UNIVERSITY OF CALIFORNIA

Los Angeles

ESP Framework:
A Middle-ware Architecture For
Heterogenous Sensor Networks

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Electrical Engineering

by

Sasank Kambham Reddy

2006

© Copyright by

Sasank Kambham Reddy

2006

The thesis of Sasank Kambham Reddy is approved.

Mark Hansen

William Kaiser

Mani Srivastava, Committee Chair

University of California, Los Angeles

2006

TABLE OF CONTENTS

1 Introduction **1**

2 Previous Work **4**

2.1 Schema Languages 4

2.1.1 SensorML 4

2.1.2 TinyML 5

2.1.3 IEEE 1451 6

2.2 Sensor Network Middleware Architectures 7

2.2.1 Atlantis Framework 7

2.2.2 TASK 7

2.2.3 Others 7

2.3 Web Interfaces 8

2.4 Wide Area Network Architectures 9

3 System Overview **10**

4 ESPml Schema **13**

4.1 Purpose and Goals 13

4.2 Components 13

4.2.1 System 14

4.2.2 Field 14

4.2.3 Platform 15

4.2.4 Sensor 16

4.3 Descriptions 17

4.3.1 Location 17

4.3.2 Functions 18

4.3.3 Mobility 22

4.3.4 Additional Ideas 22

4.4 Design Decisions 23

5 Detailed System Design 25

5.1 Registry 25

5.2 System 27

5.3 Security Components 28

5.3.1 Motivation 28

5.3.2 Encryption Scheme 29

6 Enhancement Prototypes 32

6.1 Decentralized Authentication 32

6.2 Network Functions 34

6.3 Data Logging Services 37

7 Evaluation 40

7.1 Example Systems 40

7.2 Client Application 43

7.3 Observations 44

7.4 Performance 44
7.4.1 Cryptography Components 45
7.4.2 Registry Profiling 46
7.5 Wide Area Middleware Architecture Comparison 50
7.5.1 Interoperability 50
7.5.2 Service Discovery 51
7.5.3 Intelligent Networks 52
8 Future Work 54
8.1 Standardization 54
8.2 Mobility and Availability 55
8.3 Scalability 56
8.4 Client Applications and Systems 57
9 Conclusion 58
References 59

LIST OF FIGURES

3.1 ESP framework operations overview. 10

4.1 Model view of the system component in the ESPml schema. . . . 14

4.2 Model view of the field component in the ESPml schema. 15

4.3 Model view of the platform component in the ESPml schema. . . 15

4.4 Model view of the sensor component in the ESPml schema. 16

4.5 Model view of the function element in the ESPml schema. 17

6.1 A typical OpenID interaction in the ESP framework. 33

6.2 AON PEP Showing URI Re-Direction. 36

6.3 ESP system interaction with SensorBase. 38

7.1 Screen-shot of the Google Maps client and an actuated camera. . 42

7.2 Screen-shot of the Google Maps client and a small sensor network node's photodiode sensor. 42

7.3 Screen-shot of the Google Maps client and the weather stations in the Los Angeles area. 43

7.4 register Latency. 46

7.5 listSystems Latency. 47

7.6 listSystems Simultaneous Query Latency. 49

LIST OF TABLES

5.1 SQL Table Definitions for Registry Entries 26

7.1 Timings for Cryptographic Algorithms 45

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my adviser, Professor Mani Srivastava. Thank you for guiding me as I explored this problem space. I also, would like to thank Professor Deborah Estrin, Professor William Kaiser, and Professor Mark Hansen for their input and feedback as I progressed through my research work. Finally, credit goes to Thomas Schmid since our joint work on the project resulted in the first prototype of the ESP system as a whole. Chapter 2 and 4 are joint work with Thomas, and he helped revise Chapter 5 and 8 as well.

ABSTRACT OF THE THESIS

ESP Framework:

A Middle-ware Architecture For

Heterogenous Sensor Networks

by

Sasank Kambham Reddy

Master of Science in Electrical Engineering

University of California, Los Angeles, 2006

Professor Mani Srivastava, Chair

Sensor networks are quickly becoming a flexible, inexpensive, and reliable platform to provide solutions for a wide variety of applications in real-world settings. For instance, sensor systems have been used for medical monitoring, detection and classification for defense purposes, and to perform environmental monitoring. The increase in the proliferation of sensor networks has paralleled the use of more heterogeneous systems in deployments.

This thesis presents a framework that enables interoperability between varied sensor systems and also details some of the fundamental capabilities needed in such an environment. The ESP framework aims to provide a standard method to manage, query, and interact with sensor network systems. We provide a method to describe sensor systems using ESPml, a XML schema, as the modeling language. The fundamental goal of the schema is to be able to describe sensors in a simple, compact manner while still having the ability to represent essential details such as the general setup, the type of data that can be provided, and the commands that are available. In addition, we present a web services based frame-

work that provides the basic capabilities for locating, managing, and querying the sensor systems. Finally, challenges such as authentication schemes, logging services, context integrity, and client interfaces are also explored.

CHAPTER 1

Introduction

Embedded sensor networks have started becoming useful in a wide variety of environments, and there have been numerous research efforts that show how these sensor networks can be applied to such areas ranging from household situations to scientific pursuits. For instance, sensor networks have been used successfully in improving agriculture procedures in vineyards [BBB04], providing better insight into the learning process in educational institutions [MP01], and detection and classification of objects in military settings [DS02].

As the potential for sensor networks in various fields have been realized, software frameworks have been developed to make deployments relatively easy to configure, maintain, and use. For instance, Tiny Application Sensor Kit (TASK) [BGH05] focuses on providing a sensor network system in a box, where installation is fairly easy, field tools can be used for remote management and health monitoring, and client tools are available for getting specific information from the system. Mote-View [Tur05] addresses the issue of sensor network monitoring in a fine grained fashion by providing a suite of tools to visualize the network health of a particular node in a system by providing data related to bandwidth, congestion, and throughput. But most of these architectures are focused on a single sensor network and are often coupled with the underlying software, such as the operating system, that is running on the implementation.

Since sensor networks have a large application space where they can be useful

and setting up and monitoring a sensor system is becoming easier, the proliferation of sensor networks has increased immensely. But due to the application space variance, there is tremendous heterogeneity in the logic for interfacing and collecting data on these systems. The ESP framework provides the middle-ware architecture to bridge the gap between heterogeneous sensor systems and enables a standard way to manage, query, and interact with the various sensor network setups.

The ESP framework consists of a sensor network description language, in the form of an XML schema (ESPml), that is used to fully describe a sensor network in terms of location, setup, type of data that is provided, and any commands that can be enacted on the system. These details can be provided for various granularities ranging from the system as a whole to a specific sensor that is on a particular platform. Also, the framework defines an architecture that enables heterogeneous sensor networks to be registered, queried and interacted with through a common interface available using a web services. This system architecture, along with the use of ESPml, is demonstrated by registering a variety of sensors and interacting with them through a Google Map end user interface.

The rest of the thesis is organized as follows. We discuss previous work including schema languages for describing various attributes in sensor networks that are already available, middle-ware architectures that exist in the sensor network realm, and current web interfaces for sensor network systems in Chapter 2. A general overview of the system as a whole is provided in Chapter 3. Chapter 4 contains details about the ESPml XML schema. A detailed explanation of the ESP framework registry and system architecture are provided in Chapter 5. Work related to using and evaluating the ESP framework is in Chapter 6. Chapter 7 contains concepts explored to enhance the ESP framework including the use of

a decentralized identity system for authentication, the idea of providing network attested information with the introduction of a mediator, and enabling advanced data logging services. Chapter 8 and 9 contain guidelines for future work and the conclusion for the thesis.

CHAPTER 2

Previous Work

2.1 Schema Languages

2.1.1 SensorML

The Open Geospatial Consortium (OpenGIS) is developing a standard called SensorML [Bot] which intends to describe a sensor system in great detail. It uses GML [CDL], a XML standard also from OpenGIS, to detail the geospatial properties of sensors. For example, it includes the relative positions of sensors if a system has multiple sensors as well as detailed description of the sensors themselves. It is possible to describe what physical entities a sensor measures, what accuracy the sensor can achieve, who the manufacturer is, etc.

Additionally, SensorML includes basic functionalities to describe processes which handle and transform sensor data. For example, if one has a small weather station which reports the current wind speed and the air temperature, one can create a process which uses these two values and outputs the wind-chill factor by applying the following formula

$$T_c = 35.74 + 0.6215 \cdot T - 35.75 \cdot V^{0.16} + 0.4275 \cdot T \cdot V^{0.16}$$

where T_c is the wind-chill temperature, T the current air temperature and V is the wind speed.

SensorML is part of the “Sensor Web Enablement and OpenGIS SensorWeb” area of interest of the OpenGIS consortium. This entity intends to develop “interoperability interfaces and meta-data encodings that enable real time integration of heterogeneous sensor webs”. Other standards under development in this area are, as described on the OpenGIS website:

- Observations & Measurements - Information model and encodings for observations and measurements.
- Sensor Observation Service - Service for managing deployed sensors and retrieving sensor data.
- Sensor Planning Service - Service to assist in “collecting feasibility plans” and to process collection requests for a sensor or group of sensors.
- Web Notification Service - Service to manage dialogue between a client and Web service(s) for long duration asynchronous processes.

2.1.2 TinyML

SensorML provides a sensor-centric approach for sensor coordination and description. It is also pretty heavy weight and very complex. TinyML [OK] addresses these shortcomings and implements a lightweight version but still following the basic ideas of SensorML. Furthermore, TinyML is tailored to embedded sensor networks. The fundamental elements of TinyML are the sensor, the platform, and the sensor field. A sensor describes a specific sensor and its properties. A platform represents a physical system with a processor, an energy source and a radio device. Additionally, a platform has a collection of sensors. The sensors measure basic physical phenomena, like temperature, air pressure etc. At the

larger scale, a sensor field is a collection of platforms and represents a sensor network.

