An XML Server for Networked Physical Objects

by

Ashutosh Somani

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2001

Author ....
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by .... [illegible]
Kai-Yeung 'Sunny' Siu
Associate Professor of Mechanical Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An XML Server for Networked Physical Objects

by
Ashutosh Somani

Submitted to the
Department of Electrical Engineering and Computer Science

May 23, 2001

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis examines how to return desired portions of Product Markup Language (PML) files in response to client queries. Information about physical objects will be stored in the form of PML files that reside on servers scattered across the Internet. The syntax of PML is based on XML so we make use of XML technologies to implement the PML server. We use the Simple Access Object Protocol (SOAP) as the communication protocol, Apache Tomcat Java Servlet Engine as the Internet application server, XML Query Language (XQL) as the query language, and the Infonyte XQL engine as the query handler. We also present our implementation for the Tracking Server. A Tracking Server stores location information about physical objects and resides at a node in the supply chain. It is implemented using SOAP as the communication protocol, Tomcat as the Internet Application Server, the MM MySQL JDBC driver as the query handler, and MySQL as the database. We make use of a Tracking Package that implements the distributed algorithms necessary to retrieve the tracking information.

Thesis Supervisor: Kai-Yeung Siu
Title: Associate Professor

Acknowledgements

I would like to thank my thesis supervisor, Sunny Siu, for introducing me to the Auto-ID Center and giving me an opportunity to work there. I thank him for looking out for my long-term interests and keeping my work in perspective. I am grateful to Ching Law and Dan Engels for the countless discussions that provided me with invaluable suggestions and guidance. I could not have done this without them. Ching was my primary source of assistance, and someone I heavily depended on throughout my research. I thank Sanjay Sarma for being approachable and someone from whom I could solicit any type of advice. His general friendliness and sense of humor made him a pleasure to work with. Thanks to Amar for always being there and offering help in any shape or form when I needed it throughout my education at MIT. Thanks to all the brothers of Phi Beta Epsilon for giving me a home away from home. Thanks to Parul, Sneha, Neesha, Amit, Richa, and all my friends at MIT for providing balance in my life and making the last four years so memorable. I owe a special thanks to my twin brother, Paritosh, who has provided me lifelong companionship and has been with me every step of the way. He has been a perennial source of encouragement and motivation. Lastly, I would like to thank Mom, Dad, and Anju for their everlasting love and support.

Table of Contents

Chapter 1: Introduction

The Internet has established an infrastructure that provides connectivity around the world. People can access data by searching online through numerous resources or by connecting with someone thousands of miles away. The next step being pursued in this growing web of connectivity is the idea of connecting physical objects to the Internet. The MIT Auto-ID center is designing and implementing the infrastructure to make this happen. The aim is to develop an open and scalable architecture for embedding automatic identification technology in physical objects in order to provide access to information regarding them [1]. This requires two essential components: the technology that provides for the automatic identification of an object, and the information network that allows for its information exchange. This thesis discusses the key part of the latter component, the Physical Mark-up Language (PML) Server.

1.1 Auto-ID Center Technologies

The Auto-ID center has already developed certain technologies to help realize its vision. These include the Electronic Product Code (ePC), the Object Naming Service (ONS), and the Physical Markup Language (PML). Additionally, the Auto-ID Center works with global standards bodies and the Auto-ID industry to help specify how items will communicate in a standard environment, and to help facilitate the development of the next generation hardware that will drive the system [2].

1.1.1 The Electronic Product Code

The Electronic Product Code (ePC) is a 96 bit numbering scheme capable of uniquely identifying every physical object on earth. It is broken down into four partitions: the Header, Manufacturer Code, Product Code, and Serial Code. The current specification calls for the following partition:

XX	**.XXXXXXX**	**.XXXXX**	**.XXXXXXXXXX**
Header	Manufacturer	Product	Serial Number
8 bits	*28 bits*	*20 bits*	*40 bits*
	265 million+ unique IDs	*1million+ unique IDs*	*1 trillion+ unique IDs*

The Header is used for version control and specifies how the rest of the ePC is broken up. Thus, if the specification ever changes and the partition is altered, the system should be able to transparently adapt. The Manufacturer Code corresponds to the manufacturer of the product and is needed to identify where information on that particular product can be found, as the rest of the chapter will describe. The Product Code identifies the unit-independent characteristics of the product. Finally, the Serial Code provides a unique identity for every item.

The ePC of an object will be embedded in a Radio Frequency (RF) tag that will be attached to an object. An RF tag reader can detect any tags within a certain range of the reader. This system is different from the current bar code system used to identify products in a couple of few key areas. One disparity is how the bar code system requires manual labor to properly align a scanner to read a bar code correctly. Another difference is that unlike the ePC, the numbering scheme in the bar code system can only identify a class of products (e.g. a certain brand and style of tennis shoes), not each individual product. Whereas, as the UPC bar code provides a reference to information on a computer, the ePC will provide reference to information on the Internet. There is not much more infrastructure required in the case of the bar code because once the bar code of an item is read, any pertinent information can be looked up on the computer connected to the bar code scanner. However, since the ePC will make use of the Internet, there needs to be some mechanism to determine where to look for information corresponding to a particular ePC. This necessitates the use of the Object Naming Service.

1.1.2 The Object Naming Service

The Object Naming Service (ONS) is used to locate information on the Internet about any physical object that carries an ePC. This is an application layer protocol on top of the Internet's existing Domain Name Service [1]. Given an ePC, an ONS server provides an IP address for the server that stores information about that particular ePC. These servers are called Physical Markup Language (PML) servers because they store this information in the form of PML files. The next section explains what PML is.

1.1.3 The Physical Markup Language

The Physical Markup Language (PML) is intended to be a general, standard way of describing the physical world [3]. It will serve as the lingua franca of the entire Auto-ID system. Software applications, data storage, and analytical tools will all rely on it as a common standard. Instead of using an entirely new syntax, it uses the extensible Markup Language (XML). It is just an XML scheme and for the intents and purposes of this paper, can be treated as XML. It looks like XML and can be queried with XML querying tools. Information about each tagged object will be stored in the form of a PML file. Thousands of such files will be stored in each PML server. In order to retrieve information from a PML file, an XML querying language must be employed.

Figure 1.1 summarizes the entire process of obtaining information about a product. First a tag reader is used to read the ePC of a product. Next, this ePC is used to query ONS for the IP address of the particular PML server that stores information about the product. Finally, a PML server can be queried for the desired information.

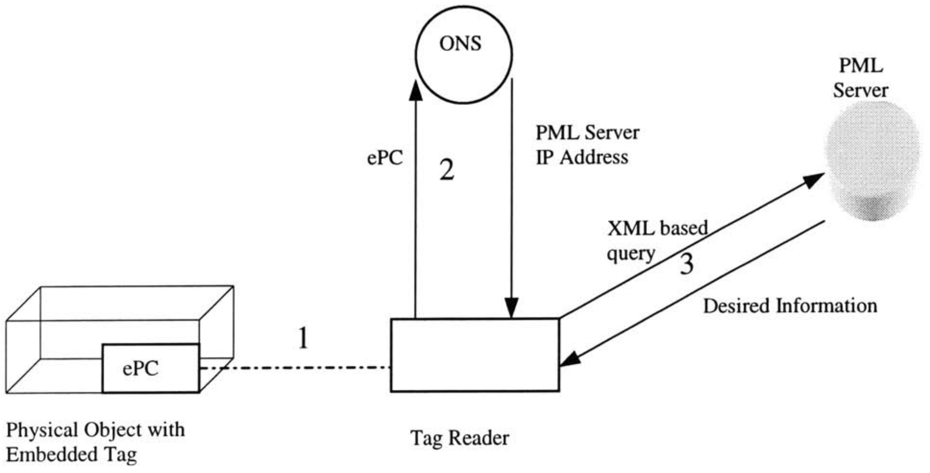


Figure 1.1: Overview of System

1.2 PML and Tracking Servers

Information about each product is stored in some type of server. We have already mentioned the PML Server, which stores most of the information in the form of PML files. We will also discuss the Tracking Server, one of the first applications of the Auto-ID Center.

1.2.1 PML Server

The focus of this thesis is on the design and implementation of the PML Server. Although information about each tagged product is stored in one or more PML files that reside on the server, the PML server is not just a file server. Due to the amount of information that may be stored in a PML file, each file can get fairly large. Thus, to minimize the Internet traffic and make the system more scalable, we make use of a PML database. Only the desired part of a PML file is sent back to the client. It is more than just a simple database because since the PML files must be queried through the use of an XML querying language, there must be some type of query handler that can process the query. At the time of our design, there was no database with a front-end for an XML querying language. Figure 1.2 provides an overview of the system and the aspects of the server that are relevant to our discussion: the communication protocol, the Internet application server, and the engine that processes the client query to return the pertinent information (the query handler and PML Database).

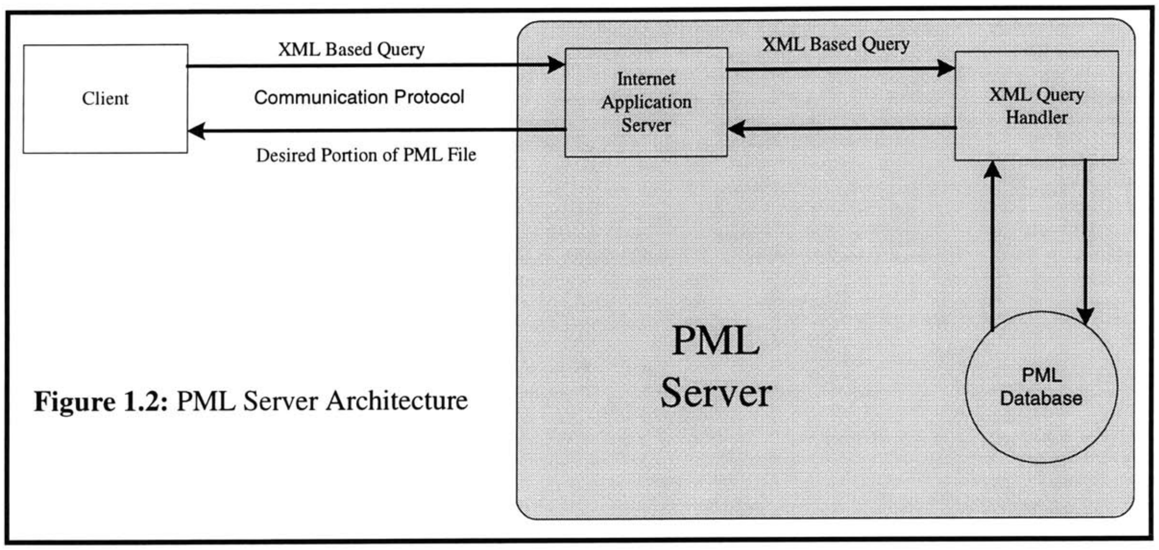


Figure 1.2: PML Server Architecture

The PML Server will be queried from a variety of locations over the Internet. Therefore, there is need for a communication protocol to transmit the query from the client to the server and then return the portion of a PML file that is the query result. The communication protocol can be viewed as facilitating a remote procedure call. The client supplies two arguments (a tag and a query), the server processes the two arguments, and the result is returned.

A PML Server is essentially an Internet application server, the application being the processing of client queries about PML files that are stored in the server. The queries that may be given to a PML Server can be broken down into two types: those about a class of products (class data) and those about an individual object/item (instance data). Instance data queries place a significantly greater load on the system. There will be much more instance data in each server since they may end up storing information about millions of individual products while only thousands of types of products. Furthermore, there may be more than one PML file that stores information about a particular product. This may become more complex since one file may contain a pointer to another. Thus, handling instance data queries is a more involved process and absorbs the majority of the system's resources.

1.2.2 Tracking Server

The Tracking Server is a server for a particular type of instance data. There will be one at every node in the supply chain. Each server keeps track of where a product is currently located in the supply chain and where it has been in the past. Since it stores very limited information about each product, PML files are not used. This makes the backend of the two servers very different. In fact, there is no need to use an XML querying language to query the tracking server. There are also very limited types of queries that a tracking server can expect and the architecture is optimized for handling only those queries. The Tracking Server is just one example of instance data that is stored in the Auto-ID system.

