The University of Kansas

Technical Report

Application of the Java Message Service in Mobile Monitoring Environments

Martin Kuehnhausen and Victor S. Frost

ITTC-FY2010-TR-41420-18

December 2009

Project Sponsor:
Oak Ridge National Laboratory
Award Number 4000043403

Table of Contents

List of Figures

List of Tables

Application of the Java Message Service in Mobile Monitoring Environments

Martin Kuehnhausen, *Graduate Student Member, IEEE* and Victor S. Frost, *Fellow, IEEE*

***Abstract*—Distributed systems and in particular sensor networks are in need of efficient asynchronous communication, message security and integrity, and scalability. These points are especially important in mobile environments where mobile remote sensors are connected to a control center only via intermittent communication, e.g. via satellite link. We present an approach that is able to deal with the issues that arise in such scenarios. In particular we focus on providing a solution that allows for flexible and efficient cargo monitoring on trains.**

The Java Message Service presents a flexible transport layer for asynchronous communication that provides a transparent *store-and-forward* queue mechanism for entities that need to be connected to each other. Previously JMS was primarily used in always-connected high-bandwidth enterprise communication systems. We present the advantages of using JMS in a mobile bandwidth limited and intermittently connected monitoring environment and provide a working implementation called the Transportation Security SensorNet (TSSN). It makes use of an implementation of JMS called ActiveMQ that is used here to enable monitoring of cargo in motion along trusted corridors.

Results obtained from experiments and a field trial show that using JMS provides not just a practical alternative to often custom binary communication layers but a better and more flexible approach. One reason for this is transparency. Applications on both communication ends only need to implement JMS connectors while the remaining functionality is provided by the JMS implementation. Another benefit arises from the exchangeability of JMS implementations.

In utilizing JMS we present a new and flexible approach to deal with challenges such as intermittent and low-bandwidth communication in mobile monitoring environments.

***Index Terms*—Telemetry, Transport protocols, Intermittently connected wireless networks, Communication system software, Data communication, Software engineering**

I. Introduction

THE primary use of Java Message Service (JMS) is in always-connected high-bandwidth enterprise communication systems but its concepts and techniques are useful in other scenarios as well. This paper describes the application of JMS in mobile monitoring environments.

One of the main advantages of using JMS is the fact that applications do not need to be modified to implement their own *store-and-forward* or *resend* mechanisms. How this is achieved is explained in detail later. Furthermore an example of an implementation of a mobile monitoring system is given that was field tested in stationary as well as intermittent mobile scenarios.

M. Kuehnhausen and V. S. Frost are with the Information and Telecommunication Technology Center, The University of Kansas, Lawrence, KS, 66045, USA; Corresponding author: mkuehnha@ittc.ku.edu

This work was supported in part by Oak Ridge National Laboratory (ORNL)—Award Number 4000043403. This material is also partially based upon work supported while V. S. Frost was serving at the National Science Foundation.

II. Problem area

Whenever disparate systems are deployed in the field that need to communicate with each other and a control center, there exist particular problems that need to be addressed. Here we used the following scenario as a motivating example.

Sensors are connected to cargo containers which they monitor. A train is then used to transport these containers. The sensors have limited capabilities and are managed locally by a more powerful sensor node which has extended functionality including a communication link back to a control center. Whenever a sensor detects an event it notifies the sensor node immediately. The sensor node then performs a simple evaluation of the event and decides whether or not to send it to the control center. In this paper we focus on the part that comes next, sending messages to and receiving control messages from the control center.

The communication link used may provide only intermittent communication. Therefore, the sensor node must deal with establishing the connection as well as transmitting messages. Especially the latter can cause problems. In a synchronous communication model the sensor node would only be able to send one message at a time and block while waiting for its acknowledgement. This is not feasible in this case because of the intermittent connection, low bandwidth and high latency of the communication link. An asynchronous communication model overcomes this blocking problem and is therefore more suitable. Furthermore, since messages cannot be send out immediately due to the intermittent connectivity they need to be stored. This is often done by implementing a queuing mechanism inside of the sensor node.

It is also possible to send control messages such as location or receive status inquiries from the control center to a specific sensor node. Again, since there does not necessarily exist an active connection to the sensor node messages need to be queued. Hence, the applications in the control center are responsible for implementing proper queuing and retry mechanisms.

Security and message integrity are critical aspects of the overall monitoring system. If thieves were able to tamper with the message contents then they could easily spoof the system. Security is essential and needs to be implemented in each application that sends or receives messages as part of the monitoring system. This brings up another issue, scalability. Implementing asynchronous communication and security components for each application in a small system may work for experiments but is not feasible for large production environments. In terms of the cargo monitoring scenario for trains there could be many control centers, thousands of sensor

nodes and even more sensors on containers. This is a very common scenario for sensor network deployments even though the particular details of the deployments may be different. In this paper we demonstrate that by using JMS in these mobile monitoring environments it is possible to overcome the common problems discussed above. In particular we focus on the problems of asynchronous communication, message security and integrity, and scalability.

A. Asynchronous Communication

Reliable communication between control centers and the sensor networks cannot always be ensured. Additionally communication is based on the form of underlying connectivity that is provided. The connectivity may vary. The system could use a 3G system when available and resort to the satellite communication only when needed; exposing several issues.

First, message sizes should be small in order to accommodate for the slow speeds such as satellite communication when needed. Possible optimizations are discussed in III-B4 but compression or conversion into binary formats are suitable options here.

Second, in order to address reliable transmission of messages either a *store-and-forward* or a *resend* mechanism needs to be implemented on both communication ends. The *store-and-forward* technique in this context would mean that the sensor networks need to hold to the data they capture until connectivity is established. By contrast, in the *resend* scenario they would attempt to transmit the data continuously or with a backoff timer.

B. Message Security and Integrity

The data that is produced by sensor networks will likely be sensitive and needs to be kept private. This is especially true for systems whose main purpose is to provide monitoring of cargo. Cargo information as well as status updates and events should only be visible to authorized entities. Furthermore it is critical that messages being transmitted cannot be tampered with, for example control messages that allow the opening of cargo containers.

In this sense it is also important to distinguish between *point-to-point* and *end-to-end* security. Using transit networks or message relay mechanisms is not possible when messages are secured in a point-to-point manner because security may be compromised at each individual connection point. However, in *end-to-end* system security it is possible for messages to pass through individual connection points. Another issue that always needs to be kept in mind is that while control centers often have adequate storage and computing power individual sensors or sensor networks may not. This can be a challenge when implementing security for the targeted scenarios.

