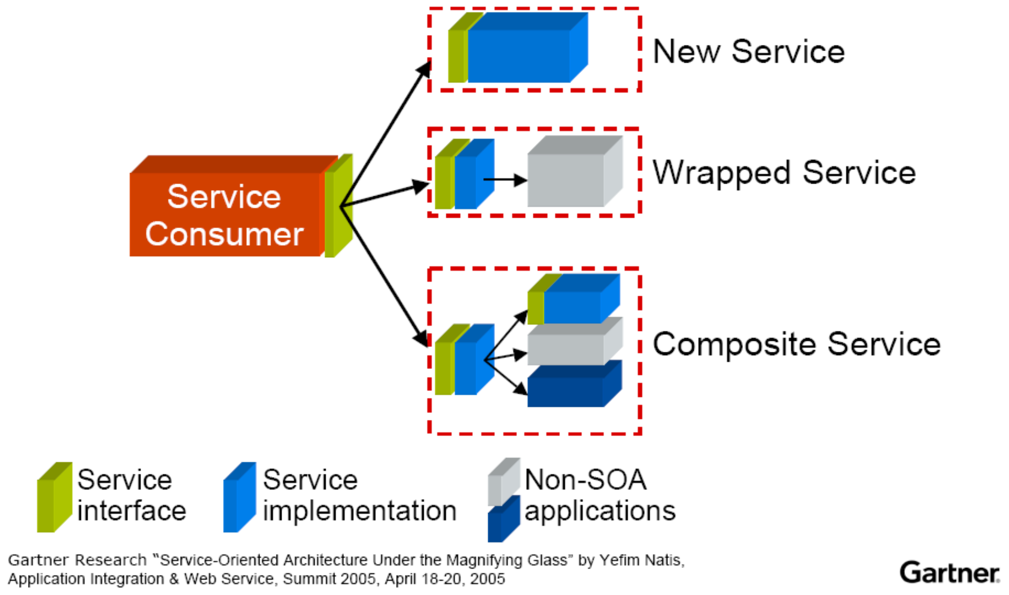

Service Consumer
New Service
Wrapped Service
Composite Service
Service interface
Service implementation
Non-SOA applications
Gartner Research "Service-Oriented Architecture Under the Magnifying Glass" by Yefim Natis, Application Integration & Web Service, Summit 2005, April 18-20, 2005
Gartner

Figure 2.3 Realizing SOA-services

The general principle of wrapping in the SOA Pilot is shown in Figure 2.4. In this figure, the WS wrapper and the WS-Notification entities together constitute the Web services adapter, as described in [25]. Each wrapped operational system delivers data (notifications) to a WS-Notification server, which is then responsible for delivering the notifications to all subscribers. In the pilot, only the Viewers acted as subscribers (in addition to being able to act as a service in itself, as described above). In addition, all services announced their presence using WS-Discovery, so that clients (in our case the viewer applications) could dynamically discover and invoke these.

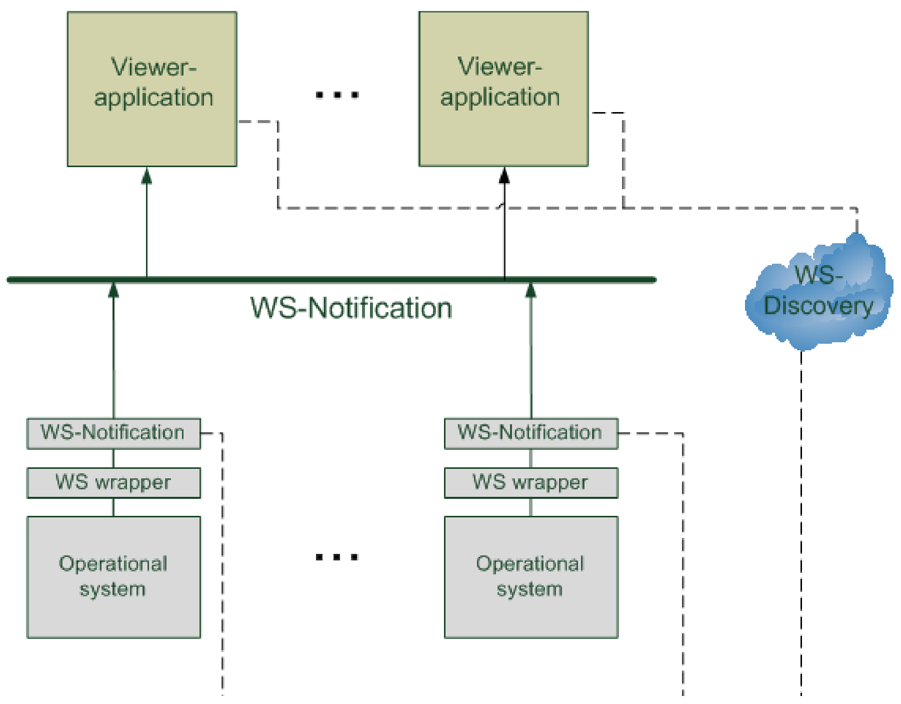


Figure 2.4 General principle of wrapping systems

2.4 Information flow

In Figure 2.5 below, we show the logical information flow in the initial situation of the scenario. In other words, this shows how the information flows between the different military units. In general, the information flows "vertically", i.e., between echelons, with the Tactical Command HQ functioning as a bridge between the two tactical units.

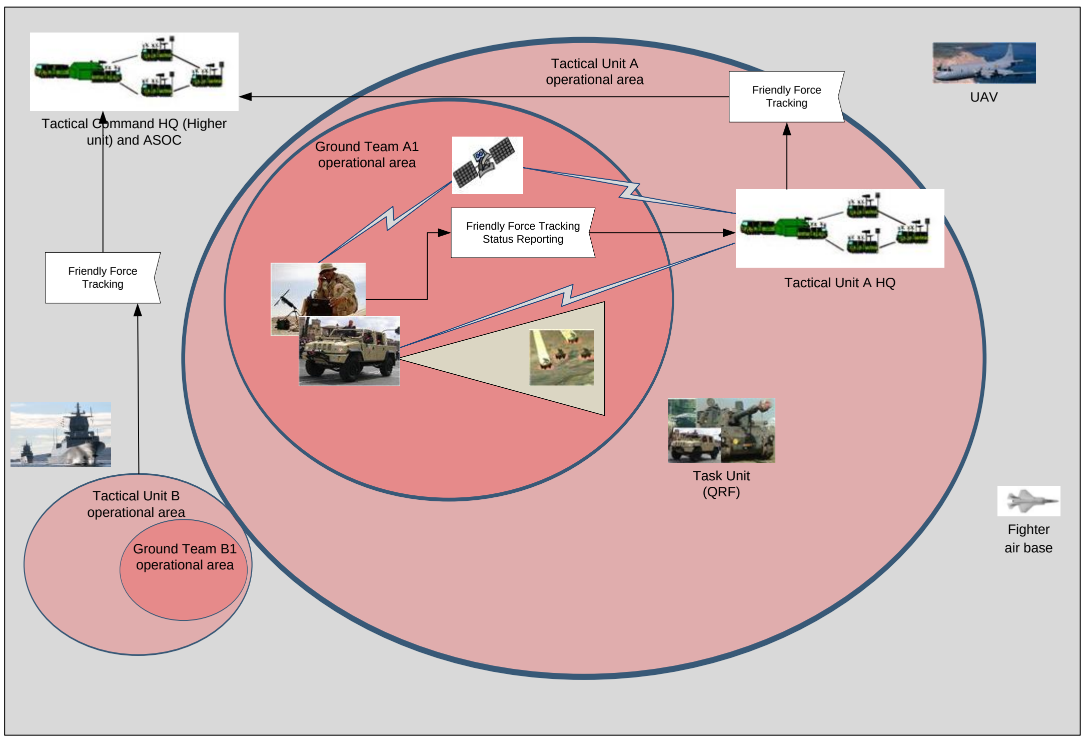

Tactical Unit A operational area
Friendly Force Tracking
UAV
Tactical Command HQ (Higher unit) and ASOC
Ground Team A1 operational area
Friendly Force Tracking Status Reporting
Tactical Unit A HQ
Friendly Force Tracking
Task Unit (QRF)
Fighter air base
Tactical Unit B operational area
Ground Team B1 operational area

Figure 2.5 Logical information flow in the initial situation of the scenario

The actual information flow is shown in Figure 2.6. As can be seen from the figure, the Viewer instances worked as aggregation points: In Ground Team A1, the local Viewer subscribed to, and visualized information from the FACNAV and NORMANS systems. This aggregated view was then offered as a Common operational picture (COP) service to Tactical Unit A, which also subscribed to a number of other systems, as shown in the figure.

The aggregated view from the Viewer in Tactical Unit A was then offered as a service to both National HQ and Tactical Command. Note that we placed an additional Viewer instance between Tactical Unit A and Tactical Command. This enabled us to control what information was sent to Tactical Command (i.e., NC3A). In the pilot, we forwarded the entire COP from Tactical Unit A, but the fact that all services were offered into one common infrastructure, provided us with great flexibility in how to distribute information. Thus, we could have chosen any combination of available services.

