Multinett II: SOA and XML Security Experiments with Cooperative ESM Operations (CESMO)

Frank T. Johnsen, Trude Hafsøe, Espen Skjervold, Kjell Rose, Ketil Lund and Nils A. Nordbotten

Forsvarets forskningsinstitutt/Norwegian Defence Research Establishment (FFI)

17.12.2008

FFI-rapport 2008/02344

1086

P: ISBN 978-82-464-1503-1
E: ISBN 978-82-464-1504-8

Keywords

Multinett II

Tjenesteorientert arkitektur

Mellomvare

Sikkerhet

Eksperimentering

Approved by

Anders Eggen	Project manager
Vidar S. Andersen	Director

English summary

This report describes the activities and experiments performed within FFI project P1086 – “Secure Pervasive SOA” during Multinett II. Multinett II is the largest joint experiment activity ever carried out in Norway. The basic concept of Multinett II is to connect different communication systems from all the military services into one common network. The goal was to experiment with technology that can contribute to further development of network based defense.

From FFI, the participation was coordinated by the FFI project P1080 - a project that focuses on cooperative ESM-operations. For P1086, collaborating with P1080 was seen as a good opportunity to test Service Oriented Architecture (SOA) principles and related end-to-end security solutions in an operational environment, and to demonstrate possibilities for operational personnel. This report focuses on our experiments using Web Services to realize SOA.

Sammendrag

Denne rapporten beskriver eksperimentene som ble utført av FFI-prosjekt P1086 – ”Sikker gjennomgående SOA” under Multinett II. Multinett II er den største militære eksperiment-aktiviteten som noensinne har blitt gjennomført i Norge. Ideen bak Multinett II var å kople sammen kommunikasjonssystemer fra alle forsvarsgrener i ett felles nettverk. Målet var å eksperimentere med teknologi som kan inngå i utviklingen av et Nettverksbasert Forsvar (NBF).

For FFI sin del ble deltakelsen koordinert av FFI-prosjekt P1080 ”ESM – nye operative muligheter og nettverksbaserte løsningskonsept for maritime operasjoner”. For P1086 innebar Multinett II en god arena for å teste tjenesteorientert arkitektur og relaterte ende-til-ende sikkerhetsløsninger i et operativt miljø, og å demonstrere de mulighetene som dette konseptet gir. Denne rapporten fokuserer på eksperimentene våre, med Web Services som teknologi, for å realisere en slik tjenesteorientert arkitektur.

Contents

1 Introduction

In October 2008, FFI participated in the exercise "Multinett II" (MNII) at Ørlandet and Jåttå. This report describes the activities and experiments performed within project 1086 – "Secure Pervasive SOA" during MNII.

MNII was the largest joint experiment activity ever carried out in Norway, and consisted of a number of field experiments. In total, more than 1000 persons were participating, involving all military services. The basic concept of MNII was to connect different communication systems from all the military services into one common network. This includes interconnections tried earlier, as well as some interconnections never tried before.

The goal of MNII was to experiment with technology that can contribute to further development of network-based defense. From FFI, the participation was coordinated by the project P1080 "Maritime ESM[1] in NBF", a project that focuses on cooperative ESM-operations (CESMO). For project P1086 "Secure Pervasive SOA", collaborating with P1080 was seen as a good opportunity to test Service Oriented Architecture (SOA) principles in an operational environment, and to demonstrate possibilities for operational personnel. This report focuses on SOA technology. For an overview of the CESMO application, see **FFI-Rapport 2008/02345**.

1.1 Cooperative ESM operations

Special radio receivers for the detection of emissions from radars and communication systems are called Electronic Support Measures (ESM) by the Electronic Warfare community. The ESM sensors measure the radar frequencies and other characteristic parameters of the signal, and the direction or bearing to the emitter.

By tracking emitters over time based on bearings from one ESM sensor, it can take a long time to establish target solutions, and the solutions may have limited accuracy.

With cooperating ESM sensors one can process simultaneous observations of an emitter from several ESM sensors, and obtain solutions more rapidly and more accurately than is possible with one sensor and one can use other and better methods that produce a more accurate emitter location. Cooperative ESM operations (CESMO) are mainly concerned with the detection and localization of radar emitters by cooperation between many sensor platforms as illustrated in Figure 1.1.

According to the present CESMO operational concept, an ESM sensor platform can have two roles:

- ordinary ESM sensor platform
- Sigint (Signal Intelligence) Identity Authority (SIA)

[1] Electronic Support Measures

All platforms can have the role of SIA, but the role is often given to the platform with the best resources, both sensor and operator wise. During the experiment the SIA was collocated with the experiment authorities, without any locally connected sensors.

Depending upon the type of operation, some hostile emitters are more interesting than others. When an interesting emitter is detected, *Signal Coordination Messages* are sent on the network from all sensor platforms that have observed the emitter. Based on the received messages the SIA computes the emitter location, and reports the emitter location in a *Geolocation Message* to all participating platforms. The participants also send *Participant Status Messages* at irregular intervals.

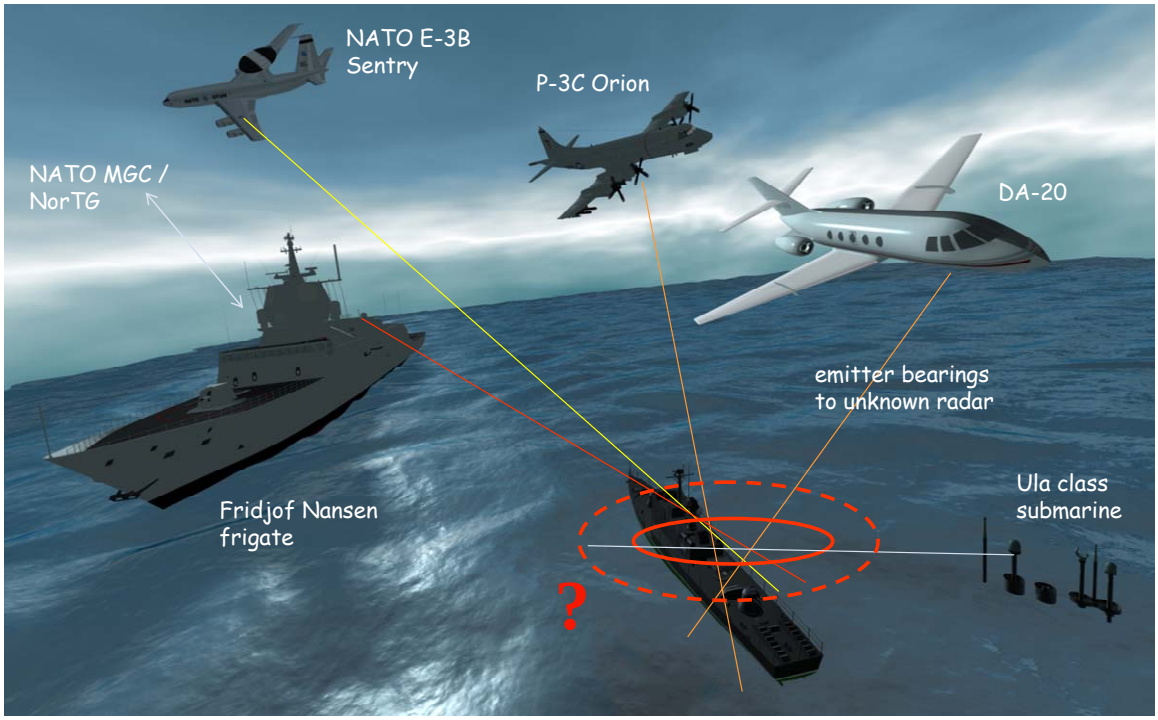


Figure 1.1 The CESMO-concept

In order to support the CESMO concept development and trials, NATO Consultation, Command and Control Agency (NC3A) started the development of a CESMO software tool. Defence Science and Technology Laboratory (DSTL) took over the development and with the latest version of the software it has become more modular, in order to facilitate integration on various platforms. The network interface of the CESMO software (more or less) follows the CESMO Interface Control Document (ICD), also referred to as STANAG 4658. The ICD defines byte oriented messages in a fixed format. Our demonstrator replaced the ICD interface, with an XML based interface compatible with NATO NEC and a Service Oriented Architecture (SOA).

This report is structured as follows: In Section 2 we discuss the motivation for introducing SOA in CESMO. Section 3 describes the SOA concept, and in Section 4 we present the setup of the CESMO network. In Section 5 we describe the SOA software used during MNII. This includes both software developed within the project, as well as third-party software. Section 6 presents the

software used for exporting data such as geolocations to NATO Friendly Force Information (NFFI), and the security solutions used are described in Section 7. The actual execution of the experiment is described in Section 8, and finally we conclude in Section 9.

2 Motivation

CESMO is a good example of operations where sharing of data is essential for the success of the operation. Traditionally signal intelligence and Electronic Warfare (EW) are areas where the sharing of data even between close allies has been limited. CESMO is a good example of a NEC, and by using the interfaces and infrastructure of SOA, could benefit favorably from a SOA with accompanying flexible security solutions, and controlled data dissemination.

The main motivation for using MNII as a case for SOA was to show that it is feasible to SOA-enable existing operational systems, which brings substantial benefits. It was also important to demonstrate these benefits for operational personnel, in order to, hopefully, trigger some ideas around how SOA can be used.

One of our goals was to show that introducing a SOA infrastructure into the CESMO network can contribute to automate and speed up cooperative ESM operations. We added a Web Services front-end to each platform, which made them appear as services. Each platform could then discover, and subscribe to, the services of other platforms.

We wanted to demonstrate how SOA enables information exchange between (stove pipe) systems, possibly in different military services. In other words, how SOA can function as an integrator, making it possible to find information and make it available across tactical and strategic systems in different military services. This was demonstrated by connecting the CESMO-network to a NORTaC-system, and then exporting the geolocations as NFFI-tracks into NORTaC. From NORTaC, the tracks were further distributed into NORCCIS-II.

Another illustration of such information exchange across operational levels is the fact that we connected the CESMO-network to the Joint Electronic Warfare Coordination Cell (JEWCC), located at the Joint Headquarter at Jåttå, Stavanger. The JEWCC is responsible for coordinating all activity related to electronic warfare, and as such it is dependent on receiving information about all radio activity in the operational area. Today, the JEWCC usually receives such information late, but by using the SOA infrastructure, we were able to provide the JEWCC with live information from the operational area.