C. Scalability

Sensor networks in general can be set up in two basic ways. First, after an initial configuration they repeatedly report their sensor data to a control center. Second, a control center sends out messages to the sensors or sensor networks in order to control their reporting or inquire for specific sensor data. Thus efficient management and scalability can become an issue.

Even though the most common scenario is running a single setup with one central control center or base station and multiple sensors or sensor networks connecting to it, the integration of multiple systems can be problematic. There are issues in dealing with multiple control centers and multiple sensor networks that need to be explored. This is especially important when it comes to managing policies and subscriptions properly.

III. Related Work

A. Java Message Service

Musolesi et al. [1] present their experiences in implementing a system called EMMA (Epidemic Messaging Middleware for Ad hoc networks) based on JMS. In particular they identified the need to adapt JMS in order to be applicable for mobile ad hoc networks. Their approach consists in synchronization of queues using a middleware layer that also manages reachability of individual nodes. For message delivery in partially connected networks they make use of an approach called epidemic routing which is described by [2] which works by propagating messages to neighbors, their neighbors and so on. In contrast the solution discussed here and implemented in the TSSN is standards based using the original JMS specification and therefore more compatible with other systems.

Vollset et al. [3] present a middleware platform built for mobile ad-hoc networks. Their solution is “serverless” in the sense that after an intial setup all the participating entities have a local copy of the JMS configuration. Furthermore they implement a new multicast protocol for delivering messages on JMS *topics* to their *subscribers*. Their platform again is an adaptation of the original JMS standard whereas the solution presented here makes use of a specified and standardized implementation and shows that JMS can be used unaltered.

In general the Java Message Service is primarily used in always connected systems such as the one described by [4]. The Mission Data Processing and Control Subsystem (MPCS) [4] utilizes JMS for different levels of event notifications. However, the communication link with the flight systems is custom. Although an extreme case, it seems that using the Java Message Service for establishing mobile connectivity is undervalued. Another, more realistic example is the Remote Real-time Oil Well Monitoring System [5] where clients receive event notifications via JMS but the data that is collected by the remote terminal units is sent to the data processing station using a custom process.

B. Web Services

Service Oriented Architectures (SOA) present a flexible approach to some of the problems mentioned earlier such as message security and scalability. The idea is to implement specific functionality in web services that communicate with each other using standardized interfaces. Message exchanges in general use the flexible SOAP message format [6]. This has a number of advantages as for instance routing and security are available as extensions to it. JMS is able to transmit SOAP

messages. Hence, applying JMS enables the use of web service specifications in mobile monitoring environments.

1) WS-Addressing: The *WS-Addressing* core specification by [7] and its SOAP binding by [8] defines how message propagation can be achieved using the SOAP message format. Usually the transport of messages is handled by the underlying transport protocol but there are several advantages of storing this transport information as part of the header in the actual SOAP message. For example, it allows the routing of messages across different protocols and management of individual flows and processes within web services.

The Java Message Service uses a similar concept for its addressing but its properties are adapted to the management of messages in queues. However, since SOAP messages can be transported over JMS flexible routing of messages is preserved.

2) WS-Security: The *WS-Security* specification as described by [9] deals with the many features needed to achieve so-called *end-to-end* message security. This provides security throughout message routing and overcomes the limitations of so-called *point-to-point transport layer security* such as HTTPS. Furthermore, the specification aims to provide support for a variety security token formats, trust domains, signature formats and encryption technologies.

Whenever SOAP messages are transported using the Java Message Service, *WS-Security* can be applied. In this scenario JMS simply acts as a tunnel.

3) WS-ReliableMessaging: Without additional specifications like *WS-ReliableMessaging* [10] the delivery of SOAP messages is based purely on best effort and cannot necessarily be guaranteed. The Java Message Service provides several mechanisms for dealing with message reliability issues. Within *transactions* messages are *acknowledged* and if necessary *redelivered*. When a message carries the *persistent* attribute, JMS message *brokers* store the message in order to be able to recover it in case of a failure.

4) Efficient Data Transmission: The SOAP 1.2 Primer [6] includes references to several enhancements of the original SOAP standard. In particular they deal with potential performance problems and the need for binary data transport in SOAP. The *XML-binary Optimized Packaging* (XOP) specification [11] defines the use of *MIME Multipart/Related* messages provided by [12] to avoid encoding overhead that occurs when binary data is used directly within the SOAP message. XOP extracts the binary content and uses URIs to reference it in the so-called *extended part* of the message. An abstract specification that uses this idea is the *Message Transmission Optimization Mechanism* (MTOM) [13].

Another extension of the SOAP standard is the *Resource Representation SOAP Header Block* (RRSHB) [14] that allows for caching of data elements using so-called *Representation header blocks*. They contain resources that are referenced in the SOAP *Body* which might be hard to retrieve or simply referenced multiple times. Instead of having to reacquire them over and over again, a service may choose to use the cached objects which speeds up the overall processing time.

ActiveMQ, which is the JMS implementation that is used in the Transportation Security SensorNet, allows several different protocols (e.g. AMQP [15], OpenWire [16], REST [17], Stomp [18], XMPP [19]), to be used for message transmission. By default it uses the OpenWire protocol, an optimized binary format for fast and efficient communication.

IV. Proposed Solution

In order to be flexible and provide a suitable solution for a mobile monitoring environment a transparent *store-and-forward* approach is used here. A *resend* mechanism is often implemented directly in the application which makes it inflexible. However, the *store-and-forward* approach allows for a more efficient and scalable centralized storage pool that is automatically forwarding the messages.

How JMS can be applied effectively is described here using the Transportation Security SensorNet (TSSN) [20] as an example. The TSSN provides monitoring capabilities in mobile environments and makes use of JMS. The TSSN uses a SOA approach for monitoring cargo in motion along trusted corridors. The system is built using web service specifications and utilizes a Java Message Service implementation for connectivity between its Virtual Network Operation Center (VNOC), the control center in this case, and the Mobile Rail Networks (MRN), which contains the sensor nodes and sensors, it monitors.

The TSSN uses the Java Message Service through one of its open-source implementations called ActiveMQ which is described in detail by Snyder et al. [21]. Each application in the TSSN is a web service. These web services can be utilized through their JMS addresses. ActiveMQ establishes a so-called *queue* for each web service and uses these *queues* to *store-and-forward* messages to them.