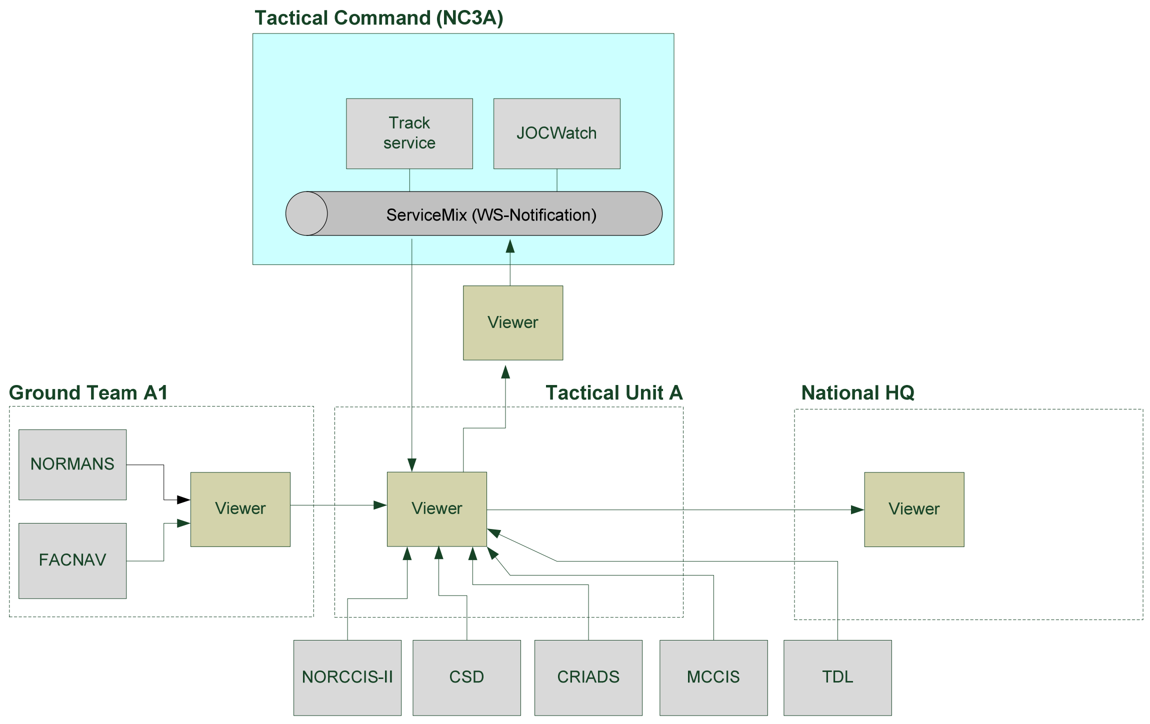


Figure 2.6 Actual information flow between the units in the pilot

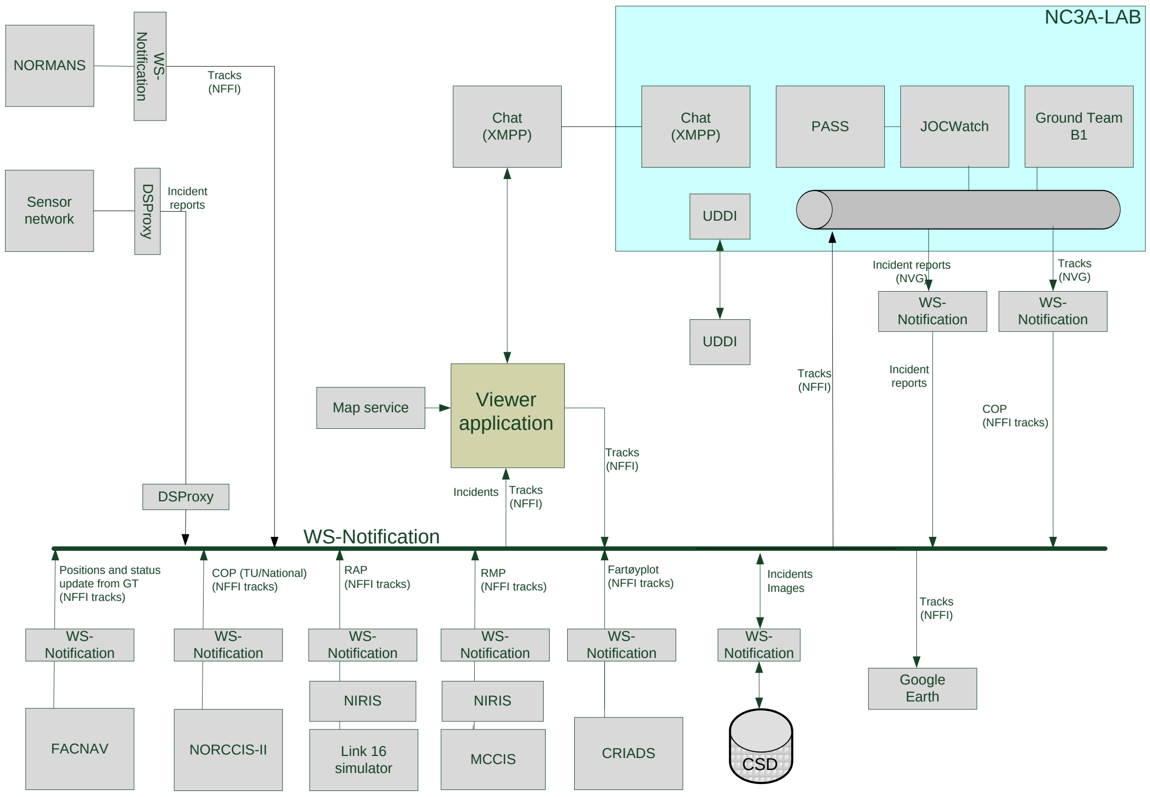


Figure 2.7 Information flow between systems

Figure 2.7 shows a more detailed picture of the information flow between the different systems. As described earlier, each operational system was wrapped with a Web service, delivering notifications to a WS-Notification service. Note that in the SOA Pilot, our focus was to demonstrate service enabling of existing operational systems. Therefore, each service announced through WS-Discovery reflected which operational system that was lying behind it. Consequently, we got a somewhat artificial set of services consisting of one CRIADS service, one NORCCIS-II service, one FACNAV service, etc. In a real-life environment, it is more likely that the services will reflect the *type of information offered*. Likewise, the user will select services based what information he or she needs, and not only based on which operational system that provides it.

In addition, we had two UDDI registries that were federated, one at NC3A and one at FFI, and we had an XMPP-based chat between all units (JChat).

We also used Google Earth as an alternative viewer, in order to demonstrate the capabilities of COTS software. This is further described in Section 3.2.

Finally, the sensor network was connected over a low bandwidth radio connection, and we therefore used the DSProxy to enable Web services over this connection. This is described in Sections 3.1.3 and 8.5.

3 Systems and software

3.1 Wireless sensor networks

Wireless Sensor Networks are expected to provide greatly enhanced situational awareness for war fighters in the battlefield. Sensors widespread in the battlefield are, however, of very limited value unless the sensors are reliable during the entire operation and the information produced is accessed in a timely manner. We have focused on these issues by enabling wireless sensor networks as a capability using Web services. In cooperation with FFI project 1141 “Situational Awareness Sensor Systems” we have shown that Web services is an enabling technology for information sharing, facilitating presentation of sensed data and alarms to a battlefield management system.

In addition, we tested the feasibility of using a Web services approach as a query processing tool enabling multi-sensor fusion and data aggregation in the wireless sensor network domain. The networking protocols can in this way inherently adjust data aggregation and -processing criteria according to the requirements posed by external subscriber systems. In this way, energy efficiency, which is paramount in wireless sensor networks, is optimized without sacrificing the flexibility of Web services. Below we present the Web services software, which was used in the SOA Pilot. For the complete wireless sensor network experiment details, see [2].

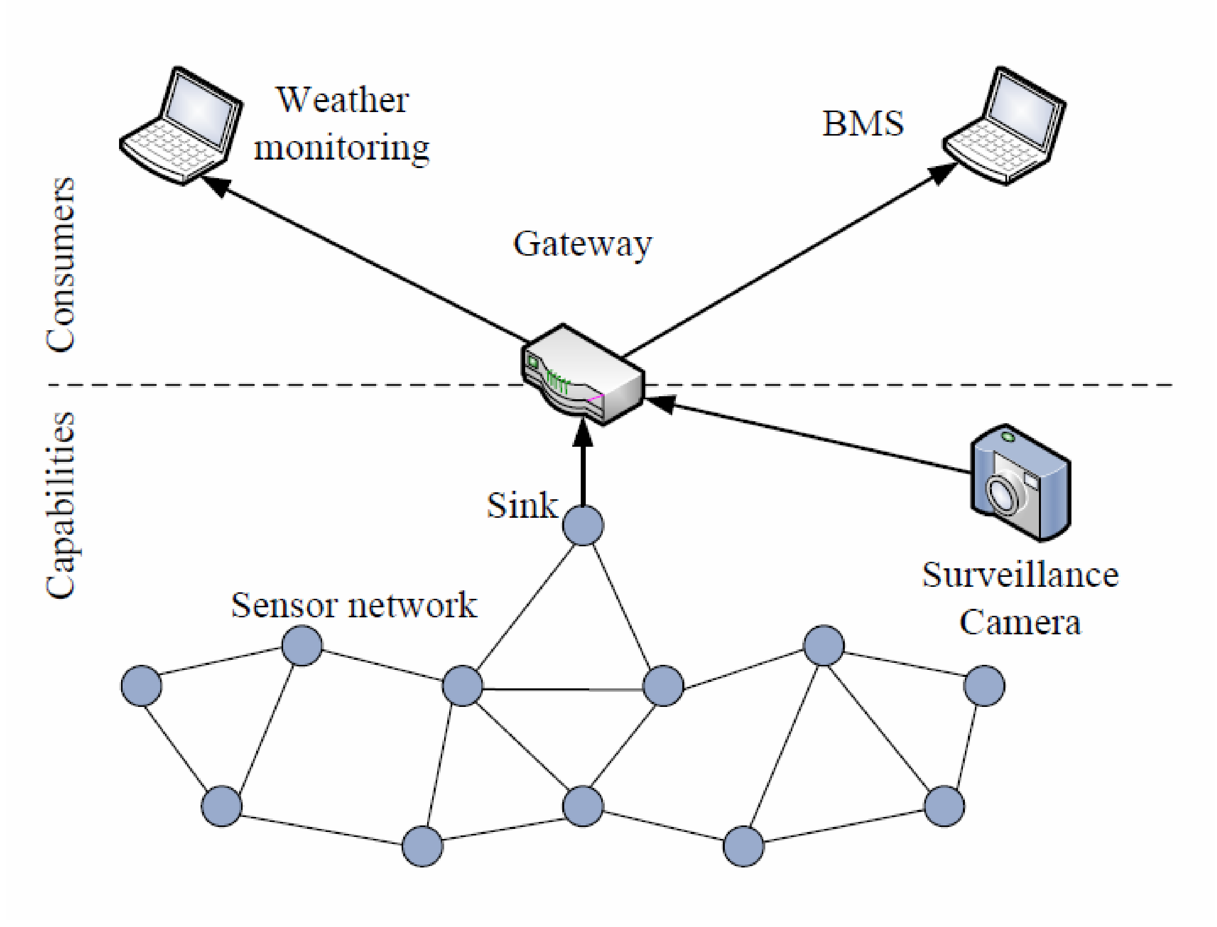


Figure 3.1 Sensor network enabled as a service providing capabilities to different consumers

3.1.1 Concerns regarding service-enabling a wireless sensor network

Since wireless sensor networks have scarce resources in terms of available bandwidth, battery, and computational power, it does not make sense to attempt to service-enable each and every sensing node. Instead, we use a wrapping approach, thus allowing existing mechanisms to be used within the wireless sensor network, while nodes external to the wireless sensor network may configure and receive information from the network using Web services. External consumer systems are for example Battlefield Management Systems (BMS) or Weather Monitoring Stations, see Figure 3.1.

We do not, however, consider Web services only as an information-sharing and interoperability entity. In our architecture, we also suggest the use of a Web services gateway as a query processing system publishing relevant sensing and alarm-criteria to the wireless sensor network domain. The networking protocols can in this way inherently adjust data aggregation and processing criteria according to the requirements posed by the external subscriber systems. In this way, energy efficiency, which is paramount in wireless sensor networks, is optimized without sacrificing the flexibility of Web services.

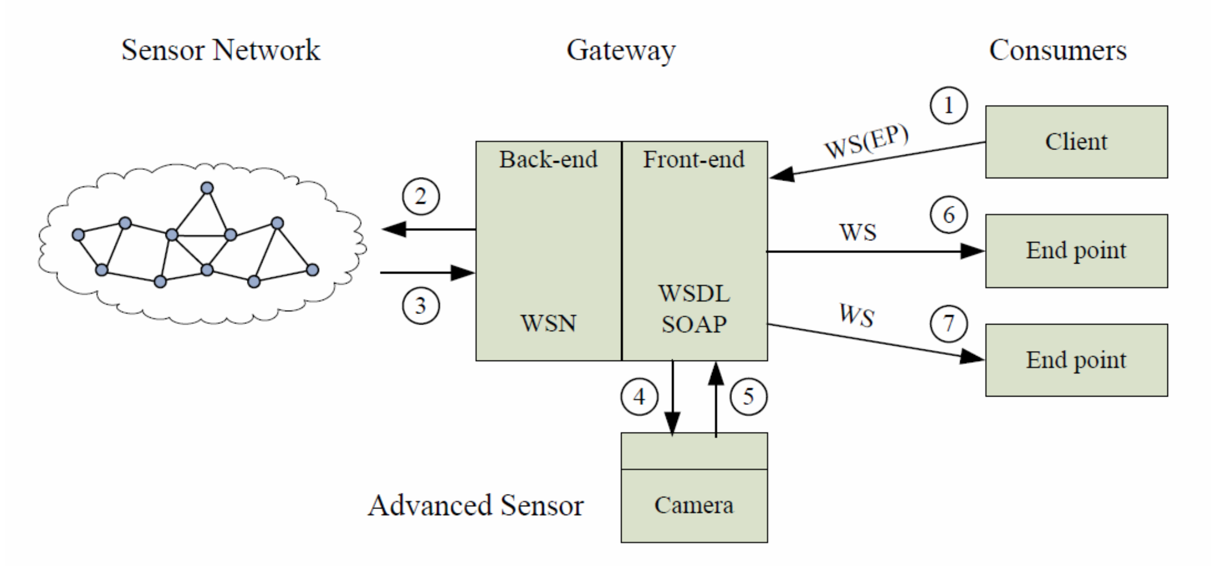


Figure 3.2 Architecture (WSN = Wireless sensor network)

3.1.2 Our sensor network gateway: A Web service wrapper

The gateway contains the Web services wrapper and provides an interface (a front-end) to the wireless sensor network using established Web services standards (see Figure 3.2). A WSDL file defines the interface, data types and message flow, whereas SOAP is employed for message transmission. This part of the wrapper is accessible to other systems using COTS Web services technology. The Web service interface allows external clients to configure queries for the wireless sensor network, and register a service endpoint (EP) for pushed information. In other words, our wrapper supports the publish/subscribe pattern, in that clients register a query (step 1, subscription providing recipient EP) and results of this query (be it periodic reports or spontaneous alarms) are sent (i.e., published directly to the consumers in steps 6 and 7) to the registered service endpoint. A client connecting to the gateway is typically a Battlefield Management System, requesting alarm reports when a subset of the sensing nodes detects an intruder which is trespassing in the area monitored. When the wireless sensor network reports to the gateway (step 3) about a detected target, the gateway sends a request to a separate Web service enabled camera (step 4) to take a picture covering the area monitored.