Since the planned CESMO network was radio-based, this was also a good opportunity to test SOA on a real, disadvantaged grid[2]. Thus, it was also a goal to show that Web Services can be employed on radio networks with relatively low bandwidth, using compaction. In addition, we

[2] A *disadvantaged grid* is a network with low bandwidth, high delay and frequent disruptions. In other words; a tactical military network.

wanted to show that the flexibility[3] and compaction friendliness of XML would lead to more efficient data transport than what is possible in traditional CESMO.

Finally, as P1086 covers both SOA and security, it was a goal to demonstrate end-to-end security solutions, and that it is feasible to integrate these into a SOA on a disadvantaged grid.

3 Service-Oriented Architecture

There exist a large number of definitions of the term SOA, although most seem to agree that it is an architectural style. The exact definition will probably also vary with area of interest, for instance, whether you define it from a business or a technical viewpoint. In our work on SOA in a (primarily operational) military context, we have arrived at the following definition:

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.

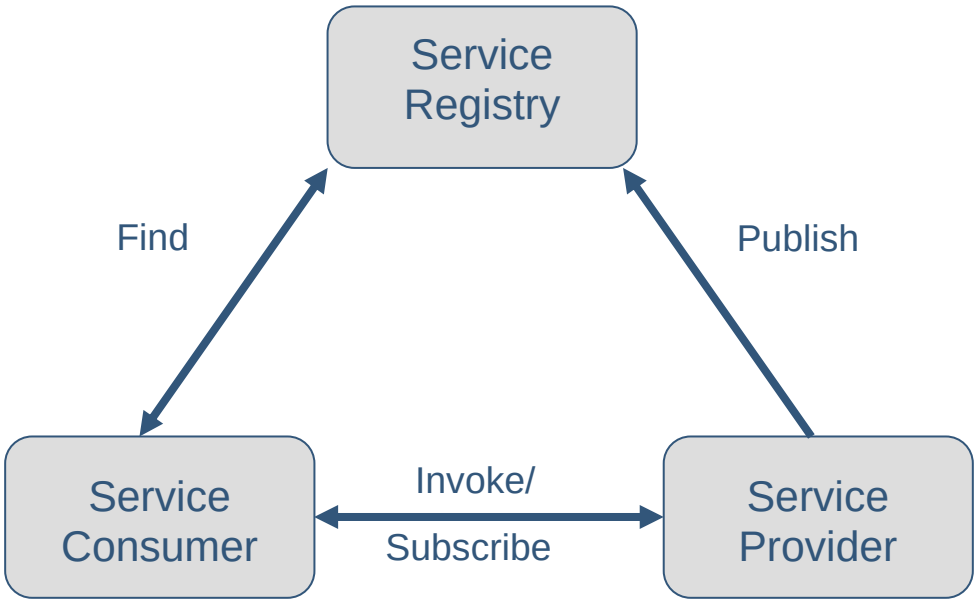


Figure 3.1 The SOA-triangle

[3] Particularly important is the ability to emit empty, optional elements entirely. This is in contrast to binary formats, where all elements usually must be included, whether they are used or not.

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 3.1.

The principle of separating the service interface from the service implementation also means that there are several different ways of realizing services. As shown in Figure 3.2, a service can be defined from scratch, allowing full control of the way the services is implemented. In addition, existing applications can be "wrapped", and made available as services. This approach requires an adaptation layer in order to adapt the existing interface of the application to the service interface. Finally, services of both types can be combined, in order to create more complex services.

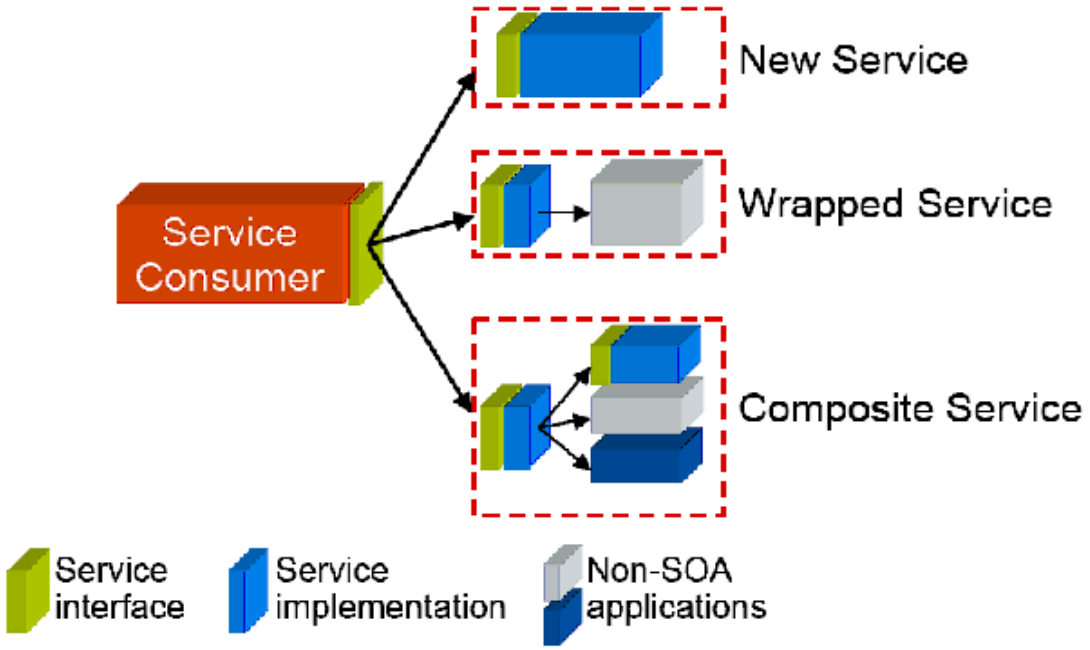


Figure 3.2 Creating services (fetched from [13])

While a Service Oriented Architecture can be implemented using several different technologies, Web Services stand out as a preferred and widely adopted standard. Web Services are defined by the W3C as "a software system designed to support interoperable machine-to-machine interaction over a network", and while the underlying protocols may vary, SOAP over HTTP is frequently utilized. In the limited network environment in which our experiment is performed, HTTP and TCP connections are unsuitable, as discussed in [16]. Therefore, substitutions for HTTP and TCP were used, and will be discussed in detail in the following paragraphs.

SOA systems and Web Services are normally based on request/response mechanisms, in which a client polls a server for data. This approach to data sharing has some fundamental challenges: The client does not know when new and interesting information is available on the server, and therefore risks polling the server and returning empty handed, adding additional traffic to the

network. Wasted polls could be minimized by decreasing the frequency of which the client polls the server, but this increases the risk of the client waiting too long, and receiving the information later than what is desirable or required. One solution to this challenge is to replace the request/response mechanism with the publish/subscribe mechanism, in which the clients subscribes to updates on a topic and the service provider notifies the nodes interested in the information, when new information arrives. In a Web services context, a basic publish/subscribe mechanism is nothing more that a reversed Web service; the server acts as a client, invoking a Web service endpoint at the subscribing clients.

Paired with underlying transmission protocols that support multicast, the publish/subscribe mechanism may further reduce network traffic by enabling the same information to be broadcasted to multiple receivers simultaneously. See **FFI/Rapport 2007/02340** for a discussion of request/response compared with publish/subscribe for use in military networks.

4 Planned network setup

Service orientation of CESMO implies that the information exchange is organized as a collection of services and clients to these services. For the experiment the following services were implemented:

- **Platform Services**. The participating platforms reported their respective positions and status as a service
- **Signal Services**. The platforms reported intercepted signals from interesting emitters using a service
- **Geolocation Service**. The SIA reported the geolocation of the emitters as a service

Service orientation facilitates a flexible organization of the operation, and makes it simple for other participants to take over the role of the SIA. The role of a participating platform can be limited to the reporting of interesting signals, but nothing prevents more capable platforms from connecting to the signal services and geolocation service, following the situation and take over the SIA role if necessary.

The present concept is centralized, with the SIA as the central element, but alternative concepts could be implemented on an ad hoc basis in order to better suit the operation, the participating platforms and the network structure.

4.1 Participants

The planned experiments had four main entities that were actively participating in the CESMO network: A group of (i.e. two) DA-20 aircraft, a group of (i.e. two) frigates, an experiment authority which functioned as SIA and an information consumer, a JEWCC. In addition to these four CESMO participants, a fifth entity, the NORTaC system, received information from the CESMO network via an NFFI service, but did not actively participate. Figure 4.1 shows the participants and the planned connections between them.

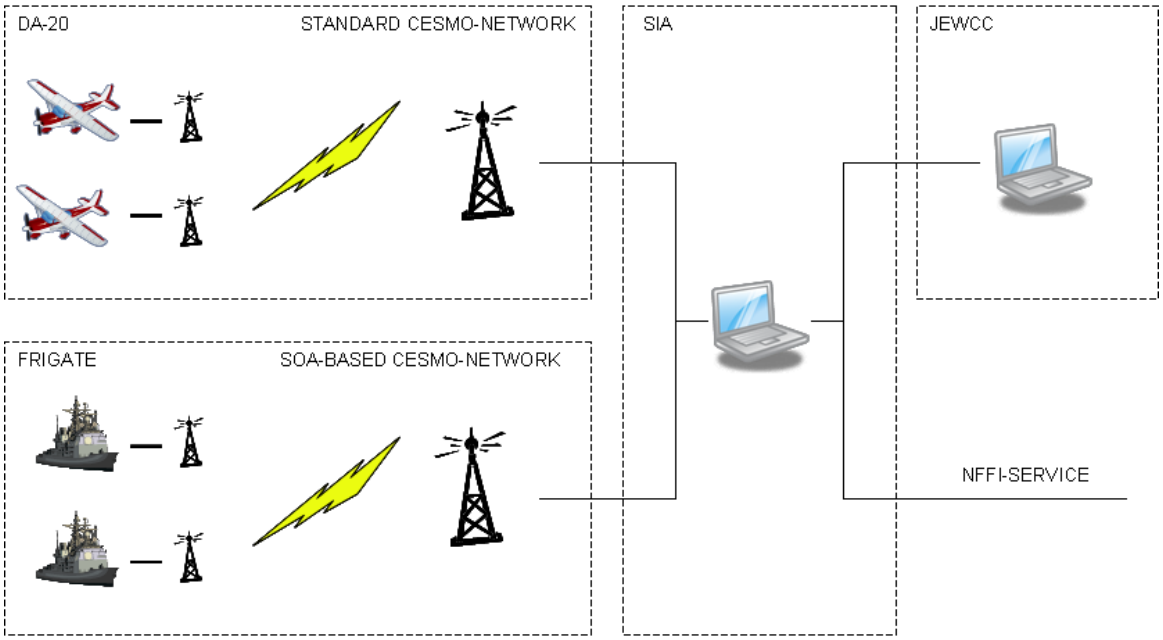


Figure 4.1 Planned experiment setup

The DA-20s were equipped with ESM sensors, and were running the above described CESMO software. However, due to the limited capabilities of the on board computer hardware, they were unable to run the SOA software. In order to connect these aircraft to the other participants, we wrapped the entire network as a Web Service. This wrapping did not only allow for the interconnection of a SOA-enabled and a non-SOA enabled network, but it also provided the opportunity to test and demonstrate how SOA can be used to bridge the gap between systems with different capabilities. The SIA, being the connection point for all the entities, performed this wrapping function.