This *queue* approach has the advantage that applications do not need to be modified and implement their own *store-and-forward* or *resend* mechanisms. It is also transparent to clients of the web services since apart from using another address, the JMS address, interfacing with the web services stays the same.

In this paper we explain the basic concepts of JMS and in particular how they relate to mobile monitoring environments and one implementation, the TSSN. Furthermore the details for the JMS implementation within the TSSN are discussed.

A. Java Message Service

The Java Message Service [22] provides a standardized specification for synchronously and asynchronously transporting messages using queues. Its implementation is vendor specific but the interfaces are clearly defined in the specification so that in theory this is an open system where changing vendors is possible. The following sections describe the Java Message Service in detail.

1) Components: In the JMS context clients are called *producers* when they create and send messages. The receiving end is called a *consumer*. Note that a client can be both, a producer and a consumer, at the same time. Clients connect to JMS *providers* which are entities that have the specified interfaces to send and receive messages.

Since most of the connections in general are point-to-point, a so-called *queue* is the most commonly used *destination* of a message. It contains messages from *producers* to a

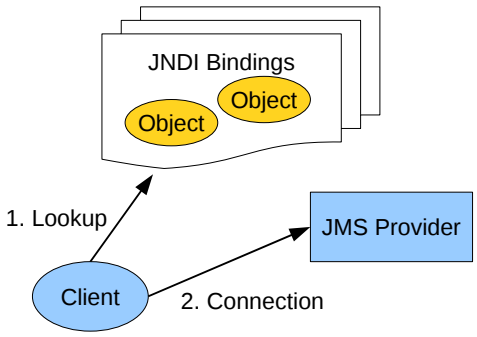


Fig. 1. JMS administration according to [22]

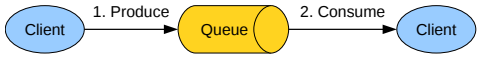


Fig. 2. Point-to-Point messaging

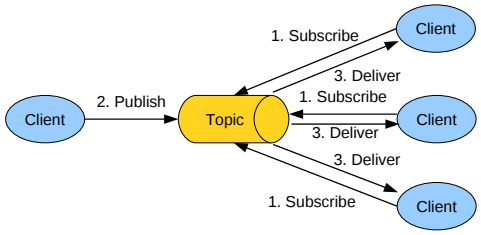


Fig. 3. Publish/Subscribe messaging

single *consumer* that have not been received. Within the TSSN unique *queues* are used to represent the individual web services. Messages are usually delivered in order *First In, First Out* (FIFO) following the basic principle of a queue but this is dependent on the underlying implementation of JMS.

Topics have multiple *consumers* and can have one or many *producers* publishing messages. They are used in publish-subscribe models and contain messages that have not yet been published.

A *message* can be any object or data that needs to be transported using JMS. The Java Message Service describes messages as entities that consist of a *header* which contains identification and routing information and a body carrying the data. Additional properties such as application, provider or standards specific properties can be attached to messages. This is effectively used in providing things like security or reliable messaging.

Note that since JMS per se does not define a message format, implementation may vary significantly. For service oriented architectures the agreed standard message format is SOAP. Easton et al. [23] describe in detail how SOAP can be used within the Java Message Service. Because the TSSN is based on SOA and uses SOAP messages it is able use web service specifications as part of JMS and therefore provide features such as *WS-Addressing* and *WS-Security* as described in III-B.

2) Java Naming and Directory Interface: In order to identify objects within the Java Message Service implementation in a standardized way the specification makes use of the Java Naming and Directory Interface (JNDI) Application Programming Interface [24]. JNDI provides a directory service for objects. This process is used for so-called *connection factories* which are used to establish connections and *destinations* which are either *queues* or *topics* in order to increase portability and ease of administration. JMS clients look up objects and use them in connections as shown in Figure 1. TSSN uses local JNDI repositories for JMS lookups and a combination of *hostname* and *web service name* for uniquely naming *queues*.

3) Messaging Models: JMS supports the two common messaging models; point-to-point and publish-subscribe. These are also called message *domains*. Both of them allow for true asynchronous communication in which the message *consumer* does not need to be connected to the *producer* at the time when the message is sent and vice versa.

a) Point-to-point: This messaging model makes use of queues and is shown in Figure 2. Its main application is a request-response type of message exchange. Messages in this model are truly unique in the sense that once the *consumer* received and acknowledged the message it is removed from the queue. While there can be only a single *consumer*, messages can be put on the queue by multiple *producers*. This is the model used in the TSSN because message propagation throughout the mobile monitoring system is done from one web service to another.

b) Publish-subscribe: Whenever there is the need for multiple *consumers* to receive messages, a subscription model is useful. The *consumers* subscribe to a specific *topic* and receive messages as soon as they are published by one or multiple *producers* as shown in Figure 3. There exists no direct connection between *publishers* and *subscribers*.

Two types of subscriptions are possible in this model. The *subscriber* is either continuously connected to the *topic* and checks for new publications or a *durable subscription* is created in which the messages are kept within the topic while the *subscriber* is not connected. Upon reconnection messages are delivered to the *subscriber*.

In the TSSN the JMS *publish-subscribe* messaging model is currently not used since publications are handled using the web service standard *WS-Eventing* for SOAP messages. However, JMS *publish-subscribe* presents a flexible approach to scalability. Since switching between messaging models is only based on configuration parameters and not on a different implementation it is possible to use JMS *publish-subscribe* effectively in mobile monitoring environments as well.