The target information (from step 3) and the picture provided (by step 5) are combined to a report sent to the BMS endpoint (e.g., step 6 and/or 7). COTS Web services technology is used to implement step 1 as well as steps 4 through 7, limiting proprietary solutions only to the functionality implemented in the back-end system, i.e., steps 2 and 3. Another Web services client could be a weather monitoring station, requesting periodic temperature or humidity reports.
In addition to supporting third party consumer applications, the architecture can also provide special case Web services for example to provide network developers with real-time information about the network at any given time, either during the initial deployment, create mid-life status reports, or to assist redeployment of energy exhausted nodes. These reports can be forwarded to a dedicated monitoring endpoint.

3.1.3 Sensor capability in the SOA Pilot

In the SOA Pilot, we used a subset of the above described implemented functionality. There, our setup was only triggered by motion, and we had no subscriptions for the temperature or camera capabilities. Since the SOA Pilot was built around two information formats – incidents and blue force tracks – the sensor network output was presented in the pilot infrastructure as an “FFI incident”, i.e., coded according to the incident schema (see Appendix A). The incident was delivered over MRR to the integration core software (point-to-point Web service interface), which would then re-distribute the incident to the appropriate subscribers using Web services notification. Since the MRR offers relatively low bandwidth, we used DSProxys [27] at each end of the connection to enable the use of Web services.

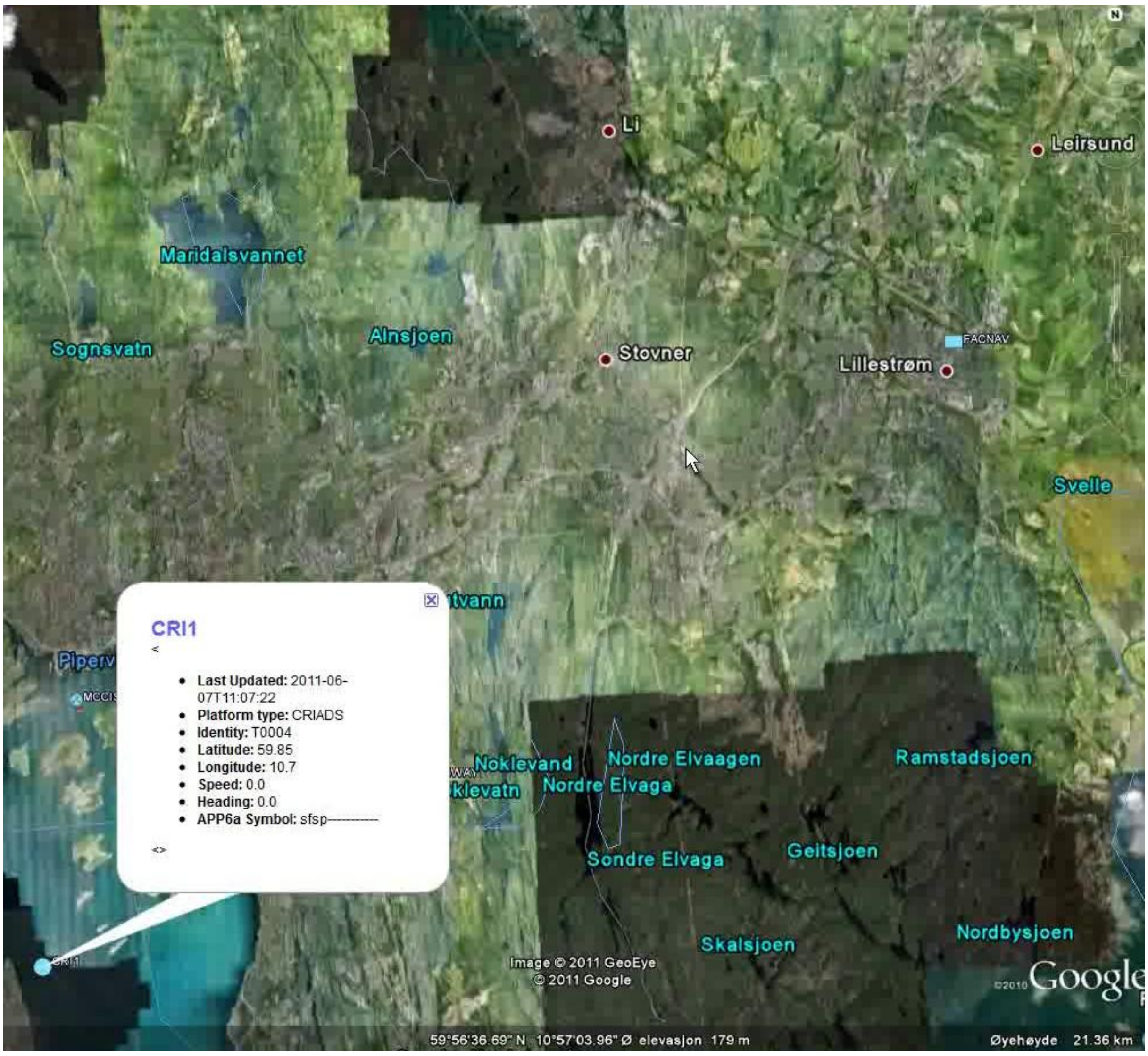
3.2 Google earth as a BFT viewer

Google earth (http://www.google.com/earth/download/ge/) is a freely downloadable application providing zoomable satellite maps of the earth. The application can be used for generic map browsing, or it can be used for special purpose visualization by feeding it with KML[3] formatted data. In the SOA Pilot, we used our Viewer as the main application for consuming and visualizing publications – both NFFI and incidents. However, as a proof-of-concept to show how simple it is to replace parts of a SOA-based infrastructure, we created an alternative blue force tracking (BFT) viewer using Google earth.

The setup was as follows: First, we installed Google earth on a computer in the network. On this computer we created a Web services notification subscriber, which set up a subscription for NFFI

[3] KML is a file format used to display geographic data in an earth browser such as Google Earth. It uses a tag-based structure with nested elements and attributes and is based on the XML standard. A tutorial is available at: http://code.google.com/apis/kml/documentation/kml_tut.html

tracks from the local HQ. This ensured that whenever new tracks were available at the HQ, they would be pushed to our subscribing application. This application could then translate NFFI to KML, and output the KML to a file. Converting NFFI to KML is straightforward, provided you are only interested in visualizing the NFFI positional data type (PDT). The PDT contains information such as latitude, longitude, and altitude which can position the track in a map. The unit description string must be replaced by an URL in KML, where the URL points to a graphical representation of the APP6A symbol in the network. Google earth was configured to read the KML file at regular intervals, and would visualize the points encoded therein. So, by doing this simple transformation we were able to show BFT information in Google earth. Figure 3.3 below illustrates a screenshot from Google earth, with an expanded view of one of the tracks originating from the CRIADS system (see the following section).



CRI1

<

- **Last Updated:** 2011-06-07T11:07:22
- **Platform type:** CRIADS
- **Identity:** T0004
- **Latitude:** 59.85
- **Longitude:** 10.7
- **Speed:** 0.0
- **Heading:** 0.0
- **APP6a Symbol:** sfsp----------

<>

Figure 3.3 Google Earth screenshot

3.3 Web services-enabling CRIADS

CRIADS is a system for coastal radar tracks. It can export several formats, among them OTH-gold (Over-The-Horizon Gold) . There is, however, currently no support for exporting NFFI tracks. In our demonstration we chose NFFI as the common format for exchange of track information in our infrastructure, and needed to provide all output coded in this format. OTH-gold is a text format for, e.g., tracking. Messages can be issued whenever there is new information available, and each track has a unique identifier. The current message can reference previous identifiers (i.e., update or delete them) so a track store is necessary to keep an overview of the current tracks. To convert OTH-gold to NFFI we first created a simplified OTH-gold track store, with support for those parts of the specification used by CRIADS. This track store connected to a TCP socket in CRIADS providing an OTH-gold feed. The track store used a mediation service to translate OTH-gold to NFFI, and exposed the NFFI track as an NFFI SIP3 request/response service. This service was then publish/subscribe-enabled by a connection from the JBridge[4]. The JBridge would invoke the NFFI track service at regular intervals, and publish the response using WS-Notification via WSMG, a freely available implementation of WS-Notification [12]. This approach is best used when you want to disseminate tracking information at regular intervals, and already have an existing request/response NFFI service, and is illustrated in Figure 3.4 below.

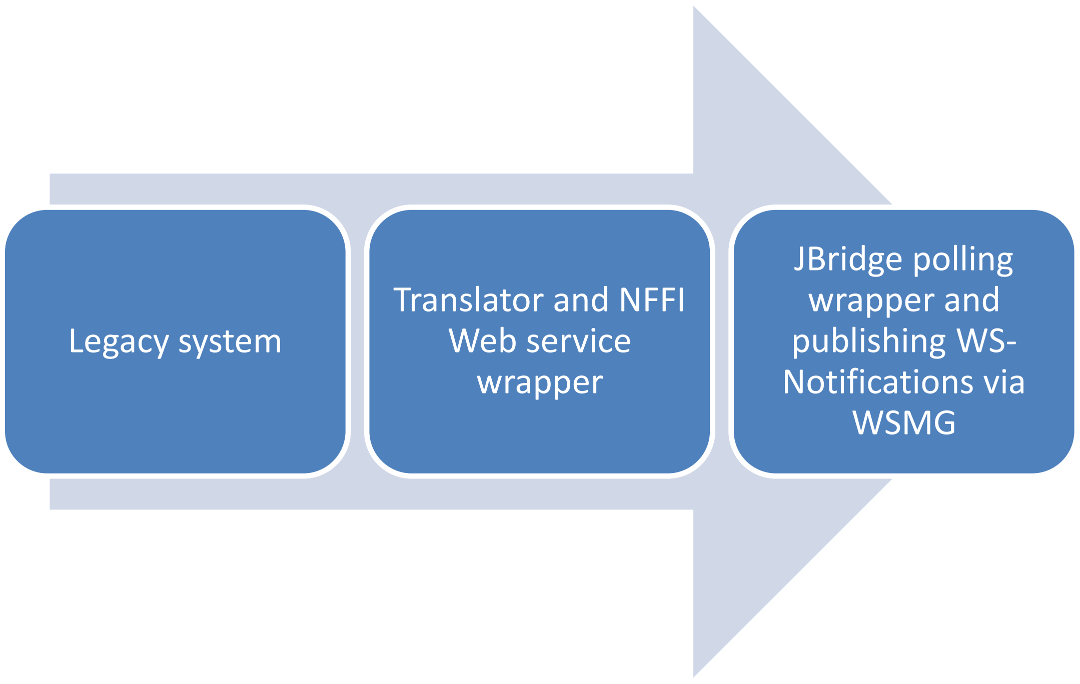

Legacy system
Translator and NFFI Web service wrapper
JBridge polling wrapper and publishing WS-Notifications via WSMG

Figure 3.4 Web service-enabling a legacy system (e.g., CRIADS) by means of a Web services wrapper and WS-Notification framework. (Information flow from left to right, consumers not shown)

[4] JBridge is a helper application developed at FFI that is used for publish/subscribe-enabling request/response Web services. This is described in Section 8.3.