The frigates were, according to experiment plans, going to run the same CESMO software as the DA-20s, but additionally also run a SOA software that would SOA enable these platforms. In other words, the CESMO application provided the user with a familiar interface, while the SOA software intercepted all network traffic and transmitted it using Web Service technology.

The role of SIA is often given to the platform with the best sensor and operator capabilities. In these experiments however, the role of SIA was given to the stationary experiment authority responsible for coordinating the entire MNII experiment. This meant that the SIA did not have a local ESM sensor, and would thus have to rely on the other participants to provide it with ESM data. The SIA used the information it received for a number of purposes:

- it graphically displayed the information it received to an ESM operator that could use this information to coordinate operations,
- calculated the location of targets,
- distributed the information generated by the CESMO network to the JEWCC, and
- made CESMO information available to other systems as an NFFI service

The JEWCC, being a coordinator, understands native CESMO messages. But, it does not have a sensor of its own. This entity can gather and observe CESMO information, and can choose to interact with the active CESMO entities, i.e. the entities with ESM sensors. Using just a legacy CESMO application, the JEWCC would have to receive and visualize all tip, tune, and geolocation messages. However, it is not necessary for the JEWCC to receive all this, for example, in a particular setting maybe only the geolocations are of interest to the operators at the JEWCC. Thus, our SOA software wraps the CESMO application at the JEWCC, enabling it to selectively subscribe to only the information it sees fit, and thus reducing bandwidth usage and reducing the possibility of information overload. Furthermore, the SOA software at the JEWCC also provided an added-value function not previously available; it exported the received information into a comma-separated values (CSV) file as well as passing it on to the CESMO application for visualization. This allowed the operators at the JEWCC to use the received data in other applications as well without having to enter the data manually in order to do so.

NORTaC is a tactical application which is totally separate from the CESMO network. It does neither understand nor consume CESMO messages. However, thanks to our SOA solution, we can provide parts of the information from the CESMO network to NORTaC through an NFFI service. The NFFI service was provided as a push service at regular intervals. Thus, NORTaC can receive tracks and positions from CESMO. CESMO specific data is first received and translated to NFFI format in the SOA solution, before the information finally is presented to NORTaC, which can visualize it by using parsing the NFFI message.

The remaining entities, i.e. the SIA, SOA-based CESMO network and the JEWCC were fully SOA capable and were set to use the full functionality of our SOA software. In other words, a complete Web Services based solution based on publish/subscribe for disseminating the CESMO information, as well as performing a value-added service by providing NFFI.

4.2 Network connections

The participants of the CESMO experiments were to be connected to the SIA with the network technologies shown in Table 4.1. The DA-20s and the frigates should use their respective installed radios, but with UIDM modems (see Figure 4.2) running on top of the voice channel to provide IP functionality.

Entity	**Connection to SIA**
DA-20	Radio network, UIDM IP modem, approximately 16 Kbps
Frigate	Radio network, UIDM IP modem, approximately 16 Kbps
JEWCC	2Mbit secure wired connection
NORTaC	100 Mbit Ethernet[4]

Table 4.1 Network connection type and bandwidth

[4] Please note that 100 Mbit is not a usual operating speed for NORTaC, but in the case of the MNII experiment the SIA was located in the experiment authority container complex, where NORTaC was being

The UIDM adds IP packet support to the network, and can be used together with existing voice radios. The modem has several limitations that need to be taken into account when using it;

- There is limited support for IP fragments, so the application should not send very large packets (or it should fragment its packets itself).
- The maximum transfer unit (MTU) is much lower than that which is usual for an Ethernet, so the messages/fragments must reflect this.
- There is limited buffer space in the modem. During testing we crashed the modem by sending too many packets to it in rapid succession. Thus, it is necessary to control the communication burstiness of the applications (congestion control).

Figure 4.2 A Universal Improved Data Modem

5 SOA coordinator

This chapter discusses the SOA coordinator, which is FFI's experimental software for SOA-enabling the CESMO application.

While the SOA Coordinator has several functions, its key role is to wrap the legacy software and connect it to the publish/subscribe middleware (WS-Messenger). The application was written in Java, and presents the user with a graphical user interface, allowing him to control and monitor the flow of information in the SOA network.

In order to enable communication between legacy software components residing on different nodes, third-party software components as well as tailor-made software were integrated with the existing software. The software stack and the system architecture can be seen in Figure 5.1. The components' functionality and roles are presented in detail in the following paragraphs.

operated from. Thus, we used an Ethernet connection from SIA to NORTaC, which yielded a local area connection with 100 Mbit.

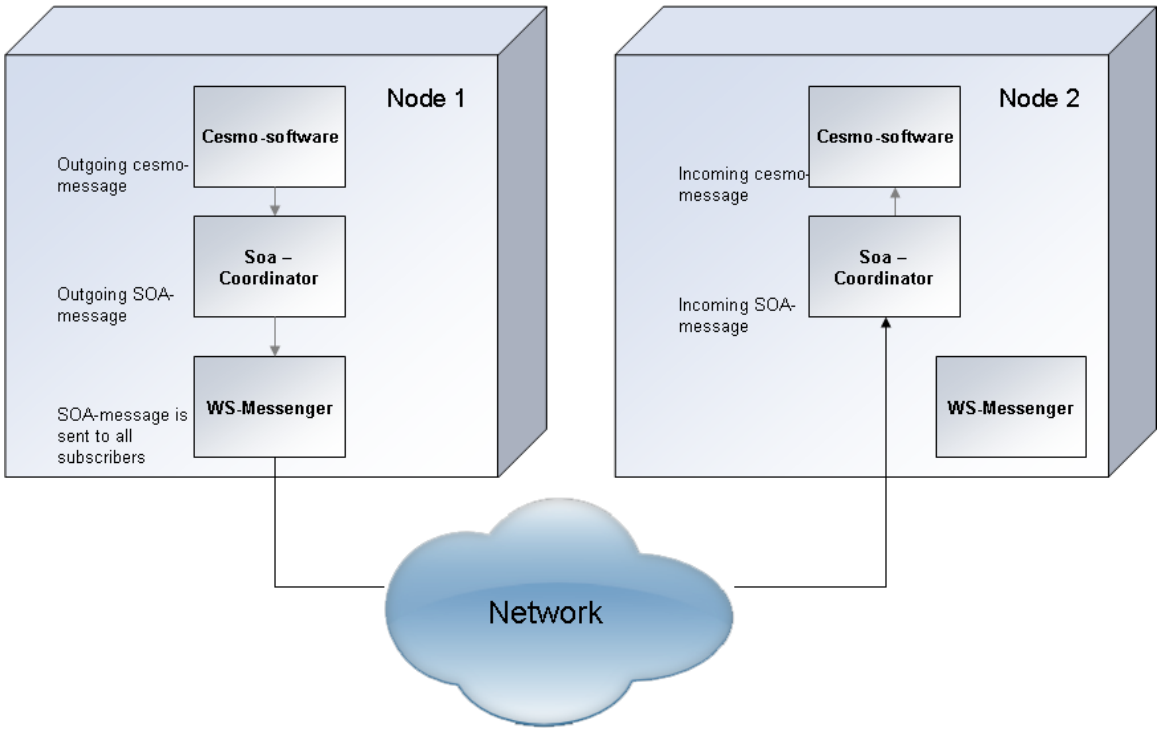


Figure 5.1 The SOA software component-node diagram

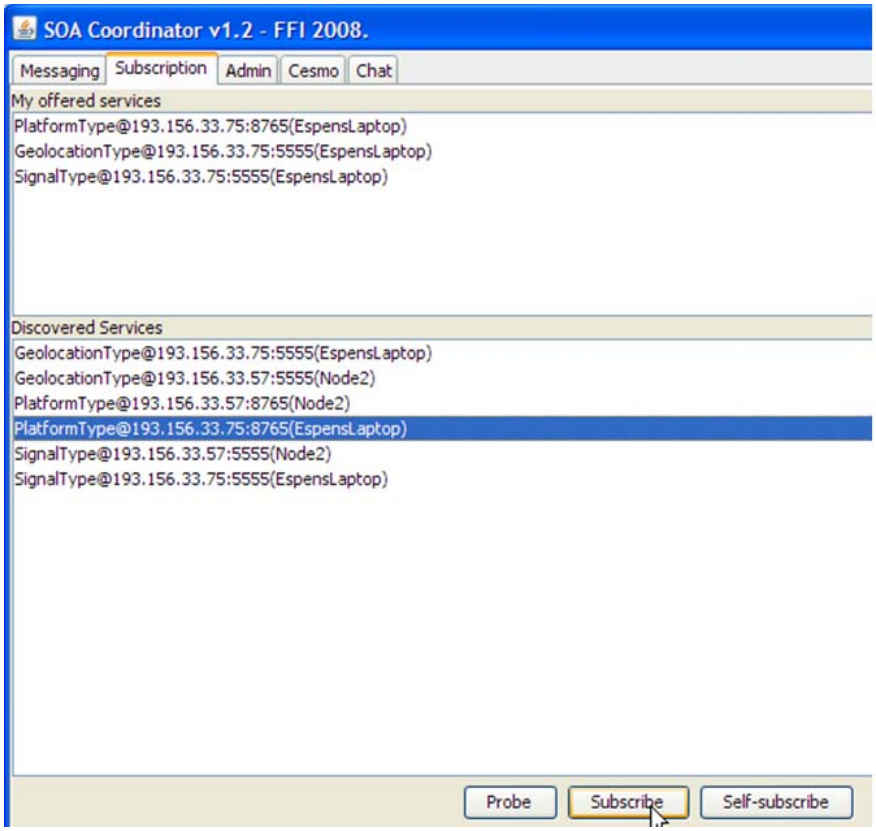


Figure 5.2 The SOA Coordinator graphical user interface

The SOA Coordinator serves as a proxy between the CESMO software and the SOA network, and enables the user to subscribe to SOA message topics from WS-Messenger instances running on both local and remote nodes. It also offers services to the network on behalf of the local CESMO software. The CESMO software communicates with other systems (and other instances of itself running on other nodes) by sending three types of XML messages: PlatformType, SignalType and GeolocationType. While the CESMO software broadcasts these messages on the local network using UDP multicast, the SOA Coordinator intercepts the messages and converts them to SOA messages. This process includes determining which topic they sort under, after which they are published to the local WS-Messenger. Three topics, one for each CESMO software message type, are offered as SOA services to the network (See Section 5.2).
Through the SOA Coordinator's "Messaging"-tab, users can monitor ingoing and outgoing SOA messages, and are provided information about their topic and content. While the "CESMO -"-tab similarly monitors the messages going to and from the CESMO-software, the "Messaging"-tab may also show messages going to and from other systems. The "Chat"-tab enables SOA Coordinator users to see other active users on the network running the same application, and allows users to send instant messaging messages to each other.