B. Transportation Security SensorNet

The Transportation Security SensorNet [20] as shown in Figure 4 uses a Service Oriented Architecture approach for monitoring cargo in motion along trusted corridors. The complete system provides a web services based sensor management and event notification infrastructure that is built using open standards and specifications. Particular functionality within the system has been implemented in web services that provide interfaces according to their respective web service

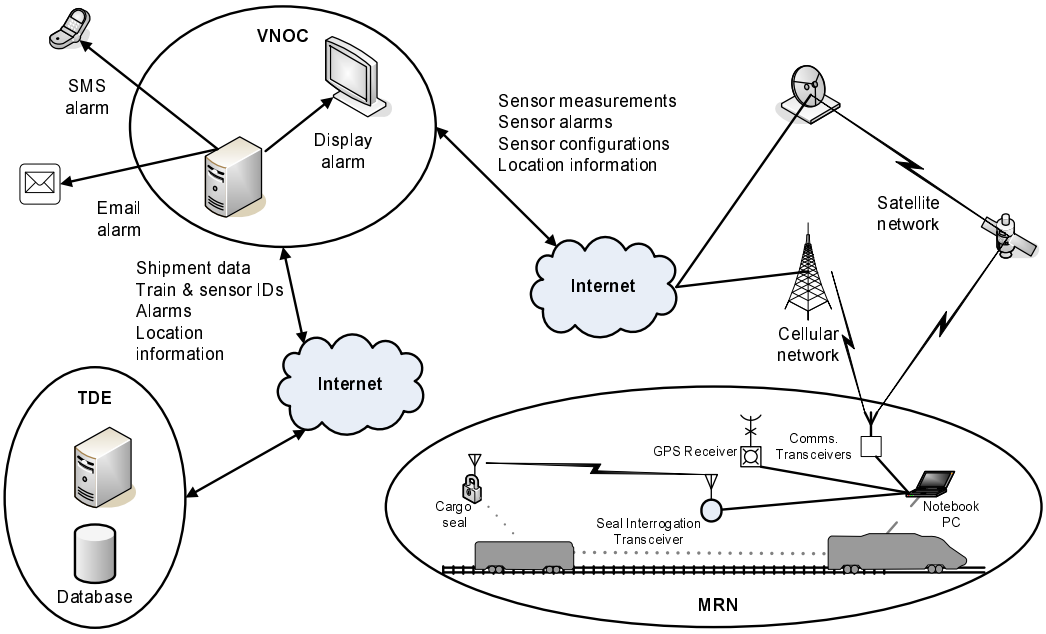


Fig. 4. TSSN physical architecture adapted from [25]

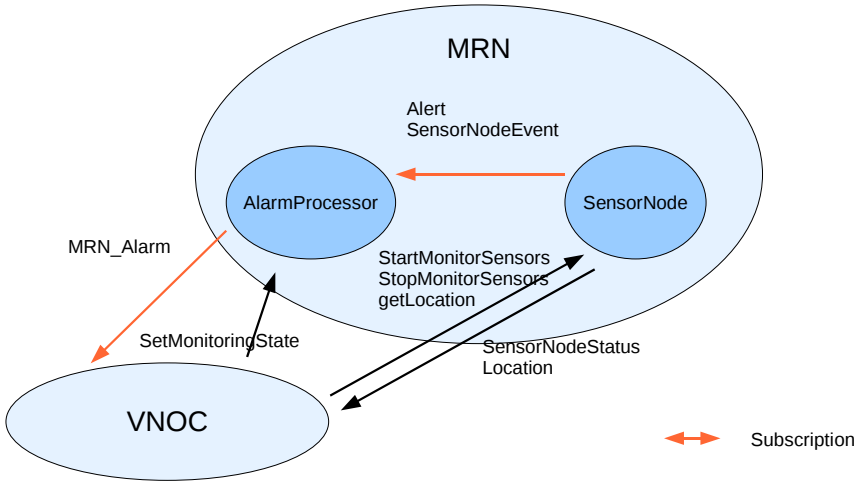


Fig. 5. Mobile Rail Network message overview from [20]

specifications. This web services based implementation allows for platform and programming language independence and offers compatibility and interoperability with other systems.

The TSSN represents the integration of SOA, Open Geospatial Consortium (OGC) specifications and sensor networks. Previous systems and research focused either on the combination of SOA and OGC specifications or on OGC standards and sensor networks. However, the TSSN shows that all three can be combined and that this combination provides capabilities to the transportation and other industries that have not existed before. In particular, the preeminent lack of performance in mobile sensor network environments has previously limited the application of web services because they have been perceived as too slow and producing a lot of overhead. The TSSN, as shown by the results in [20], demonstrates that with proper architecture and design the performance requirements of the targeted scenario can be satisfied.

Furthermore, unlike existing proprietary implementations the Transportation Security SensorNet allows sensor networks to be utilized in a standardized and open way through web services. Sensor networks and their particular communication models led to the implementation of asynchronous message transports in SOA and are supported by the TSSN.

Within the TSSN the Mobile Rail Network (MRN), which is shown in Figure 5, represents a train-mounted sensor network (sensors and sensor node) that monitors seals on cargo containers. The MRN is able to receive control messages such as when to start and stop monitoring. When an event is

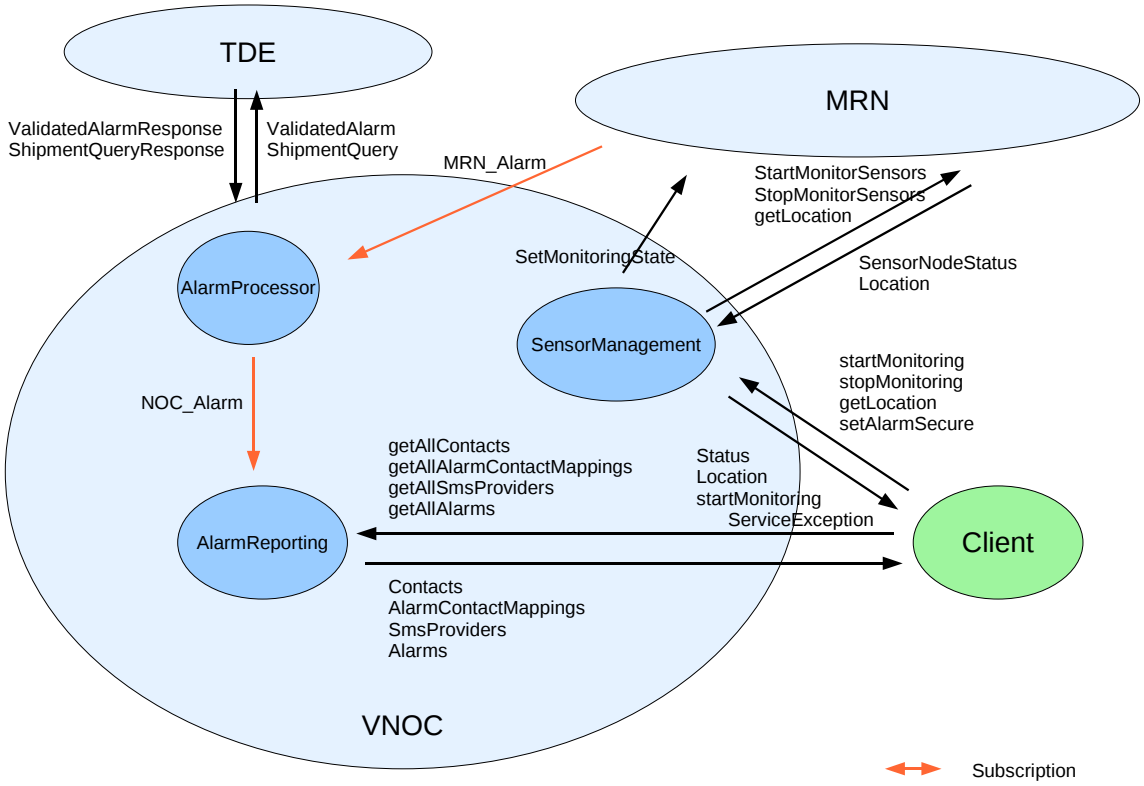


Fig. 6. Virtual Network Operation Center message overview from [20]