1.3 Thesis Contributions

In this thesis, we present the design and implementation for the PML and the Tracking Server. We implemented simple, flexible solutions and succeeded in building a proof-of-concept prototype for one portion of the MIT Auto-ID Center system. Our thesis contributions can be divided up into three main areas: the communication protocol, the server side for the PML Server, and the server side for the Tracking Server.

Communication Protocol

For both servers, we chose the Simple Object Access Protocol (SOAP) as the communication protocol. There were a couple of alternatives but SOAP is better suited for the Internet and is more flexible. It is simple and minimizes the infrastructure that needs to be built. We justify our use of SOAP by providing a detailed discussion of the alternate communication protocols. We set up a SOAP service for both the PML Server and the Tracking Server. SOAP was used in conjunction with the Apache Tomcat Servlet Engine. It was used as the Internet application server. The use of the two technologies provides clients on the Internet access to the procedures on each of the servers.

PML Server

The primary contributions of this thesis pertain to the server side processing of the PML Server. We describe a way to store PML files and return the desired portions in response to client queries. We describe how the XML Query Language (XQL) and the Infonyte XQL engine are used to query the PML files and process the queries. Again a justification for their use is provided. The selection of the two technologies was a joint decision. We needed an XML based query language that had some type of support to process the queries. Without a standard for XML query languages at the time of our implementation, there were very few choices available. However, we found a stable solution in the form of XQL and the Infonyte engine.

We present a simple and efficient methodology to process a PML Server query. Each query necessitates a search through multiple files since information may be stored on any number of files relevant to a particular product. We provide a file searching algorithm to search all the relevant files. We also support queries for a range of tags or regular expression queries that may match multiple tags. This additional functionality saves bandwidth by making use of server side processing.

Tracking Server

The Tracking Server stores information about a particular class of instance data. In order to respond to a client query, however, a Tracking Server may need to query other Tracking Servers as well. We demonstrate a mechanism to do this. We make use of the distributed algorithms described in [4] in the form of a Tracking Package. The Tracking Package specifies data storage and network methods that need to be implemented. The network methods are implemented using SOAP. We make use of a relational database, MySQL, and a corresponding JDBC driver, MM MySQL, to implement the data storage and access methods. In the case of the Tracking Server, there is no need to store the information in PML files this time since only a very specific type of information is stored. PML files add an overhead to the system that would be detrimental to the efficiency. We describe our implementation of two methods that can be called when querying a Tracking Server: `track` and `locate`. The first traces the entire history of a particular item in the supply chain while the latter provides the current location of an object. Thus, we implemented a basic Tracking Server that can be extended with additional functionality once additional methods are implemented in the Tracking Package.

Chapter 2: The Communication Protocol

A key part of the PML Server is the communication protocol responsible for transporting a client query to the server and returning the result of the processed query back to client. As mentioned earlier, SOAP was chosen as the communication protocol. Section 2.1 gives an overview of SOAP, Section 2.2 provides some background information about some alternate communication protocols, Section 2.3 justifies our use of SOAP, and Section 2.4 describes how we used SOAP in the PML Server.

2.1 Overview of SOAP

SOAP stands for Simple Object Access Protocol because it is a lightweight protocol for the exchange of information in a decentralized, distributed environment. It is intended to provide remote procedure call (RPC) facilities which work across platforms and languages [5]. The current specification is for HTTP transport of messages encoded in XML. Although HTTP transport is the only type specified, other types are not excluded from consideration.

2.1.1 Breakdown of SOAP

SOAP is an XML based protocol that consists of three parts: an envelope, a set of encoding rules, and an RPC representation [6]. The envelope defines an overall framework for expressing what is in a message, who should deal with it, and whether it is optional or mandatory. The set of encoding rules define a serialization mechanism that can be used to exchange instances of application-defined datatypes. Finally, the SOAP RPC convention defines a convention that can be used to represent remote procedure calls and responses. SOAP has no explicit programming model because one of its primary goals is to make existing programs more accessible to a broader range of users.

As a result, SOAP assumes that existing technology will be used as much as possible instead of inventing new technologies. As mentioned earlier, two existing and widely deployed protocols are used by SOAP: XML and HTTP. HTTP is SOAP's RPC-style transport and XML is its encoding scheme. SOAP simply codifies the use of XML as an HTTP payload [7].

2.1.2 Apache SOAP Architecture

We use the Apache SOAP implementation. Its architectural diagram is shown in Figure 2.1:

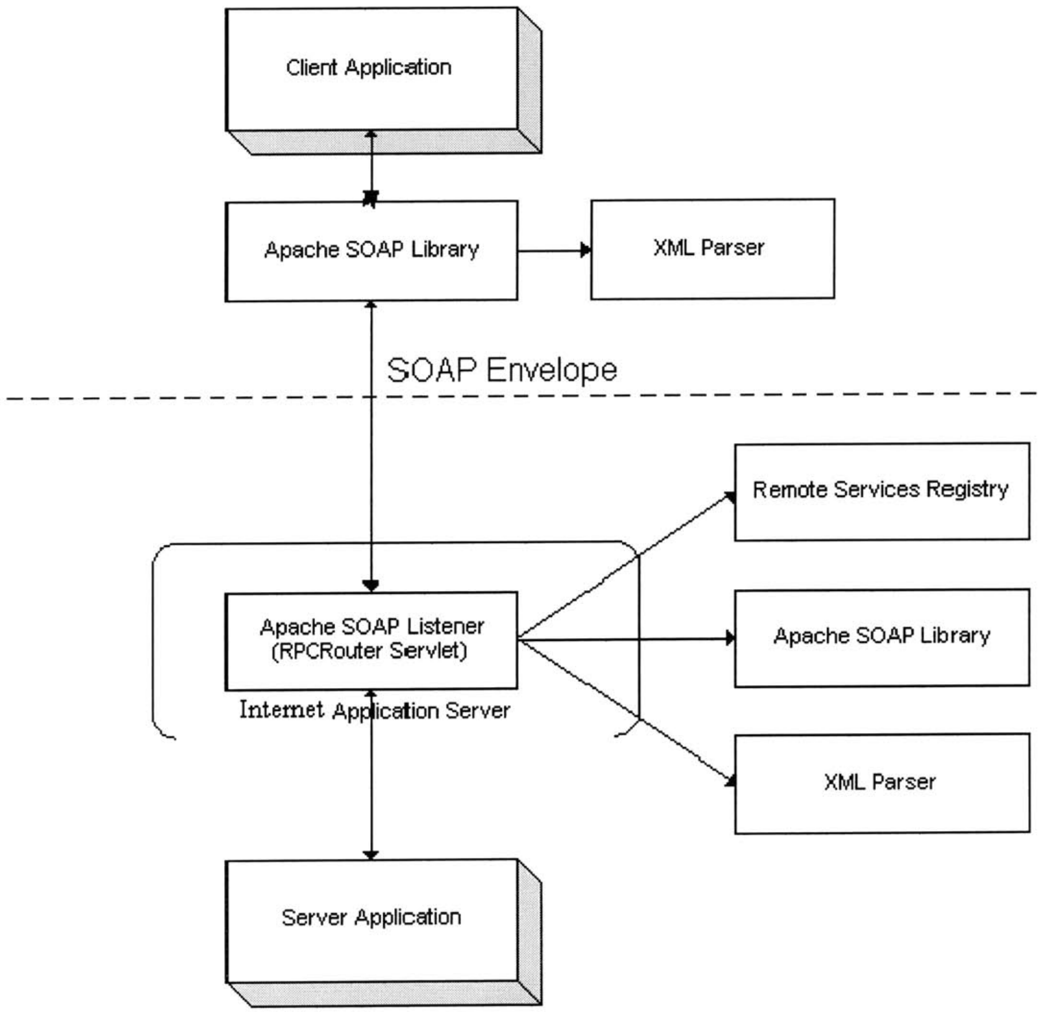


Figure 2.1: Apache SOAP Architecture [8]

The figure shows that the client side requires a client application, the Apache SOAP Library, and an XML Parser. The server side requires the Apache SOAP Library, an XML Parser, and an Internet application server. The Apache SOAP implementation assumes the use of the Apache Xerces XML Parser, which is why we chose it as our XML parser. We chose the Apache Tomcat servlet engine as the Internet application server, but this will be discussed in Chapter 3.

As Figure 2.1 shows, the Apache Listener waits for a client request and once one comes, it makes sure that the service requested has been registered. If it has, the registry is examined to find out what to do with the client request (i.e. where to pass the parameters in the server application). Once the request has been processed, the result is returned to the client application.

2.2 Alternate Communication Protocols

There were a variety of alternate communication protocols for the PML Server but we chose SOAP because it is flexible, scalable, and simple. Justification for its use begins by looking broadly at communication models for building distributed applications. We then examine the two dominant Object RPC (ORPC) protocols: Microsoft's Distributed Component Object Model (DCOM) and OMG's Common Object Request Broker Architecture (CORBA).

2.2.1 Message Passing vs. Request/Response

The two most dominant communication models for building distributed applications are message passing and request/response. Message passing systems allow messages to be sent at any time by either party. This necessitates an explicit design style since an application will be aware that it is communication with external concurrent processes. Request/response protocols restrict communication to request/response pairs, however, and thus more closely resemble a single-process application. The requesting process is essentially stalled until it receives the response. This makes request/response communications a natural fit for RPC applications [7]. SOAP messages are fundamentally one-way transmissions from a sender to a receiver, but most commonly combined to implement patterns such as request/response.

2.2.2 Overview of DCOM and CORBA

A DCOM server is a body of code that is capable of serving up objects of a particular type at runtime. Each DCOM server object can support multiple interfaces, each representing a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. CORBA relies on a protocol called the Internet Inter-ORB Protocol (IIOP) for remoting objects. Everything in the CORBA architecture depends on an Object Request Broker (ORB). The ORB acts as a central Object Bus over which each CORBA object interacts transparently with other CORBA objects located either locally or remotely. Each CORBA server object has an interface and exposes a set of methods. To request a service, a CORBA client acquires an object reference to a CORBA server object. IIOP is just CORBA's version of the General Inter-ORB Protocol (GIOP) [7]. Thus, both protocols use an object endpoint ID to identify the target object, as well as a method identifier to determine which method to invoke. Figure 2.2 shows that the request formats for the two protocols are nearly identical.

There are two key differences between IIOP and DCOM relevant to our discussion: the interface ID and the format of parameter values. One is that with IIOP, the interface ID is implicit since a given CORBA object only implements one interface.

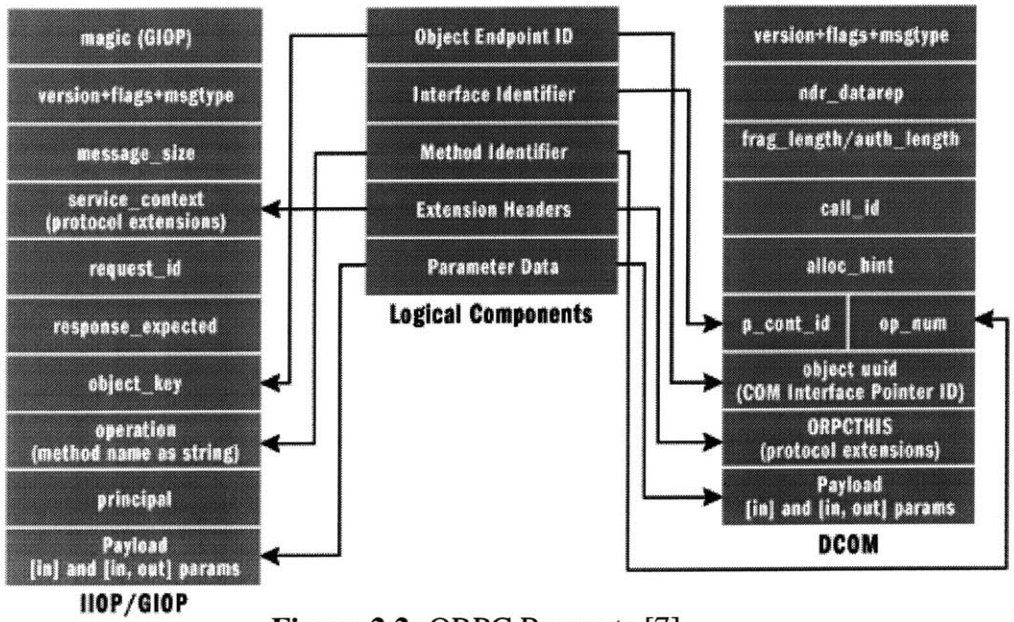


Figure 2.2: ORPC Requests [7]

The other difference worth noting is the format of parameter values in the payload. In DCOM, the payload is written in a format called Network Data Representation (NDR) and in IIOP, the payload is written using Common Data Representation (CDR). There are differences between the two formats that render them incompatible.