TinyML has capabilities to define virtual devices. These virtual devices can be generated in an ad-hoc fashion and group together different platforms or sensors. This allows the ability to easily query a subset of platforms and sensors by taking advantage of virtual devices. Each element in TinyML can have a query or set a flag to indicate a response / actuation of that component. This is a very simple mechanism to interact with a sensor network.

The advantages of TinyML are that it gives a universal interface to interact with a sensor network and it is very lightweight. Unfortunately, TinyML is centered too much on sensor networks and can not easily be adapted to other sensor systems. Furthermore, it does not have any notion of mobility, i.e., for a sensor network where nodes move around, and there is no rigorous implementation available.

2.1.3 IEEE 1451

The Institute of Electrical and Electronics Engineers (IEEE) developed a standard called IEEE 1451 [Lee00], a standard to easily network transducers. The first part, IEEE 1451.2 was already adopted in 1997, and was closely followed by IEEE 1451.1 in 1999. The standard defines ways to find out what transducers are on a network. All of this is done at a very low level, i.e., this would be used on a platform where the micro-controller wants to know what sensors are attached to its input ports and buses.

2.2 Sensor Network Middleware Architectures

2.2.1 Atlantis Framework

The Atlantis Framework [Arn05] is based on TinyML but addresses several of its shortcomings. The basic elements are the same, i.e., it can describe fields, platforms, and sensors. Additionally, the Atlantis Framework adds data handling abstractions, and a query field for more detailed queries. It makes further improvements by defining a field task object which can handle asynchronous data retrieval. For this purpose, it adds an additional data broker which handles the tasks, and specific broker behaviors to describe how to handle the task itself. As a nice roundup, the Atlantis Framework adds data filters and event subscription possibilities. On the downside, there is not a standard way to manage the sensor systems since a registry does not exist.

2.2.2 TASK

The "Tiny Application Sensor Kit" (TASK) [BGH05] was designed for use by end-users with minimal sensor network knowledge. TASK uses TinyDB as a back-end running on nodes and is thus tailored towards sensor networks running on Mica2 mote type systems. Additionally, it can handle only one deployment at a time and needs to be reconfigured to be used for a different deployment, i.e., it can not handle multiple deployments simultaneously.

2.2.3 Others

F. C. Delicato et al describe in [DPP03] a general schema to use Web Service technology in a sensor network, i.e., their main physical components are sensor and sink nodes. The paper fails to give an implementation of the schema and

does not go beyond general web service concepts.

In [TP05] D. Trossen and D. Pavel describe a middleware application for smart phones. Their architecture consists of several different components. They have entities for event delivery, acquisition, query resolution, aggregation, access control, storage and even registration and availability. The framework is mainly tailored towards cellular systems and they claim to have a test-bed running, though an implementation was not available at the time of writing this report.

2.3 Web Interfaces

The overall architecture of the ESP framework revolves around using web services as a platform construct. By using web services, the framework provides support for interoperable machine to machine interaction. Specifically, we take advantage of the communication protocol and service description areas of the web services protocol. The interface to the registry and also to the actual systems is through the Simple Object Access Protocol (SOAP) and the actual interfaces are described using the Web Services Description Language (WSDL) [CDK02]. ESPml is used to convey the specific intent of the various calls by providing data necessary to perform the actions defined by the interface calls.

Current use of web services is very limited in the sensor network realm. Instead of providing access to the actual nodes that are on the system, in terms of sensors, most implementations use web services as a method to access data that is stored on a database management unit or an aggregation unit [BGH05],[BMM06]. The queries to these nodes might be complex and involve the query being disseminated throughout the sensor network, but the actual web service interface is still typically confined to interacting with a small set of nodes that are at a

higher data granularity level than the actual sensors in the system.

Other uses of web services in the sensor network area include services that enable sensor data to be published to a particular source that then provides querying services on the data that is collected [DRK06],[SNL06]. Also, there are numerous applications that provide web interfaces not based on web services but instead custom interface methods. Overall, based on the current landscape of sensor networks and how web services are being used, there is definitely a gap that can be filled by a unifying system such as the ESP framework that provides multiple services for a sensor network platform.

2.4 Wide Area Network Architectures

There are a number of projects that share the same vision as the ESP framework in terms of interconnecting devices and services together to support searching, dissemination, and sharing capabilities with wide area networks in mind. For instance, the Jini architecture exploits mobile byte-code and proxies for remote method invocation with a registry service to serve as a lookup component [Wal99]. The Ninja project focuses on providing a method for a large set of heterogeneous devices to access services running on clusters of workstations using network intermediaries, service discovery service, and remote message invocations [GWB01]. Silph uses Jini as a service discovery module and adds an abstraction layer to manage data-centric queries that interact with sensor modules and applications [CMY02]. The ESP framework aims for the same type of functionality as these architectures but uses different architecture components or has additional feature requirements for some of the modules involved. We will discuss these wide area network architectures in relation to the ESP framework in more detail as part of the evaluation section.

CHAPTER 3

System Overview

The overall architecture for the ESP Framework involves three main components. First, there is the actual system, which represents the sensor network that is being registered. The registry is the second element in the architecture and is the location where sensor systems are registered and where the user can actually query for systems. Finally, there is the client that accesses both the registry and the individual systems. Figure 3.1 contains all three entities in the architecture and typical operations that can occur during normal interaction.

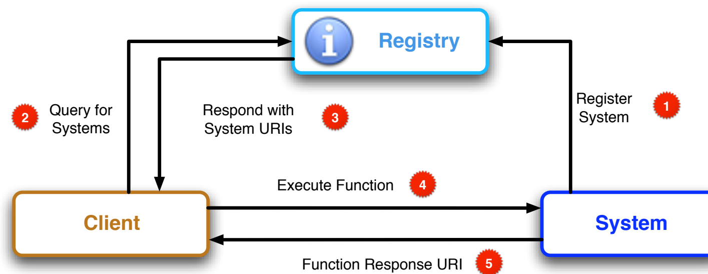


Figure 3.1: ESP framework operations overview.

In order to illustrate how the framework works in general, usage scenarios will be described below.

1. **Register System**

   During the registration process for a sensor, the first activity that needs to be performed is to create a ESPml document that describes the sensor

network system as a whole. This can be done manually by an application developer or done automatically by analyzing the sensor system and generating the appropriate XML to represent the system. In addition, one must define all the functions that are described as being available in the ESPml document in the actual system. Finally, the ESPml document must be sent to the registry. The process of sending information to the registry occurs through a web services interface where a method for registering the system is exposed. At the actual registry, the system ESPml is populated into a database. More details about the database structure will be given in Chapter 5.

2. **Query for Systems**

   In order to query for systems, the client sends a request to the registry with an area of interest. The area is described as a polygon and the actual coordinates are encoded as a ESPml document. Again, there is a method exposed on the registry through web services that enables communication between the client and the registry to occur.

3. **Respond to Query**

   After a request for sensor systems is sent, the registry responds to the client by sending back uniform resource identifiers (URIs) for the corresponding systems in the polygon area submitted. The actual response is an ESPml document that contains all sensors, platforms, and fields within the polygon. The URIs are addresses to the system's web service interfaces. Furthermore, descriptions for the different aspects of the system and the functions provided by these entities are detailed in the ESPml document.

4. **Function Execution Request**

Once the client gets back all the systems in a certain area of interest, a possible next step is to execute a method on the systems. In this case, a light weight version of the ESPml document, which contains only the URIs and the function to execute with any required variables filled in, is sent to the specific system. The systems have a web service exposed function to take in generic queries that are defined by the ESPml document.

5. **Function Response**

   The final step in a typical interaction will be the actual system responding to the function execution request. This occurs by the system returning a ESPml document that contains the output tag for the function. The output tag contains the name of the function, the return type for the result, and a URI that points to the result.

Overall, the ESP framework is simple, yet robust enough to handle a wide variety of heterogeneous systems. Furthermore, the use of ESPml and web services makes interoperable communication possible in a standard fashion. More details about each of the different parts of the architecture will be given in Chapter 5.

CHAPTER 4

ESPml Schema

4.1 Purpose and Goals

The purpose of the ESPml schema is to define a standard language protocol with which the different entities in the framework can communicate and also to have a unifying grammar that systems can describe their abilities. ESPml is held very generic such that it can describe a wide variety of systems including databases, sensor networks, weather stations or even web cameras. The goal was to design the ESPml language to be based on a minimum number of powerful abstractions that can be easily adopted to describe any type of sensor deployment. Furthermore, the language should be kept fairly light-weight so that only necessary information for describing sensor systems is represented.

4.2 Components

The different components in ESPml describe the different parts of a system. The main entity is the system element. The system element contains one or more fields, which are a collection of one or more platforms. And finally, a platform is a collection of sensors, the smallest entity possible. We will now go into details of each of the elements.

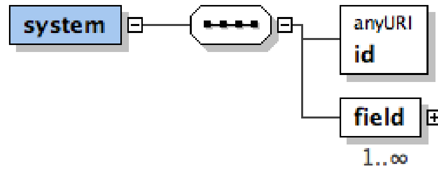


Figure 4.1: Model view of the system component in the ESPml schema.

4.2.1 System

Figure 4.1 depicts the model view of the top-level element of the schema, the system component. The system consists of an id element of type anyURI and one or more field elements. The id element points to the address where the system's web services can be reached, i.e., where one can execute remote functions through web service calls. The field elements represent actual sensor networks. For example, if a system has multiple physical deployments, and there is one common interface for all the fields in the system, then the different deployments can be represented as different fields. We will describe the field element in the next section.