3.4 Web services-enabling NORCCIS II

NORCCIS II can export its tracks using several formats, NFFI included. However, the NFFI support in the version of NORCCIS II that we used follows an older specification using NFFI over TCP, and not the Web service interface from the current NFFI SIP3. We solved this by creating an NFFI TCP client that we connected to the configured NORCCIS II export socket. The tracks pushed over this socket were NFFI 1.3 formatted, i.e., they used the same schema as SIP3 does. Thus, creating a Web service wrapper was a straight forward task, where tracks received were exposed as a SIP3 Web service. This means that even if both NORCCIS II and CRIADS are fundamentally different systems, the Web services offered from them used the exact same interface. Thus, the JBridge was used in this case as well, in the same manner as described above for CRIADS.

3.5 Web services-enabling NIRIS

NIRIS (Networked Interoperable Real-time Information Services) is a middleware integration core exploiting Internet technologies to provide a recognized air picture in near real-time. It can export its track store in different formats. The version we got in our lab was unable to export NFFI, but it could export NVG. NVG is a flexible XML-based format which can describe incidents, positioning information, etc. When used for tracking, NVG contains a lot of information which cannot be expressed by NFFI. In our NVG to NFFI conversion module we extracted the data that could be mapped to the mandatory parts of NFFI. We used the same NFFI Web service as for CRIADS and NORCCIS II, but in this case populating it with data converted from the NVG interface in NIRIS.

3.6 Web services-enabling the CSD

We had an installation of the Coalition Shared Database (CSD). Its role in the SOA Pilot was providing us with a source for images. The CSD was wrapped in a Web service that could provide incidents with images (see Appendix A for the incident schema). The images needed to play the scenario were loaded into the CSD by an operator at the appropriate time in the timeline, and thus made available for publishing over WS-Notification using the framework described below via a Web services wrapper.

3.7 Web services-enabling NORMANS

FFI develops concepts, requirements and technology for the future network enabled soldier. An overview of the Norwegian Modular Network Soldier (NORMANS) is given in [9], and the C2I system is presented in [10].

Previously, we have Web services-enabled NORMANS at CWID 2007. Since then the NORMANS protocol has been revised. We created a new wrapper for NORMANS in the Viewer in a similar manner as that used at CWID 2007 (see [11] for details of the experiment).

4 Publish/subscribe framework: OASIS WS-Notification standard

Publish/subscribe is a well-known communication pattern for event-driven, asynchronous communication. The pattern is particularly well suited in situations where information is produced at irregular intervals. Simply speaking, publish/subscribe means that you will only receive the information that you have subscribed to. This concept utilizes a combination of push and pull. As opposed to a general *push* mechanism there are benefits in that you may select (subscribe) to the information sent to you, and when using pure *pull* principles you are not able to notify listeners when events occur.

4.1 WS-Notification

WS-Notification [5] is an OASIS standard defining publish/subscribe for Web services. There are three parts in the specification:

- WS-BaseNotification [6] which defines standard message exchanges that allow one service to subscribe and unsubscribe to another, and to receive notification messages from that service.
- WS-BrokeredNotification [7] which defines the interface for notification intermediaries. A notification broker is an intermediary that decouples the publishers of notification messages from the consumers of those messages; among other things, this allows publication of messages through a chain of proxies.
- WS-Topics [8] defines topic-based filtering using an XML model to organize and categorize classes of events into “Topics”. It enables users of WS-BaseNotification or WS-BrokeredNotification to specify the types of events in which they are interested.

In summary, the specifications standardize the syntax and semantics of the message exchanges that establish and manage subscriptions and the message exchanges that distribute information to subscribers. For further discussion of publish/subscribe in general and WS-Notification in particular in a military context, see [4].

4.2 Publish/subscribe in the SOA Pilot

In the SOA Pilot we used a freely available implementation of WS-Notification called WSMG [12], which is provided as open source by the University of Indiana. The framework implements the WS-BaseNotification standard, among other things. There are some software bugs present that limit WSMG’s usability, and being a student project the software is no longer maintained. Thus, future experiments should explore alternative frameworks for offering a more complete and reliable WS-Notification implementation[5].

[5] After the SOA Pilot, Project 1176 has developed its own implementation of the core parts of WS-BaseNotification and WS-BrokeredNotification. This implementation will be used in future SOA activities at FFI.

5 Service discovery: OASIS WS-Discovery standard

Discovery of Web services can be performed either by utilizing a service registry, or a decentralized non-registry based solution. There are three standards addressing Web services discovery, two registries and a non-registry solution. The registries, UDDI and ebXML, suffer from liveness and availability problems in dynamic environments, as described in [3]. The third standard for Web services discovery is WS-Discovery [17]. It is better suited to dynamic networks than the registries in that it is a decentralized discovery mechanism, thus removing the single point of failure that a centralized registry constitutes.

We attempted to use WS-Discovery in a type of disadvantaged grid, and found that it was unsuitable for use there since it generated too much traffic in the network and flooded the modem buffers, as described in [19]. If we can reduce the overhead of WS-Discovery, however, it may be better suited for use in military networks as well. Recent work by the W3C regarding efficient XML interchange (EXI) can potentially make WS-Discovery suitable for both civil and military networks. For evaluation purposes we have combined an open source implementation of WS-Discovery with an open source implementation of EXI.

5.1 WS-Discovery overview

WS-Discovery is the newest standardized Web services discovery mechanism. After being a draft since 2005 [16], it became a standard in 2009 [17]. WS-Discovery is based on local-scoped multicast, using SOAP-over-UDP [18] as the advertisement transport protocol. Query messages are called probe messages. Services in the network evaluate probes, and respond if they can match them. To ease the burden on the network, WS-Discovery specifies a discovery proxy (DP) that can be used instead of multicast. This means that WS-Discovery can run in two modes, depending on whether there is a DP available or not. However, this DP is not well-defined in the standard. The standard fully describes the decentralized operation of WS-Discovery, but the functionality of (and integration with) the DP is left to be implemented in a proprietary manner for now. We evaluate only the standardized parts of WS-Discovery in this report, focusing on decentralized operation.

5.2 Compression

Efficient XML (EFX) was one of the formats the W3C XML Binary Characterization Working Group investigated during their work with requirements for a binary XML format. It was later adopted by the W3C Efficient XML Interchange Working Group (EXI) as the basis for the specification of the efficient XML format. The objective of the EXI Working Group is to develop a specification for an encoding format that allows efficient interchange of the XML Information Set, and to illustrate effective processor implementations of that encoding format. EFX was originally developed by Agile Delta and provides a very compact representation of XML information. There also exists an open source Java implementation of the EXI specification called

"EXIficient[6]". In this report we use the open source implementation of EXI. For further discussion of EXI and its applicability to SOAP-over-UDP in WS-Discovery, see our paper [20].

5.3 Implementation and evaluation

An open source implementation of WS-Discovery written in Java is available from [15]. The release at the time of the experiment (0.2.0) was only draft compliant[7], but the version in the repository adhered to the standard. We therefore downloaded the most recent WS-Discovery revision from the open source repository (which was revision 116 at the time we performed our experiments). The evaluation was performed in two iterations: First, we evaluated the WS-Discovery standard on its own. Second, we evaluated WS-Discovery with added EXI compression. For the evaluation of the standard we compiled the sources and used the software unmodified. To evaluate the standard with EXI compression, however, we had to make some modifications:

First, we modified parts of the SOAP-over-UDP library, where we added a new transport class that would apply EXI compression and de-compression to outgoing and incoming UDP packets, respectively. We enabled all the compatibility parameters for EXI (along with the parameter for maximum compression), thus ensuring that the lexical integrity of the XML documents was preserved. This was done to ensure that WS-Discovery functioned properly: Enabling these EXI compatibility parameters mean that compression rate is slightly reduced, but it ensures that all namespaces and other metadata are preserved. This is especially important if one wants to employ security measures, as changes to the document will break cryptographic signatures.

Next, we made two changes to the WS-Discovery library, where we added our new EXI capable transport under available transport types, and finally set this transport to be the default to be used. Finally, we compiled the libraries and repeated the tests made with the standard implementation. Our modified code has since been submitted back to the open source domain, so that the current version of WS-Discovery in the repository mentioned above now also has EXI support.

We evaluated WS-Discovery using WSDLs for services such as finance, news, weather services, etc. These WSDLs were fetched from [13] and [14], which provide lists of freely available Web services. Also, the WSDLs from Google and Amazon’s search services were included, yielding a set of 100 WSDLs in total. This provided us with a representative set of interfaces (see Table 5.1) for a wide array of Web services which we could use in our evaluation.

Minimum size	Maximum size	Average	Standard deviation	Median
1643	149342	13830	19202	8514

Table 5.1 Sizes (in bytes) for our 100 WSDL files

[6] Available from http://exificient.sourceforge.net/.
[7] At the time of writing, the current release is version 1.0beta1. This version adheres to the WS-Discovery standard.

We used Wireshark[8] version 1.2.1 for Windows to capture data traffic in a small network with two nodes. This enabled us to capture actual WS-Discovery traffic, and examine the packet payload sizes, thus giving a foundation for further analytical study.

5.4 WS-Discovery network usage evaluation

The standardized WS-Discovery behavior is a decentralized discovery protocol. Services are required to send UDP multicast HELLO messages that advertise when they become available. Also, services should send UDP multicast BYE messages when they go away. If services are able to do this, then each node will have an up-to-date view of the available services. In a dynamic network we cannot rely on receiving all such messages. It is also possible to actively query the network by sending PROBE messages. In order to accurately mirror the current network state of a dynamic network probing must be used, in which case each node replies with UDP unicast PROBE MATCH messages. This generates a lot of data traffic, but is required to ensure an up-to-date view of the available services. The standard requires all multicast packets (i.e., HELLO, PROBE, and BYE messages) to be sent twice, and the unicast PROBE MATCH messages to be sent once. Since we are concerned with WS-Discovery in dynamic environments, we focus on the HELLO, PROBE, and PROBE MATCH messages in this report.

WS-Discovery is based on a query-response model, where a multicast query (probe) triggers unicast responses (probe match). The load incurred on the network by the number of querying nodes (q) in a network with a total number of n nodes can be calculated using the formula below. If all nodes should have an up-to-date view of the currently available services, then q = n, conversely, if only one node is querying, then q = 1.

$$\text{LOAD} = (\text{sizeof(probe)} + \text{sizeof(probe match)} * (n - 1)) * q$$

In our tests the HELLO messages yielded different sizes (see Table 5.2) depending on the different WSDLs that were published (two HELLO messages generated per WSDL).

An uncompressed PROBE message was always 581 bytes (using a generic probe querying for all available services with no scope limitations). An EXI compressed PROBE message varied between 272 and 274 bytes (compression varying with varying UUID and time stamp in message; for simplicity we assume a compressed size of 273 in our calculations below as this is the average over time). According to the standard the message had to be transmitted twice, meaning that sizeof(probe) = 2 $*$ 581 bytes for uncompressed traffic (EXI compressed sizeof(probe) = 2 * 273 bytes).

[8] Available from http://www.wireshark.org/.

Compression	Minimum size	Maximum size	Average	Standard deviation	Median
Uncompressed	807	887	834	17,04	830
EXI	373	420	390	9,22	388

Table 5.2 HELLO message payload statistics (in bytes), calculated from HELLO messages corresponding to the Web services described by our 100 test WSDL files. For each Web service that is published, WS-Discovery sends two identical such messages

The PROBE MATCH varied in size with the number of services published, since it contained all the services published by a node. Table 5.3 shows the different sizes of PROBE MATCH messages sent by a node with 1 to 100 services published. We see that publishing just one service incurs a lot of overhead (1092 bytes to disseminate information about it), whereas for a larger number of services this overhead is reduced (more actual Web services porttype information in the response compared to SOAP headers, etc). Calculating the average when publishing multiple services (i.e., the average of the message sizes divided by the number of services) yields 497 bytes for uncompressed WS-Discovery, and 130 bytes for EXI compression. UDP can carry a payload of 65507 bytes, meaning that WS-Discovery has a theoretical upper limit of publishing approximately 65507/497 ≈ 131 Web services per node when considering results from our 100 WSDLs. Conversely, with EXI compression we may publish around 65507/130 ≈ 503 Web services per node. Naturally, the number is approximate, because in practice varying namespace lengths in different WSDLs can affect the PROBE MATCH size. We can also see that an increase in the number of services in a PROBE MATCH leads to an increased compression rate, because of recurring patterns in the XML encoding of the service information. For just one service, the compression rate is 511/1092 ≈ 0.47, whereas for a 100 services the compression rate has increased, yielding 5009/38200 ≈ 0.13.