detected it is transmitted from a sensor (seal) to the sensor node where it is analyzed to determine whether or not to send out a notification to the VNOC (Figure 6). Sensor management and correlation of events with shipment and route information is then performed in the VNOC. According to specified mappings, people and organizations that subscribed to these notifications then receive emails and/or sms messages containing detailed information about the nature of the event.

In terms of the communication the critical link is between the Mobile Rail Network and the Virtual Network Operation Center because it cannot be guaranteed that there always exists a link and hence an asynchronous communication model had to be implemented. An approach that is able to deal with message queuing on both ends of the communication is the integration of the Java Message Service as the transport. The TSSN implementation fully supports asynchronous communication using the so-called *Enterprise Service Bus* queues in order to send and receive messages.

1) Axis2: The TSSN is based on the Apache Axis2 web services software stack. By default Axis2 uses request-response in a *synchronous* manner. This means that the client has to wait and is therefore *blocking* until it receives the response from the service. In certain scenarios, for instance when the service needs a large amount of processing time, the client can experience timeouts. Furthermore, in the TSSN where the MRN is only intermittently connected to the VNOC, *synchronous* communication is not feasible.

A better option is to make the communication between services *asynchronous*. This resolves timeout issues and deals with connections that are only temporary. The following aspects were taken into consideration when developing the *asynchronous* communication for this case:

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

b) Transport Level: Depending on the transport protocol that is being used, Axis2 supports the following approaches.

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

For asynchronous communication to work in the TSSN the two-way approach was modified as part of [20] through the Axis2 *client API* which provides the option of using a *separate listener*. This tells the service that it is supposed to use a new channel for the response. In order to correlate request and response messages Axis2 makes use of the *WS-Addressing* specification, in particular the *RelatesTo* field.

c) Service: The final piece of asynchronous communication is to make the service processing asynchronous as well.

```
<transportReceiver name="jms"
 class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory">
  <parameter name="java.naming.factory.initial">
   org.apache.activemq.jndi.ActiveMQInitialContextFactory
  </parameter>
  <parameter name="java.naming.provider.url">
   tcp://localhost:61616</parameter>
  <parameter name="transport.jms.ConnectionFactoryJNDIName">
   TopicConnectionFactory</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory">
  <parameter name="java.naming.factory.initial">
   org.apache.activemq.jndi.ActiveMQInitialContextFactory
  </parameter>
  <parameter name="java.naming.provider.url">
   tcp://localhost:61616</parameter>
  <parameter name="transport.jms.ConnectionFactoryJNDIName">
   QueueConnectionFactory</parameter>
 </parameter>

 <parameter name="default">
  <parameter name="java.naming.factory.initial">
   org.apache.activemq.jndi.ActiveMQInitialContextFactory
  </parameter>
  <parameter name="java.naming.provider.url">
   tcp://localhost:61616</parameter>
  <parameter name="transport.jms.ConnectionFactoryJNDIName">
   QueueConnectionFactory</parameter>
 </parameter>
</transportReceiver>
```

Fig. 7. JMS transport receiver configuration in axis2.xml

```
<transportSender name="jms"
 class="org.apache.axis2.transport.jms.JMSSender" />
```

Fig. 8. JMS transport sender configuration in axis2.xml

This is done by specifying so-called *asynchronous message receivers* in the services configuration in addition to the *synchronous* ones.

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

There exist various forms of transport protocols that are suitable for *asynchronous* communication. Axis2 by default supports HTTP, SMTP, and JMS as *asynchronous* transports but other transports can easily be defined and plugged in.

In order to allow for the TSSN to use JMS as a transport the following items were added to the Axis2 configuration by the authors. First, a so-called *transport receiver* for JMS as shown in Figure 7. This represents the receiving end of the communication and allows web services and clients to consume JMS messages by creating a JMS address for them. In particular, connection factories are set up for *queues* and *topics*. Second, a *transport sender* shown in Figure 8 allows JMS messages to be produced.

Axis2 by default sets up a *queue* for each of the services and uses the service name as the *queue* name. Since a service is not necessarily unique this name can be changed in the service configuration (Figure 9). For the Mobile Rail Network this naming consists of the node id which is used to represent a sensor network and the name of the service. For the Virtual Network Operation Center the name is made up of the host

```
<parameter name="transport.jms.ConnectionFactory">
 myQueueConnectionFactory</parameter>
<parameter name="transport.jms.Destination">
 TSSN_NODE/2222/MRN_SensorNode</parameter>
```

Fig. 9. JMS service queue name configuration in services.xml

```
<broker xmlns="http://activemq.apache.org/schema/core"
 brokerName="mrn2222"
 dataDirectory="\${activemq.base}/data">
 ...
</broker>
```

Fig. 10. ActiveMQ broker configuration in activemq.xml

```
<transportConnectors>
 <transportConnector name="openwire"
  uri="tcp://localhost:61616" />
 <transportConnector name="ssl"
  uri="ssl://localhost:61617"/>
</transportConnectors>
```

Fig. 11. ActiveMQ transport connector configuration in activemq.xml

on which the service is run and its name. This makes it possible to easily identify queues and avoid misconfiguration of ActiveMQ while also offering a naming scheme that is scalable.

2) ActiveMQ: Apache ActiveMQ is an open source implementation of the Java Message Service and is used by the TSSN for JMS messaging. A detailed introduction is given by Snyder et al. [21]. It is a JMS message broker mostly used in enterprise systems where high bandwidth connectivity is a given and throughput is most important. Note that the version used within the TSSN had to be modified by the authors because ActiveMQ could not work correctly without an existing and permanent Internet connection. However, being able to function without constant connectivity is essential in mobile monitoring environments. For example, the connection between the Virtual Network Operation Center and the Mobile Rail Networks in the TSSN may be intermittent.