4.2.2 Field

Illustrated in Figure 4.2 is the field element of ESPml. The field consists of an id of type integer. This id should be unique within the system and is used to identify the field. The location can be a point or a polygon which describes the physical location of the field. We will describe the location element in more detail in Section 4.3.1. Each field has a description element of type string which describes the type and purpose of the field in a verbal manner. A field can also have zero or more functions associated with it. In general, these functions should act on the whole field, for example, they could return the average value of a sensor reading

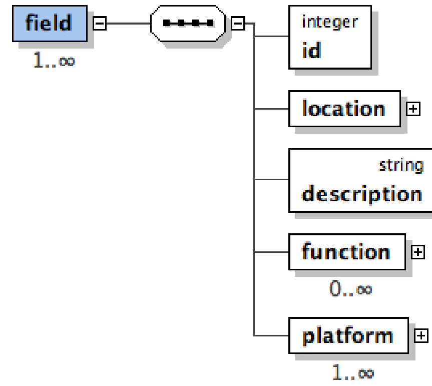


Figure 4.2: Model view of the field component in the ESPml schema.

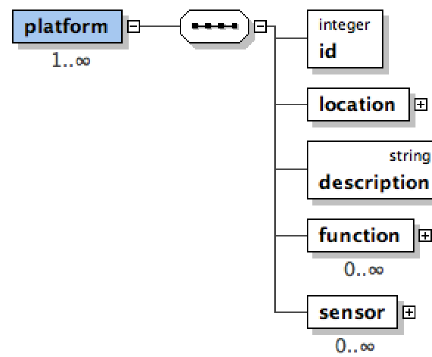


Figure 4.3: Model view of the platform component in the ESPml schema.

over the whole field, etc. The last element in a field is one or more platforms.

4.2.3 Platform

Figure 4.3 shows the model view of the platform element. A platform element represents a physical entity with a micro-controller, some communication device and a possible energy source. The integer id in a platform has to be unique within the field. Thus, the field id and the platform id can uniquely identify any platform in the system. Each platform has a location, a string description, and

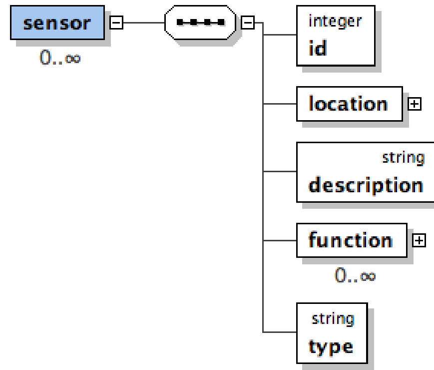


Figure 4.4: Model view of the sensor component in the ESPml schema.

functions which can be executed. These functions can affect the platform for actuation purposes, retrieve data values such as the exact location, the local time at the platform, or to move the platform around. Each platform also contains zero or more sensors.

4.2.4 Sensor

Probably the most important component of the ESPml schema is the sensor. Figure 4.4 depicts the model view. The sensor element represents a sensor on a platform, such as a thermistor, light-sensor, camera, etc. The id must be unique on the platform. Similar to the other elements, the sensor element has a location, description, and function element. One additional important element is the type element which describes the type of sensor. This type is not standardized right now, and can thus be defined by the system developer. This will change in the future since we intend to provide a standard for describing sensor types.

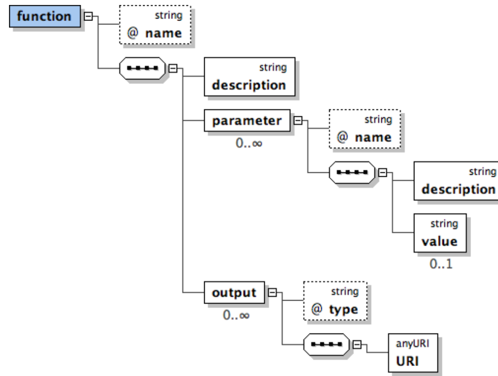


Figure 4.5: Model view of the function element in the ESPml schema.

4.3 Descriptions

The next four sections will explain the different additional fields for the field, platform and sensor element in more detail.

4.3.1 Location

The location element describes, as the name suggests, the location of the enclosing element. The location is described either by a point or a polygon element. The point element contains a simple string of the format "*latitude*, *longitude*, *altitude*" and is derived from the GML standard. The polygon is composed of a multi line string, where each line represents a corner of the polygon and is also derived from the GML standard. The format for each line is the same as the point element. Additionally, the polygon has to be closed, i.e., the first and last point have to coincide.

4.3.2 Functions

The function element is a little bit more complicated than the other elements. The location of the function within the XML determines to which component it is attributed to. For example, a function in the field element should affect the whole field, whereas a function in a platform element should affect only that platform.

Figure 4.5 depicts its model view. Each function has a name attribute. The name attribute should be reflective of the function's purpose. For example, a function which returns a sensor's data, should be called `getSensorData`, a function which sets the sampling rate of a sensor, should be called `setSamplingRate`, etc. We do not enforce a naming schema for functions. Though we encourage the user to choose meaningful names. In a later version we will develop a standard naming schema for commonly used functions.

Each function element contains a description which details what the function actually does. Additionally, a function element contains zero or more parameter elements and zero or more output elements. What role they play will be explained in the following example where we show how one defines, executes and then collects the data from a function.

- **Declaring a Function**

  The function declaration is done in the system's XML schema file which is sent to the registry. Each field, platform, and sensor should define the functions it supports, including each function name attribute, the description element, and the parameter elements with name and description, which need to be provided to execute the function later on. The following code example is an excerpt of a ESPml XML file a system sends to the registry,

and defines a sensor with two functions. The first function gets the current sensor value and has no parameter. The second function gets the average value over a certain number of recorded data. Thus, it needs a parameter which tells the function how many elements it should calculate the average.

```
...
<sensor>
  <id>1</id>
  <location>
    <point>
      <pos>34.0682,−118.44,0.00</pos>
    </point>
  </location>
  <description>PhotoSensor</description>
  <function name=”getCurrentValue”>
    <description>Get the current value for this sensor.
    </description>
  </function>
  <function name=”getAverageValue”>
    <description>Gets the average for this sensor.
    </description>
    <parameter name=”numberElements”>
      <description>The number of elements for which the
          average should be calculated.
      </description>
    </parameter>
  </function>
  <type>Light</type>
</sensor>
```

```
24 ...
```

- **Executing a Function**

  If a client wants to execute a function, then it sends a ESPml XML file to the system's URI. In the XML file, the user will add the function element that needs to be executed on a field, platform, or sensor. It is also possible to execute multiple functions in one request. The following listing shows parts of a ESPml XML file which a client sends to a system to execute functions. This excerpt will execute `getAverageValue` on platform 1, sensor 1 with a parameter value of 10, and it will get the current value of platform 2's sensor 1.

```
1  ...
2 <platform>
3    <id>1</id>
4    <sensor>
5       <id>1</id>
6       <function name="getAverageValue">
7          <parameter name="numberElements">
8             <value>10</value>
9          </parameter>
10      </function>
11   </sensor>
12 <platform>
13 <platform>
14   <id>2</id>
15   <sensor>
16      <id>1</id>
17      <function name="getCurrentValue">
```

```
18      </function>
19    </sensor>
20 <platform>
21 ...
```

- **Collecting Function Output**

  The system will respond with another ESPml XML file which will be filled with the output elements of the received function calls. The output elements consist of a type attribute and a URI field. The type element describes the mime-type of the file the URI points to. The following listing is an example response to the function executed from the last example.

```
1 ...
2 <platform>
3   <id>1</id>
4   <sensor>
5     <id>1</id>
6     <function name="getAverageValue">
7       <output type="text/comma-separated-values">
8         <URI>http://foobar.org:8080/sensor_getAverageValue
            /852760e7177ac9eeebcc16621ec2e83c
9         </URI>
10      </output>
11    </function>
12  </sensor>
13 <platform>
14 <platform>
15   <id>2</id>
16   <sensor>
```

```
17      <id>1</id>
18      <function name="getCurrentValue">
19        <output type="text/comma-seperated-values">
20          <URI>http://foobar.org:8080/sensor_getCurrentValue
              /76344f22bdbc47211407b085bf5da147
21          </URI>
22        </output>
23      </function>
24    </sensor>
25 <platform>
26 ...
```

4.3.3 Mobility

The location element has an optional element "mobile". This indicates that the enclosing element is mobile and can move around. This is an indicator that the location might not be the real location where the element is right now and that one has to take some precaution. In the future, we will require a special function to get the enclosing elements exact current location if the mobile element is defined.

4.3.4 Additional Ideas

The current ESPml schema is not complete and is open for further development. It has all the functionality to get a simple example system up and running but it will be extended in the future. Some points which will be addressed are:

- standards for function names and types

- confidentiality, privacy, and authentication
- improved location element for different coordinate systems
- scalability
- etc.

We have a closer look at these extensions in Chapter 8.

4.4 Design Decisions

One aspect of actually creating the ESPml schema is to point out the main design criteria used to make decisions on what should be included in the specification and what should not be included. First, there are four components involved in the specification, and they include systems, fields, platforms, and sensors. The best way to analyze why these specific components were used is to look at the design from a bottom-up traversal. The fundamental element that is involved in any time of sensing system are the actual sensors. These systems need to be described in great detail since that is the essential property that is being described. Typically, we think sensors will be housed in some type of higher-order platform. This platform needs to be able to handle several sensors and can mitigate functionality and management of the various sensors. Thus, we introduced the the idea of a platform element. In addition, by having a platform element, we have the ability to perform aggregation functions based on several sensors. In a typical deployment, there are several of these platforms that are deployed. Typically, these platforms are organized either by their objective or by their high-level capabilities whether it is based on communication, computation, or sensing. The concept of a field covers this organization. Several platforms

can be organized in unique fields. Finally, these fields make up a overall system. Instead of this multi-level organization, we could have had just a flat design but having multiple levels lets us more carefully describe a deployment and also helps in having the ability to perform aggregate functionality that could be based on a set of system elements.