5.1 WS-Messenger

WS-Messenger is a third party, open source framework that enables publish/subscribe mechanisms in Web Service oriented systems, and a modified version was utilized as middleware in the SOA system and integrated with the SOA Coordinator. The software implements two different standards associated with publish/subscribe and Web Services, namely WS-Eventing and WS-Notification. While the latter includes WS-BaseNotification, WS-BrokeredNotification and WS-Topics, the former mainly implements the functionality found in WS-BaseNotification. See **FFI/Rapport 2007/02340** for a detailed overview of these two competing standards. As interoperability is central to the project, this support was one of our main motivations behind the choice of framework. WS-Messenger supports both WS-Eventing and WS-BaseNotification and is capable of translating between the two, enabling messages to be published on different formats to different nodes, depending on which formats the nodes support. WS-Messenger has store-and-forward functionality, and offers retransmission of messages that failed to reach the intended subscribers.

The publish/subscribe mechanism introduced not only the loose coupling of Web Services, but also asynchrony between senders and receivers into the system, and resulted in increased scalability and a more dynamic network topology. By sending XML messages compliant with WS-Eventing or WS-Notification, nodes are able to subscribe and unsubscribe to topics of interest. When a node publishes a message to WS-Messenger, WS-Messenger forwards the message as a notification message to all nodes currently subscribing to the message topic.

The WS-Messenger application library, *Wsmg.jar*, contains both the client and server-side code used when connections are initiated by clients to the server, respectively. Out of the box, WS-Messenger supports only the HTTP-protocol over TCP/IP for sending and receiving messages. While the TCP-protocol offers reliable point-to-point communication, it relies on

acknowledgements, and the TCP header introduces relatively large amounts of overhead. Furthermore, TCP initiates slow start after packet loss, yielding poor throughput when there is packet loss. As the system is required to perform in disadvantaged grids offering narrow bandwidths, another transmission protocol was required. Despite the lack of reliability and ordering, UDP was chosen for transporting SOA messages between the nodes. Not being dependant on ACKs/NACKs makes it suitable for low bandwidth environments. XML messages sent over UDP were first compressed using the GZIP[5] algorithm, and encoded into Base64 to avoid special characters and possible hardware incompatibilities. Because the SOA system was developed for use in radio networks, and because in a radio network all information is broadcasted to all participants, support for UDP multicast was added to WS-Messenger/SOA Coordinator.

5.2 Service Discovery

Service discovery is a mechanism for discovering available services in a network. Different networks require different solutions, and in MNII we used a proprietary FFI-deviced mechanism tailored to low bandwidth networks. The main focus in MNII was not on the service discovery mechanism, but rather on how service discovery can be used to reduce the need for manual configuration. Thus, a simple service discovery mechanism (described below) was sufficient for these purposes. The solution was distributed without a registry, and was based on multicast communication.

Service discovery was implemented into the SOA Coordinator software using a custom Java library developed at FFI. The library, called UdpDiscovery, was optimized for disadvantaged grids, and generates very little network traffic. It adheres to the following principles:

- Small, portable Java-objects are used as information-carriers over the network.
    - The objects are serialized, compressed using GZIP and encoded using Base64 to ensure compactness and robustness.
- All clients, in this case all instances of the SOA Coordinator implementing the UdpDiscovery-library, send multicast keep-alive messages at a specified interval (here, once every minute) including information about the nodes' perceived view of the network (in the form of a list of active nodes).
    - When nodes B and C receive a keep-alive message from A, they know the node is alive and active and update their keep-alive timestamp for node A.
    - They also compare the received node-list with their own network information, and, if the received information is newer, update their own information about active and inactive nodes.

[5] We used the GZIP algorithm because it has an overall good performance and comes in a native Java library, and thus was very easy to integrate into our Java-based software. However, as our previous research has shown (see [14] and [15]), Efficient XML (EFX) yields slightly better compression than GZIP, and allows more flexible use due to it being tailored to XML. For our participation in MNII we were pressed for time, and did not have the resources to integrate EFX in our software.

    - This mechanism ensures that all nodes have updated information about the network, even in environments where packets are lost and information fails to reach all nodes.
- The more nodes that are part of the service discovery overlay network, the more redundancy is added to the network, increasing the robustness (but also the overhead) of the system.
- When a node joins the network, instead of waiting for incoming keep-alive messages it sends a request, which is responded to with a keep-alive message.
    - In order to minimize the number of duplicate responses, the nodes that receive the request wait for a random period of time before responding, and skip the response if a response is sent by another node.

The implementation of service discovery provided the SOA Coordinator operators with information about active services in the network, enabling them to manually subscribe to services and topics they were interested in. Future experimentation will include using an implementation based on the WS-Discovery draft specification for improved system compatibility (see 8.2.2).

5.3 CESMO Messaging software

The CESMO Messaging software is the glue between the CESMO software and the SOA software. The CESMO Messaging software is not running as a separate process, but consists of two main classes and an interface implemented by the SOA software:

- CesmoReceiver
- CesmoListener
- CesmoSender

In addition to classes supporting the main classes, the CesmoMessaging software package contains classes for marshalling and unmarshalling of Java (JAXB) objects, and a class for writing emitter data to a CSV file, to be imported into a live Excel spreadsheet. Figure 5.3 illustrates the service orientation of the CESMO software. The various parts are presented below, except for the NFFI track provider, which is discussed in Chapter 6.

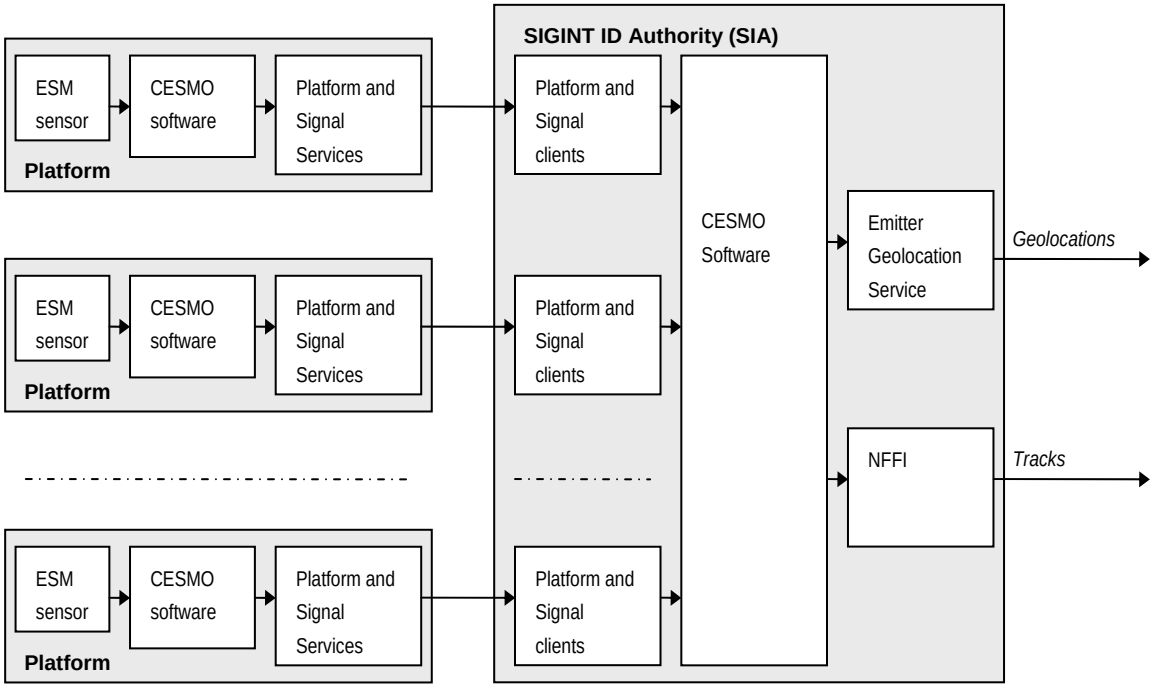


Figure 5.3 Service Oriented ESM cooperation

5.3.1 Cesmo Receiver

The CesmoReceiver handles the reception of data from the CESMO software. It listens on a multicast socket on the CESMO network for CESMO messages. On reception of:

- Participant Status Message
- Signal Coordination Message
- Geolocation Message

The messages are decoded and turned into the objects of the respective classes:

- PartStatus
- SCMWrapper
- Geolocation

and translated into objects of the (JAXB) classes:

- PlatformType
- SignalType
- GeolocationType

and sent to the CesmoListener interface in the SOA software.

5.3.2 CesmoListener interface

The CesmoListener interface is a Java interface that is implemented by the SOA software to receive the data from the CesmoReceiver.

5.3.3 CesmoSender

The CesmoSender receives Java objects from the SOA software. When objects are passed to the CesmoSender they go through the reverse of the process that is described for the CesmoReceiver and finally the CESMO messages are sent on the CESMO network as multicast messages.

6 NFFI software

The NATO Friendly Force Information (NFFI) Standard for Interoperability of Force Tracking Systems (FTS), to become STANAG 5527, was used for the delivery of sensor platform locations, signal observations and emitter locations to the army's NORTaC command and control system.

All of the ESM-related information in the signal observations and emitter locations was lost in the translation to NFFI, but the bearing to the emitters in the signal observations was translated to bearing in the NFFI positional data, and could be shown on a NFFI display. Thus, the platform locations were translated to NFFI with the significant part of the information content intact.