2.2.3 Weakness of DCOM and CORBA

Although they are both solid protocols, the industry has not completely shifted to either and adopted one. Part of the reason may have to do with cultural issues, but the technical applicability of both protocols has also been called into question. The consensus has been that both DCOM and IIOP are reasonable protocols for server-to-server communications, but have been shown to have severe weaknesses for client-to-server communications, especially when the client machines are scattered across the Internet [7].

- They rely on single-vendor solutions to use the protocol to maximum advantage.

- They rely on a closely administered environment.
- They rely on fairly high-tech runtime environments.
- They have severe inabilities to work in Internet scenarios.

2.3 Advantages of SOAP

Next, we examine how SOAP solves some of the problems faced by DCOM and IIOP.

2.3.1 HTTP as a Better RPC

HTTP is a protocol very similar to an RPC-type protocol. It is widely deployed and also more likely to function when faced with a firewall than any other communication protocol available today. Thus, it is considered well suited for the Internet. HTTP layers its request and response communications over TCP/IP just like DCOM and CORBA. An HTTP client connects to an HTTP server using TCP. Once a connection has been established, the client can send the HTTP request message to the server and expect an HTTP response message once the request has been processed.

HTTP headers are just plain text. This makes problems easy to diagnose by using packet sniffers or even text-based Internet tools like telnet. This also makes it easily adaptable to low-tech programming environments popular in Web development. For instance, the following is a legal HTTP request message:

```
POST /foobar HTTP/1.1
Host: 18.236.0.18
Content-Type: text/plain
Content-Length: 11

Hello World
```

The first line contains the HTTP method, the Request-URI (Uniform Resource Identifier), and the protocol version. GET is the HTTP method used to surf the Web.

POST is the most commonly used HTTP method for building applications. The Request-URI is simply a token used by the HTTP server software to identify the target of the request. The third and fourth lines of the request specify the size and type of the request payload.

Upon processing the request, the server is expected to send an HTTP response back to the client. This must contain a status code indicating the outcome of the request and may also contain arbitrary payload information. If the server application reverses the argument string, the response may look something like:

```
200 OK
Content-Type: text/plain
Content-Length: 11

dlroW olleH
```

As an RPC protocol, HTTP provides nearly all the functionality of IIOP or DCOM in terms of framing, connection management, and support for serialized object references. In conclusion, the major advantages of HTTP are that it is simple enough for the low-tech programming environments found on the Web, widely deployed, widely accepted, and works well in an Internet environment.

2.3.2 XML as a Better NDR

One piece missing from HTTP as an RPC protocol is a single standard format for representing the parameters of an RPC call. This is where SOAP makes use of XML. XML is a platform-neutral data representation protocol. It allows data to be serialized into a transmissible form that is easily decoded on any platform. It has the following advantages over NDR and CDR (the format of the parameter values used in the payload of DCOM and IIOP):

• Encoding and decoding software for it is widely available for nearly every programming language and platform

• It is text-based and easy to handle from low-tech programming environments

• It is extremely flexible and easily extended.

2.3.3 SOAP as a Better Communication Protocol

SOAP basically codifies the use of XML as an encoding scheme for request and response parameters using HTTP as a transport [7]. A SOAP method is just a HTTP request and response that complies with the SOAP encoding rules. Thus, it is an ORPC that takes advantages of the strengths of HTTP and XML. Figure 2.3 shows how SOAP maps to the ORPC protocol concept that was discussed earlier. One of the major advantages of using existing technologies (XML and HTTP) instead of inventing new ones is that incompatibilities are a lot less likely to arise. Furthermore, HTTP is already widespread while XML soon will be. This bodes well for the likelihood of SOAP becoming a dominant communication protocol.

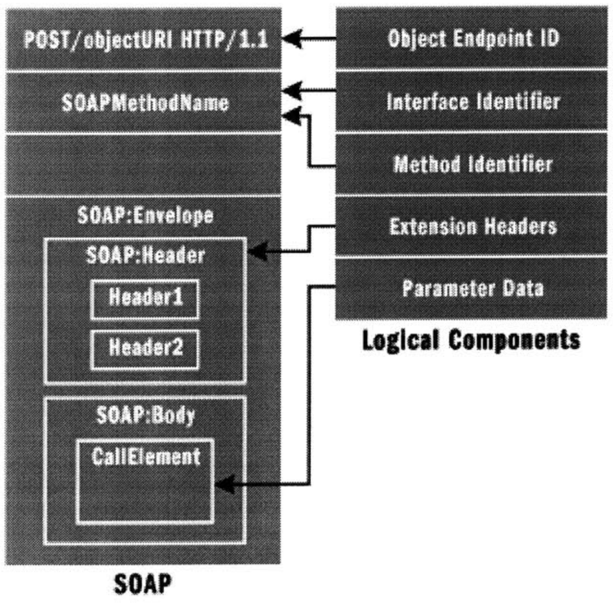


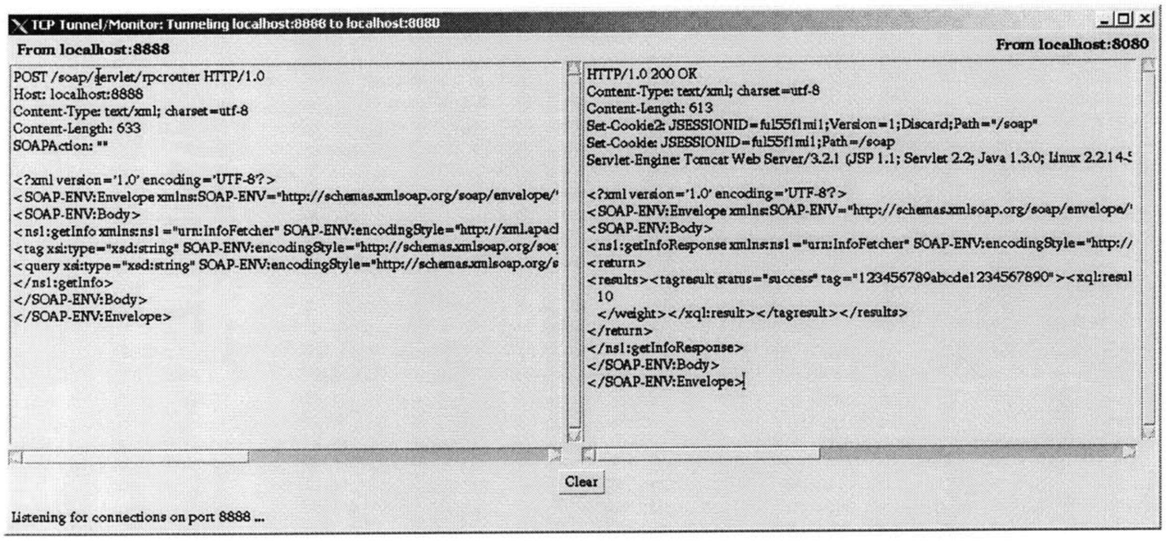
Figure 2.3: SOAP and ORPC [7]

From the perspective of the Auto-ID Center, a flexible technology was desired. We did not want to force our users to use a particular machine in order to be compliant with our system. SOAP offers this flexibility. Furthermore, it is simple and does not warrant the use of new technologies. This minimizes the necessary new infrastructure, something very important for a project of the magnitude of the Auto-ID Center.

2.4 SOAP in the PML Server

The PML Server uses SOAP to pass the query from the client to the server and then the result back to the client. The query consists of two parts: the ePC (or tag) to be queried in hexadecimal form and the query itself. The client can expect part of a PML file in return. In summary, the client sends two strings to a PML Server and gets an XML document back.

To use SOAP, we constructed a SOAP Service which we called 'InfoFetcher' and registered it to accept two string parameters. Thus, a client can make a call to the PML Server by adding two string parameters to its request (the ePC and the query) and setting the proper encoding type so SOAP knows how to serialize the request and response values. For the client request, each parameter is serialized as a string while the response is serialized as literal XML. We employed a TCP Tunnel Gui to see what the request and response looked like at the server side. This is shown in Figure 2.4:


TCP Tunnel/Monitor: Tunneling localhost:8888 to localhost:8080
From localhost:8888
From localhost:8080
Clear
Listening for connections on port 8888 ...

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: localhost:8888
Content-Type: text/xml; charset=utf-8
Content-Length: 633
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/'
<SOAP-ENV:Body>
<ns1:getInfo xmlns:ns1 ="urn:InfoFetcher" SOAP-ENV:encodingStyle="http://xml.apacl
<tag xsi:type="xsd:string" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soa
<query xsi:type="xsd:string" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/s
</ns1:getInfo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 613
Set-Cookie2: JSESSIONID=ful55f1mi1;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=ful55f1mi1;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.1 (JSP 1.1; Servlet 2.2; Java 1.3.0; Linux 2.2.14-

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/'
<SOAP-ENV:Body>
<ns1:getInfoResponse xmlns:ns1 ="urn:InfoFetcher" SOAP-ENV:encodingStyle="http://
<return>
<results><tagresult status="success" tag="123456789abcde1234567890"><xql:resul
 10
 </weight></xql:result></tagresult></results>
</return>
</ns1:getInfoResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 2.4: TCP Tunnel GUI

```
1  POST /soap/servlet/rpcrouter HTTP/1.0
2  Host: localhost:8888
3  Content-Type: text/xml; charset=utf-8
4  Content-Length: 633
5  SOAPAction: ""

6  <?xml version='1.0' encoding='UTF-8'?>
7  <SOAP-ENV:Envelope xmlns:SOAP-
   ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
8  <SOAP-ENV:Body>
9  <ns1:getInfo xmlns:ns1="urn:InfoFetcher" SOAP-
   ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
10 <tag xsi:type="xsd:string" SOAP-
   ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
11 123456789abcde1234567890
12 </tag>
13 <query xsi:type="xsd:string" SOAP-
   ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
14 //weight
15 </query>
16 </ns1:getInfo>
17 </SOAP-ENV:Body>
18 </SOAP-ENV:Envelope>
```

Figure 2.5: SOAP Message Embedded in HTTP Request

The left hand column is the SOAP Request and this is shown in more detail in Figure 2.5. The first few lines are the HTTP Request message that was discussed earlier. The remaining part is the SOAP Envelope. Line 9 tells the method name to be called on the server side (in this case ‘getInfo’) along with the encoding type for the response. Lines 11 and 14 are the parameter values, each of which are enclosed with the proper XML tags with the parameter names (in this case, ‘tag’ and ‘query’) along with their encoding types.

Similarly, the right hand column of Figure 2.4 is the SOAP Response and is shown in more detail in Figure 2.6. As before, the first few lines are the HTTP Response message followed by the SOAP Envelope. Line 10 tells us that this is the response for the call to the ‘getInfo’ method. Line 16 is the actual result of the query, in this case ‘10’.

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 613
Set-Cookie2: JSESSIONID=dv3kfn1yk1;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=dv3kfn1yk1;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.1 (JSP 1.1; Servlet 2.2; Java
1.3.0; Linux 2.2.14-5.0 i386; java.vendor=Sun Microsystems Inc.)

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getInfoResponse xmlns:ns1="urn:InfoFetcher" SOAP-
ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
<return>
<results>
<tagresult status="success" tag="123456789abcde1234567890">
<xql:result xmlns:xql="http://metalab.unc.edu/xql/">
<weight>
10
</weight>
</xql:result>
</tagresult>
</results>
</return>
</ns1:getInfoResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 2.6: SOAP Message Embedded in HTTP Response

Chapter 3: PML Server Side

This chapter details the server side processing of a query in the PML Server. We first examine the core technologies behind the PML server: the Internet application server, the XML querying language, and the XML Query Handler. Then we provide a discussion detailing the methodology used to process a query. This includes which files are searched in response to a query and what types of queries the system supports.