Another component of the ESPml schema that needs to be justified is what properties actually are first-order. For instance, we included such elements as types, functions, mobility, and location as first order grammar elements in our language. The guideline for this choice was based on the idea of static and dynamic data. When designing the schema, we wanted to include descriptions that were static in nature. In another words, elements that describe a system that would not change in general. Description elements that change often should be queried from the systems themselves using functions. To further illustrate this concept, lets look at location. We provide a location tag that represents a systems typical home location, but we expect clients to get the specific location at a given time of a system by querying through the use of a function. Overall, the ESPml schema is a work in progress and other elements can be added as first-order elements, but they will have to meet the general guideline that the data is static in nature.

CHAPTER 5

Detailed System Design

In addition to the ESPml schema language, the framework provides an architecture to enable registration and interaction with sensor networks. The basic two components involved in this prototype are the registry and a system. The following section details the design of these entities and goes over their interfaces.

5.1 Registry

The registry is the component that serves as a repository for sensor systems. It provides services that enables sensor systems to be added, updated, deleted, and queried for. In terms of its architecture, one can think of it as a database backend with a web service front-end. The sensor systems are registered by providing an ESPml document. Thus, the database structure is modeled similar to the hierarchy of the ESPml schema. Table 5.1 shows the definitions of the various tables that are involved in storing a sensor system in the registry database. One of the main aspects of the database structure that needs to be pointed out is how locations, polygons, and points are represented. Instead of storing them as pure XML strings, the basic components of each of these structures are represented as fields in the tables. This enables efficient searching especially when location based queries are initiated on the registry. To make the registry faster, a more thorough design analysis of the database structure needs to be done.

Name	**Fields**					
System	id	systemURI	description	xml		
Field	id	systemId	fieldKey	xml		
Platform	id	fieldId	platformKey	xml		
Sensor	id	platformId	sensorKey	xml		
Location	id	referenceTable	referenceId	xml		
Polygon	id	locationId	xml	posList		
Point	id	locationId	xml	latitude	longitude	altitude

Table 5.1: SQL Table Definitions for Registry Entries

In terms of interfaces, the registry uses web services via SOAP to implement remote procedure calls. The following functions are exposed as part of the service: `register`, `unRegister`, `update`, and `listSystems`.

- **register**

  The register function takes an ESPml document that describes a system and adds it to the database. If there is already a system in the registry with the same identifier, then the old system will be deleted and the new one will be added.

- **unRegister**

  In the unRegister case, again a ESPml document will be provided as input, and the registry simply deletes the system described from the database.

- **update**

  The update function is similar to sending the same system twice via the register command. The purpose is to update an existing entry. If the entry

does not exist, then it will be added.

- **listSystems**

  The listSystems function takes a low overhead ESPml document that contains a polygon area that is described as a point string. The registry responds with a ESPml document with all the systems, that are part of the polygon area.

If there is a problem due to improper ESPml formatting or inconsistencies exist in the actual call when compared to the database, then an error string is returned. Standardization of the return value types needs to be made and this is considered an essential next step.

5.2 System

The system component represents the actual sensor system that client applications and other sensor systems can access. Essentially, the system component is a gateway to the sensor network as a whole. Interactions with the sensor network occurs through a web service interface named execute.

- **execute**

  Once a consuming program finds the appropriate system it wants to interact with, the component runs the execute command with an ESPml that specifies what functions to run on the system. The result of the execute command is a lightweight ESPml document that contains the output of the functions. The output elements consist of the type attribute and a URI field. The actual code necessary to execute the function is defined by

the individual sensor network. Also, the availability of the output is also determined by the sensor network.

Overall, the framework dictates that the system needs to implement the functions that it specifies, provides a standard method to actually interact with the system through the execute web service remote procedure call (RPC), and provides a standard way to format the response or output for the execution.

5.3 Security Components

There are two major issues that need to be addressed in terms of security: authentication and encryption. Authentication for clients and other components in the architecture is addressed in Chapter 7 using the OpenID model. In this section, we focus on encrypted communication between the various parts of the ESP framework.

5.3.1 Motivation

In order to motivate the use of encryption in the framework, we will first cover the types of messages that typically get exchanged among the ESP components. First, there is a register message in which a system details its capabilities and the services it offers. Often times, there are update messages that are sent whenever a certain characteristic of a system changes. Also, there are query messages that occur between clients and the registry. The query contains the type of information or service a particular client wants, and the response to the query contains all the systems that can address this type of query. Finally, there is the actual execute message, and its associated reply, in which the client asks a particular system to perform some action and gets an appropriate reply, either data that exchanged

or an action that happens on the actual system, in response.

These various messages contain very critical information that can be used to profile users and also the components involved in the ESP framework. For instance, by monitoring the register and update messages coming for a system, a malicious user can accurately model such attributes such as the system's location or how its capabilities change over time. Also, by monitoring the queries that occur from clients one can determine the exact interests of a client. Furthermore, a malicious user can send incorrect information as a reply to a execute message. Thus, this motivates us to provide a mechanism to encrypt the various messages involved in the ESP framework.

5.3.2 Encryption Scheme

In order to implement the encryption scheme for the ESP framework, we use Secure Sockets Layer (SSL) in conjunction with SOAP [Cho02]. Before detailing the SSL scheme, first lets analyze why this approach is better than other encryption methods that are available. One method that could be employed is the use of symmetric encryption. Essentially, both the sender and the recipient use the same key to encrypt and decrypt the data. There are several popular algorithms, including Data Encryption Standard (DES), Triple DES, Advanced Encryption Standard (AES), and RC4, that can perform symmetric encryption. Encrypting and decrypting is typically not CPU intensive. The problem with symmetric encryption is that there has to be a reliable and secure method to distribute the keys. If the key distribution is done insecurely, then anyone can decrypt the messages and interpret the data [Sch94]. Another method that can be used for encrypting messages is public-key encryption. This method of encryption involves two separate keys: a public key and a private key. Typically,

the public key is used to encrypt a message and only the private key can be used to decrypt it. Thus, one can freely distribute the public keys. The downside is that decrypting a message that has been encrypted by a public key is quite CPU intensive [Sch94].

SSL capitalizes on the advantages of both symmetric and public-key encryption algorithms. Essentially, SSL uses public-key mechanisms to boot-strap the use of secret-keys that are symmetric. Thus, most of the traffic that gets exchanged is done through the use of less expensive symmetric key algorithms. SSL consists of two phases: handshake and data transfer. During the handshake phase, the two parties involved in the communication use public-keys to determine secret-keys. This phase involves the two parties involved in the communication to agree on the set of cipher suites that they both support. Furthermore, they both verify that each other's public keys are actually authentic through the use of a certificate authority. The client, in another words the user initiating the connection, creates the secret key to be used. During the data transfer, the secret key is used to encrypt and decrypt transmissions. Additionally, a client making several new connections with the same server in a short time could use a session identifier so that a particular secret key that has been established can be used without having to perform the initial handshake once again. But using this session resumption technique opens up an inherit security risk but this is avoided usually by renegotiating the secret-keys periodically [Cho02].

Using SSL in conjunction with SOAP requires the various components of the ESP framework to register with a certificate authority. Although this requirement seems to be a hefty price to pay to use the system, in reality it is not since a certificate authority can simply reside under the control of ESP and can even run on the same system as the registry. To justify the use of SSL, we evaluate both

public and symmetric key algorithms in the evaluation section. Furthermore, we make the use of SSL over SOAP optional so that clients or components that are not necessarily concerned with security or do not have the ability to perform the encryption schemes can still use the ESP framework.

CHAPTER 6

Enhancement Prototypes

6.1 Decentralized Authentication

One of the main issues that needs to be addressed for the ESP framework to further succeed is concerned with setting up an authentication system. There are several reasons for having authentication for various components, especially the clients, and they include: having the ability to control access to certain systems, providing selective services based on identity, and for auditing purposes. Also, the actual authentication process could occur on different modules in the ESP framework.

In order to implement an identity/authentication system, we adopted the OpenID model [BD06]. OpenID is essentially a decentralized identity system, where the actual identity is just a URL. All the OpenID system does is provide a method to prove that the owner of the URL is who he claims to be. This process is done without passing any sensitive information around. Only identifier that needs to be passed is the OpenID URL, which can be thought of as a username for an entity.

The ESP framework will use the OpenID system for authentication purposes. A web services wrapped interface has been created so that systems, clients, or even the registry could take advantage of this OpenID model Figure 6.1 shows a typical interaction and goes over some of the technical details regarding how the

model works. In this particular scenario, the client is trying to authenticate to the system.

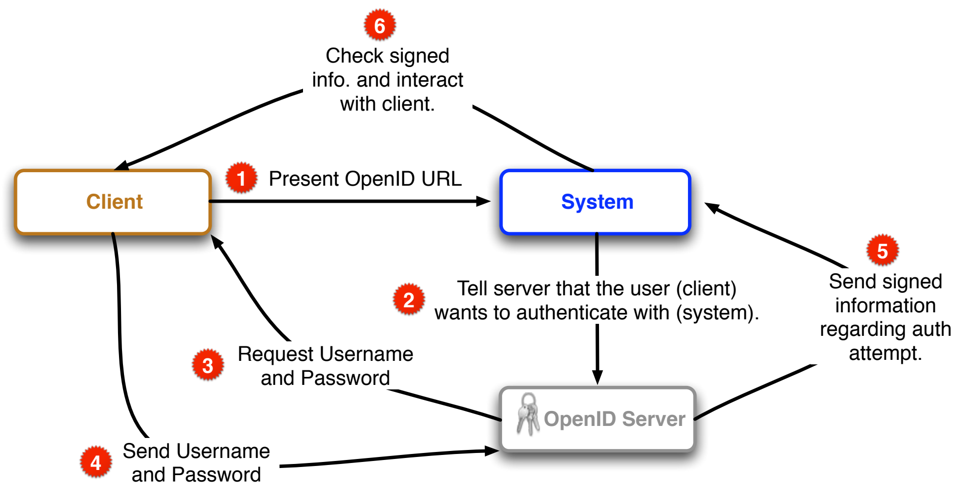


Figure 6.1: A typical OpenID interaction in the ESP framework.