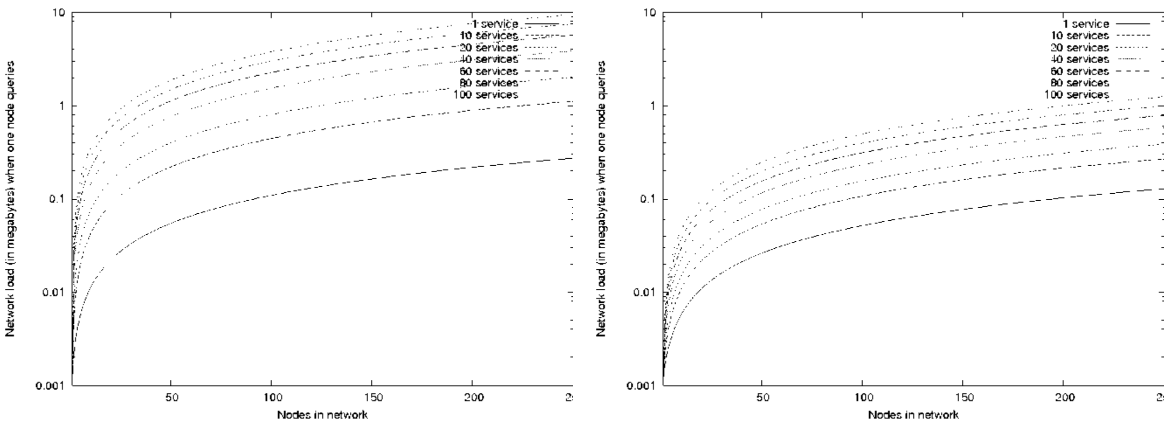
Number of services	Uncompressed PROBE MATCH	EXI compressed PROBE MATCH
100	38200	5009
80	30292	4000
60	22902	3146
40	15531	2339
20	8122	1552
10	4476	1072
1	1092	511

Table 5.3 The size (in bytes) of the unicast PROBE MATCH message sent by a node publishing a certain number of Web services with WS-Discovery

Using the LOAD equation, we fill in values for sizeof(probe match) using values from Table 5.3, as well as the above mentioned sizeof(probe). The number of nodes in the network, n, is varied from 1 to 250. First, we set q = 1, meaning that only one node is querying. Figure 5.1 illustrates

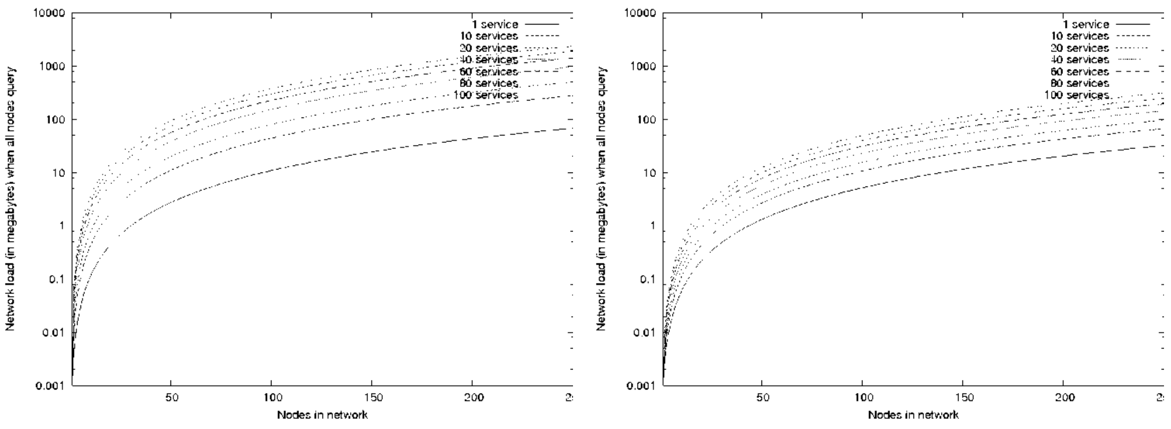
WS-Discovery’s resource use (in megabytes) in this case when one node is querying in networks with up to 250 nodes. This means that in such a network, for every query issued, we get the resource use indicated by the graph.

Next, we set q = n, so that in a network of a given size, all nodes query. In both cases, this means that the querying node(s) send PROBEs and receive(s) n PROBE MATCHES. Figure 5.2 shows WS-Discovery’s resource use in the case where q = n. In both graphs, all nodes are equal and publish the same number of services. Please note that the graphs have a logarithmic Y-axis to ease comparison between uncompressed and compressed results. We see that with an increasing number of nodes and published services, the overall resource use increases substantially.



Uncompressed WS-Discovery EXI compressed WS-Discovery

Figure 5.1 Resource usage of WS-Discovery when one node queries



Uncompressed WS-Discovery EXI compressed WS-Discovery

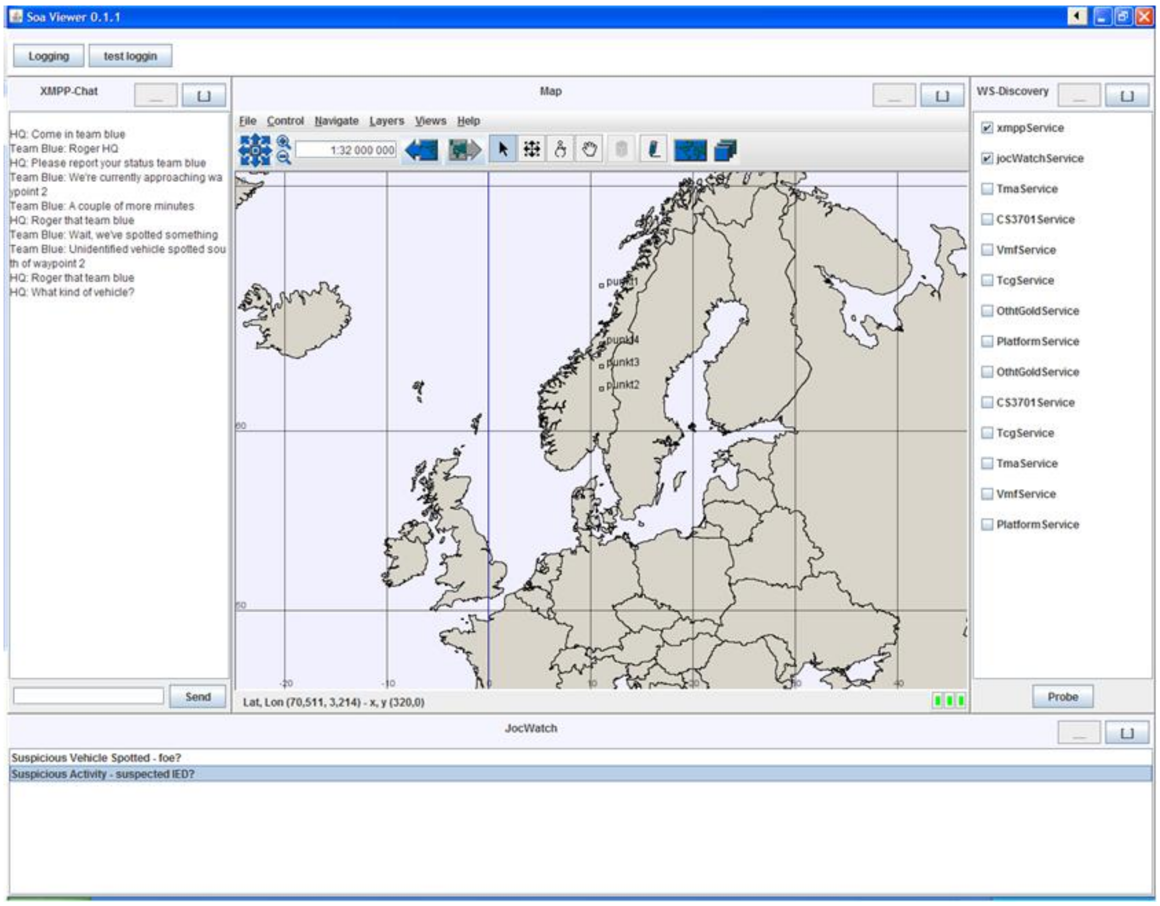
Figure 5.2 Resource usage of WS-Discovery when all node queries

5.5 WS-Discovery in the SOA Pilot

In the SOA Pilot we used the EXI-enabled version of WS-Discovery. The library was integrated into the Viewer, and used there to provide the user with an overview of available services. The Viewer is further described in the next section.

6 The Viewer

The Viewer was developed by the FFI project 1176 for use as an information integration and visualization core during the SOA Pilot. It was not intended to compete with existing visualization systems in any way, but rather created to illustrate how existing services can be utilized by new software, in the same manner as we used Google earth as described earlier.



HQ: Come in team blue
Team Blue: Roger HQ
HQ: Please report your status team blue
Team Blue: We're currently approaching waypoint 2
Team Blue: A couple of more minutes
HQ: Roger that team blue
Team Blue: Wait, we've spotted something
Team Blue: Unidentified vehicle spotted south of waypoint 2
HQ: Roger that team blue
HQ: What kind of vehicle?

- [✓] xmppService
- [✓] jocWatchService
- [ ] TmaService
- [ ] CS3701Service
- [ ] VmfService
- [ ] TcgService
- [ ] OthtGoldService
- [ ] PlatformService
- [ ] OthtGoldService
- [ ] CS3701Service
- [ ] TcgService
- [ ] TmaService
- [ ] VmfService
- [ ] PlatformService

Lat, Lon (70,511, 3,214) - x, y (320,0)

Suspicious Vehicle Spotted - foe?
Suspicious Activity - suspected IED?

Figure 6.1 Screenshot of the Viewer

The graphical user interface (GUI) of the Viewer is shown in Figure 6.1. In the middle, the map is displayed, together with tracks received from the operational systems. The list on the right side displays services that are available at the moment, while the left side is a chat window. The list at the bottom shows incidents that have been reported. Based on the mapping specified in the Viewer's configuration, the GUI can be, by hiding or showing the different panels described above.

The Viewer was developed using Java, and provides functionality for consuming, visualizing, aggregating and re-publishing services in a service oriented architecture. Specifically, the user is able to subscribe to available services, and the Viewer will receive push-based notifications whenever the services offer updated information. WS-Discovery was chosen for publishing and discovering services, and the Viewer was integrated with the EXI-enabled library discussed above.

The Viewer's GUI lists all available sources found by using WS-Discovery, and when checked by the user, the services are subscribed to using WSMG's client library, effectively sending a subscription message to the WS-Notification subscription manager. The WS-Discovery messages were created so that the *portType name* field would contain the human-readable service name, the *scope* field would contain the service topic, and the *Xaddr* field would contain the WS-Notification server address. By sending WS-Discovery probe-messages at regular intervals, the Viewer offers liveness-information to the user, by making the GUI only display services currently up and running.

In order to present geographic information-based services in a map, the Viewer was integrated with OpenMap, a Java-based open source toolkit for showing map data. The toolkit fetched maps from the Norwegian Mapping Authority (Statens Kartverk) using Web Map services (WMS). Thus, the Viewer was able to display detailed maps over Norway in multiple scales. The XMPP protocol was chosen for instant messaging in our experiments, as this is the de facto chat protocol in NATO. The Viewer offers multi-user chat by integrating with Ignite Realtime's Smack API, communicating with XMPP servers such as Openfire and Prosody. The Viewer's information flow is illustrated in Figure 6.2.

Upon receiving incident reports, the Viewer will present the entries in a list, and when clicked upon, a popup dialog will display detailed information about the event, together with an image of the incident if provided.