3.1 Technologies

Three primary technologies are present in the server side architecture. The Apache Tomcat Java servlet engine is used as the Internet application server. We chose XQL as the XML querying language and the Infonyte XQL Engine to process the XQL queries.

3.1.1 Tomcat

Apache Tomcat was chosen as the Internet application server to host the SOAP Service (discussed in the previous chapter) for the PML Server. Tomcat is a servlet container with a Java Server Pages (JSP) environment. More specifically, Tomcat is the Reference Implementation for the Java Servlet 2.2 and Java Server Pages 1.1 Technologies [9]. A servlet container is a runtime shell that manages and invokes on behalf of users. Servlet containers can be partitioned into three groups: stand-alone servlet containers, in-process servlet containers, and out-of- process servlet containers. In our implementation, we use Tomcat as a stand-alone servlet container because all we need it to do is to host a SOAP service on the Internet. We implemented the PML server with the latest version available at the time, Tomcat 3.2.

By default, Tomcat accepts HTTP connections on port 8080. So if a PML Server were deployed at some IP address, `18.236.0.18`, Tomcat would listen for connections at `http://18.236.0.18:8080`. Since the PML Server methods are embedded in a SOAP Service, a client request would need to be sent to the default SOAP URL, `http://18.236.0.18:8080/soap/servlet/rpcrouter`.

Tomcat is not the only Internet application server that can be used with the Apache SOAP implementation. Some of the other options include JRun or IBM's WebLogic. We chose Tomcat because it was the most commonly used out of the three in conjunction with SOAP. Given that SOAP was a new technology at the time of our development with very limited documentation, choosing Tomcat was quite beneficial because we were able to solicit a lot of help from the SOAP community. Tomcat is also fairly simple and since it was already known to work well with SOAP, it was a logical choice as our application server.

3.1.2 XML Query Language (XQL)

XQL is an XML querying language that provides an extension to the XML Stylesheet Language (XSL) pattern language. XSL provides a very simple and comprehendible way to describe a class of nodes. More detailed information about XSL can be found in [10]. XQL builds upon the features found in XSL for identifying classes of nodes by adding such features as Boolean logic and filters. It is a simple and concise querying language designed specifically for XML documents.

3.1.2.1 Basic Syntax

A full discussion with all the technical details can be found in [11] but here we just provide an overview of the basic syntax needed to query an XML document using XQL.

• A plain string is assumed to be an element name. Thus the following query would return all the <product> elements from an XML document:

```
product
```

• Hierarchy can be indicated using the child operator ("/"). Paths are always described top down, however, and the right-most element on the path is always returned (unless otherwise specified). Thus, the following query returns all the color elements that are children of car elements:

```
car/color
```

• The content of an element can be specified using the "=" operator. The result of the following query would be all the red cars:

```
/car/color='red'
```

• Likewise, the value of an attribute can also be specified using the "=" operator. Attribute names begin with "@" and are treated as children of the elements to which they belong. Therefore, this query would return all cars that are Toyotas:

```
/car/@type='toyota'
```

• Whereas the previous query returns type attributes, elements can be also be returned for the same query by using the filter operator ("[]"). This operator filters the set of nodes to its left based on the conditions inside the brackets. This query returns all the car elements that have an attribute called "type" whose value is "toyota":

```
/car[@type='toyota']
```

• Any number of intervening intervals can be specified using the descendant operator ("//"). The following returns all cars anywhere within product:

```
product//car
```

• If a descendant operator is found at the beginning of the path, it specifies all nodes descended from the document. This query would find any car in the document:

```
//car
```

This overview of the basic syntax of XQL should be sufficient to get most of the functionality out of the PML Server. As the above examples show, the syntax is very simple, straightforward, and logical. Even if a user has no experience with XQL, it is easy to gain a basic understanding and carry out desired queries to the PML Server.

3.1.2.2 Justification

Unfortunately, there was no standard for XML querying languages at the time of our implementation and there is still none in place today. Thus, there is no dominant XML querying language that has emerged. Two of the more popular alternatives are Quilt and XML-QL. When trying to decide on a querying language to use, our decision was based more on which could provide the most for the PML Server rather than which language had the most functionality. We wanted a querying language that had some type of database support. Given that there is no standard as of yet, very few database vendors had built in some type of front-end support for an XML querying language at the time of our decision. We found that XQL did have some support in the form of an XQL Engine built by Infonyte. This is primarily why we chose it. Our justification is that once a standard comes out, most database vendors will most likely build a front-end for the particular XML querying language. We left the architecture of the PML Server flexible enough so when a standard does emerge, a new querying language/database pair can be "plugged" into the PML Server and replace the current implementation. Our goal was to build a prototype of the PML Server so we chose XQL based on which language had the most support. We did not have a need for a querying language with any more functionality than XQL. This implementation decision is flexible, however, and can be changed at a later date.

3.1.3 Infonyte XQL Engine

The Infonyte XQL engine is the workhorse of the server. It implements the XQL Specification published in the W3C-QL '98 proposal "XML Query Language (XQL)" [12]. As we discussed earlier, the XQL Specification provides a set of query facilities for XML documents that are sufficient for the needs of the PML Server.

3.1.3.1 Functionality

The Infonyte XQL Engine implements the complete set of operators and functions described in the XQL Specification. It can process any XQL query for a properly formed XML document. In addition, proprietary extension operators are provided for variables, multi document queries, restructuring of query results, full text searches, result reconstruction, and sequencing. The added functionality makes the Infonyte XQL engine a comprehensive solution as a query handler. It has support for any required functionality for the PML Server.

The Infonyte engine does not set up the use of any relational database. Instead we simply use a flat-file database where all the PML files for a PML server reside in a particular directory. We built our PML Server on a Unix machine so we chose a directory in the Unix filesystem. There are then two steps to processing a query. Given a filename and an XQL query, the XQL engine first parses the specified PML file and builds a Document Object Model (DOM) representation in its memory. The DOM represents documents by a logical structure similar to a tree of nodes. The XQL engine then applies the query and returns the result. By processing the query in this manner, we are consistent with one of our major design goals. We did not want to use significant bandwidth and send the entire PML file back to the client. By making use of some server side processing we only need to send back a small portion of the PML file (the result of the query).

3.1.3.2 Justification

The justification for using XQL as the querying language and the Infonyte XQL engine as the query handler go hand in hand. We were looking for some type of support for the XML querying languages and the only feasible solution available at the time was the Infonyte XQL engine for XQL. Not only did it meet our needs functionally, but it was also built entirely using Java. Since this was the programming language we planned to use for our server code, and the one needed to interface with the Apache SOAP implementation, it fit in very nicely with our system. Furthermore, it has built in performance optimizations such as automated and transparent indexing on the document structure of an XML file.

Two other options we considered were Kweelt and Software AG's Tamino database. Kweelt is a framework to query XML data and among other things, it offers an evaluation engine for the Quilt XML query language [13]. It offers multiple backends because its query evaluator does not impose any specific storage for the XML files being stored, but instead relies on a set of interfaces. This gives a lot of flexibility for the storage medium, including a relational database (MySQL). While the performance of Kweelt may eclipse that of the Infonyte XQL Engine, we chose not to use it because there were too many uncertainties with the querying language. In particular, Kweelt does not claim to be faithful to the Quilt proposal. Our system could not tolerate such an uncertainty because we needed a query handler that would work accurately with any one dominant XML querying language. Although Kweelt did have support for the majority of Quilt queries, there was the danger of building a query handler that would not recognize some queries supplied by a client. Furthermore, Kweelt was still a work in progress at the time of our development whereas the Infonyte XQL engine had already been known to work well with XQL queries.

The reason for not choosing Software AG's Tamino database product was purely monetary. The Tamino product has many attractive features [14]:

- Built-in native-XML data store for fast database access
- Storage and retrieval of XML data via HTTP and TCP/IP
- Connection to standard Web servers
- Virtual DBMS capability by integrating existing DBMS sources
- Interfacing with XML authoring, schema, and other XML tools
- Customizable server functionality
- Support of read-only databases
- Centralized Web-browser-based administration
- Based on XML standards

However, the cost was far too much to justify. Given that we were just attempting to build a prototype, the Infonyte XQL engine met our needs for a much cheaper cost.

As we stated earlier, the choice of the XML querying language and the query handler are not fixed by any means. Our goal was to build a prototype for proof of concept and with this in mind, we made the best decisions possible under our constraints. Our architecture is flexible enough so that if a better solution emerges in the future (in terms of cost or functionality), it can be easily incorporated into the server to replace the current implementation.

3.2 Processing a Query

Thus far, we have provided a discussion on all of the technologies used in our PML Server: the communication protocol (SOAP), the XML querying language (XQL), the Internet application server (Tomcat), the query handler (Infonyte XQL engine), and the PML database (flat-file system on Unix). We will now present the algorithms and details of how a query is actually processed.

3.2.1 PML File Storage Format

The first point of interest is how the PML files are actually stored. We have already discussed how we store the files in a particular directory on the server to comply with the Infonyte XQL engine. The naming scheme used is a hexadecimal string representation of the 96 bit ePC (or some portion of it) that corresponds to the tag(s) for which the file stores information. We chose this out of simplicity. We were considering storing the files using the bit representation itself, but this would have led to extremely long filenames (96 characters not including file extensions). Furthermore, given the complexity of the entire system and the number of times that an ePC needs to be passed around, the most reliable form that the data unit can take is the string [15]. Using a hexadecimal representation also cuts down the length of the filename to 24 characters, which is the hexadecimal equivalent of 96 bits. Thus, we have the following partition for an ePC (where each X corresponds to a character):

XX	**.XXXXXXX**	**.XXXXX**	**.XXXXXXXXXX**
Header	Manufacturer	Product	Serial Number
8 bits	*28 bits*	*20 bits*	*40 bits*

It works out very nicely since the partitions fall on the nibble boundaries.

We have yet to discuss how the PML server knows where all the files are being stored. We expect the user to supply this information in a configuration file. When the first call is made to a PML Server, it looks in "/etc" directory for a file called "pmlserver.conf." It parses the file looking for the phrase "PMLFilePath:" at which point it expects the absolute pathname to the directory in which the PML files reside on the server. This location is then stored in a local variable so any subsequent calls to the server do not necessitate this expensive file-read operation. The method to parse the configuration file and find the pathname to the files is part of the constructor for the SOAP service in the PML server. This constructor is only called once, when the first SOAP call to the server is made.

Besides just the hexadecimal string of the ePC, each PML file also has a .pml extension. This is transparent to the client, however, because whenever a call is made to the server, it automatically appends a .pml to the tag parameter and then proceeds with the query. The extensions were handled in this manner so that the ePC can still be passed around easily in the system without having to worry about the .pml extension that may be relevant to the PML Server but not other parts like ONS. It should also be noted that the PML Server assumes that any alphabet characters in the filename are lower case. We had to pick lower case or upper caser for consistency when the XQL engine searches for a particular PML file. We chose lower case because the regular expression library we use (discussed in Section 3.3.3) returns filenames with every alphabet character in lower case after matching an expression. The PML Server converts any alphabet tags in the tag field of a query to lower case. This additional step is there in case the client submits a tag with upper case characters.

3.2.2 File Searching Algorithm

There is a nice hierarchy in the structure of the ePC (Manufacturer Code, Product Code, and Serial Number). This also means that information about a particular product may be scattered across a number of PML files in a server. It would be extremely inefficient to store information that pertains to a class of products (such as weight) in every single PML file for those products. Instead, it is likely that there will be just one file that will store this information for the entire class of products. The name of this file would be a subset common to all the ePCs for which it was storing information. A probable length for this file (without the extension) would be 15 characters. This is the total length of the Header, Manufacturer Code, and Product Code:

$$\underbrace{\underset{\text{Header}}{\textbf{XX}}\qquad\underset{\text{Manufacturer}}{\textbf{.XXXXXXX}}\qquad\underset{\text{Product}}{\textbf{.XXXXX}}}_{\text{15 Characters}}\qquad\underset{\text{Serial Number}}{\textbf{.XXXXXXXXXX}}$$

Similarly, some piece of information common to all products made by the same manufacturer (e.g. country of production) would probably be stored in a file of length 9 characters. This is the total length of the Header and Manufacturer Code:

$$\underbrace{\underset{\text{Header}}{\textbf{XX}}\qquad\underset{\text{Manufacturer}}{\textbf{.XXXXXXX}}}_{\text{9 Characters}}\qquad\underset{\text{Product}}{\textbf{.XXXXX}}\qquad\underset{\text{Serial Number}}{\textbf{.XXXXXXXXXX}}$$

A filename does not necessary have to fall on one of the partition borders (9 or 15 characters) and may be any length between 9 and 24 characters. This is because there may be some subdivisions within the Product Code or the Serial Number.