1. **Send OpenID URL**

   The client sends its OpenID URL to the system. This URL can be hosted anywhere but should be on a server that is highly reliable.

2. **Inform OpenID Server of Request**

   The system then goes to the OpenID URL and looks for a openid.server tag. The system sends a message to the OpenID server that is referred to by the openid.server tag to inform that it wants the client authenticated and specifies a return to address for the OpenID server to visit once the authentication process is completed. This message is actually transferred via HTTP GET interface call.

3. **Request Client for Username and Password**

This is the key part of the OpenID system. Instead of passing username and password (authentication information) to the system, the OpenID server directly asks the client. This is typically done using a web page interface where a user interactively fills the necessary information but in our framework this was automated by using Python scripts.

4. **Send authentication information to OpenID server.**

   The client provides the necessary authentication information to the OpenID server. The OpenID server checks this information in its database system.

5. **Send confirmation to system.**

   At this point, if the OpenID server got correct information from the client, then a call is made to the return to address of the system specified earlier with signed information, which is typically the OpenID client URL, OpenID server URL, and the type of authentication. If the information provided by a client is not valid, a cancel message is returned to the address specified by the system.

6. **System checks signed information.**

   The system then contacts the OpenID server again with the signed information to check whether it was a real request or not. If all information is valid, then it starts interacting with the client as an authenticated user.

6.2 Network Functions

Another concept that is needed in the ESP framework are nodes that provide in-network functions on data streams. One can imagine there are several services these mediator type nodes can provide including: attested contextual in-

formation, resolution control of data streams, verification on data values, and duplication of data streams [SHB06]. These mediator nodes are different from such efforts as Active Networks in which individual users or groups can inject customized programs into nodes of a network. We imagine a much more controlled scheme where a set of trusted nodes offer services for verification and scalability purposes. The network functions will be manipulated by disclosure and verification rules set by the systems in the environment.

To further examine the concept of mediator nodes that provide network functions, we present different application scenarios. In the first, we examine a sensor that provides images of a certain location. If two different clients enact functions to receive these functions, one of which the sensor system is aware off and trusts and another which is a anonymous user, the system might provide different resolutions for the image. The mediator would actually provide the resolution control on these images based on the rules set by the image sensor and the clients that are interacting with the sensor system. Another example is concerned with attestation location and time. For instance, if there was a mobile sound sensor that obtained samples from various environments that a user can query, the mediator can be involved in verifying or at least providing a network testimony to the actual data that is being provided by this sensor. The mediator node would essentially be an inter-mediator between the audio sensor and the client that wants to get information from it and when information passes through it, the mediator would provide its location and time. This information can be used by the client as a verification technique to attest to see if the location and time the actual sound sensor is advertising is reliable. Finally, we can have a scenario where information is simply dropped or re-directed due to a policy. For instance, data streams can be filtered so that they do not reach users that have not be authorized to actually receive them.

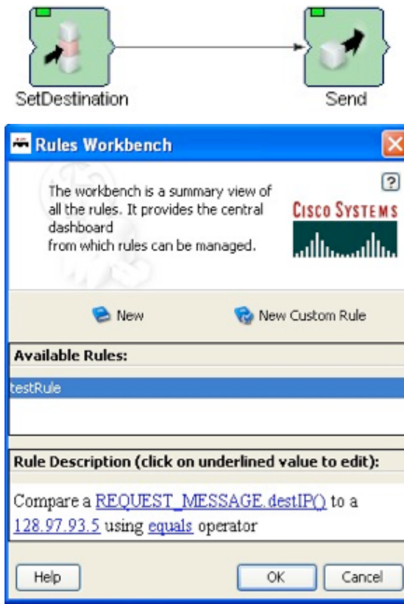


Figure 6.2: AON PEP Showing URI Re-Direction.

To prototype a mediator node, we chose the Cisco Application Oriented Networking (AON) technology [AS06]. Essentially, AON enables application oriented networking functionality with capabilities to inspect the contents of messages and then perform some type of action on these messages ranging from routing decisions to content manipulation. In our setup, AON is included as part of a blade of a typical Cisco router. AON provides a programming and architecture interface in order to easily manipulate network traffic based on a message layer. The basic unit in AON is a Bladelet, which is a Java program that enables one to perform operations on a message. For example, Cisco offers a variety of pre-defined Bladelets to perform such functions as logging message data to a database, getting http header information, obtaining cached information to manipulate an incoming message. Bladelets are then used in conjunction with Policy Execution Plans (PEPs) that enable one to perform more complex message level manipula-

tion tasks. PEPs basically consist of a combination of Bladelets with the ability to provide conditional functionality and other control mechanisms. Figure 6.2 shows a PEP that does a simple URI based redirection. Essentially, the PEP looks at the header information of the message and filters based on the URI and redirects to another URI. Although this is a fairly simple example PEP, one can see that more complex manipulations of messages can be added using the AON infrastructure.

6.3 Data Logging Services

Using the ESP framework, one is able to share a sensing system for various consumers to use. But what happens to data that is not being consumed by a consumer? What if a consumer wants to perform queries based on historical data? Can data from different sensors be mashed up to provide data sources that are more robust? These questions can be answered by the introduction of a data logging service. In another words, a component that can collect sensing data (slog) and then provide services based on this data.

This slogging component has several design ideals that it should be based off. These design requirements can be divided to how recording of information is done, what information is actually logged, and how this information is presented to consumers of the data. In terms of actually logging the data, we imagine having both a push and pull model. In the push model, sensors would essentially send data to a data slogging component and have their data logged. While the pull model works by having the data slogger subscribe to sensor systems and actively pulling data based on a set of rules that control sampling technique and rate. In the pull model, we expect a triggered interface where a time, sensor system, duration, sampling frequency, and event can be set to actually obtain

the information. Once the information is received by the slogger, it will store it in different resolutions. For instance, various time and space scales can be used to store the data so that one has different sampling frequencies of data or even varying amount of sensors used to cover a particular area. Also, one can imagine custom sloggers that combine various sets of data from different sensing systems and providing aggregation services. An example might be a slogger that combines sound and image data from different sensors and providing location/time based media streams of information gathered. Finally, we imagine the data can be presented to the consumers as a programmable playback device. Essentially, the user can setup schedules of sensor streams that take in time, location, sensor type, resolution, and duration and this stream gets played back to a consumer.

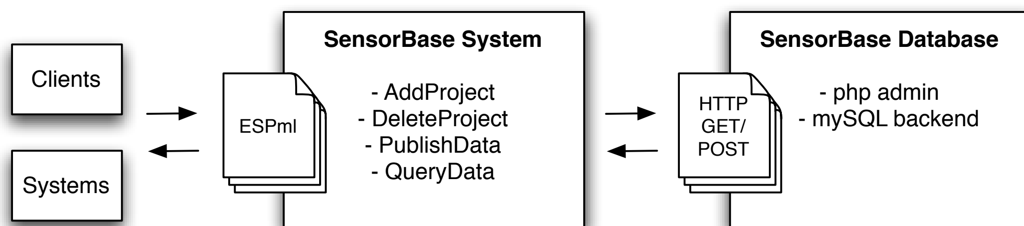


Figure 6.3: ESP system interaction with SensorBase.

As a first cut of a slogger, we implemented an interface to SensorBase. SensorBase is database back-end that has a REST type interface to add projects, send data, and retrieve data. For instance, users can publish data by performing HTTP POST with a string that represents the data. Furthermore, the queries can be made on a certain project and the results are returned in an XML format, which is defined by SensorBase. The ESP SensorBase interface is currently a system component and has functions that enable entities in the framework to create projects, publish data, and query for data. Figure 6.3 contains a high level

diagram of the message flow that occurs in this system. As you can see, we just simply have a ESP front-end to the SensorBase functions. This currently allows a way for various systems to log sensor data. We imagine this type of interface to grow and provide the capabilities of the slogger that was described earlier.

CHAPTER 7

Evaluation

In order to evaluate the ESP framework, we created several systems that would be registered and also a geo-centric client that can actually query and interact with the example systems. The following section details the different types of systems implemented and also the architecture of the client.

7.1 Example Systems

One of the main goals of the ESP framework was to be able to represent several different types of networked systems. Furthermore, we wanted to have a low development overhead to use the framework. Currently, we provide a Python based system template that users can utilize to add their system in to the framework. Essentially, a system would need an ESPml document describing its capabilities and must modify the system template to add the proper hooks for the functions that are specified in the ESPml description.

There are several systems that were added to the architecture to demonstrate its robustness. The systems are shown visually as Figures 7.1, 7.2, and 7.3.

- **Virtual Weather Stations**

  In order to demonstrate that the ESP framework can handle many systems in terms of quantity, we added virtual weather stations for every zip code

for a particular region. In this case, it is for the state of California. Essentially, a system for each zip code in the state was added to the registry with function capabilities to get the current weather information, a ten day forecast, and an hourly forecast. The results of running these functions results in a URI that represents a web page that contains the actual information about the weather forecasts.

- **Community Web Cams**

  Another type of system that was added using the ESP framework are web cams that exist already for community monitoring. Specifically, we implemented three web cams in the Los Angeles region that show pictures of UCLA, Santa Monica pier area, and Venice. The systems have a function that gets the current picture of the region they are monitoring.

- **Actuated Network Camera**

  To demonstrate the ability to take parameters as part of a function call to a system, we implemented a Sony RZ30n network camera as a system. Using the system, a user can set the pan, tilt, and zoom values and then obtain either a picture or a movie.

- **Photodiode Sensor Network**

  Another capability that the ESP framework enables is aggregation functions through the use of platforms or fields. A photodiode sensor gateway that operates over five MicaZ motes is registered using the framework. Not only can one get photodiode values from any of the individual motes, but one can also perform aggregation on the whole field to get an average value for the photodiode readings. Furthermore, one can set the individual sampling rate for any of the photodiode sensors.