The following sections explain the important components of ActiveMQ that are used in the TSSN and provide configuration details.

a) Broker: A *broker* is responsible for managing queues and topics. It receives message from *producers* which connect to it and delivers them to the according *consumers*. The configuration for a broker at a Mobile Rail Network is shown in Figure 10.

b) Transport Connectors: Brokers allow *producers* and *consumers* to use various protocols to connect to it. In ActiveMQ these connectivity entities are defined as *transport connectors*. The TSSN configures the services and clients use TCP in order to connect to the *broker* (Figure 11). Another use of the specified protocols is for inter-broker communication which is explained in detail later.

c) Network Connectors: Multiple *brokers* can form a *network of brokers* using *network connectors*. This is allows the use of *distributed queues* and is the setup that is used to connect Virtual Network Operation Center and Mobile Rail Networks. In order to be flexible the configuration of a so-called *network bridge* is initiated by the Mobile Rail Networks (Figure 12). Establishing a *duplex connection* then enables

```
<networkConnectors>
 <networkConnector
  name="MRN2222network"
  uri="static://(tcp://laredo.ittc.ku.edu:61616)?
   initialReconnectDelay=5000&amp;
   useExponentialBackOff=false"
  duplex="true"
  dynamicOnly="false"
  networkTTL="5"/>
</networkConnectors>
```

Fig. 12. ActiveMQ MRN network connector configuration in activemq.xml

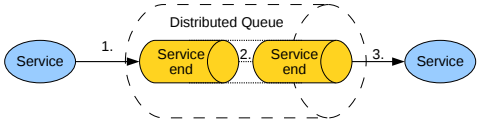


Fig. 13. One-way JMS message transmission

messages to be forwarded in both directions. The advantage here is that the VNOC does not need to be reconfigured every time a new MRN is set up.

ActiveMQ makes use of the OpenWire protocol [16] which is an optimized binary compressed format tailored to efficient management of JMS *queues* and *topics* as well as network connectivity. This is another advantage of using ActiveMQ since it makes sure that communication between brokers is bandwidth efficient which is essential for slow and unstable connections such as satellite links.

d) Distributed Queues: Connections from the VNOC to the MRN and vice versa are *point-to-point* which corresponds to *queues* in the Java Message Service. *Queues* can be distributed across several *brokers*. Whenever the *brokers* are connected to each other they exchange information about which *broker* has the *consumer* and the other *brokers* forward their queue messages to that *broker*. The two common types of message exchanges are explained in the following paragraphs.

Notification messages require only one-way communication as shown in Figure 13. Within the TSSN a web service acts as a *producer* and puts the notification onto the queue which corresponds to the web service it wants to notify. This is done by using the specified *transport connector* to connect to the local *broker* and deliver the message to it. The *broker* then puts the message on the *queue end* that it manages. Whenever the *broker* can contact the *queue end* with the consumer it forwards the message. The receiving web service uses a listener to detect when its *queue* at the broker contains new messages. It then uses its local *transport connector* to consume the notification.

Control messages that are sent by the VNOC to the MRN are good examples of two-way communications. As shown in Figure 14 in a request-response scenario the client creates a *temporary queue* at its local broker that only itself knows about. This is where the response message will be put. The request then follows the usual path from the local *transport connector* to the local *broker*, from the local *broker* to the *broker* with the specified *consumer* and then using the remote *transport connector* to the according web service. The JMS message that is transmitted contains a *ReplyTo* field with the

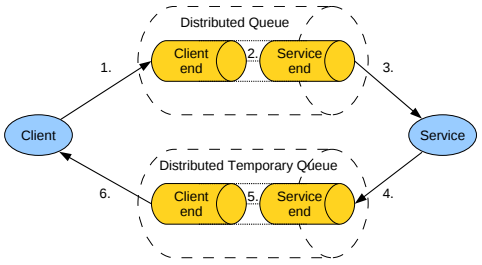


Fig. 14. Two-way JMS message transmission

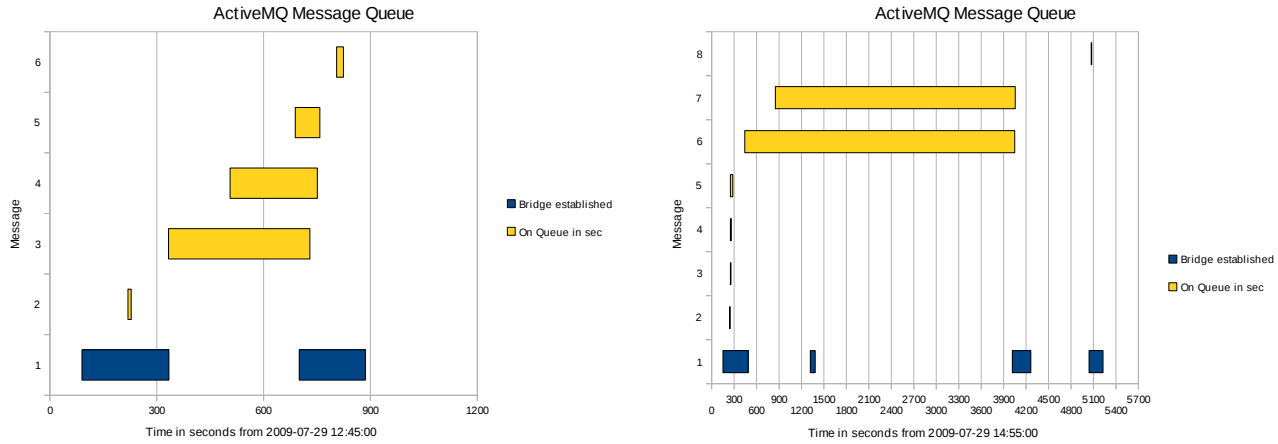
Fig. 15. Route for longhaul field trial, route starts at San Luis Potosi and ends approximately 210 miles down the track

temporary queue that is used for the response. The response is then sent to back using the web service's local *transport connector*, local *broker*, remote *broker* until it is consumed from the *temporary queue* by the original client.