The CESMO software produces the emitter locations with uncertainty ellipses, and during the experiment the ellipses were oblique and oblong. NFFI defines uncertainty in latitude and longitude, and to translate the uncertainties of the emitter locations to this uncertainty model would give a wrong impression of the accuracies of the emitter locations, and the translation of uncertainties was therefore simply dropped.

One weakness of the proposed STANAG 5527 is that it does not give any requirement for the implementations of the STANAG, which is in our opinion a serious omission for a STANAG promoting interoperability. Most of the elements of a NFFI document are optional. This is done in order to facilitate interoperability with platforms with limited capabilities. More capable platforms should be required to implement the complete specification in order not to filter out essential data. That an element is optional in the NFFI XML schema should not necessarily mean that the element may be discarded upon reception.

ESM-sensors have the unique property of being able to identify the emitters and their corresponding platforms, and the experiment has shown that NFFI can be used for the dissemination of data from ESM operations, and that this can be done without compromising the security restricted ESM-parameters.

6.1 NFFI implementation

CesmoRosa is a test and monitoring application that was used during the development of the demonstrator. It was the main tool for developing and testing the CesmoMessaging software. During the experiment CesmoRosa was used for monitoring the traffic on the CESMO network.

Since a simple socket connection was used for the NFFI connection, the NFFI software was kept separate from the SOA coordinator, and was implemented in CesmoRosa.

The interface protocol to be used by the demonstrator was NFFI-IP1, which specifies that TCP should be used as a transport protocol. The CesmoRosa implemented a TCP push client. When messages are received on the CESMO net, they are translated into the corresponding NFFI tracks and sent to the NORTaC NFFI server.

7 SOA Software Security

Considering that the SOA software was based on Web Services and XML, this provided a good opportunity to experiment with security standards for Web Services and XML in a military environment. The following XML and Web Services security standards were therefore used in conjunction with the SOA software:

- XML Signature [1]
- WS-Security [2]
- The Security Assertion Markup Language (SAML) [3]
- The eXtensible Access Control Markup Language (XACML) [4]

While confidentiality was already provided by the underlying network infrastructure[6], the above listed standards provided the additional benefits of end-to-end integrity, authentication, and policy-based access control.

A brief description of the each of the above standards will now be provided, before their use in the experiment is described.

7.1 An Overview of the Security Standards Used

XML Signature defines how to apply well established digital signature algorithms to XML. This includes:

- A standardized way to represent signatures and information about the associated key(s) in XML, independent of whether the signed resource is an XML resource or not.
- The possibility to sign selected parts of an XML document.
- The means to transform two logically equivalent XML documents, but with syntactic differences, into the same physical representation. This is referred to as canonicalization. In order to be able to verify the signature of an XML resource that has had its representation changed, but still has the same logical meaning, it is essential that canonicalization is performed as part of the XML signature creation and verification processes.

WS-Security defines a security header for use within SOAP messages. Using this security header, WS-Security defines how XML Signature (and XML Encryption) can be applied to SOAP

[6] All networks used link-level cryptography, except the SIA-JEWCC link which used IP crypto.

messages. WS-Security also provides other functionality, however, including a way to include security tokens in SOAP messages. Security tokens are typically used to provide authentication and authorization. In our case, SAML assertions were used as security tokens.

The Security Assertion Markup Language (SAML) defines how to express security assertions in XML. Conceptually, an assertion is a set of statements, made by an asserting party (i.e., a SAML issuer), that a relying party may trust.

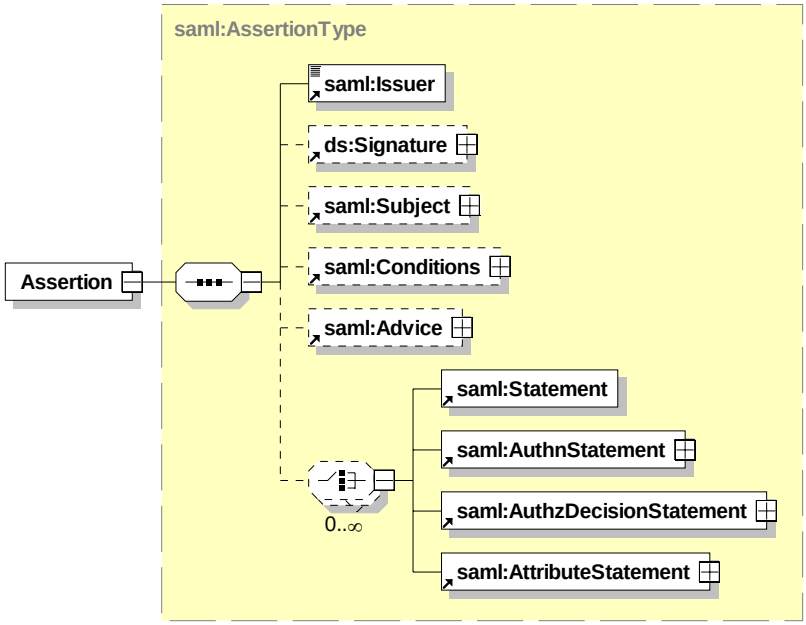


Figure 7.1 The SAML assertion element

The SAML assertion XML element is shown in Figure 7.1. As can be seen, the SAML assertion identifies the issuer of the assertion and may contain one or more statements. In our case, the attribute statement was used for communicating the roles of the subject specified in the subject element. The SAML assertions were also digitally signed by the issuer, using XML Signature, in order to provide integrity protection.

In addition to specifying the SAML assertion format, SAML also defines protocols by which SAML assertions can be exchanged and requested.

The eXtensible Access Control Markup Language (XACML) is a specification for defining access control policies using XML. In addition to defining the policy language for expressing policies, XACML also provides an architectural model.

A more complete description of these security standards can be found in [9].

7.2 Implementation and Usage at Multinett II

We will now describe the security architecture of the SOA software used at MNII. It should be noted that the main intention of the experiment with regard to security has been to gain

experience on the use of the mentioned XML and Web Services security standards in a military environment. Thus, securing the system has not been a primary goal.

Recall from Chapter 5 that a SOA Coordinator may subscribe to one or more topics from a WS-Messenger instance. In order to perform access control, we require all subscription requests to contain a SAML assertion (with an attribute statement) specifying the role of the requester. The SAML assertion is obtained by the SOA Coordinator from a SAML issuer (i.e., identity provider). The SOA Coordinator includes the SAML assertion within the request message using the security header of WS-Security. The request message is also signed using XML Signature, in compliance with WS-Security.

The use of digital signatures obviously requires some type of key management to be in place. In our case, this was solved through predistributed keys, where the keys were stored in a Java key store at each node.

Upon reception of a subscription request at the WS-Messenger side, the request message is first subject to signature verification. If the signature verification succeeds, further access control is performed based on the role specified in the attribute statement of the SAML assertion.

The access control decision itself is taken by a XACML policy decision point (PDP) based on the applicable XACML security policy. Both the SAML identity provider and the XACML policy decision point (PDP) were run as services on a WSO2 Web Services Application Server, and were part of a software delivery from Thales [5]. The SAML identity provider was based on Open SAML 2 [6]. The PDP used was the one contained within JBoss XACML [7] which again is based on Sun's XACML implementation [8].

Digital signatures were also applied to notification messages published through WS-Messenger (i.e., the PlatformType, SignalType, and GeolocationType messages) ensuring the integrity and authenticity of these messages.

The above described security functionality was made available to WS-Messenger and the SOA Coordinator through a Java jar-file (i.e., *esmsec.jar*). In particular, this jar-file contained one class for outbound security and one class for inbound security. The outbound security class contains a method to add a signature to an outbound message and a method to obtain a SAML assertion and add this assertion to an outbound message. For signature verification, the inbound security class contains a method to verify the signature of an incoming message. The inbound security class also contains a method to determine if access (i.e., a subscription) should be granted or not. This method works by first obtaining the role of the requester from the SAML assertion, contained in the subscription request message, and then sends a request to the PDP point whether access should be granted or not. The XACML PDP then makes the access decision, based on the supplied role and the access control policy for the given subscription service.

Because the methods in *esmsec.jar* interacted with the SAML identity provider and the XACML PDP, it had dependencies to many of the libraries/jar-files used by the XACML and SAML implementations. Some of these jar-files were in conflict with the jar-files required by WS-Messenger. Thus, in order to resolve this problem, *esmsec.jar* was run in a separate Java Virtual Machine. A wrapper (i.e., *SecurityServer.jar*) was built around *esmsec.jar* in order to make it available to WS-Messenger and SOA-Coordinator (through *SecurityServerClient.jar*). This way, a modular architecture was obtained, where *esmsec.jar* (through *SecurityServer.jar* and *SecurityServerClient.jar*) shielded WS-Messenger and SOA-Coordinator from the details of the security related implementation. For instance, all communication with the security services running within the WSO2 Web Services Application Server (i.e., the XACML PDP and the SAML identity provider) was handled within *esmsec.jar*.

8 Experiment execution

8.1 Experiment network configuration

In Chapter 4, we presented the planned network setup for the experiment. However, due to unforeseen circumstances, the frigates were unable to take part in the UIDM-based network. Thus, we had to change our configuration, and connected the two laptops that were supposed to be on the two frigates to the local network at the SIA location instead (see Figure 8.1). That meant replacing the radio connection with a wired connection, as shown in Table 8.1.

Entity	**Connection to SIA**
DA-20	Radio network, UIDM IP modem, approximately 16 Kbps
Frigate	100 Mbit Ethernet (local connection within the experiment authority location)
JEWCC	2Mbit secure wired connection
NORTaC	100 Mbit Ethernet[7]

Table 8.1 Final network connection overview

This change of configuration meant that we would not be able to test the SOA solution over an actual disadvantaged grid. The SOA solution would now use only 2 Mbit and 100 Mbit connections, and the ESM information that was supposed to be gathered and provided to the SIA by the frigates was now unavailable. However, after some discussion we came up with an ad-hoc solution to this problem: One of the frigates would collect and manually send information to us using a manual system, whereas a local operator at the SIA would have to read these messages and manually input the data into the CESMO software. While not ideal, this was better than receiving absolutely no information from the frigates whatsoever. As an added bonus, we were

[7] Please note that 100 Mbit is not a usual operating speed for NORTaC, but in the case of the MNII experiment the SIA was located in the experiment authority container complex, where NORTaC was being operated from. Thus, we used an Ethernet connection from SIA to NORTaC, which yielded a local area connection with 100 Mbit.

also able to receive ESM information from both the Orion aircraft and a submarine that were participating. They reported their findings via voice to a frigate, which in turn could include this information in its messages to us. So, all in all, we received ESM data from quite a few platforms, even if the method of delivery (i.e. e-mail) did not allow us to showcase the full functionality of the SOA software.