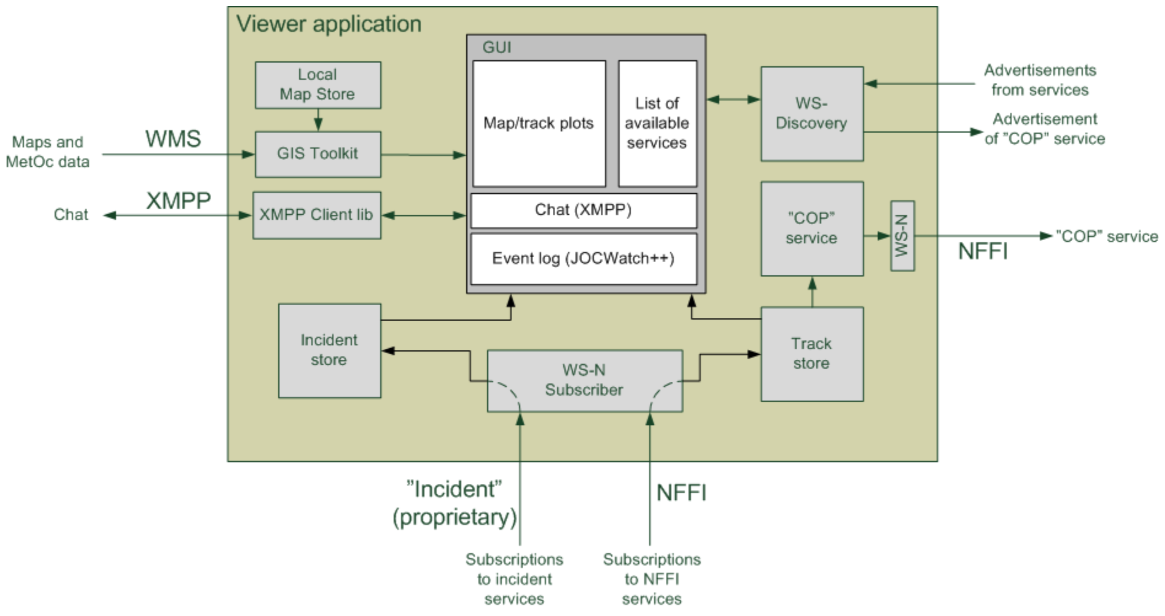


Figure 6.2   Information flow in Viewer

In addition to consuming services, the Viewer provides functionality for aggregating information and re-publishing it as a new COP service. For the experiment, this functionality was configured to aggregate all geographic information based entities (Tracks), and re-publish them as a COP service. By specifying a topic and a WS-Notification broker address, the Viewer would publish the COP whenever its internal model changed, allowing other systems to subscribe to the COP-topic and to receive notifications.

Being a lightweight Java-application, the Viewer can be run on many types of devices including many tablets and even some mobile phones.

7 Security: Data diodes

Military communication systems have strict security policies, which in some cases require ensuring that no piece of information, however small, is allowed to flow from one information domain to another. This challenge can be addressed by having networks that are fully disconnected from others, and moving information between domains over an air gap. However, when there is a need for a steady flow of information into, but not out of, a system, a common approach in military systems is to use what is known as a *data diode*[9]. Such a diode ensures that information can only flow in one direction, from what is called the low side to the high side.

A data diode is in principle a simple concept based around a hardware device that only supports one way communication. In addition to the hardware device, commercial diode solutions also include software functionality that can be used to move data across the diode. One example of such functionality would be file transfer, such as a one-way FTP service.

7.1 Web Service Communication Patterns

As mentioned previously, the basic communication between Web service entities can be divided into three simple steps. The situation in real deployments can however be more complex, and in this section we present the most common communication patterns in more detail, and discuss if, and how, each of them must be adapted for use across a data diode.

Web services technology is based on two-way communication between nodes, and most communication between parties can involve messages being acknowledged also on the Web service level. This means that the introduction of a data diode might require changes to the message exchange patterns between Web service entities in addition to the changes on the transport level.

7.1.1 Service Discovery

Service Discovery is a term used for any mechanism that allows a service provider to make its service known to potential service consumers. The traditional way to implement service discovery for Web services is in the form of a stand-alone registry. In this case the communication with the registry is limited to two request/response patterns, one when the service provider publishes its services, and one when the service consumer queries the registry.

[9] Also known as *information diode*.

More recent service registries are not limited to function in a stand-alone manner, but have the ability to work together in a federation, or share service information between registries. This leads to a new set of communication patterns that must also be taken into consideration.

When two or more registries form a federation, or set up information sharing, this can be done in one of two ways. Either the registries replicate data between them, or they use so-called federated queries, where the queries sent to one registry is passed on to the other members of the federation. In the latter case, the responses sent by the other registries are then consolidated, and sent back to the requesting consumer by the original registry.

The third standard for Web service discovery, WS-Discovery, is a fully distributed solution, which relies on IP multicast for service announcements. In this case, a service Provider will send out a multicast message describing its services. Consumers will then either query a local cache or send a multicast probe which the service providers will respond directly to.

In cases where several independent networks are interconnected, it is possible that the different networks will be using different service discovery mechanisms. In this case it is possible to share information between the different mechanisms using either a common translation format known to both mechanisms, or by using a gateway that translates between the mechanisms. In this case, the translation between mechanisms is performed within one network node, which means that no new communication patterns are introduced by such an interconnection.

Discovery Type	**Communicating entities**	**Pattern Name**	**Pattern Description**
Registry-based	Service Provider Registry	Publish	The service description is sent to the registry
Registry-based	Service Consumer Registry	Lookup/ Find	A query is sent from the consumer, and the registry responds
Registry-based	Multiple registries	Replication	A registry sends its service information to a different registry
Registry-based	Service consumer Multiple registries	Federated query	A query is sent from the consumer, the registries share the query and return a response to the consumer
Distributed	Service Provider Service Consumer (optional)	Service announcement	The service provider sends an announcement, which may be received by one or more consumers
Distributed	Service Consumer Service Provider (optional)	Probe	The consumer multicasts a probe message, and service providers may respond with their matching services

Table 7.1 Service discovery communication patterns

7.1.2 Service Invocation

Once a service consumer knows of the existence of a service, it can contact the service directly. The communication between the service and the consumer can be done according to one of two main patterns, namely request/response and publish/subscribe.

The simplest form of communication between service and consumer follows the request/response pattern. In this case, the consumer sends a request message to the service, which immediately responds to the consumer.

The publish/subscribe pattern is more complex, and can involve more entities than just the service and the consumer. The basic principle is that the consumer sends a subscription request to a subscription manager, letting it know that the consumer is interested in all or a subset of the information produced by the service. In this case, the subscription manager can be either the service itself, or a separate entity. After the subscription has been established, the service provider will send so-called notification messages to the consumer whenever there is new information available.

Communication type	**Communicating entities**	**Pattern name**	**Pattern description**
Request/Response	Service Provider Service Consumer	Polling	The consumer sends a request, which the provider responds to immediately
Publish/Subscribe	Consumer Subscription Manager	Subscribe	The consumer sends a subscription request to the subscription manager
Publish/Subscribe	Provider Consumer	Notification	The provider sends a message containing new information directly to the consumer.

Table 7.2 Service invocation communication patterns

7.2 Adapting Web service communication

When adapting Web services for use with a data diode, it is important to make a distinction between which communication patterns can be adapted, and which patterns it makes sense to adapt. One example is service discovery; while it might be possible to adapt registries to share service information across a diode, one need to consider the value of such information sharing. If a service registry on the low side provides a registry on the high side with information about the services on the low side, potential consumers on the high side will be able to discover the services on the low side as well. These consumers will, however, not be able to invoke these services directly, since they have no way of sending a request to the service. The only time it would make sense to have information about the low end service available on the high side is if there is some other means available through which the low side service can be invoked. One example of such a mechanism could be the end user logging in on a system on the low side, and issue the request on behalf of the high side consumer, e.g., if equipped with a multilevel security (e.g., MILS[10]) terminal [22].

In this section we consider the communication patterns from Table 7.1 and Table 7.2 in turn, and identify those patterns that will function in conjunction with a data diode.

For each pattern, we first need to identify the primary information producer. Due to information only being able to pass from the low side to the high side, the information producer will have to be on the low side of the diode. Note that the information producer is not always the one that initiates the communication between the entities, which and that this might stop some patterns from functioning across a diode.

[10] Multiple Independent Levels of Security

7.2.1 Service Discovery

For the first pattern, the *publish* pattern, the information producer is the service, which has a service description that it needs to pass to the registry. Having the service on the high side and the registry on the low side would make this pattern impossible, so we concentrate on the opposite case. A service on the low side could, by using this pattern, make its existence known to potential consumers on the high side. The high side consumers on the other hand, would not be able to contact the service directly. Therefore, this pattern is only useful if the consumers either simply want to know which services are available (but do not plan on using them), or if they have another way of contacting the service on the low side. The latter can be achieved by, for instance, the user logging in on the low side and subscribing to information that is later pushed through the diode to the high side.

A similar pattern is the *replication* pattern, in which cooperating registries share service information. This pattern can be used for one-way information sharing, in which all or a part of the content of the low side registry is copied into its partner registry on the high side. Interconnecting registries in this manner does not allow for automatic usage of the services on the low side (as the high side consumers cannot contact these services), but it can still be useful. It makes the high side users aware of which capabilities exist on the low side, which enables them to perform manual set-up of communication from the low side to the high side by for instance contacting the service owner, or logging in on the low side as described above.

The two other registry related patterns, namely the *lookup/find* pattern and the *federated query* pattern, both rely on information being provided as a direct response to a query initialized by the service consumer. The strict two-way demand of these patterns means that neither of them will function in conjunction with a diode.

Distributed service discovery is intended for use within a single network, and relies on multicast. Under the assumption that cross-diode multicast traffic is supported, the *service announcements* may cross from the low to the high domain. It does however suffer from the same usage limitations as the registry publish pattern.

WS-Discovery, the only standardized distributed Web service discovery solution, uses a pattern similar to the request/response invocation pattern for its *probes*. Both the probe and the probe match are of equal importance, and none can be omitted. This means that probing is not possible across a diode.

7.2.2 Service Invocation

The *request/response* pattern, which is the most common invocation pattern for Web services, relies on the consumer first issuing a request which the service then responds to. In most cases, the primary information producer is the service, whose response supplies the consumer with the requested information. The service cannot be made responsible for initiating the communication because it has no way of knowing which consumers will be interested in the information it can supply. Because of the synchronous nature of this pattern, its usefulness across a diode is fairly limited.

However, there exist a few potential service types where the majority of the information is carried in the request. These services, such as event logging services (reporting alarms from a software system etc), could function even if the response is not delivered to the consumer. These services would then function on a best-effort basis, since reliable delivery cannot be guaranteed, making it unsuitable for critical systems.

The other Web service invocation paradigm, publish/subscribe, relies on two patterns that are executed in order. The first step is the *subscription* pattern, which is generally executed once, which leads to the *notification* pattern being executed a number of times in the opposite direction. In the subscription pattern the eventual service consumer acts as both the information provider and the initiating entity, and supplies either the service or an external third party, a subscription manager, with its request for information. Even if the subscription pattern, seen isolated, can function from the low to the high side of a diode, its usefulness is limited when executed in this manner. This is because the subscription pattern is only an initial step in setting up the *notification* pattern, in which the information flow goes in the opposite direction of the subscription pattern.

The notification pattern is a one-way concept, where the service periodically sends a notification message to the service consumer without needing a reply. This means that notifications are well suited for use across diodes, but the service does rely on first having received a subscription message (either directly, or by delegating this responsibility to a subscription manager).
The WS-Notification standard, which is one of the two Web service notification standards, does allow for a third party to issue this subscription request on behalf of the service consumer [6, references therein]. This means that it is possible for a user on a high system to receive notifications from a low system without having initiated the subscription herself. If the user can either log in to the low system to issue the request, or have a user in the low system issue the request for them, notification can flow from the low to the high systems without requiring any feedback from the high side.