In the TSSN all of the queue creation, message queuing and brokering is transparent to the web services. Whenever asynchronous communication using the Java Message Service is required the clients and web services simply use JMS addresses instead of the default HTTP ones which can be set in the their configuration files. This makes the solution very scalable and flexible since a *store-and-forward* mechanism does not need to be implemented in each web service but is provided by an ActiveMQ JMS message broker. This holds true not only for Service Oriented Architectures that make use of web services but is also applicable to other systems.

V. Results

The described system above has been successfully tested in a field trial. Here are results from two scenarios which were explored: a stationary scenario and a mobile scenario, carried out by mounting the equipment onto a train. Throughout the experiments communication between the VNOC and the MRN in the TSSN is using a satellite link, in this case Iridium at 2.4 kb/s. Both scenarios were performed as part of a longhaul trial in Mexico (see Figure 15).



(a) Initial test

(b) Follow-up tests

Fig. 16. ActiveMQ message queue for stationary scenario

A. Stationary

Figures 16a and 16b show tests results acquired from when the MRN was set up in a rail yard but not mounted on a train and not moving. The time each message spent on the distributed queues as well as when the MRN and the VNOC brokers had a JMS network bridge established and were therefore fully connected is displayed. It can be seen that while all messages were successfully transmitted the time a message was on a queue is dependent on the quality of the satellite connection. Only once the satellite connection is stable enough for ActiveMQ to establish a network bridge messages can be transmitted.

TABLE I
ELAPSED TIME FROM MRN TO VNOC DURING TRIAL IN SECONDS

	Minimum	Maximum	Mean	Median	Std. Dev
Link Down	31.29	1273.1	616.26	553.23	411.36
Link Up	5.85	40.53	11.62	6.02	10.77
Average Case	5.85	1273.1	481.90	430.97	441.86

TABLE II
COMPARISON OF ELAPSED TIME FROM MRN TO VNOC OF LONGHAUL TO SHORTHAUL TRIAL IN SECONDS

	Minimum	Maximum	Mean	Median	Std. Dev
Shorthaul	0.45	2.90	1.89	1.94	0.62
Longhaul	5.85	1273.1	481.90	430.97	441.86

B. Mobile

The more interesting scenario is to use TSSN as a mobile monitoring environment. For this purpose the MRN was deployed on a train and sensors attached to cargo containers. Figure 17 shows results of roughly the first hour (8:30-9:30am July 30th 2009) of the longhaul trial along the path shown in Figure 15. Due too a hardware problem after about 9:30am the system clock synchronization is significantly off and the remaining data, especially time measurements, cannot be analyzed. Therefore, only the time before the hardware issue occurred is shown in Figure 17. However, it is important to note that the TSSN kept operating correctly for more than 32 hours and ActiveMQ was successfully transmitting and receiving messages.

Figure 17 shows in detail when messages were put on a queue, when they were consumed and at which time the JMS network bridge, actual connectivity, was established. A comparison of times message required to be transmitted from the MRN to the VNOC is shown in Table I. Whenever connectivity, in ActiveMQ a so-called bridge, is established the actual message transmission takes about 11.6 seconds on average. This is in stark contrast to when the satellite link is down and needs to be established before sending out messages. In that case it took 616.2 seconds on average with the slowest message being received 1273.1 seconds or more than 21 minutes after it was sent.

The key characteristic here is the availability of the satellite link. The trial was performed in a mountainous environment where the satellite view was partially obstructed and hence the times measured may not be the same in a different geographic region. Looking at the average case of about 7 minutes per message transmission though the system is found to be in range of mobile monitoring environments.

A comparison of these results to a previous shorthaul trial described in [20] is shown in Table II. During the shorthaul the MRN was continuously connected to the VNOC using a GSM modem with a peak throughput of about 700 kb/s. Looking at the minimum times and assuming this as the best case scenario the satellite configuration is slower by a factor of about 13.

VI. CONCLUSION

As discussed in the previous sections the approach of using JMS in mobile monitoring environments works. We showed that Java Message Service technology can be utilized to provide drop-in connectivity between disparate and delay tolerant systems. Previously JMS was primarily used in always connected high bandwidth scenarios but its concepts and

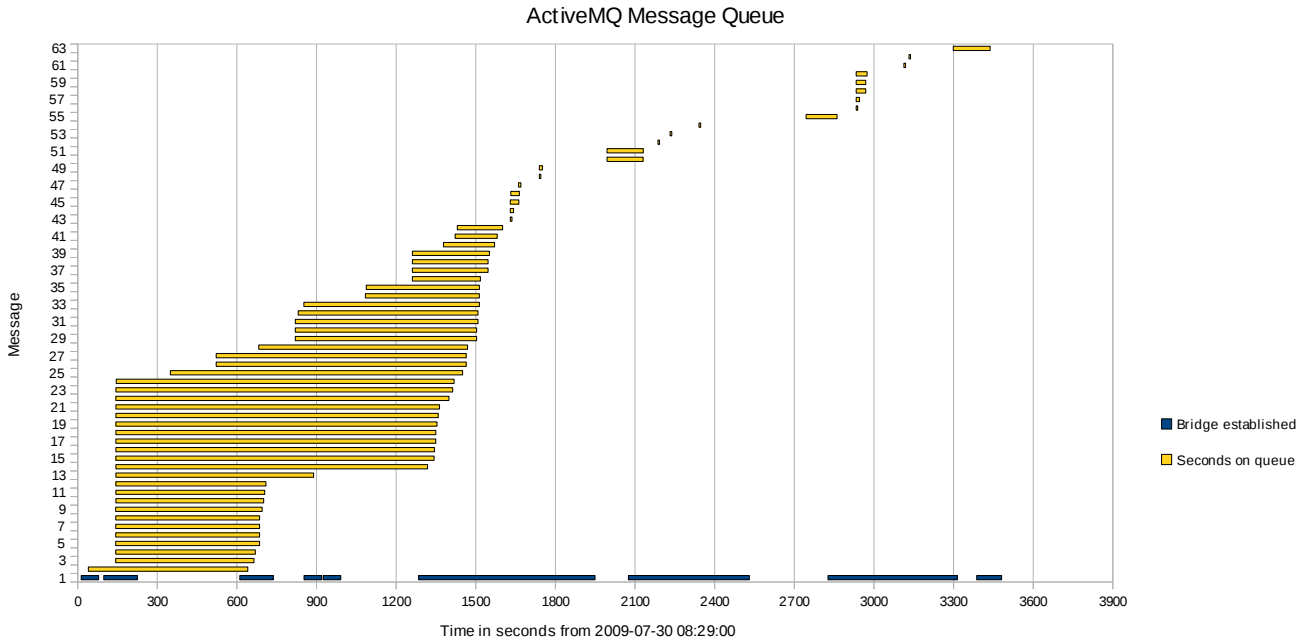


Fig. 17. ActiveMQ message queue for mobile scenario

techniques are useful in mobile monitoring environments as well.