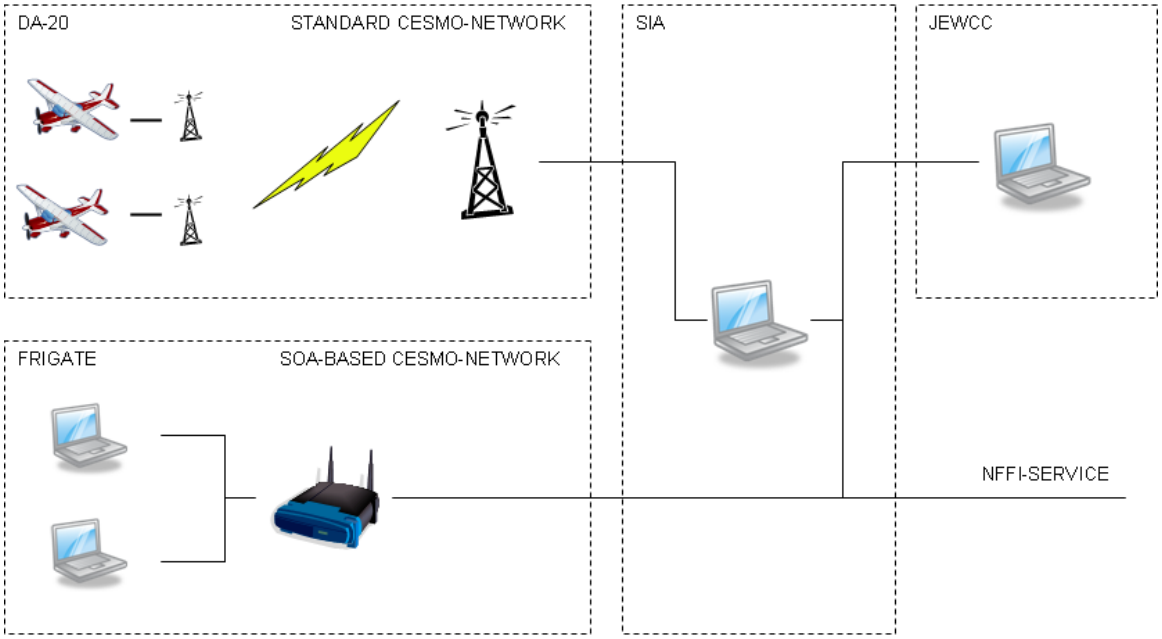


Figure 8.1 The final MNII network setup

8.2 SOA experiment execution

The experimentation with the SOA solution (see Chapter 6) eventually went as planned, and we were able to achieve most of the goals we had set. The only thing we were unable to test was the performance of the SOA solution over disadvantaged grids, since the frigates did not participate in the UIDM network.

In summary, we were able to investigate

- Service discovery, which yielded a simpler setup and bootstrap of the software, and led to less need for manual configuration.
- The added value of SOA: We were able to wrap non-Web Service enabled CESMO software in a Web Service and connect it to our software. Additionally, we were able to offer several new services based on information from the CESMO system, namely a NFFI service and a CSV service. These new services allowed previously separate systems to receive information from the CESMO network.
- Using the SOA software with real data under real workloads was a benefit; previously we had verified its functionality in our lab environment. During the experiment we saw that the software could handle the data and usage patterns of an operational system.
- The benefits of pub/sub, in that one easily could control the flow of information between the different parts of the system. Subscriptions allowed information consumers to

subscribe to only those messages which were of interest to that particular system. Published messages went out only to subscribers, and were sent asynchronously with no need to poll for new data. This reduced network load since only relevant and needed information was sent over the network at any time.

- Using compression on the XML messages meant that the data exchange of the SOA system was comparable to that of the standard CESMO messages. In other words, we got a lot of added value from introducing just a little overhead.

However, before we were able to get the software up and running properly we had to tackle some hardware configuration issues. Basically, the configuration of the network at the SIA turned out to be a problem, since it overloaded the UIDM modem. A detailed description of this problem and how we solved it is presented below in Section 8.2.1. Also, we had to replace the service discovery solution in order to get our software ready in time for the MNII experiment. Originally we started out with WS-Discovery in our initial lab tests, but we found that it did not work particularly well together with the UIDM modems. This issue is discussed below in Section 8.2.2.

8.2.1 UIDM issues

The UIDM modems had an issue with internal buffer space. If too much information was sent to the UIDM modem, *even if the packets were not addressed to it*, then the modem would choke because of too much information and hang. After that it would have to be reset in order to function again. In our initial setup at the SIA (shown in Figure 8.2), we had all the machines constituting the SIA connected to one HUB, which in turn was connected to another HUB where the SOA part of the SIA and the simulated frigates were attached. The problem with using a HUB is that all packets it receives on one interface is sent out again on all the other interfaces. This meant that every single IP packet that was transmitted from any of the machines at the SIA would eventually make its way to the UIDM, thus filling up and overflowing its internal buffer. One would think that the UIDM would filter out any packet which was not addressed to its network, but this seems not to be the case since its internal buffer overflowed.

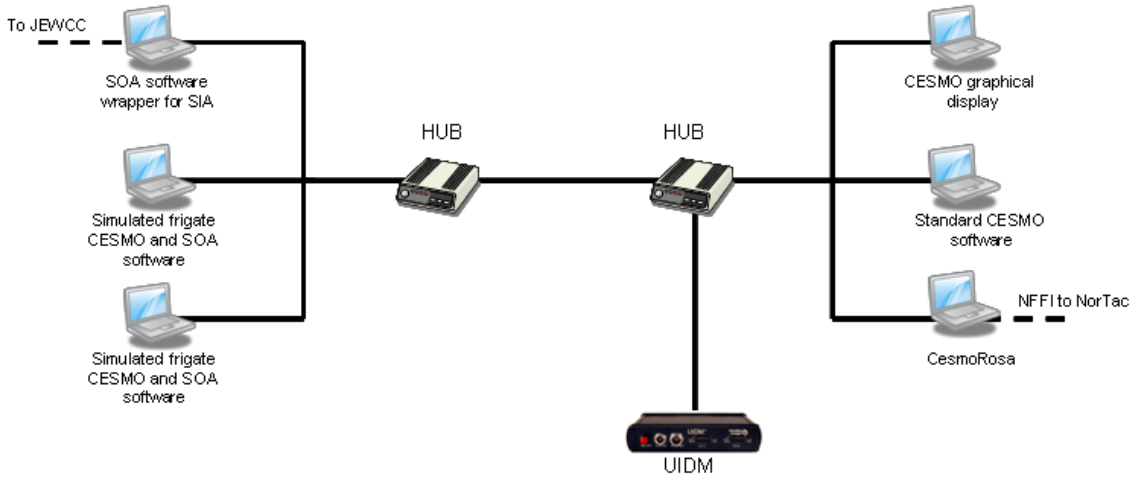


Figure 8.2 Initial SIA configuration

After having reset the UIDM modem several times and analyzed the traffic on the network, we concluded that we would have to reconfigure the network at the SIA if we were to have a successful experiment. We replaced the HUB connected to the UIDM modem with a switch and made some other minor reconfiguration (see Figure 8.3). A switch, unlike a HUB, will over time learn which interface is connected to which port, and selectively transmit IP packets only on the port that has the corresponding interface connected to it. This meant that using this setup, the UIDM could see only the packets that were intended for it (i.e. the CESMO packets to and from the DA-20s).

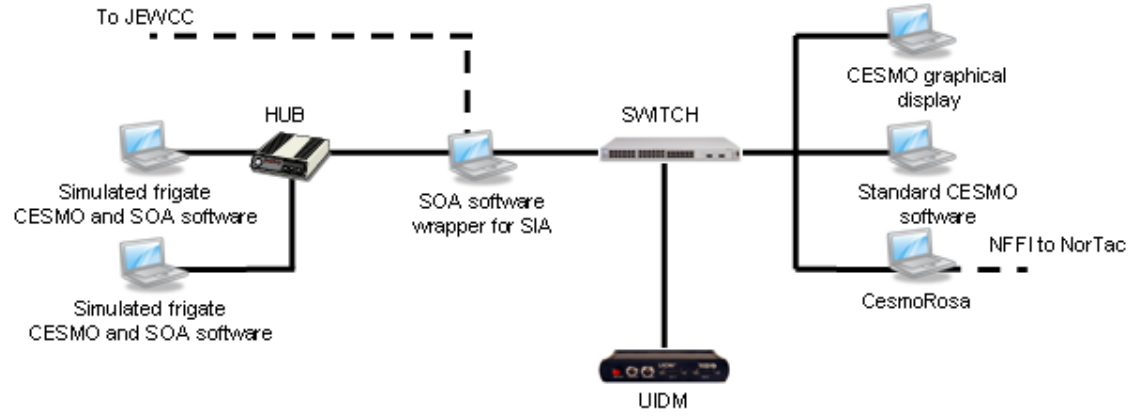


Figure 8.3 Final SIA configuration

With this setup the UIDM performed flawlessly, indicating that one should be very careful with how one goes about to configure a network. Basically, one should connect the UIDM directly to an Ethernet interface on a computer using it, or, failing that, one should use a switch and *not* a HUB. The other reconfiguration we did which you can notice in Figure 8.3 was that we connected the frigates to an interface on the SOA wrapper machine and not to the switch. The reason for doing this was that we could then totally separate the traffic between (simulated) frigates and the rest of the SIA functionality.

8.2.2 Service discovery issues

Our initial service discovery module in the SOA software was WS-Discovery. WS-Discovery is a draft specification of a service discovery system for Web Services in local area networks. The current WS-Discovery draft is tightly coupled with the SOAP over UDP specification [10]. We have a Java implementation of the mandatory parts of the WS-Discovery draft, see [11] for further details.

In our lab setup we encountered two issues with WS-Discovery, or rather the communication part of it; SOAP over UDP:

1. Excessive transmission redundancy jammed the network.
2. Probe matches were too large to be sent over the UIDM modem.

The first issue arose because the SOAP over UDP specification dictates that every message should be sent four times to ensure its delivery across the network. While this may well work as intended in an office Ethernet or civil WLAN, it is not a very good idea in a tactical network with low bandwidth. All in all, WS-Discovery used too many resources in the UIDM modem, exhausting buffer space when we allowed this default behavior, and no useful information could get through. We addressed this issue by reducing the number of times to transmit each message from four to one.