We now present our algorithm for querying files. As mentioned earlier, a call to a PML Server requires two arguments: a tag and a query. The PML Server first looks for a filename that matches the tag (with a .pml extension). If the file is found, then it applies the query. However, if the file is not found or the query result is empty, the next largest substring of the tag is taken and this process is repeated. The PML Server will keep taking the largest substring of the tag and attempting to query the matching file (if it exists) until a non-empty result is returned or a filename equal to some minimum length is tried. This minimum length tried is derived from the length of the ePC Header and Manufacturer Code. Under the current specification, this length is 9 characters. Given a hypothetical tag to query, Figure 3.1 shows which files would be searched and in what order.

Original tag in query:	123456789abcde1234567890
Files searched on server:	123456789abcde1234567890.pml
↓	123456789abcde123456789.pml
	123456789abcde12345678.pml
	123456789abcde1234567.pml
	123456789abcde123456.pml
	123456789abcde12345.pml
	123456789abcde1234.pml
	123456789abcde123.pml
	123456789abcde1.pml
	123456789abcde.pml
	123456789abcd.pml
	123456789abc.pml
	123456789ab.pml
	123456789a.pml
▼	123456789.pml

Figure 3.1: File Searching Algorithm

There is no need to search for a file less than the length of the Header and Manufacturer partition because no information about a product would be stored in a file so broad that it is not even specific to one manufacturer. Information about a particular manufacturer is the most general type that will be stored by any file. The PML Server has a method to determine the minimum length. When it receives a tag and query, it strips off the header from the tag (first 2 characters) and examines it to find out the length of the Manufacturer Code partition. Since the length of the Header will remain constant at 2 characters, the length of the Manufacturer Code will determine the minimum file length the PML Server should search.

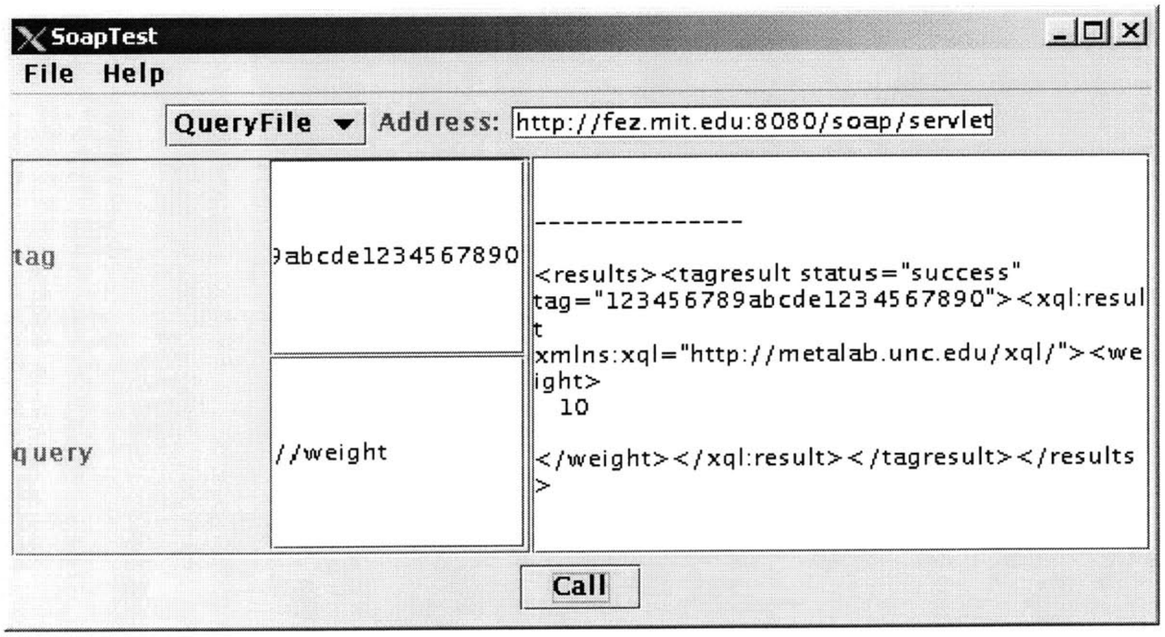
3.3 Possible Query Formats

We have already stated that one of our goals was to decrease the amount of network bandwidth our system used by making use of server side processing wherever possible. The support of multiple query formats helps us achieve this goal. In particular, we allow a user to submit a range of tags as well as a regular expression. In both cases, multiple calls to the PML Server are replaced with just one encompassing call. The PML Server processes all the tags specified and then sends back the cumulative result.

When a tag is submitted to the server, it checks for the "-" character to see if it is a range and handles it appropriately. If it is not a range, the PML Server tests if the tag string contains a "?", "*", or a "+" in order to detect whether the string is a regular expression. If the string does not contain any of the three characters, it is treated as a single ePC. Otherwise, it is handled as a regular expression. We will now discuss how the PML Server handles each of the three possibilities (single ePC, range, and regular expression).

3.3.1 Single ePC

The case of a single ePC is fairly straightforward and is handled using the file searching algorithm described in Section 3.2.2. We built a client graphical user interface (GUI) to easily build client calls and view the results. Figure 3.2 shows the case for a single ePC. The address text box is where the URL for the SOAP service is entered. We named the SOAP service for the PML Server 'QueryFile' which is why it is specified in the drop down menu. The ePC is entered in the tag field and the query is entered in its respective field. The right hand column shows the result of the query. The entire result is embedded in a <results> element. The first child of the <results element> is the <tagresult> element and is specific to one tag. The <tagresult> element has two attributes: status and tag. The status attribute tells whether the query was successful or not. It shows a value of "success" if the desired information was found on the PML Server. It shows a value of "error" if the desired information was not found and is also accompanied by some message explaining the reason why the query failed. This will be discussed in greater detail in Section 3.4. The tag attribute shows the file in which the information was found. Our file searching algorithm makes it possible to find information in a file other than the one actually queried (specifically, some substring of the original file). The XQL Engine adds the <xql:result> element before returning the



```
http://fez.mit.edu:8080/soap/servlet
```

```
9abcde1234567890
```

```
//weight
```

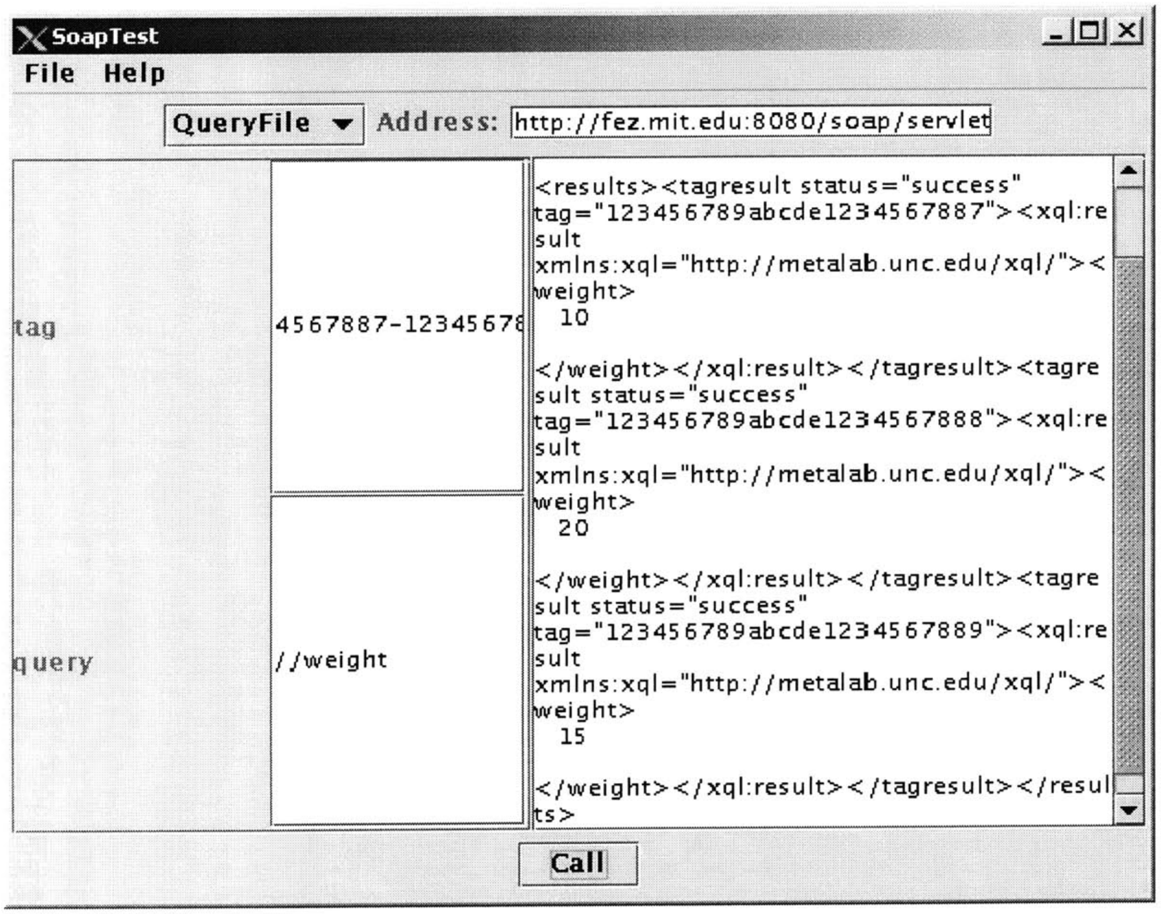
```
---------------

<results><tagresult status="success"
tag="123456789abcde1234567890"><xql:resul
t
xmlns:xql="http://metalab.unc.edu/xql/"><we
ight>
  10

</weight></xql:result></tagresult></results
>
```

Figure 3.2: Single ePC Query

result. It has one attribute, which is just its namespace. In Figure 3.2, the result of the "//weight" query can be seen embedded in the <weight> element. In this case, it has a value of "10."

3.3.2 Ranges

A client call can also specify a range of tags to query. This way, instead of having to query the server n times for each of the n tags in a range, there only needs to be one call made. Upon receiving a range of tags, the PML Server first checks to see if the hexadecimal value of the starting tag in the range is less than the hexadecimal value of the finishing tag in the range. If it is not, an error message is returned. Next, the PML Server extracts the range of n tags and processes each of the n tags individually. Thus, it



Address: http://fez.mit.edu:8080/soap/servlet

tag: 4567887-12345678

query: //weight

```
<results><tagresult status="success"
tag="123456789abcde12345678 87"><xql:re
sult
xmlns:xql="http://metalab.unc.edu/xql/"><
weight>
  10

</weight></xql:result></tagresult><tagre
sult status="success"
tag="123456789abcde1234567888"><xql:re
sult
xmlns:xql="http://metalab.unc.edu/xql/"><
weight>
  20

</weight></xql:result></tagresult><tagre
sult status="success"
tag="123456789abcde1234567889"><xql:re
sult
xmlns:xql="http://metalab.unc.edu/xql/"><
weight>
  15

</weight></xql:result></tagresult></resul
ts>
```

Figure 3.3: Range of Tags

treats this case as if n individual calls to the PML Server were actually made. The results from each of the individual query are compiled into one XML document and are returned to the client. Figure 3.3 shows a sample call using a range of tags.

As expected the entire set of results is contained within a single <results> element. There are three <tagresult> elements, each of which correspond to a tag in the range. In this case, all three queries were successful and the results can be seen in the three <weight> elements.

3.3.3 Regular Expressions

The third query format we support besides a single ePC and ranges are regular expressions. Regular expression matching allows you to test whether a string fits into a specific syntactic shape [16]. You can also search a string for a substring that fits a pattern. A regular expression describes a set of strings. The simplest case is one that describes a particular string; for example, the string 'foo' when regarded as a regular expression matches 'foo' and nothing else. Nontrivial regular expressions use certain special constructs so that they can match more than one string. They contain special characters such as '+', '?', and '*' to specify their syntactic shape.