JMS provides a transparent asynchronous communication model that is scalable and flexible since a *store-and-forward* mechanism does not need to be implemented in each component but is provided by a JMS message broker. This is done using distributed queues that are managed by network connectors as described above.

Since JMS allows the transport of all types of messages including SOAP messages, web service specifications were used here to provide features such as *end-to-end* message security and integrity. In terms of scalability JMS makes it possible to connect disparate systems with limited effort without having to implement *store-and-forward*, *resend* mechanisms and security again and again. Furthermore, JNDI and support for different messaging models enhance scalability for JMS based systems.

In this paper we presented a new and flexible approach to deal with challenges such as intermittent and low-bandwidth communication in mobile monitoring environments. This approach is to utilize the features that the Java Message Service provides to address the issues of asynchronous communication, message security and integrity, and scalability. We have shown that this is possible and presented an implementation of a mobile monitoring system called TSSN that successfully uses the approach.

ACKNOWLEDGMENT

This work was supported in part by Oak Ridge National Laboratory (ORNL) Award Number 4000043403. This material is also partially based upon work supported while V. S. Frost was serving at the National Science Foundation.

REFERENCES

[1] M. Musolesi, C. Mascolo, and S. Hailes, "Adapting asynchronous messaging middleware to ad hoc networking," in *MPAC '04: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*. New York, NY, USA: ACM, 2004, pp. 121–126.

[2] A. Vahdat and D. Becker, "Epidemic Routing for Partially-Connected Ad Hoc Networks," Duke University, Tech. Rep., 2000.

[3] E. Vollset, D. Ingham, and P. Ezhilchelvan, "JMS on Mobile Ad-Hoc Networks," in *In Personal Wireless Communications (PWC*. Springer-Verlag, 2003, pp. 40–52.

[4] D. Allard, "Development of a ground data messaging infrastructure for the mars science laboratory and beyond," in *Aerospace Conference, 2007 IEEE*, March 2007, pp. 1–8.

[5] L. Hongsheng, W. Yu, D. Yongzhong, and P. Zhongxiao, "Implementation of network-computing and nn based remote real-time oil well monitoring system," in *Neural Networks and Brain, 2005. ICNN&B '05. International Conference on*, vol. 3, Oct. 2005, pp. 1810–1814.

[6] Y. Lafon and N. Mitra, "SOAP version 1.2 part 0: Primer (second edition)," W3C, W3C Recommendation, Apr. 2007, http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[7] M. Gudgin, M. Hadley, and T. Rogers, "Web services addressing 1.0 - core," W3C, W3C Recommendation, May 2006, http://www.w3.org/TR/2006/REC-ws-addr-core-20060509.

[8] M. Gudgin, M. Gudgin, M. Hadley, T. Rogers, T. Rogers, and M. Hadley, "Web services addressing 1.0 - SOAP binding," W3C, W3C Recommendation, May 2006, http://www.w3.org/TR/2006/REC-ws-addr-soap-20060509.

[9] K. Lawrence, C. Kaler, A. Nadalin, R. Monzillo, and P. Hallam-Baker, "Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)," OASIS, OASIS Standard, Feb. 2006, http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[10] P. Fremantle, S. Patil, D. Davis, A. Karmarkar, G. Pilz, S. Winkler, and mit Yalinalp, "Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.1," OASIS, OASIS Standard, Jun. 2007, http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf.

[11] N. Mendelsohn, H. Ruellan, M. Gudgin, and M. Nottingham, "XML-binary optimized packaging," W3C, W3C Recommendation, Jan. 2005, http://www.w3.org/TR/2005/REC-xop10-20050125/.

[12] E. Levinson, "The MIME Multipart/Related Content-type," RFC 2387 (Proposed Standard), Aug. 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2387.txt

[13] M. Nottingham, H. Ruellan, N. Mendelsohn, and M. Gudgin, "SOAP message transmission optimization mechanism," W3C, W3C Recommendation, Jan. 2005, http://www.w3.org/TR/2005/REC-soap12-mtom-20050125/.
[14] M. Gudgin, Y. Lafon, and A. Karmarkar, "Resource representation SOAP header block," W3C, W3C Recommendation, Jan. 2005, http://www.w3.org/TR/2005/REC-soap12-rep-20050125/.
[15] "Advanced Message Queuing Protocol (AMQP) version 0-10 Specification," Specification, 2009, http://www.amqp.org.
[16] "OpenWire Version 2 Specification," Apache ActiveMQ, Specification, 2009, http://activemq.apache.org/openwire-version-2-specification.html.
[17] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.
[18] J. Strachan, "Stomp Protocol Specification, Version 1.0," FuseSource, Specification, 2005, http://stomp.codehaus.org/Protocol.
[19] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Core," RFC 3920 (Proposed Standard), Internet Engineering Task Force, Oct. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3920.txt
[20] M. Kuehnhausen, "Service Oriented Architecture for Monitoring Cargo in Motion Along Trusted Corridors," Master's thesis, University of Kansas, Jul. 2009.
[21] B. Snyder, D. Bosanac, and R. Davies, *ActiveMQ in Action*. Manning Publications, 2009.
[22] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, "Java(TM) Message Service Specification," Sun Microsystems, Specification, 2002, http://java.sun.com/products/jms/.
[23] P. Easton, B. Mehta, and R. Merrick, "SOAP over java message service 1.0," W3C, W3C Working Draft, Jul. 2008, http://www.w3.org/TR/2008/WD-soapjms-20080723.
[24] "Java Naming and Directory Interface Application Programming Interface (JNDI API)," Sun Microsystems, Specification, 1999, http://java.sun.com/products/jndi/.
[25] D. T. Fokum, V. S. Frost, D. DePardo, M. Kuehnhausen, A. N. Oguna, L. S. Searl, E. Komp, M. Zeets, J. B. Evans, and G. J. Minden, "Experiences from a Transportation Security Sensor Network Field Trial," Information Telecommunication and Technology Center, University of Kansas, Lawrence, KS, Tech. Rep. ITTC-FY2009-TR-41420-11, June 2009.