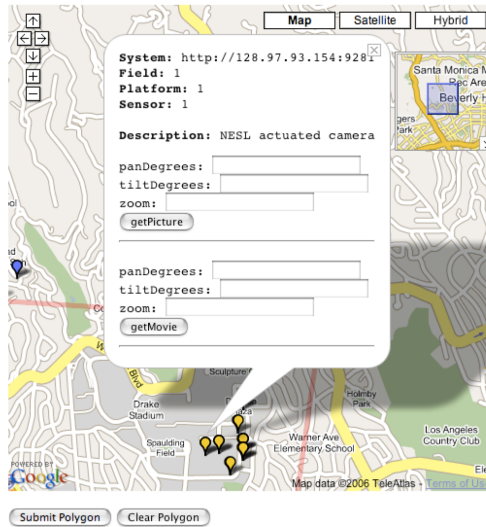
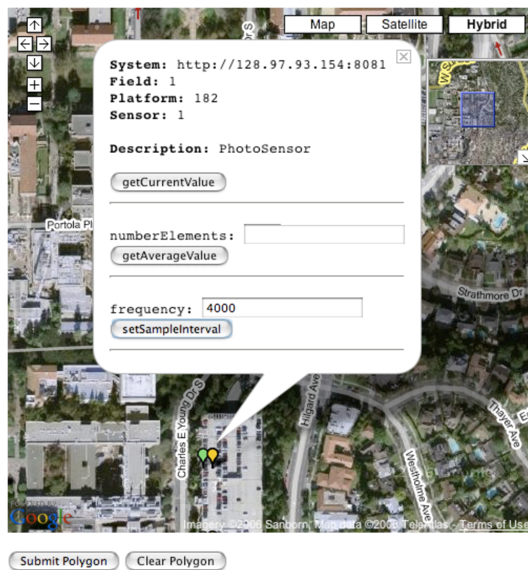


Figure 7.1: Screen-shot of the Google Maps client and an actuated camera.



Function: setSampleInterval
Type: OK
Response:

Figure 7.2: Screen-shot of the Google Maps client and a small sensor network node's photodiode sensor.

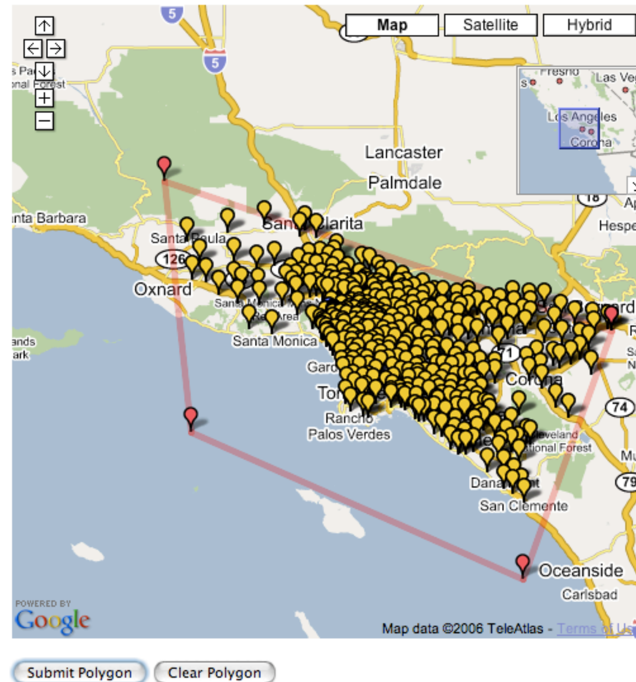


Figure 7.3: Screen-shot of the Google Maps client and the weather stations in the Los Angeles area.

7.2 Client Application

To interact with the ESP framework and also the systems that were implemented using the architecture, a geo-centric web client that relies on Google Maps was created. Essentially, the user has the ability to draw a polygon box over a certain area and then query for all the different networked systems in that space. When the results come back from the registry, the networked systems in that area are represented as click-able point icons. Different colors for the point icons represent the levels of abstraction for the systems. For instance, fields are represented as purple icons, while platforms and individual sensors are green and yellow. Once a user clicks on the icons, a box layer pops up that shows a description of the entity

that is being interrogated and then the user can interact with the component by clicking on buttons that represent function calls. The results of the function call show up on a side frame. Overall, the client application provides a visual interface that users can use to interact with the sensor systems that are currently registered.

7.3 Observations

Based on implementing the various example systems with the framework and creating the client application, several observations were made. While evaluating the speed of queries, we realized that the database is a bottleneck. We suspect that this is the case due to the fact that there needs to be some optimization techniques applied and the actual design of the database model needs to be improved. Another point that became clear is that for systems with numerous functions, a more robust interface then a window listing all the functions needs to be obtained. Furthermore, we realized the limitations of the Google Map interface as we tried to map more advanced functionality onto it. There are situations when non-location based queries might be necessary and also the interface does not provide advanced mapping functions such as temporal or spatial visualization overlays and analysis techniques. Overall, there is a need for different types of client applications and also optimizations in the various parts of the framework.

7.4 Performance

In this section, we examine the performance of the ESP framework and use the results to validate the design decisions made in the architecture. Specifically, we focus on analyzing both public and secret-key encryption algorithms and then we

overview the latency breakdown for a typical query by a client to the registry. The actual profiling was done using a Apple Power Mac G4 with 1.25 GHz processor and 1 GB of memory. Furthermore, Python 2.4 was used as the implementation language with the SOAPpy, PyCrypt, and MySQLdb extensions added.

7.4.1 Cryptography Components

Table 7.1 lists the costs of the cryptography mechanisms that can be used in security systems. We profile the use of RSA and DSA public key (asymmetric) systems along with several secret-key (symmetric) algorithms including AES, DES3, Blowfish, and RC5. All execution times were determined by using a 1KB input blocks and averaging results from thirty specific profile runs. The measurements show the asymmetric algorithms, such as RSA and DSA, are much more computationally expensive than symmetric algorithms in the key generation, encryption, and decryption phases. These results justify the use of SSL, which is a hybrid combination that uses public keys to bootstrap symmetric key encryption for actual transmission. Thus, public keys are used only once per session while symmetric keys are used to transport data over a period of time.

	Key Gen.	**Encryption**	**Decryption**
RSA	489.6 ms	1.981 ms	130.8 ms
DSA	4550.0 ms	1.305 ms	100.1 ms
AES	0.091 ms	0.025 ms	0.017 ms
DES3	0.087 ms	0.039 ms	0.026 ms
Blowfish	0.166 ms	0.017 ms	0.067 ms
RC5	0.024 ms	0.017 ms	0.092 ms

Table 7.1: Timings for Cryptographic Algorithms

7.4.2 Registry Profiling

In addition to security, performance profiling was performed for tasks that the registry is responsible for. The two main procedures that can occur on the registry are to register a system and also to query for specific systems based on location string that is provided. Figure 7.4 and 7.5 contain the computation time overheads associated with each of these types of registry procedures while adjusting the payload obtained or inputed to the actual procedures. The systems used to register and also for querying using listSystems contained a field with one platform. The platform contained one sensor that had a single function that could be called. The results for the profile analysis for both the register and listSystems procedures were obtained by running each specific scenario twenty times and then taking the average of the results.

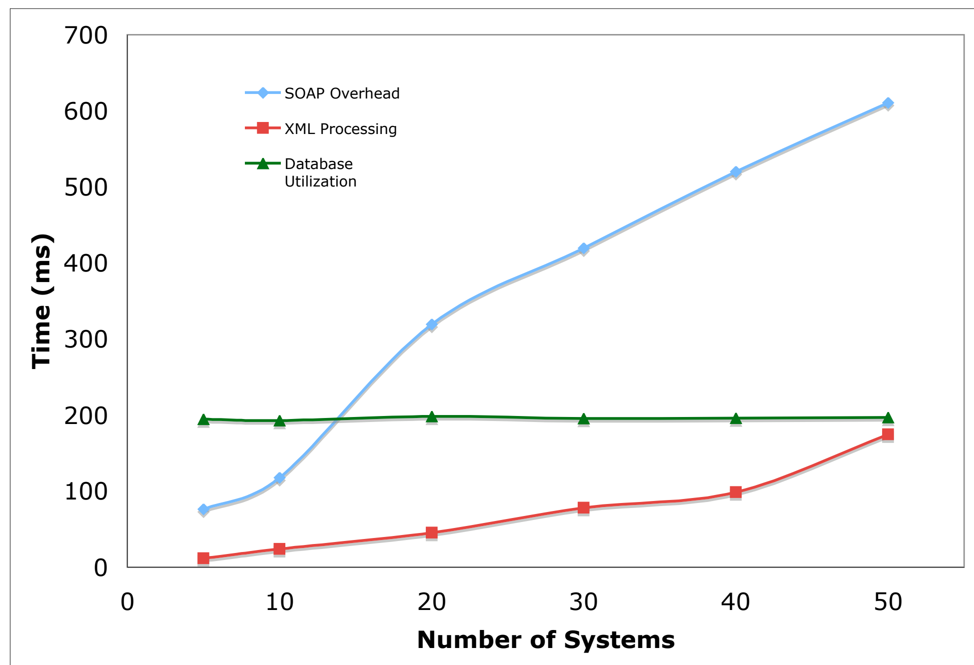


Figure 7.4: register Latency.