We use the Regexp package from Apache. It is a 100% Pure Java Regular Expression package and thus, fits in nicely with the rest of our system components. There are many regular expression packages available, but we needed one that was built in Java and had support for the basic types of regular expressions. The full documentation on constructing regular expressions using the Regexp package is given in [17]. An overview of how to build some basic regular expressions, enough for efficient use of the PML Server, will be presented here.

PML files are named using basic alphanumeric characters (the hexadecimal representation of the 96 bit ePC) so there is no need to try to match a certain type of character. The only important one is the period, which matches any one character. For example, the regular expression "d.d" would match "dad", "dbd", "dcd", etc. The other important special characters are those that specify the number of times a character is matched. These are '?', '*', and '+'. There are two types of closures, greedy closures and reluctant closures. Table 3.1 summarizes the details:

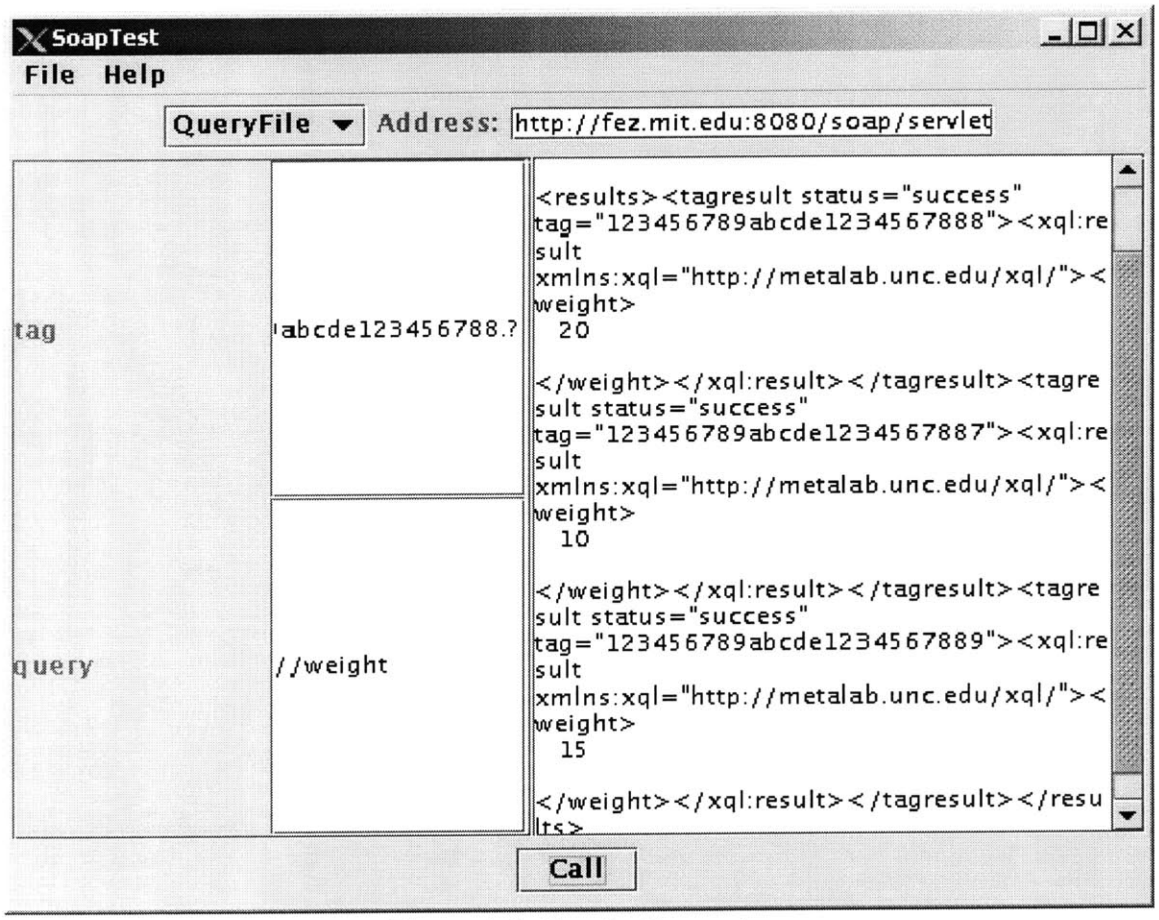
Greedy	Reluctant	Explanation
A*	A*?	Matches A 0 or more times
A+	A+?	Matches A 1 or more times
A?	A??	Matches A 0 or 1 times

Table 3.1: Greedy and Reluctant Closures

All closure operators (+, *, ?) are greedy by default, meaning that they match as many elements of the string as possible without causing the overall match to fail. A closure can be changed to reluctant (non-greedy) by adding a '?'. A reluctant closure will match as few elements of the string as possible when finding matches.

When a PML Server finds that tag parameter is not a range, but contains a '*', "+', or a '?', it is treated as a regular expression. In this case, all the files in the PML Server are matched against the regular expression. Then every file that matches is treated as an individual tag and processed the same way as a single ePC as described in Section 3.3.1. This is not the same as taking a regular expression, extrapolating every string that would match it, and treating each of those possible matches as an individual tag in the query. Although this may be possible to implement if the regular expression only contained a '?' or a '+', it would be impractical to extrapolate all the possible matches if there was a '*' in the regular expression. Thus, we implemented a compromise. While it is not a complete solution, it can still be used effectively to save bandwidth and time.

Figure 3.4 shows a sample use in our system. The regular expression matches three tags and the results are compiled and displayed in the right hand column for each of the three tags. Just like a range of tags is handled, each result is contained in a <tagresult> element. Also, the use of a regular expression saves the user from making three separate PML Server calls. Instead, extra workload taken on by the server makes the client call much easier.



Address: http://fez.mit.edu:8080/soap/servlet

tag: 'abcde123456788.?

query: //weight

```
<results><tagresult status="success"
tag="123456789abcde1234567888"><xql:re
sult
xmlns:xql="http://metalab.unc.edu/xql/"><
weight>
  20

</weight></xql:result></tagresult><tagre
sult status="success"
tag="123456789abcde1234567887"><xql:re
sult
xmlns:xql="http://metalab.unc.edu/xql/"><
weight>
  10

</weight></xql:result></tagresult><tagre
sult status="success"
tag="123456789abcde1234567889"><xql:re
sult
xmlns:xql="http://metalab.unc.edu/xql/"><
weight>
  15

</weight></xql:result></tagresult></resu
lts>
```

Figure 3.4: Regular Expression

3.4 Error Handling

There are many reasons that a query might fail for a particular tag. It could involve a problem with SOAP, the PML file may be badly formed, or the desired information just may not be stored on the server. We investigated some existing error codes, but found that we would have to devise our own. The primary reason was that we are not using a relational database and thus most of the error codes for database systems are not applicable to our system.

The first place that an error might occur in the system is the communication protocol. Fortunately SOAP already has a fairly exhaustive error code system. An error message from a SOAP application is carried inside a fault element. The sub elements of a fault element are shown in Table 3.2 [18].

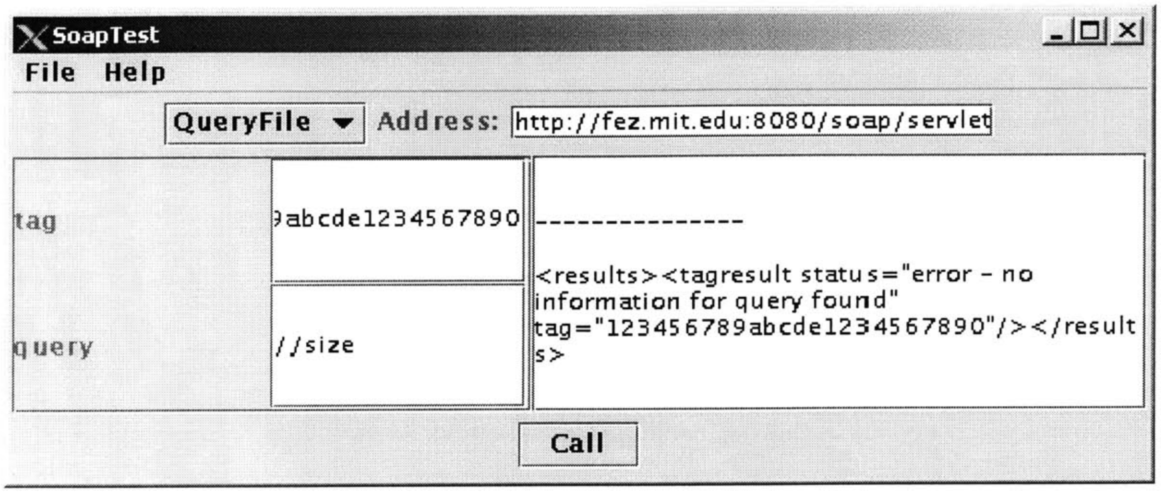
Sub Element	**Description**
<faultcode>	A code identifying the error
<faultstring>	The error as a string
<faultfactor>	Who caused the error
<detail>	Specific error information

Table 3.2: The SOAP Fault Element

The first sub element, faultcode, can take on four values as detailed in Table 3.3 [18].

Error	**Description**
VersionMismatch	Invalid namespace for the SOAP Envelope element
MustUnderstand	A child element of the Header element, with the mustUnderstand attribute set to "1", was not understood
Client	The message was incorrectly formed or contained incorrect information
Server	There was a problem with the server so the message could not proceed

Table 3.3: SOAP Fault Codes



```
http://fez.mit.edu:8080/soap/servlet
tag     9abcde1234567890
query   //size
---------------
<results><tagresult status="error - no
information for query found"
tag="123456789abcde1234567890"/></result
s>
```

Figure 3.5: Query Failure

To handle failure messages with anything that may occur in the backend, we make use of the status attribute in a <tagresult> element. As describe earlier, it has a value of "error" if the desired information is not found and is also accompanied by an explanation of why the query failed. Figure 3.5 shows an example of such an error message with a client call querying for size information that does not exist in any of the files searched. In this case, the error message returned reads "error – no information for query found." Similarly there are other specific error messages for various reasons a query may fail. Table 3.4 summarizes the error cases and messages that we have implemented.

Error Scenario	**Error Message in Status Attribute**
finish tag for a range comes before the start tag	error – badly formed tag request
improper regular expression	error – badly formed tag request
no matches for a regular expression	error – no matching files found
queried PML file is improperly formed	error – badly formed PML file for "whichever tag was queried"
server cannot find desired data	error – no information for query found

Table 3.4: Error Messages

The error handling we implemented covers the basic reasons why a query may not return a result. Additional restrictions on types of queries or possible values for the tag field can be added easily in this manner by inserting a check in the code and returning the proper error message.

Chapter 4: Tracking Server

The Tracking Server stores a particular type of instance data. Namely, it stores and delivers information related to the location of a product. If a PML Server is queried for location information, it will need query a Tracking Server to get the desired data. Under the Auto-ID system, a Tracking Server will reside at every node in the supply chain. Thus, there would be one at every manufacturer, distributor, and retailer. Client queries to a Tracking Server may require recursive calls to other Tracking Servers to gather the requested information. For instance, a client may request the full history of a particular tag in the supply chain (i.e. every node that it visited). The implementation of the distributed algorithms used to handle such tracking queries is described in [19]. We make use of them as part of a 'Tracking Package,' but will treat it as a black box in this thesis and not provide any further details. Here, we will be focusing on our implementation of the Tracking Server only. We first provide a discussion of the architecture of the Tracking Server and then describe how we use the Tracking Package in our implementation.

4.1 Architecture

As in the PML Server, there are two major parts to the Tracking Server architecture: the communication protocol and the server side technologies. Although SOAP is still used as the communication protocol, we use a relational database (MySQL) to store the data. Thus, the Structured Query Language (SQL) is used as the querying language. The query handler is a Java Database Connectivity (JDBC) driver, the MM MySQL Driver.