The second issue was that service discovery lookups sometimes could not be sent to the requesting node. The way WS-Discovery works is that someone wanting to discover a service sends a *probe* message, and then the node(s) which provide(s) the services will send a *match* back. This match message contains all WSDLs of all services on the node(s), and resulting in a large message if the node has a couple of services. Basically, the probe matches were too big to be sent over the UIDM (due to low MTU), and the modems had issues with IP fragments. This meant that in order to get WS-Discovery to work over the UIDM network we would either have to implement application level fragmenting of messages or violate the WS-Discovery draft specification by allowing a *probe* to be answered by several *match* messages, thus putting one WSDL in each reply and in this way circumventing the UIDM MTU problem. Obviously, we would not be able to get WS-Discovery as it was specified in the draft to work. We were also pressed for time, and thus chose to use a simpler, well-tried service discovery mechanism (see Section 5.2) for use at MNII instead.

8.3 Security experiment execution

The experiments at MNII illustrated how civilian standards for SOA security can be used in a military setting. Some issues require further discussion however.

Although we were able to utilize all the intended XML and Web Services security standards successfully during the experiment, we experienced problems related to XML Signature. More specifically, we were only able to verify the signatures when UDP was used as transport protocol by WS-Messenger. When HTTP was used as transport protocol, signature verification

consistently failed and had to be disabled. The reason for this was apparently that WS-Messenger performed XML deserialization and serialization of the messages when HTTP was used, which caused changes to the layout of the XML that could not be undone by XML canonicalization. In particular, XML canonicalization retains all whitespace that is between a start/end tag pair. Therefore, any attempts by an intermediary to introduce blanks or newlines (e.g. in order to improve readability) will change the checksum of the document and break the signature. Consequently, an XML signature is broken by layout changes such as changes in indentation.

The availability of the security services is also a critical point that needs to be taken into consideration. In our case, the security services (i.e., the SAML identity provider and the XACML PDP) were replicated at each location. An advantage of such local replication is that the services remain available even in the case that the communication between locations fails. Considering that the unavailability of a security service may prevent access to other services, this is an important consideration. On the other hand, if the information relied on by the security services are of a dynamic nature, security will likely be affected unless consistency can be ensured. In addition, there are also bandwidth considerations that may play a role between a more centralized or decentralized/replicated architecture, depending on the usage patterns and the frequency of updates.

As mentioned previously we also relied on predistributed keys for the experiment. In a scenario with few participants and limited bandwidth this may provide a good solution. However, such a solution clearly does not scale to a high number of participants, in which case a more sophisticated key management scheme would be required.

9 Conclusion

In this report we have presented the experiments performed by P1086 at the exercise “Multinett II”. The goal of our participation was, by using CESMO as a case, to show that introducing SOA into an existing operational system can contribute to automate and speed up existing processes, as well as add value through new and extended functionality.

This exercise was particularly interesting to us since it provided an opportunity to demonstrate SOA on an operational disadvantaged grid (the IDM network). However, due to a number of circumstances, this turned out to be impossible, which in turn meant that the value of the experiment was somewhat reduced for us. Although we had tested and verified the SOA infrastructure on the IDM network (modems connected back to back), not being able to run our infrastructure on a radio network meant that valuable testing and measurements could not be performed.

Despite a few problems, we were able to test quite a few aspects of SOA:

- Service discovery, which yielded a simpler setup and bootstrap of the software, and led to less need for manual configuration.

- The added value of SOA: We were able to wrap non-Web Service enabled CESMO software in a Web Service and connect it to our software. Additionally, we were able to offer several new services based on information from the CESMO system, namely a NFFI service and a CSV service. These new services allowed previously separate systems to receive information from the CESMO network.
- Using the SOA software with real data under real workloads was a benefit; previously we had verified its functionality in our lab environment. During the experiment we saw that the software could handle the data and usage patterns of an operational system.
- The benefits of pub/sub, in that one easily could control the flow of information between the different parts of the system. Subscriptions allowed information consumers to subscribe to only those messages which were of interest to that particular system. Published messages went out only to subscribers, and were sent asynchronously with no need to poll for new data. This reduced network load since only relevant and needed information was sent over the network at any time.
- Using compression on the XML messages meant that the data exchange of the SOA system was comparable to that of the standard CESMO messages. In other words, we got a lot of added value from introducing just a little overhead.
- We were able to utilize all the intended XML and Web Services security standards successfully during the experiment, except for XML Signature. More specifically, we were only able to verify the signatures when UDP was used as transport protocol by WS-Messenger. See section 8.3 for a discussion.

All in all the experiment was an important proof-of-concept, allowing us to test SOA in a operational setting. The problems we encountered gave us an insight into limitations of the existing solutions, providing us with issues to tackle in future research.

References

[1] D. Eastlake, J. Reagle, and D. Solo “XML-Signature Syntax and Processing,” W3C Recommendation, 2002.

[2] A. Nadalin et al., "Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)," OASIS Standard, 2006.

[3] S. Cantor et al., "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) v2.0," OASIS Standard, 2005

[4] Tim Moses, "eXtensible Access Control Markup Language (XACML) version 2.0," OASIS Standard, 2005

[5] Morten Andresassen, “SAML XACML XKMS Software,” 18.8.2008.

[6] Open SAML, https://spaces.internet2.edu/display/OpenSAML/Home/

[7] Anil J. Saldhana, “User Guide for JBoss XACML – A Guide for Developers”, http://www.jboss.org/file-access/default/members/jbosssecurity/freezone/releases/jbossxacml/2.0.2.GA/guide/pdf/jbossxacml.pdf

[8] Sun’s XACML Implementation, http://sunxacml.sourceforge.net/

[9] N. A. Nordbotten, “XML and Web Services Security,” FFI-rapport 2008/00413, 2008

[10] Harold Combs et al., “SOAP-over-UDP”, 2004, http://specs.xmlsoap.org/ws/2004/09/soap-over-udp/soap-over-udp.pdf

[11] M. Skjegstad, F. T. Johnsen, T. Hafsøe, “A Java Implementation of the WS-Discovery 2005 Draft”, FFI-Notat 2008/02209

[12] Indiana University, “WS-Messenger (WSMG)”, http://www.extreme.indiana.edu/xgws/messenger/, 2006

[13] Yefim Natis, Gartner Research, “Service-Oriented Architecture Under the Magnifying Glass”, Application Integration & Web Service, Summit 2005, April 18-20, 2005

[14] F. T. Johnsen and T. Hafsøe, “Using NFFI Web Services on the tactical level: An evaluation of compression techniques”, 13th ICCRTS, Seattle, USA, June 2008

[15] L. Johnsrud, D. Hadzic, T. Hafsøe, F. T. Johnsen, L. Ketil, “Efficient Web Services in Mobile Networks”, ECOWS’08, Dublin, Ireland, November 12-14, 2008

[16] K. Lund et al., “Using Web Services to Realize Service-Oriented Architecture in Military Communication Networks”, IEEE Communications magazine, Special issue on Network-Centric Military Communications, October 2007

Appendix A SOA software details

A.1 Installation

Several software components must be in place to run the SOA software system on each network node. The legacy CESMO software must be installed, see paragraph [XX] for details. Subsequently, the SOA Coordinator must be installed. It's contained within the "WsmgFFIClients" folder, and includes the following components:

- WsmgFFIClients.jar –SOA Coordinator executable application library.
- Wsmg.jar – Third party WS-Messenger application library, here used for client side operations (for initiating connections to the server).
- Xsul-2.7.9-FFI-1.0.jar - Third party utility-library for handling network communication, utilized by WS-Messenger. Modified for FFI needs.
- UdpDiscovery.jar –Service discovery library
- NifProvider.jar –Utility-library providing support for multiple network interfaces on the same node.
- GlobalProperties.jar –Centralized configuration library for managing consolidated configuration across Java applications.
- Log4j-1.2.8.jar – Third-party library for logging in Java applications.
- EfxSever.jar – Utility-library providing support for XML compression using the EFX algorithm. As the compression utility itself runs inside a virtual machine, EfxServer.jar provides functionality for remote invocation of the compression utility. Here used for client side operations.
- SecurityServer.jar – Executable application library running in conjunction with Wso2 server. Isolates namespaces preventing namespace collisions and versioning issues.
- Crypto.properties – Properties file needed by wso2 server.
- SecurityServerClient – Client side utility library for invoking Security Server methods.
- Xpp3-1.1.3_7.jar – XML Pull Parser, third party utility library used by Xsul and WS-Messenger.
- Wso2 lib folder – Folder containing all ws02 server files.

WS-Messenger requires a MySQL database to be installed, to which it persists all subscriptions. MySQL should be downloaded from " http://dev.mysql.com/downloads/", and installed with standard settings. The WS-Messenger database should be installed using the database creation script, see WS-Messenger database creation script. By default the database name is "WSMG", and username and password is also "WSMG". To change this, update the file "wsmg.db.JdbcStorage.Java" and change the following settings:

```
jdbcUrl="jdbc:mysql://localhost:3306/wsmg?user=root&password=wsmg";
jdbcDriver="com.mysql.jdbc.Driver";
```

Note that the default port is set to 3306.

A.2 Configuration

In order to consolidate configuration and facilitate easy configuration management, the software library *GlobalProperties.jar* was used by SOA Coordinator, WS-Messenger, XSUL-library (communication library used by WS-Messenger) and UdpDiscovery. The jar-file contains the properties-file storing the configuration settings as well as the Java library needed to access them. This ensures independency of the underlying file- and operation-system, and requires only that Java applications using it have access to it through their class path.