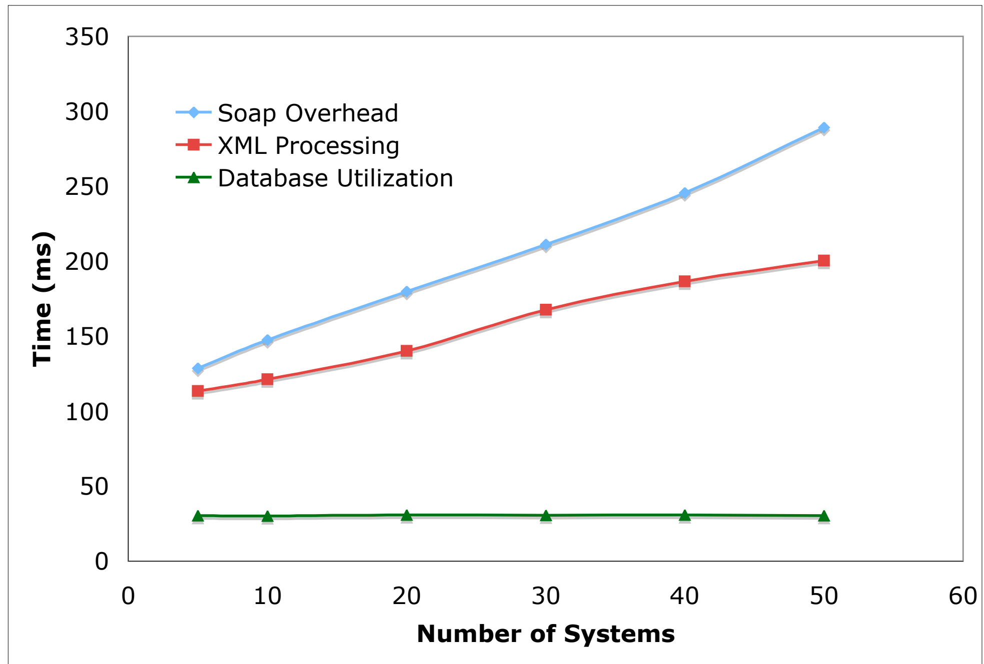


Figure 7.5: listSystems Latency.

The three types of overhead that could occur with both the register and listSystems calls are SOAP, XML, and database. SOAP overhead accounts for creating a SOAP message on the client/system side, transporting the message via HTTP, decrypting the message on the registry, and sending a SOAP response back via a HTTP. XML processing accounts for any computation done on the registry dealing with parsing and creating XML documents in response to the procedure calls. Finally, the database overhead concerns computation time spent on accessing data sets from the registry repository. In Figure 7.4, the number of systems represents the components that were actually registered. On the other hand, in Figure 7.5 the number of systems represents the components that get returned.

Analyzing the results, one can clearly see that as the data payload increases,

in another words more systems registered or more systems returned as query responses, SOAP overhead increases. But in both procedures, even when the number of systems is high, the actual SOAP overhead is still relatively small. For instance, for registering fifty systems the SOAP computation took about 600 ms, while a call to listSystems for 50 systems took 300 ms. Thus, we can conclude that SOAP scales fairly well for our application usage scenarios.

Another attribute that is noticed is that there is a price to pay for using XML as the interoperable language. As the number of systems increases, the XML processing time also increases. In the case of listSystems, XML processing is more prominent than registering systems because a complete XML node entry for each component needs to be created. Database utilization is fairly constant in both procedures just due to the overall efficiency of the model used to store the data and also the query used to obtain data. In general, register has a higher database utilization than listSystems due to the fact that there are entries being added as opposed to just being accessed. Overall, we can conclude that both SOAP and XML processing overhead increase as the data payload for each procedure increases. But the net computation time is still fairly low even when interacting with a large set of systems.

Lastly, we focus on the listSystems request to the registry. In terms of actions that are performed on the registry, we believe that listSystems will be the most common request because typically users will be adding and deleting systems much less often than making queries to obtain systems based on certain criteria. To analyze how such queries will scale as the number of users increases, we performed simultaneous requests. The results, presented as Figure 7.6, show that as the number of simultaneous requests increases so does the latency time in an linear fashion. One might assume that this latency is caused by the SOAP

server having to handle multiple requests and blocking, but in reality this is not the case. The SOAP server is in fact multi-threaded and handles each SOAP request using an independent thread. Serialization of the requests occurs due to the database. Currently, the way the database queries are setup requests will lock the tables associated with making such a query. This can be vastly improved by using temporary tables in which copies of the data of interest are made and each request only acts on the copied data. Although there is some overhead with making copies of the information and then cleaning up the temporary data, we believe that this will be much less overhead then locking the database until the request is acted upon and then letting the next request access the database. Thus, database optimization will be a task that will be focused on as future work.

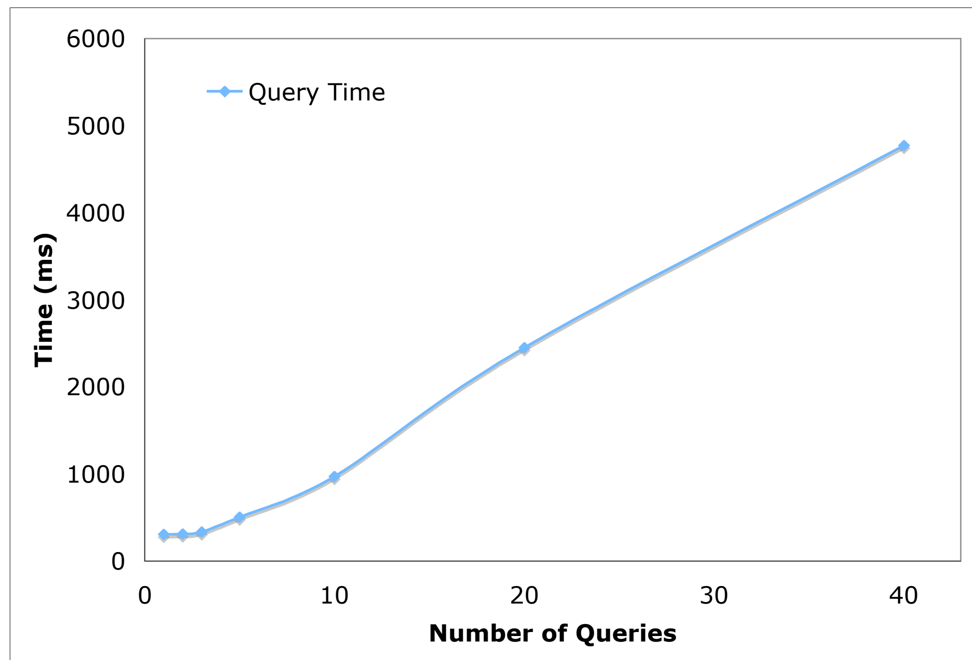


Figure 7.6: listSystems Simultaneous Query Latency.

7.5 Wide Area Middleware Architecture Comparison

It is particularly interesting to look at other wide area middleware architectures, such as Jini, Ninja, and Silph, as comparison units to the ESP framework because they employ a complementary design approach to solve the fundamental problems involved in connecting, discovering, and using heterogeneous systems. In this section, we will focus our discussion on the similarities and difference in how the designs approach interoperability between components, discovery of resources, and intelligent networks.

7.5.1 Interoperability

Flexible middleware that enables the interaction of heterogeneous devices in a distributed fashion is not a new concept. But the implementation choices associated with achieving this goal usually affect the types of devices that can be connected. Jini exploits the use of mobile byte-code, using Java as the programming model, to enable interaction between systems (services) and the clients. Specifically, systems provide clients with portable Java objects that give clients access to the services that the systems provide using predefined network communication protocols such as remote method invocation (RMI) or SOAP [Wal99]. Unfortunately, by using a Java-centric approach that relies on running byte-code, devices that are not able to run the Java Virtual Machine (JVM) will not be able to interact in this architecture. Sylph uses Jini but adds an abstraction layer on top to limit the set of commands to only a few specific procedures to set and get data [CMY02]. Ninja uses a variation of the Java RMI as the model to enable interaction between systems and clients [GWB01]. Java RMI is based on a binary wire protocol with remote procedure call semantics. Furthermore, the use of Java RMI introduces a much more tight coupling between the clients and

systems due to the fact that whenever an interface or communication protocol changes then the client needs to be informed and often times initiate some type of setup procedure to use the existing system again.

Although all these architectures share the common goal of inter-connecting various heterogeneous devices, the implementation decisions affect the scale, in terms of capabilities, of components that can be used and the ease of maintenance and compatibility with future standards. Since ESP uses XML as the interface language and SOAP to enable interaction, more devices including components that have limited computation or memory capabilities can be involved in the framework. This is unlike Jini which requires components to run a JVM. Another aspect of using SOAP is that it is universally accepted as a remote procedure call method while distributed object technology, such as Java RMI which Ninja uses, have other competing implementations in the same space and thus lack a large continuing maintenance and development forum. Finally, SOAP is based on message passing for RPC and has a very lose coupling between components which enables interfaces and communication protocols to be easily changed without strict coordination requirements.

7.5.2 Service Discovery

Jini, Sylph, and Ninja all have concepts of a service discovery mechanism that is parallel to the idea of a registry in the ESP framework. But the architectures have a difference in what type of data is actually registered. Jini registers a small set of meta-data information regarding a service and a proxy Java object [Wal99]. Silph, which uses Jini service discovery module, has a proxy core module that manages the queries between a client and the actual sensors or systems that are registered [CMY02]. Finally, Ninja enables systems to describe what services

they provide using XML and has a layer to take care of data queries [GWB01].

The difference between these other architectures and the ESP framework is that the registry is geared toward system capability lookups while the other designs are aimed for data-centric searches. In another words, the types of queries associated with the other architectures are similar to SQL type queries and act on the actual data provided by the sensors. The ESP registry uses such attributes as location, type of sensors, functions provided, and service descriptions as first order elements for searching. Searching for data is left for a back-end application to provide as a service in the ESP framework.