4.1.1 Communication Protocol

We chose to use SOAP as the communication protocol, just as we used it in the PML Server. A detailed discussion with an overview of SOAP and justification for its use is provided in Chapter 2. Using SOAP is an added convenience since the PML Server and the Tracking Server will need to communicate with each other. This communication becomes is trivial if both servers are using the same communication protocol. The likely scenario is for a PML Server to be queried for location information. Since this type of instance data is stored in the Tracking Server, the PML Server will in turn make a call to the Tracking Server and upon receiving the desired location data, return this in a newly constructed XML file. The PML Server will make the call to the Tracking Server using SOAP.

4.1.2 Server Side

The server side is where the Tracking Server begins to deviate from the PML Server. Apache Tomcat is still used as the Internet application server since it works well with SOAP, the communication protocol. Further details and justification for using Tomcat can be found in Section 3.1.1.

The tracking information is stored in a relational database, MySQL. A relational database offers a very structured and efficient way to store data. Storing the data in separate tables rather than putting it all in one storeroom offers speed and flexibility. The information can be accessed very quickly and efficiently. We chose MySQL because it is simple, free, and the most widely used open source relational database available. We did not need a database management system that was too complex or had a great amount of functionality besides efficient storage and retrieval of data. Furthermore, the connectivity, speed, and security make MySQL well suited for access through the Internet [20].

Naturally, the language needed to query a MySQL database is SQL. This is the most standardized language to access databases. As we will discuss in the next section, a client does not make a call to the Tracking Server with an SQL query. Instead, a client calls specifies a method corresponding to its desired information. The only parameter needed by each of these SOAP services is the tag in question.

Since the client does not supply the SQL queries, they are built into the server side code for each of the methods supported. This requires the use of a JDBC driver. It allows for connections to a relational database through a Java program (our server side code). The most commonly used and most stable JDBC driver is the MM MySQL JDBC driver [21]. Again, we did not require anything too elaborate or with great functionality. This driver was known to be stable and was more than sufficient.

4.2 Processing Methodology

Now that we have discussed the architecture and the technologies used to implement the tracking server, we can present how each of the parts are used with one another. First, we present the details relevant to the data storage. Then we discuss how a call to the Tracking Server is handled.

4.2.1 Data Storage

We use a relational database with multiple tables to store the location data. Specifically, the use of the Tracking Package described in [19] necessitates the use of two tables. One is called 'incoming' and stores information about all tags that have arrived at the particular node in the supply chain. The other is the 'outgoing' table and stores information about all tags that have left the Tracking Server's node. The structure of these two tables is nearly identical. Tables 4.1 and 4.2 give the names of each of the five fields in the tables and what they store.

first	last	next	stock	exp
first tag in the range	last tag in the range	previous node the tag visited	total number still active in supply chain	expiration date

Table 4.1: Incoming Table

first	last	next	stock	exp
first tag in the range	last tag in the range	next node the tag visited	total number still active in supply chain	expiration date

Table 4.2: Outgoing Table

The first two fields are the 96 bit ePC in hexadecimal format. Thus, they are each 24 characters. Since it is likely that shipments of the same product will have sequential numbering (consecutive ePCs), we lessened storage requirements by storing the entire range of tags in the group as one record (row) in the table. It would be an unnecessarily expensive operation to store each tag in the range as an individual record in the table. The third field, 'next', is a URL for the SOAP service of another Tracking Server in the supply chain. Depending on the table, this represents either where the tag came from (for the incoming table) or where it went (for the outgoing table). We use an IP address instead of a URL because one URL may have multiple IP addresses. Stock is an integer that cannot exceed the range dictated by the 'first' and 'last' fields. Finally exp is a date in XXXX-XX-XX (year-month-day) format.

4.2.2 Tracking Package

The Tracking Package implements the distributed algorithm behind each querying function. Details about it can be found in [19]. In order to use the Tracking Package in the actual implementation of the Tracking Server, we implemented a set of methods involving database reads, database updates, and calls to other Tracking Server. We implemented two tracking queries: `track` and `locate`. `Track` returns the full history of every node a tag has visited starting with its current location. `Locate` returns the current location of a tag. Any methods that require access to the local information at a Tracking Server were implemented using the MM MySQL JDBC driver to query the local MySQL database. The algorithms in the Tracking Package require calls to other Tracking Servers if the desired information is not found in the local database. These calls were implemented using the SOAP communication protocol.

Currently, only the `track` and `locate` methods have been implemented but as other algorithms and methods are developed in the Tracking Package, they can easily be added to the SOAP service that is deployed on the Tracking Server. This SOAP Service is called 'TrackFetcher'. Besides the supported `track` and `locate` methods, there are two additional methods it needs to operate. One returns the name of the database in MySQL that has the 'incoming' and 'outgoing' tables. The other returns the URL of the local SOAP service. These two pieces of information are needed in order to properly pass information between Tracking Servers. As in the PML Server, we make use of configuration files to store the database name and SOAP URL. This allows users to easily change this information later. Thus, the server looks for a 'trackingserver.conf' file in the '/etc' directory. It should have the format shown in Figure 4.1.

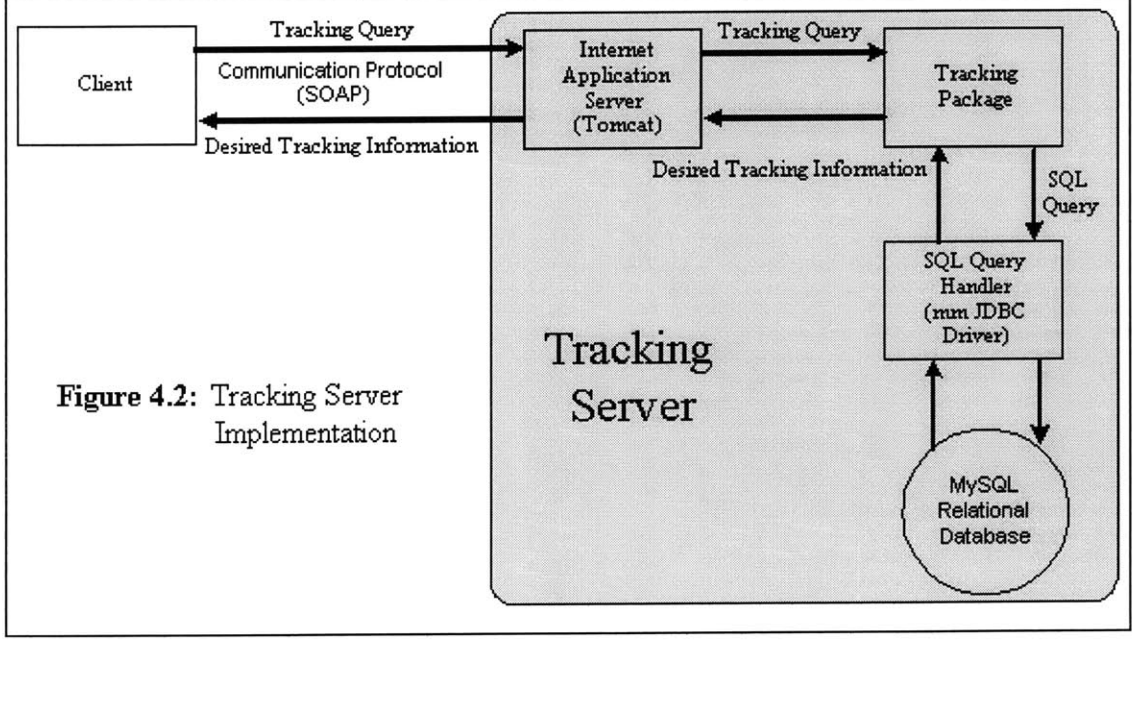
```
DatabaseName: trackingserver
SoapServiceUrl: http://18.0.236.18:8080/soap/servlet/rpcrouter
```

Figure 4.1: Tracking Server Configuration File

A Tracking Server with this configuration file would expect the 'incoming' and 'outgoing' tables in the 'trackingserver' database. Furthermore, it would expect its SOAP service to be deployed at the URL given. The user can easily modify this file if any of this information changes.

The structure of the Tracking Server is similar to that of the PML Server. Both use one SOAP service. Whereas the PML Server only has one method that a client can call in its SOAP Service (to query a file), the Tracking Server has two (this number will increase as more methods are implemented in the Tracking Package). When a client makes a call to the PML Server, it specifies its desired information with a tag and a query. In the case of Tracking Server, a client gains access by specifying a tag and a method (for the type of tracking information it desires). Figure 4.2 summarizes the implementation of the Tracking Server.

The diagram shows that the client does not know which database is used on the Tracking Server. Only the Tracking Package interfaces with the database. This leaves



Client
Tracking Query
Communication Protocol (SOAP)
Desired Tracking Information
Internet Application Server (Tomcat)
Tracking Query
Tracking Package
Desired Tracking Information
SQL Query
SQL Query Handler (mm JDBC Driver)
Tracking Server
MySQL Relational Database

Figure 4.2: Tracking Server Implementation

our design flexible in case we want to change it in the future. One point to note in Figure 4.2 is that before the Tracking Package returns the desired tracking information, it may need to make a SOAP call to another Tracking Server. This may be a recursive process where the second Tracking Server may need to call a third and so on. Thus, one call to a Tracking Server may involve calls to many Tracking Servers. The client in Figure 4.2 may be some process making a direct call, a PML Server, or a Tracking Server.

Chapter 5: Conclusions and Future Work

In this chapter, we present some of the problems we faced when building the PML and Tracking Server. Most of the issues we faced were related to the use of relatively new technologies in implementing the two servers. We will also include a discussion regarding future work for both servers. Finally, we will present our conclusions.

5.1 Challenges

Some of the key technologies we used for the PML Server and the Tracking Server have not been used extensively as they are very new. This posed many challenges in our implementation. They revolved around a lack of documentation and a lack of standards.

5.1.1 New Technologies

The communication protocol used for the PML Server and the Tracking Server (SOAP) and the query handler used for the PML Server (the Infonyte XQL Engine) are both technologies that have only recently been developed. In fact, both are still actively being developed.

The greatest challenge in using SOAP was that at the time of our implementation there was little or no documentation available. Simply installing the software package took significant effort, let alone trying to use it. There are now numerous well written tutorials on the Internet describing how to use SOAP, but they were not available until just recently. Instead of documentation, we were forced to rely on the SOAP user community mailing list for relevant details. This was challenging because information retrieval took days and many times there was no answer to our questions. Our original implementation of the PML Server was with Apache SOAP version 2.0. However, when version 2.1 was released, we upgraded our system. There were some minor compatibility issues between the two versions that needed to be ironed out. Again, this was one of the struggles of working with software that was still in development.

The Infonyte XQL Engine was another key element of the PML Server. It too was going through some major development while we were implementing our solution. The XQL Engine began as a research project at Darmstadt, but a startup company, Infonyte, was soon spun off to continue development. This structural change made it very difficult to get support for the product. The first version we received did not support the XML parser (Xerces) that SOAP required. We were forced to contact Infonyte and after some time were fortunate enough to get a version that did support Xerces. We also wanted to make use of some additional functionality called the Persistent Document Object Model (PDOM), a caching mechanism for DOM pages. However, the Infonyte software was going through some major changes so we deemed this feature unstable for our implementation. It will be discussed in greater detail in Section 5.2.

5.1.2 Lack of Standards

One of the major issues in working with XML related technologies is a lack of standards. This became an issue when choosing our XML querying language and the corresponding query handler. Because the W3C (World Wide Web Consortium) has yet to decide on a standard for an XML querying language, most database vendors have held off from supporting XML querying languages. This is why we did not have a relational database but instead had implemented the PML Server with a flat file database. Once there is a standard, most database vendors will likely build a front end for the chosen XML querying language.

As discussed in Chapter 3, we were forced to choose between XML-QL, Quilt, and XQL as the XML querying language. At the time, it appeared that XQL was emerging as the likely standard. This is why we went with XQL and the appropriate XQL Engine. However, it now appears that Quilt will be chosen as the standard [22]. Thus, the implementation will probably have to be changed. Because there was not a standard, we were forced to go with our best guess at the time.

5.2 Future Work

The PML and Tracking Server have room for both additional functionality and performance optimizations. Future work on the PML Server revolves around the PML database. The Tracking Server only has two tracking methods implemented but others have been developed. Future work will be to implement these new functions.