Below follows a detailed description of the non-self-explanatory settings in GlobalProperties.jar.

multicastTTL=4 – The maximum number of hops for UDP multicast packages.
useMulticast=true – Whether WS-Messenger should use UDP multicast or unicast for notifications.
multicastAddress=224.224.224.224 – Specifies the multicast port used by WS-Messenger when multicasting notifications.
compressionLevel=soapPart – Specifies whether the whole message should be compressed (soapMessage) or whether only an inner XML element should be compressed (soapPart).
innerMessageElementName=StringNotification - InnerMessageElementName is required when compressionLevel is set to soapPart. Specifies the element containing the #compressed soap part. "StringNotification" is the default parent element created by wsmg.
compressorClass=ffi.GZipCompressor - Specifies which compression utility/algorithm to use (e.g ffi.EfxCompressor)

nonNotificationMessagesProtocol=udp - Protocol used for sending system messages, e.g subscriptions. (Notification-messages are sent using the protocol specified by their address. (e.g udp://localhost:10)
logFile=[appPath]\\xsulLog.html – Logfile path and file name. ”[appPath]” is optional and substituted with the application path.
logLevel=INFO – Specifies the desired message logging granularity.
mtu=500 – Specifies the UDP max MTU in bytes. Messages larger than the MTU will be automatically fragmented and defragmented at the application level. Useful when transmitting through UIDM modems with a maximum MTU of below 600 bytes.
delayBetweenFragmentsInMillis=10 – Specifies the number of millseconds between the transmission of UDP fragments. Avoids buffer overrun problems in the UIDM modem.
defaultNetworkInterfaceIpAddress=193.156.33.75 – Specifies the network interface address associated with the default network adapter on the machine. Used by the SOA software components.
removeSubscriptions=False – Instructs WS-Messenger not to remove client subscriptions after a time of inactivity.

offeredServices=serviceA:[localIp]:8765:node1 serviceB:[localIp]:5555:node2 - Specifies which services the UdpDiscovery component will offer to the network.

topicMappings=topic_A:topic_B – Creates a mapping between two service topics. When Soa Coordinator receives a message on topic_A, it will automatically republish it on topic_B. Note that the mapping does not also work the other way around.

localNodeId=node1 – Specifies the name the UdpDiscovery component should broadcast as it's own node name.

chatMulticastAddress=224.224.224.222 – The multicast address used for chat.
chatMulticastPort=9753 – The multicast port used for chat.
chatCompressorClass=ffi.GZipCompressor – Specifies a fully qualified class name from which an object should be instantiated to handle the XML compression. The class is loaded dynamically at run-time, and new implementation could be added to the system at any point without recompilation. The class must implement the interface ffi.xsul.compression.Compressor.

acceptedCesmoidIDs=41 53 – Specifies which Cesmo IDs, local or remote, that will be handled by Soa Coordinator.

startScreenLoggger=True
screenLoggerPath=c:/temp/img/ - The path where the captured GIFs are stored.
screenLoggerInterval=5000 – Screen capture interval.

udpDiscoveryMulticastAddress=224.224.224.224 – The UDP multicast address used for service discovery.

udpDiscoveryMulticastPort=6666 - The UDP multicast port used for service discovery.

advertisementInterval=60000 – Specifies the interval in milliseconds between keep-alive messages used for service discovery. Should be lower than ServiceTTL-setting.

udpDiscoveryRandomResponseSeconds=8 – The maximum number of seconds to wait before responding to a request. The higher the value, the lesser chance for multiple nodes replying to the same request.

serviceTTL=80000 – Specifies the maximum number of milliseconds that may pass since last keep-alive message, without regarding the node as inactive. Should be higher than the advertisement interval.

addAssertion=false

checkAssertion=false

addSignature=false
checkSignature=false

cesmoSoaSidePort=8000
cesmoCesmoSidePort=9000
cesmoMulticastPort=4666
cesmoCesmoSideAddress=pckjru
cesmoMulticastAddress=230.0.0.1

localCesmoidId=RR
cesmoNetworkInterfaceIpAddress=192.168.2.1 – The network interface ip address to use for the Cesmo network (if other than default network).
cesmoMessageFile=c:/cesmoMessage.txt

A.3 Execution

A.3.1 Startup

Before operating the SOA Coordinator, three applications should be started on each node: The CESMO-software, SOA Coordinator and WS-Messenger. The order in which they are bootstrapped is insignificant.

”WSMG STARTUP.bat” starts WS-Messenger. It assumes that the MySQL database is in place, and that WS-Messenger itself is installed at the following path: ”C:\SOA\wsmg jar\”;
The output is presented in a DOS-shell, and should look something like this:

Get init()

Calling cleanup()

Batch update successful.

Queue is cleaned.

Finished Calling cleanup()

WS-Eventing/Notification Broker(XSUL) service (version:1.76.2 After_StressTest_addedStat_f1 XPathFilterEnabled=true) started on http://192.168.2.1:8765

***Delivery Thread started.

Starting Subscription Cleaning up Thread.

TotalRecord=3

******Topic=GeolocationType

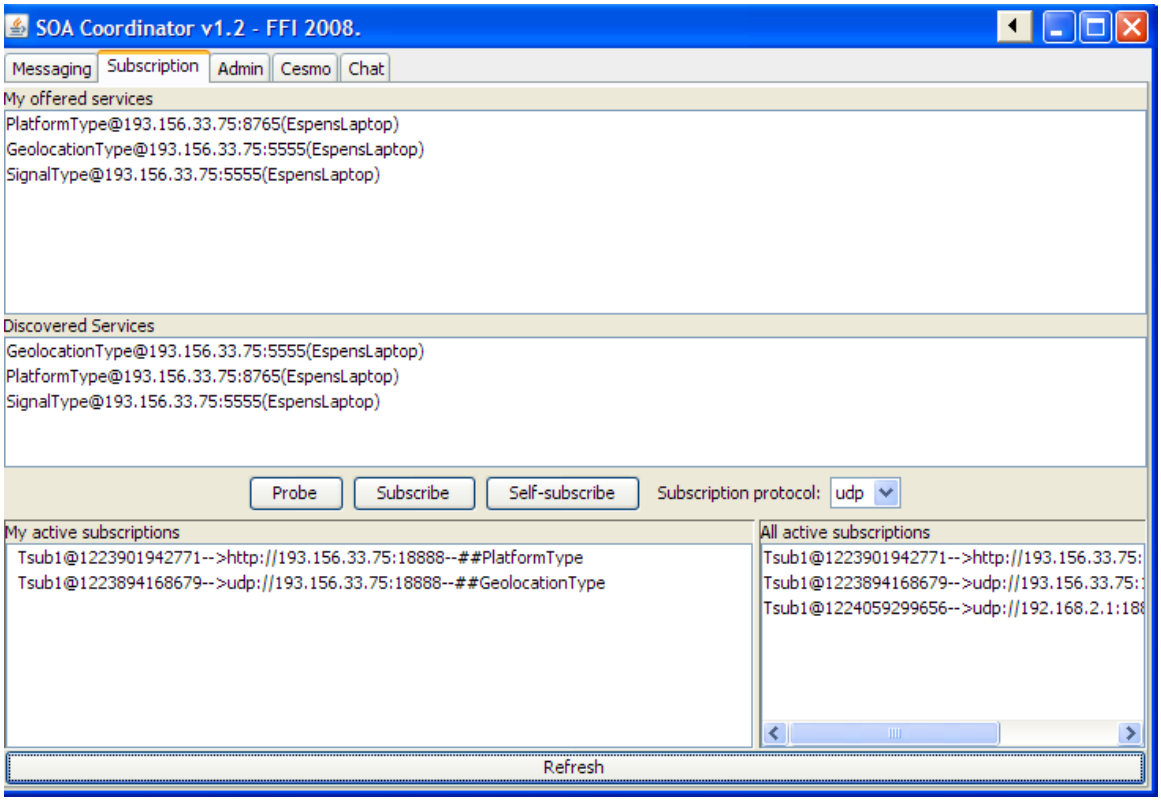
******Topic=PlatformType

******Topic=GeolocationType

```
0    [Thread-4] INFO  xsul.ws_addressing.WsaInvoker  - Listening for UDP datagrams...
469  [Thread-4] INFO  xsul.ws_addressing.WsaInvoker  - using multicast.
```

Similarly, SOA Coordinator should be started by executing the batch file ”SOA STARTUP.bat”, and presents the user with a graphical user interface.

A.3.2 Operation



SOA Coordinator has several tabs, each presenting the user with different groups of functionality.

Messaging: This tab enables the user to manually publish SOA messages to an WS-Messenger instance. The WS-Messenger server address and port is specified, as well as the topic the messages should be publish under. The tab also offers a button to manually start the HTTP-listener (The UDP-listener is started automatically).

Subscription: This tab enables the user to manually create SOA subscriptions. The user selects the discovered services of choice, and presses ”Subscribe”. The subscription-messages it sent to the WS-Messenger server instance specified on the ”Messaging”-tab. ”Self-subscribe” can be used for debugging purposes, and allows the user to receive the messages it publishes itself.

Admin: Used to create and remove topic mappings. Causes received messages to be republished under a different topic, causing SOA Coordinator to behave as an ”intelligent” proxy.

CESMO: This tab allows the user to monitor CESMO-messages received and sent from SOA Coordinator. The panel on the left hand side show CESMO-messages received from a remote CESMO (transmitted over the SOA network) while the panel on the right hand side displays messages coming from the local CESMO software, which is subsequently published to the WS-Messenger instance (again, specified on the ”Messaging”-tab).

Chat: This tab allows users to send IM-messages to one another. The panel on the left hand side displays a list of active nodes (including yourself). The list is based on the service discovery functionality, and removes any nodes that fail to send keep-alive messages at the specified interval. Messages sent from the chat-tab will be multicasted, and received by all active nodes. Submitting the ”cls”-command clears the screen.

A.4 WS-Messenger database creation script

The following SQL-script should be used to create the MySQL database used by WS-Messenger for storing subscriptions.

```
##Used for mySQL database
CREATE TABLE `subscription` (
                `SubscriptionId` varchar(200) NOT NULL default '',
                `Topics` varchar(255) default '',
                `XPath` varchar(200) default '',
                `ConsumerAddress` varchar(100) default '',
                `ReferenceProperties` blob,
                `xml` blob,
                `wsrm` tinyint(1) NOT NULL default '0',
                `CreationTime` datetime NOT NULL default '0000-00-00
00:00:00'
              );

CREATE TABLE `msgbox` (
  `id` int(11) NOT NULL auto_increment,
  `xml` blob NOT NULL,
  `msgboxid` varchar(100) NOT NULL default '""',
  PRIMARY KEY  (`id`),
  KEY `MSGBOXID` (`msgboxid`)
) ;

CREATE TABLE `disQ` (
          `id` bigint(11) NOT NULL auto_increment,
          `trackId` varchar(100) default NULL,
          `message` mediumblob,
          `status` int(11) default NULL,
          PRIMARY KEY  (`id`)
        );

CREATE TABLE MaxIDTable(
  maxID integer
  );

CREATE TABLE MinIDTable(
  minID integer
);
```