7.5.3 Intelligent Networks

Several projects have had the concept of using the network to perform transformations on data. Active networks allow customized programs to be injected into nodes of the network to enable new network protocols to be implemented without changes to existing edge systems, filtering of packets for security purposes, and traffic routing on a granular scale [TW02]. The key distinction between Active Networks and the ESP mediator design is that we aim to work on higher layers, such as the transport or application layer, as opposed to the packet layer. This enables the ESP mediator architecture to have much more of a context, in terms of who is using the data and for what purposes, when making decisions about routing, filtering, or distribution of data. Ninja has the concept of Active Proxies that also share the idea of working on higher level application semantics [GWB01]. The main focus of the Ninja Active Proxies seems to be in adapting service content to better suit smaller devices and also implementing security transformation for resource constrained units. The ESP mediator aims to have additional goals than just content and security protocol manipulation. For in-

stance, the ESP mediator architecture calls for providing such services as adding network context attested information, such as location and time, providing privacy control and selective sharing of data using policy controls, and data stream duplication for scaling purposes.

CHAPTER 8

Future Work

There are a few different areas that should be investigated further in terms of this project. First, some additions need to be made to the ESPml schema language in order to more accurately describe certain types of systems. Also, there are changes that need to be made to the architecture in order to scale the platform as a whole. Furthermore, there are some issues related to standardization that need to be addressed. Finally, some additional systems and clients need to be made for the architecture that can act as services that other components could use.

8.1 Standardization

One of the main actions that needs to be performed in terms of ESPml is standardization of a few different aspects of the schema. ESPml allows different sensor types to be defined. Currently, these types can be any arbitrary name. But there should be a standard naming convention for the sensor types. This will enable users that use the framework to interpret the sensor types programmatically. Furthermore, certain sensor types should have standard functions that need to be defined. These functions will have a certain prototype associated with them and a standard output as well. For instance, one can expect all sensors that observe some type of phenomena and quantify it in terms of a measurement

to have a function to get the current value. In addition, they should be able to get the average over a period of time and to set the sampling rate. Having this type of standardization will make describing sensors that are similar easier since the same schema constructs can be repeated and guarantee the end user certain basic functionality for most sensor types.

8.2 Mobility and Availability

When describing sensor systems, two attributes that need to be expressed in some fashion is mobility and availability. Mobility deals with the notion of whether the system actually moves. Since the ESPml schema has a location tag as part of the description language, if the systems are mobile then this location tag may not be reliable. To represent the difference between mobile systems and non-mobile ones, a mobility tag can be added to different constructs of a system to indicate that the location might not be reliable based on the information in the registry. At this point, the location that is in the actual ESPml description serves as the last provided location by the sensor system or the location that the sensor system is actually located at with the highest probability. In the next revision of the schema, a confidence interval can be given to the location indicating how likely the node will be at that particular location. Also, if the mobile tag exists, a function will be required to specify the exact location for that system.

In terms of availability, one would introduce timestamps and a time to live counter on the registry. Basically, when a system is registered it would be assigned a time to live in which the system is required to re-contact the registry to verify that it is still available. The client program can then check the registry to get the time to live requirement and at what time the system last updated the registry to evaluate the availability confidence level for a system. Furthermore, statistics

can be kept on the registry about the quality of the system connection. Overall, both mobility and availability are important aspects of a sensor system that need to be addressed in the ESPml framework.

8.3 Scalability

As more sensors are added and query requirements diversify, scalability becomes an issue with the ESP framework. Managing a large amount of sensors with one registry is not feasible without having performance issues related to queries and management. Thus, a hierarchy for registries needs to be developed. The Domain Name System (DNS) platform can be used as an example model [MD]. When a system is registered, a unique name could be provided to the system. An example naming scenario would be to adopt the unique name based on how domain names are formatted. Thus, there will be top level domains and then sub domains associated with sensors that are somehow related. For instance, sensor systems located in the same region might share a certain level in the naming convention. Then, the actual registries could be arranged in a tree fashion where each registry is responsible for a certain zone of entries. There would be a a set of root registries that handle the top-level construct in the naming convention. If the query that comes cannot be addressed by the top-level registry then the request would be passed along the hierarchy to a registry that can actually handle the request.

In terms of queries, location is the only aspect that can be searched for in terms of finding systems. But one can imagine other querying methods. For instance, the type of sensor, functionality of sensor systems, and a common naming convention can all be attributes that can be queried for by clients. These types of searching capabilities for sensor systems need to be incorporated into the registry

to make it usable by a larger set of clients.

8.4 Client Applications and Systems

Currently, there are many different types of network systems that are incorporated into the ESP framework. But there are not many clients that use the actual system. An example Google Map client interface is provided that enables searching and interactivity through a map interface, but one can imagine a larger plethora of services that can be created on the client side. There are more sophisticated analysis services that can be provided as clients or systems by using the capabilities of popular statistical and signal processing software components such as R, ESRI, and Matlab. Also, we can imagine using Google Earth or other mapping tools to serve as further client programs for users to visually query and interact with various components in the system. Finally, one can explore ideas related to signal search including how to quantify such a search, what are the features of a signal to characterize it, and what the query language would be for such a search. These signals would be data sets that are obtained from various sensors or even archived data that is stored in a slog.

CHAPTER 9

Conclusion

One of the fundamental characteristics of networks systems, especially sensor networks, is that they are heterogeneous in nature. With the ESP framework, we provided a standard method to manage, query, and interact with these varied sensor systems. ESPml serves as the model for describing sensor network systems at different levels of granularity. The ESP architecture, which includes a registry and system entities, uses a web service based framework for locating sensor systems and then methods to actually interact with the systems themselves including getting information and running specific procedures. A geo-centric web interface is provided as a portal for users to interact with the various sensor systems that are available on the framework as well.

REFERENCES

[Arn05] V. Arnaudov. "Unified Management of Heterogeneous Sensor Networks In the Atlantis Framework." *Brown U*, 2005.

[AS06] T. Anthias and K. Sankar. "The network's new role." *Queue*, 4(4):38–46, 2006.

[BBB04] T. Brooke, J. Burrel, and T. Beckwith. "Vineyard Computing: Sensor Networks in Agricultural Production." In *Pervasive Computing Magazine*, pp. 38–45. IEEE, March 2004.

[BD06] Fitzpatrick B., , and Recordon D. "OpenID." http://www.openid.net, 2006.

[BGH05] P. Buonadonna, D. Gay, J.M. Hellerstein, W. Hong, and S. Madden. "TASK: Sensor Network in a Box." *Proceedings of the Second IEEE European Workshop on Wireless Sensor Networks and Applications*, 2005.

[BMM06] Sam Baird, Dan Myung, Steve Moulton, Mark Gaynor, Matt Welsh, and Stephen Dawson-Haggerty. "Communicating Data from Wireless Sensor Networks Using the HL7v3 Standard." *Body Sensor Network*, 1:183–186, 2006.

[Bot] Mike Botts, editor. *OpenGIS Sensor Model Language (SensorML), Implementation Specification.*

[CDK02] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI." *Internet Computing, IEEE*, 6(2):86–93, 2002.

[CDL] Simon Cox, Paul Daisey, Ron Lake, Clemens Portele, and Arliss Whiteside, editors. *OpenGIS Geography Markup Language (GML) Implementation Specification.*

[Cho02] W. Chou. "Inside SSL: the secure sockets layer protocol." *IT Professional*, 4(4):47–52, 2002.

[CMY02] A. Chen, RR Muntz, S. Yuen, I. Locher, SI Sung, and MB Srivastava. "A support infrastructure for the smart kindergarten." *Pervasive Computing, IEEE*, 1(2):49–57, 2002.

[DPP03] FC Delicato, PF Pires, L. Pirmez, and LF Rust da Costa Carmo. "A flexible web service based architecture for wireless sensor networks." *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pp. 730–735, 2003.

[DRK06] Estrin D., Guy R., Chang K., Hansen M., , and E. Graham. "SensorBase." http://www.sensorbase.org, 2006.

[DS02] Y. Hu D. Li, K. Wong and A. Sayeed. "Detection, Classification, and Tracking in Distributed Sensor Networks." In *IEEE Signal Processing Magazine*, pp. 17–29. IEEE, March 2002.

[GWB01] S.D. Gribble, M. Welsh, R. von Behren, E.A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, et al. "The Ninja architecture for robust Internet-scale systems and services." *Computer Networks*, 35(4):473–497, 2001.

[Lee00] K. Lee. "IEEE 1451: A standard in support of smart transducer networking." *Instrumentation and Measurement Technology Conference, 2000. IMTC 2000. Proceedings of the 17th IEEE*, 2, 2000.

[MD] P.V. Mockapetris and K.J. Dunlap. "Development of the Domain Name System.".

[MP01] R. Muntz M.B. Srivastava and M. Potkonjak. "Smart Kindergarten: Sensor-based Wireless Networks for Smart Developmental Problem Solving Environments." In *Mobile Computing and Networking*, pp. 132–138. ACM, March 2001.

[OK] N. Ota and W.T.C. Kramer. "TinyML: Meta-data for Wireless Networks.".

[Sch94] B. Schneier. *Applied cryptography.* Wiley New York, 1994.

[SHB06] M. Srivastava, M. Hansen, J. Burke, A. Parker, S Reddy, G. Saurabh, M. Allman, V. Paxson, , and D. Estrin. "Wireless Urban Sensing Systems." April 2006.

[SNL06] A. Santanche, S. Nath, J. Liu, B. Priyantha, and F. Zhao. "SenseWeb: Browsing the Physical World in Real Time." http://research.-microsoft.com/nec/senseweb, 2006.

[TP05] D. Trossen and D. Pavel. "Building a Ubiquitous Platform for Remote Sensing Using Smartphones." *Proceedings of the The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services-Volume 00*, pp. 485–489, 2005.

[Tur05] M. Turon. "MOTE-VIEW: A Sensor Network Monitoring and Management Tool." *Embedded Networked Sensors, 2005. EmNetS-II. The Second IEEE Workshop on*, pp. 11–18, 2005.

[TW02] DL Tennenhouse and DJ Wetherall. "Towards an active network architecture." *DARPA Active NEtworks Conference and Exposition, 2002. Proceedings*, pp. 2–15, 2002.

[Wal99] J. Waldo. "Jini architecture for network-centric computing." *Communications of the ACM*, 42(7):76–82, 1999.