5.2.1 PML Database

Currently, a flat file database is used to store information on the PML Server. Since we use XQL to query the files, we utilize an XQL engine to handle the queries. This querying language/query handler/database combination will eventually need to be changed, especially when the W3C makes its decision on a standard XML querying language.

A performance optimization that can be made without changing the database or the querying language is to upgrade the Infonyte XQL Engine. Recently, Infonyte released a new product, the Infonyte DB. This is based on two technologies: the Persistent Document Object Model (PDOM) engine and the XQL engine. We have already implemented the PML Server using the XQL engine, but the use of the PDOM engine would increase performance of the server. The Infonyte PDOM engine allows the storage and retrieval of XML-documents in and from binary database files where each file represents one document. Documents are parsed once and stored in binary form, accessible without the overhead of parsing them again. The PDOM engine swaps XML documents between the main memory and the binary database files transparently for the application. The engine also has a self-optimizing cache architecture to further increase performance. This approach scales beyond the limitations of main memory. It offers a lightweight alternative to heavyweight database management systems.

Upgrading the Infonyte XQL engine would still necessitate the use of XQL as the querying language. However now that it is looking like Quilt, not XQL, will become the W3C standard it may be a better idea to change querying languages and use a product like Kweelt. As discussed in Section 3.1.3, Kweelt is a framework to query XML data and among other things, it offers an evaluation engine for the Quilt XML query language [13]. It offers multiple backends because its query evaluator does not impose any specific storage for the XML files being stored, but instead relies on a set of interfaces. This gives a lot of flexibility for the storage medium, including a relational database (MySQL). Kweelt may be a good alternative to Infonyte because not only does it make use of Quilt, but it also allows the use of a heavyweight DBMS which may be necessary for the large amount of data stored in PML Servers.

One alternative to completely replacing the Infonyte engine with Kweelt is to use both technologies. The PML Server would then support two dominant XML querying languages: XQL and Quilt. Depending on which query was submitted to the PML Server, the SOAP service would pass the arguments to the appropriate query handler. This implementation may be ideal if it appears that no standard will emerge for XML querying languages for some time. If a standard is foreseeable, however, implementing both query handlers may be excessive.

When the W3C releases its standard for XML querying languages, it is likely that most heavyweight DBMS solutions will begin supporting this querying language. At that point, the ideal implementation will include the use of a heavyweight database solution. Ultimately, we do not want the PML database to be a collection of XML text files, even if the files are cached. We want a PML database that can map the XML logical structure into an efficient data-structure. Once this is done, the database system can perform inverse mappings of its datastructure and generate XML documents [1]. Hopefully, one of the solutions offered by a vendor in the future will satisfy this constraint once an XML querying language standard is issued.

5.2.2 Tracking Server Methods

The current implementation of the Tracking Server supports two types of client calls: track and locate. Track returns the full history of a tag and locate returns the current location of a tag. These are the only two methods supported in the current implementation because these were the only two methods whose algorithms were implemented in the Tracking Package. However, in the future there will be other algorithms implemented in the Tracking Package [4]. It will then be necessary to implement those methods in the Tracking Server.

5.3 Conclusions

In this thesis, we presented our design and implementation of the PML Server and the Tracking Server. Information about each tagged object in the Auto-ID system is stored in the form of one or more PML files in a PML Server. This information may be about a class of products or a unique product. The Tracking Server stores data relevant to a particular object's location. We had to carefully evaluate and choose an appropriate communication protocol and server side technologies for each server.

For both servers we chose SOAP as the communication protocol. We found it to be well suited for use over the Internet, simple, and flexible. Furthermore, since it is built using existing technologies (XML and HTTP[1]) instead of inventing new ones, incompatibilities are less likely to arise. HTTP is already widespread while XML soon will be. From the perspective of the Auto-ID Center, a flexible technology was desired. We did not want to force our users to use a particular machine in order to be compliant with our system. SOAP offers this flexibility. Furthermore, it is simple and does not warrant the use of new technologies. This minimizes the infrastructure that needs to be built.

We chose Tomcat as the Internet application server for both the PML Server and the Tracking Server because it was the most commonly used in conjunction with SOAP. Given that SOAP was a new technology with very limited documentation, choosing Tomcat was quite beneficial because we were able to solicit substantial help from the SOAP community. Since Tomcat was already known to work well with SOAP, it was a logical choice.

In the PML Server, we chose XQL as the querying language, the Infonyte XQL engine as the query handler, and a flat file database system. The choice of the querying language and the query handler went hand in hand. Given that there is no standard for querying languages, very few databases had front-end support for an XML querying

[1] Although we use HTTP in our implementation, SOAP does not preclude the use of other transport protocols.

language at the time of our decision. We found XQL to be perhaps the only querying language with some type of support in the form of an XQL engine built by Infonyte. This is primarily why we chose it. The XQL engine uses only a flat file database. However, the architecture of the PML Server is flexible enough so that new querying language/database pair can be easily "plugged" into the PML Server and replace the current implementation when a standard is released. Our goal was to build a prototype of the PML Server so we chose technologies that were most widely used with the most support.

The technologies used in the Tracking Server turned out to be very different from the PML Server. A relational database, MySQL, is employed to store the data. A more heavyweight database was desired because there will be an enormous amount of information stored at each of these servers and there is a lot of overhead with PML files. Thus, the querying language is SQL. However, this is transparent to the client because it just needs to call the relevant tracking method and the database queries are built into these methods. The MM MySQL JDBC driver is utilized as the query handler. Its stability and common use justified its choice. The Tracking Package described in [19] was used for the code design and the distributed algorithms necessary to implement the tracking methods.

We designed and implemented simple, flexible solutions for the PML and Tracking Server. We succeeded in our primary goal, which was a proof-of-concept of one part of the Auto-ID infrastructure. We built stable prototypes that can be further developed and extended to be part of the robust automatic identification system envisioned by the MIT Auto-ID Center.

Appendix A: Building a PML Server

There are six major components to the PML Server:

a) Simple Object Access Protocol (SOAP)

b) Apache Tomcat Java Servlet Engine

c) Infonyte XQL Engine

d) Apache Xerces XML parser

e) Jakarta Regular Expression package

f) PML Soap Service

We will show how to setup the necessary components and build the PML Server from the bottom up. First we assume, that each of these pieces has been downloaded from the Web or attained otherwise. For ease of explanation, we will assume that each of the six components has been placed under a super directory `/foo/PMLServer`. Thus, `PMLServer` has six directories:

a) soap

b) tomcat

c) infonyte

d) xerces

e) regexp

f) pmlsoapservice

We assume that each of the respective distributions have been renamed to correspond to these directory names. Next SOAP and Tomcat need to be properly configured. First, the `tomcat.sh` file in the `/foo/PMLServer/tomcat/bin` directory needs to be modified. After line 111, the following line will be seen:

```
CLASSPATH=path-to-xerces/xerces.jar:${CLASSPATH}
```

The "path-to-xerces" portion needs to be replaced with /foo/PMLServer/xerces so that the line reads:

```
CLASSPATH=/foo/PMLServer/xerces/xerces.jar:${CLASSPATH}
```

Next, the `server.xml` file in the `/foo/PMLServer/tomcat/conf` directory needs to be modified. The easiest way to do this is to add a new context to the file as follows:

```
<Context path="/soap" docBase="/foo/PMLServer/soap/webapps/soap"
         debug="1" reloadable="true">
```

All that is left to finish the configuration is to properly set the necessary environment variables.

The environment variable JAVA_HOME needs to point to the root directory of the JDK hierarchy (the Java interpreter needs to be added to the PATH variable as well).

The environment variable TOMCAT_HOME should point to the root directory of the Tomcat hierarchy. In our case it would be `/foo/PMLServer/tomcat`.

Lastly, the CLASSPATH variable must include the following items:

- /foo/PMLServer/xerces/xerces.jar
- /foo/PMLServer/tomcat/lib
- /foo/PMLServer/
- /foo/PMLServer/soap/lib/soap.jar
- /foo/PMLServer/infonyte/libs/xqlpdom.jar
- /foo/PMLServer/regexp/Jakarta-regexp.jar

The CLASSPATH variable is the one that causes the most trouble so the following points are important:

a) Xerces needs to be the first item in the classpath
b) the filenames will vary from distribution to distribution so they should be checked before adding them to the classpath
c) if running version 2.1 of SOAP, the mail.jar and activation.jar files need to be downloaded from the SUN site and added to the classpath as well

Once the environment variables are set, Tomcat is ready to be started. This can be done on a UNIX machine by going to the `/foo/PMLServer/tomcat` directory and typing the command:

```
bin/startup.sh
```

To stop Tomcat, the following command needs to be typed (in the same directory):

```
bin/shutdown.sh
```

All that is left is to deploy the PML SOAP service. This can be done by going to the `/foo/PMLServer/pmlsoap` directory and typing:

```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter deploy
DeploymentDescriptor.xml
```

The PML Server is now ready to receive client requests.

Appendix B: Building a Client for the PML Server

To use the client we built for the PML Server, there are only three components needed:

a) Simple Object Access Protocol (SOAP)

b) Apache Xerces XML Parser

c) PML Client GUI code

First we assume, that each of these pieces has been downloaded from the Web or attained otherwise. For ease of explanation, we will assume that each of the three components has been placed under a super directory `/foo/PMLClient`. Thus, `PMLClient` has three directories:

a) soap

b) xerces

c) pmlclientgui

We assume that each of the respective distributions have been renamed to correspond to these directory names. Next, the CLASSPATH environment variable needs to be set properly. It must contain the following elements:

- /foo/PMLClient/xerces/xerces.jar
- /foo/PMLClient/soap/lib/soap.jar

A couple of points to note about environment variables are:

- Xerces needs to be the first item
- It is assumed that a Java interpreter has been added to the PATH variable. It works best if this interpreter is the same version as the one on the server side.

The client GUI is now ready to be deployed. Go to the `/foo/PMLClient/` directory and type:

```
java pmlclientgui/SoapTest
```

This should launch the GUI. This GUI itself contains a field for the tag, query, and soap URL of the desired PML Server.

References

[1] Niranjan Kundapur, *On Integrating Physical Objects with the Internet*, S.M. Thesis, MIT, 2000.

[2] MIT Auto-ID Center, [http://auto-id.mit.edu].

[3] David Brock, *The Physical Markup Language*, 2001.

[4] Ching Law, *Tagged Object Tracking in Distribution Networks*, 2001.

[5] The Computing Sciences Organization at Berkeley Labs, "Soap Info", [http://www-itg.lbl.gov/~dang/soapInfo.html].

[6] World Wide Web Consortium, "Simple Object Access Protocol", [http://www.w3.org/TR/SOAP], 2001.

[7] Don Box "Guide to SOAP," [http://msdn.microsoft.com/msdnmag/issues/0300/soap/soap.asp], 2000.

[8] Apache SOAP Document, [http://parikhfamily.virtualave.net/soap.htm].

[9] Apache Tomcat Site, [http://jakarta.apache.org/tomcat].

[10] World Wide Web Consortium, "Extensible Stylesheet Language", [http://www.w3.org/TR/XSL], 2000.

[11] World Wide Web Consortium, "XML Query Language", [http://www.w3.org/TandS/QL/QL98/pp/xql.html], 1998.

[12] Infonyte.com, "Infonyte XQL Suite", [http://www.infonyte.com].

[13] Penn Database Research Group, "Kweelt", [http://db.cis.upenn.edu/Kweelt/].

[14] Software AG, Inc., "Tamino", [http://www.softwareag.com].

[15] Jocelyn Tait, *Data Succession*, 2001.

[16] University of Kentucky, "Regular Expression Overview", [www.uky.edu/ArtsSciences/Classics/regexp.html].

[17] The Jakarta Project, "Jakarta-Regexp 1.2 API", [http://jakarta.apache.org/regexp/apidocs/].

[18] W3Schools.com, "SOAP Tutorial", [http://www.w3schools.com/soap/].

[19] Paritosh Somani, *Real Time Tracking in Distribution Networks*, M.Eng. thesis, MIT, 2001.

[20] MySQL, [www.mysql.com].

[21] M3ySQL, “MM MySQL JDBC Drivers”, [http://www.worldserver.com/mm.mysql].

[22] Sun Microsystems, Inc., MIT Auto-ID Center, “Discussion” March 2001.