7.3 Proof-of-Concept Implementation

As previously mentioned, making Web service technology function one way across data diodes is best done by using an alternative transport binding (e.g., SOAP-over-UDP), in addition to the adaptations that have to be made on the Web service level. However, since most existing Web service solutions use the standard HTTP over TCP binding, we have implemented a solution that shows that is it possible to make these Web services function without any significant modifications to the service or consumer software. The only modification necessary is to adapt the addresses used so that the communication goes via a Web service proxy.

The data diode we used in our experiments is a proprietary prototype, which in addition to the fiber-based hardware, has software that supports automatic duplication of files across the data

diode. Commercial diodes can support more advanced communication patterns than this, but the prototype we used for our experiments was sufficient for our proof-of-concept implementation. Our proof-of-concept implementation and test use a piece of diode hardware, which ensures that information can only flow one way through it. In other words, it is a data pump capable of moving data from a domain of low classification to a domain of high classification. The diode comes with custom software, which can be configured to achieve the desired operation. The make and model of the diode and software is not important, because several products which all offer similar characteristics and high assurance exist. It suffices to say that we configured the software to take immediate action whenever new data was available, so that when new data became available in the low domain, it would be pushed to the high domain as soon as possible.
The diode is traditionally used to move data files across domain boundaries, often with a manual review and release inspection step along the way. Since we wanted to allow Web services traffic to flow across the domain, we set up a directory with the proper rights so that any new file (or update to an old file) which was placed in that directory would be duplicated in the high domain in a corresponding directory structure. This gave us the necessary infrastructure to enable one-way Web services invocations across classification domains, by going through an intermediary file which could be moved by the diode.

One of the main reasons for using Web services is that the technology is founded on standards[11], which ease interoperability and reduce chances of vendor lock-in. Thus, for our proof-of-concept implementation we have strived to employ standards according to what is defined in NATO Core Enterprise Services [28] where possible.

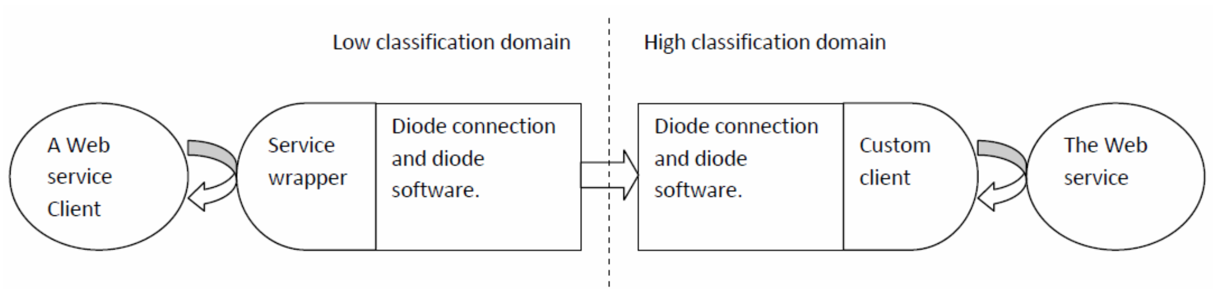

Low classification domain
High classification domain
A Web service Client
Service wrapper
Diode connection and diode software.
Diode connection and diode software.
Custom client
The Web service

Figure 7.1 Proof-of-concept architecture

Figure 7.1 gives an overview of our setup, where the circles indicate COTS implementations following Web services standards (i.e., W3C Web services where the interfaces are defined by a WSDL, and information exchange is performed over the SOAP protocol). Typically, Web services use HTTP/TCP as transport for the SOAP messages. However, when we have a diode as a gateway between networks, TCP (being bi-directional) will not function. Along the same lines, even Web services which do not have a return parameter usually need some acknowledgement on the HTTP layer for the completed transaction. As a consequence, we need a piece of software on both sides of the gateway that can function as a wrapper service and a wrapper client,

[11] Web services are defined in the WS-* suite of standards and recommendations. Furthermore, the definition of the NATO Core Enterprise Services [28] builds to a large extent on the WS-* suite.

respectively. We implemented a service wrapper in the low domain using the WSDL from the Web service we want to submit data to in the high domain. This ensures that to a client, the two are not distinguishable and can be used interchangeably. Thus, the COTS Web service client in the low domain is able to invoke our service wrapper without modification. Conversely, in the high domain we have created a client using the same WSDL, so that data posted by the COTS client can be submitted to the COTS service in the high domain once the data has passed through the diode. This means that the COTS client can use SOAP over HTTP/TCP when connecting to our service wrapper, whereas our custom client can use the same protocol suite when connecting to the COTS Web service. This allows us to use existing services and clients over a diode, provided the services are information sinks only (i.e., they do not need to issue any data as a response to the invocation). For each such service that we want to expose in the low domain, we must create a service wrapper and custom client that can accommodate SOAP messages over the diode. Since the diode we used operated on file structures, the service wrapper would take incoming requests, serialize the XML to a file, and then place this file in the directory structure the diode software was monitoring. In the high domain, our custom client was monitoring the corresponding directory structure, so that when the diode software had moved new data across from the low domain it would be read by our custom client. Our client could then de-serialize the XML, and use this as its data when invoking the Web service.

This experiment was performed to investigate the viability of Web services over a data diode (see our paper [21] for the complete discussion). Data diodes are a legacy solution, however, and in the SOA Pilot we used a more flexible prototype solution as described below[12].

7.4 Security in the SOA Pilot

Data diodes are the current way of realizing communication security for information flows between different domains. However, as described above, this is quite restrictive as it does not allow any information whatsoever to flow from the high to the low domain. In the SOA Pilot, an experimental XML guard gateway was used instead to secure the information in the networks. For complete details, see the report on security [26].

8 Recommendations and lessons learned

The complete SOA Pilot was a complex and heterogeneous system, with a large number of autonomous parts that were interconnected. In addition, several of the operational systems are difficult to use, normally requiring trained personnel. Thus, it was a challenging task making everything run at once. The main cause of this, however, was the choices we made with respect to infrastructure software. This will be further explained in this Section.

[12] The diode experiment was performed on request from the procurement project P8009, and is therefore included in this report, even though diodes were not used.

8.1 WS-Notification infrastructure (WS-Messenger)

Our original plan was to use an implementation of WS-Notification developed by IABG (Germany) as part of the CoNSIS project[13]. However, the deliverance of the software was delayed to such an extent that we were forced to find an alternative. We chose to use a freely available implementation called WS-Messenger (WSMG) [12], which is provided as open source by the University of Indiana. Part of the reason for this choice was that we had some previous experience with this software.

When used in the SOA Pilot, however, WSMG displayed a number of shortcomings, and it was the main cause of difficulties and errors in the pilot. The main problems of WSMG were:

- Proprietary tags in notification messages, meaning that the notification receiver had to be tailored for use with WSMG.
- No support for XML in the payload, which meant that the payload had to be Base64 encoded.
- If WSMG was unable to deliver a notification to a subscriber (for instance because the subscriber had crashed), it would cease submitting notifications to all subscribers of that topic. Instead, it would start buffering the notification, pending the return of the missing subscriber. This behavior was a major cause of delays in the experiments, since one subscriber crashing or losing connection would mean that everything had to be set up again from the start.
- WSMG has no easy way to delete subscriptions from the server. Instead, it was necessary to run an SQL script to delete the entry from the database. In a test environment where clients may be unstable, this was an annoying factor that slowed down the experiments.
- Finally, WSMG proved to scale poorly, even in a relatively small setup like ours.

Consequently, we will advise against using WSMG for WS-Notification, even in a test environment. It should also be noted that since WSMG is a student project, support and further development of the software is relatively uncertain.

In addition to WSMG, we did some brief testing with Apache Servicemix, an open-source ESB that supports WS-Notifications. The version we tested, 3.2.3, had several limitations:

- It was not 100% standards compliant. Among other things, the notifications were not sent as SOAP messages, but instead in a more REST-like way. This means that subscribers must be tailored for use with Servicemix. Also the required format of the subscribe messages deviated from the standard.
- Subscriptions could not be deleted without stopping and restarting the entire Servicemix instance.

[13] Coalition Network for Secure Information Sharing

Thus, we cannot recommend using Servicemix version 3.2.3 for WS-Notification. Later versions may have better support, though, but we have not looked into this. However, it should also be noted that being a complete ESB, Servicemix is a large and complex system with a considerable learning curve and it also takes quite some effort to configure it. This means that using Servicemix only as a WS-Notification framework may be considered as overkill.

As mentioned earlier, after the SOA Pilot was finished, Project 1176 developed its own implementation, called ‘microWSN’, of the core parts of WS-BaseNotification and WS-BrokeredNotification. This is a dedicated piece of software that *only* focuses on WS-Notification. We plan to use microWSN in future SOA activities at FFI that require publish/subscribe. However, microWSN is not intended for a production environment, in particular because it does not implement the entire WS-Notification standard. However, it should be no problem to extend it to support the full standard, if required. We are also considering releasing microWSN as an open source project, although no decision has been made yet on this.

8.2 Wrapper software

The actual principle of wrapping existing software with a Web services front-end works very well. Since Web services normally are stateless, the wrapper is relatively simple to implement, and with a WSDL file available, parts of it can be auto-generated.

On the other hand, the complexity of the wrapper is dependent on the type of connection it has into the actual operational system. Several of the systems offered an outgoing TCP connection that the wrapper was listening to. The messages received on the TCP socket were then transformed to an XML document and made available on the request/response interface.

As mentioned earlier, we only received information *from* the systems; we did not send information or commands *into* the system through the wrapper. Consequently, we had a rather weak integration between system and wrapper. The main reason for this is that the operational systems are large, complex systems with closed source code. It was not possible for us to interface directly with the internals of these systems, and we had to manage with whatever out-channels that were available, and the information provided through these.

This means that programming Web services wrappers for production use should be done by the vendor of the system in question. When new systems, or new version of systems, are purchased, Web services interfaces should be included as part of the specification, to avoid having to retrofit such interfaces.

Finally, in the process of transforming the networking and information infrastructure (NII) into a service oriented architecture, it is important to think beyond just Web services enabling existing systems. By turning systems into services, they become available to a much broader community than the existing, often stove-piped, systems are. This fact should be taken advantage of by analyzing business processes and required data flows in order to establish which interfaces are required and where. For instance, in some cases it may be advantageous to break existing monolithic systems into smaller, individual modules. In other cases, it may be necessary to go the other way and service enable an entire network (i.e., the entire network appears as a single service to the outside world. In the SOA Pilot, service enabling of both NORMANS and the sensor networks are examples of the latter).

8.3 JBridge

The JBridge is a helper application developed at FFI that is used for publish/subscribe-enabling request/response Web services. This is done by having a JBridge instance polling a request/response Web service at pre-defined intervals, and each time compare the response with the previous one (see Figure 8.1). If the response has changed, it is sent to the WSMG broker as a new notification (it is also possible to configure the JBridge to send a notification every time it polls, regardless of whether the response has changed or not).

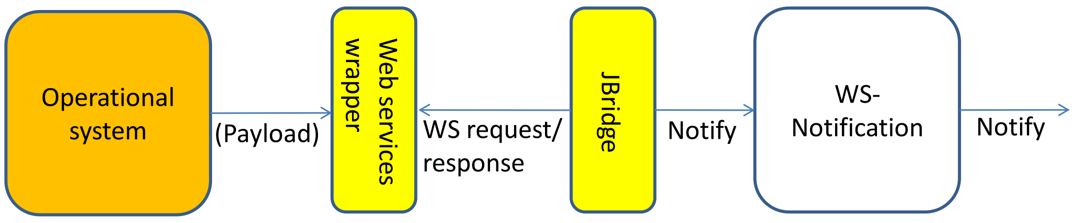


Figure 8.1 Using JBridge to Publish/subscribe-enable request/response Web services

The JBridge was only intended as a helper component for use in the SOA Pilot. It was introduced because we wanted to avoid having to use WSMG libraries in the wrapper to enable publisher functionality. Instead of building the publisher functionality into each wrapper individually, we chose to make one generic component that could be used in front of all Web services enabled operational systems. However, the JBridge brought increased complexity into the pilot, and it did represent an additional source of possible errors.

When Web service enabling operational systems in the future, we therefore strongly recommend that the publisher functionality is integrated into the Web services wrapper. Furthermore, when building new systems that support Web services in the first place, it is also important that the publisher functionality is included. In cases where integration of publisher functionality is not possible, the poller function (i.e., the JBridge equivalent) should be located on the same machine as the actual Web service, in order to avoid network traffic.

It should also be noted that although we used NFFI as data format, we did *not* use the publish/subscribe mechanism specified by NFFI SIP3 [23], which is WS-Eventing [24]. Since both Norway and NATO have chosen WS-Notification as the standard for publish/subscribe in the NII, it does not seem reasonable to deviate from this standard for one single data format.

8.4 Service discovery

Our chosen mechanism, WS-Discovery, is a decentralized service discovery mechanism intended for use in local area networks, relying on UDP multicast for disseminating information. Consequently, WS-Discovery can only be used in networks that support multicast, and this is not necessarily the case for all types of military networks. It is also a relatively “chatty” mechanism, which means that it should not be used on networks with very limited bandwidth. As described in Section 5, some measures can be introduced (e.g., compression) in order to reduce the network traffic.

Our work in the SOA Pilot shows that WS-Discovery works fine over medium bandwidth radios like the Kongsberg WM600, while it is not possible, even with compression, on radios with very low bandwidth, such as the MRR.

WS-Discovery works very well when it comes to liveness, i.e., the announced services reflect quite well which services are actually available.

It should also be noted that WS-Discovery is only intended for *run-time* use. This means that a query for a service will only return the endpoint address of that service, not the WSDL. Consequently, it is only possible to invoke services that you already have a client for, i.e., that you have previously downloaded a WSDL for the service, generated the client stub and integrated this with the client software. This is not a problem, however, since the required WSDLs are normally obtained during design time, either by using a registry and repository, or by downloading them directly from the service provider.

There is a more general problem concerning publish/subscribe and service discovery, though, that also affects WS-Discovery, and that is that all service discovery mechanisms (as the name implies) focus on discovery of *services*. The problem is that seen from a WS-Discovery point of view, all publish/subscribe (WS-Notification) services are equal. They all provide the same *service* (which is to deliver notifications), but they can provide notifications on different *topics*. Consequently, the information about which topics each service published on had to be distributed beforehand.

Furthermore, in the SOA Pilot, our focus was to demonstrate service enabling of existing operational systems. Therefore, the services announced through WS-Discovery reflected the underlying operational system. In a real-life system, it is more likely that the user will select services mainly based on the *information* it offers (i.e., the *topics*), and not only based on which operational system that provides it.

It is clear that when using publish/subscribe for information dissemination, it is necessary to focus both on how to search for and discover available topics, and also the semantics of topics, i.e., how to structure the description of the information in the notifications into topics and subtopics.

8.5 DSProxy

In the SOA Pilot, we used DSProxy on the MRR connection between the sensor network and the main network, as shown in Figure 2.7. The MRR represents a considerable challenge when using Web services, since the available bandwidth is very low.

The DSProxy, being intended for low bandwidth networks, worked very well, and we successfully invoked the Web service on the gateway of the sensor network. However, we did identify a couple of problems that we will focus on in our work with the DSProxy:

- No support for HTTP-chunking. The consequence of this is that the DSProxy can only be used on Web services that are hosted on IIS (Internet Information Server) from Microsoft.
- No support for IPv6. This was not a problem in the SOA Pilot, but should nonetheless be fixed.

Note that these are not *design* problems, but rather related to the concrete implementation. In our further work we will test alternative libraries for HTTP support, in order to solve these problems. For more information on the DSProxy, see [27].

8.6 Format translation

In the SOA Pilot we had to handle a number of different formats on the data coming from the different operational systems. For tracks, these included OTH-gold, NVG and proprietary formats, and all were translated into NFFI, which we used internally.

The translation was done programmatically, i.e., we did not use XSLT. This was mainly because XSLT engines are normally a part of ESBs, which we did not use. The translation code was placed within the wrapper, and the main reason for this was that messages in the OTH-gold format refer to earlier messages, and thereby required state. Thus, for those systems that delivered messages on OTH-gold format, we had to break the principle of Web services not having state. However, this was still considered the easiest solution.

In general, translation of formats should be done as separate services, known as *mediation services*, part of the NATO Core Enterprise Services [28]. Only when translation from non-self-contained message formats such as OTH-gold is required, may the translation function be integrated in the wrapper. Thus, as a rule, message-formats used in SOA environments should be self-contained.

Another challenge when integrating information of different formats is loss of information. For instance, when translating from NVG to NFFI, relatively much information is lost. If, at a later stage, it is necessary to translate back from NFFI to NVG, a lot of the original information has been lost. There are also examples of the opposite problem; when translating from OTH-gold to NFFI, it is necessary to add information that is required in NFFI, but lacking in OTH-gold, since the latter is a much simpler format. This is either information that is derived, or it is “dummy” information.

In general, when introducing SOA, which data formats to use and how to handle them is a matter that needs careful consideration. Closely related to this, it is very important to be aware of how data flows between systems, and what to do with circular data. That is, data that origin in on system, flows to another system, and then eventually (possibly via further systems) flow back into the originating system. This may lead to false confirmation of data, i.e., where users of system A believes data has been confirmed by system B, while system B have actually just returned data it has received from system A. Thus, managing data origin will probably be very important.

9 Conclusion

The SOA Pilot was demonstrated at FFI in June 2011 in cooperation with NC3A. Our goal was to give a practical demonstration of some of the benefits of using a service-oriented architecture. We started with a number of existing military operational systems and service-enabled these by using so-called wrappers. All these systems were then connected in a common information infrastructure, offering their information through services.

From a practical point of view, we had two main challenges during the execution of the pilot:

- Setting up and using several large and complex operational systems was a non-trivial task
- Our choice of software implementation of the WS-Notification infrastructure caused a lot of problems. However, we have later implemented our own, standalone WS-Notification module that will be used in future experiments, as explained in Section 8.1.

As a demonstration of the benefits of SOA, however, we consider the pilot a success. We proved that service-enabling of existing operational systems is a manageable task, although it should preferably be done by the system vendors. Also, new systems should be designed as services from the start.

We also showed that once the systems are service enabled and deployed in a SOA infrastructure, this gives a whole new flexibility with respect to information dissemination and being able to dynamically configure and reconfigure an information infrastructure. We therefore recommend that the experiences from this pilot are taken further by the Norwegian Defense for realizing a common information infrastructure.

References

[1] "Pervasive web services discovery and invocation in military networks", F.T. Johnsen, FFI-report 2011/00257.

[2] "Integrating Wireless Sensor Networks in the NATO Network Enabled Capability using Web Services", J. Flathagen and F.T. Johnsen, IEEE MILCOM 2011

[3] "Interoperable service discovery - experiments at Combined Endeavor 2009", F.T. Johnsen, J. Flathagen, T. Hafsøe, M. Skjegstad, N. Kol, FFI-report 2009/01934

[4] "Information exchange in heterogeneous military networks", K. Lund, T. Hafsøe, F.T. Johnsen, E. Skjervold, A. Eggen, L. Schenkels, J. Busch, FFI-report 2009/02289

[5] OASIS WS-Notification (2006) TC
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn

[6] WS-BaseNotification 1.3 OASIS Standard, approved October 1st 2006
http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf

[7] WS-BrokeredNotification 1.3 OASIS Standard, approved October 1st 2006
http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-os.pdf

[8] WS-Topics 1.3 OASIS Standard, approved October 1st 2006
http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf

[9] "Norwegian Modular Network Soldier (NORMANS)", R. Lausund, S. Martini", FFI Facts, November 2006

[10] "NORMANS KKI" (in Norwegian), J. Flathagen and L. Olsen, FFI Facts, November 2006

[11] "Experiment report: 'SOA – Cross Domain and Disadvantaged Grids' – NATO CWID 2007", R. Haakseth et al., FFI-report 2007/02301

[12] WS-Messenger (WSMG) home page
http://www.extreme.indiana.edu/xgws/messenger/

[13] WebserviceX.net home page
http://www.webservicex.net/

[14] Web Service List home page
http://www.webservicelist.com/

[15] A WS-Discovery implementation written in Java home page
http://code.google.com/p/java-ws-discovery/

[16] "Web Services Dynamic Discovery (WS-Discovery)" J. Schlimmer (editor).
http://specs.xmlsoap.org/ws/2005/04/discovery/ws-discovery.pdf, April 2005.

[17] "Web services dynamic discovery (ws-discovery) version 1.1 OASIS standard 1 july 2009" V. Modi, and D. Kemp (eds.)
http://docs.oasis-open.org/ws-dd/discovery/1.1/wsdd-discovery-1.1-spec.pdf

[18] "SOAP-over-UDP version 1.1 OASIS standard 1 july 2009" R. Jeyaraman (ed.)
http://docs.oasis-open.org/ws-dd/soapoverudp/1.1/os/wsdd-soapoverudp-1.1-spec-os.pdf

[19] "Multinett II: SOA and XML security experiments with Cooperative ESM Operations (CESMO)", F.T. Johnsen et al., FFI-report 2008/02344

[20] "Adapting WS-Discovery for use in tactical networks", F.T. Johnsen and T. Hafsøe, 16th ICCRTS, June 2011.

[21] "Employing Web services between domains with restricted information flows", T. Hafsøe and F.T. Johnsen, 16th ICCRTS, June 2011.

[22] “Multiple independent levels of security (MILS) – a high assurance architecture for handling information of different classification levels”, T. Gjertsen and N.A. Nordbotten, FFI report 2008/01999

[23] “NFFI Service Interoperability Profile 3 (SIP3) Technical Specifications (Version 1.1.5)”, Vincenzo de Sortis, Working Paper EPW002625-WP05, NC3A, The Hague, March 2009

[24] “Web Services Eventing (WS-Eventing)”, http://www.w3.org/TR/ws-eventing/, W3C Proposed Recommendation, 27 September 2011

[25] “Experiment Report: SOA Pilot 2011 “, R. Rasmussen, FFI Report 2011/02407

[26] “Towards a certifiable MILS based workstation “, N. A. Nordbotten, T. Gjertsen, FFI report 2012/00049

[27] “Robust Web Services in Heterogeneous Military Networks, Lund, K., Skjervold, E., Johnsen, F. T., Hafsøke, T., Eggen, A., IEEE Communications Magazine, Vol. 45, No. 10, October 2010, pp. 78-83

[28] NATO Core Enterprise Services Standards Recommendation, The SOA Baseline Profile, Version 1.7, Jan 2012

Appendix A FFI SOA pilot incident format schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  FFI SOA pilot incident format v1.1
  fields:
    name                  (mandatory) - incident name
    id                    (mandatory) - unique incident ID (e.g., system+nr)
    dateTime              (mandatory) - incident timestamp
    longitude             (optional)  - position, if applicable
    latitude              (optional)  - position, if applicable
    originSystem          (mandatory) - system where the incident occurred
    sender                (mandatory) - system sending this report
    incidentDescription   (mandatory) - text describing the incident
    picture               (optional)  - base64 encoded image, if applicable
-->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="urn:no:ffi:reports:incident"
  xmlns:tns="urn:no:ffi:reports:incident"
  elementFormDefault="qualified">
  <xsd:element name="Incident">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="name" type="xsd:string"></xsd:element>
        <xsd:element name="id" type="xsd:string"></xsd:element>
        <!-- ISO 8601 in the format YYYYMMDDhhmmss format (14 digits) -->
        <xsd:element name="dateTime">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:pattern value="\d{14}"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="latitude" type="xsd:double" minOccurs="0"></xsd:element>
        <xsd:element name="longitude" type="xsd:double" minOccurs="0"></xsd:element>
        <xsd:element name="originSystem" type="xsd:string"></xsd:element>
        <xsd:element name="sender" type="xsd:string"></xsd:element>
        <xsd:element name="incidentDescription" type="xsd:string"></xsd:element>
        <xsd:element name="picture" type="xsd:base64Binary" minOccurs="0"></xsd:element>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```