Subscribing to a Publisher Subscriber System with Dynamic HTML

Master of Science Thesis IT Program

MIKAEL FROSTHAGE

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, Augusti 2010

Subscribing to a Publisher Subscriber System with Dynamic HTML

MIKAEL FROSTHAGE

Examiner: MARINA PAPATRIANTAFILOU

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2010

Abstract

Middleware is software which acts as a bridge between different kinds of systems which are not able to communicate with each other. This thesis presents how a webpage can subscribe to different events which are generated by an arbitrary system that does not communicate with the HTTP protocol. In order for the webpage to fetch data, it communicates with webservices which act as middleware.

Three different ajax techniques are evaluated on how they fetch data without doing any postbacks, periodic refresh, long polling and HTTP streaming.

On the back end side, two different publisher subscriber specifications are evaluated: WS-Eventing and WS-Notifications.

The result of the thesis is a prototype system where the webpage uses the periodic refresh technique to retrieve data from WS-Notification webservices.

Sammanfattning

Middleware är mjukvara som agerar som en brygga mellan olika system som inte kan kommunicera med varandra. I detta examensarbe presenteras hur en websida kan prenumerara på olika händelser som genereras av ett godtyckligt system som inte kommunicerar med HTTP protokollet. För att kunna hämta data kommunicerar websidan med olika webservicar som aggerar som middleware.

För att hämta data från webservicen utan att göra post backs så utvärderas tre olika ajax tekniker. Periodic refresh, long polling och HTTP streaming.

På back end sidan utvärderas två olika publisher subscriber specifikationer för webservicar, WS-Eventing och WS-Notification.

Resultatet av exjobbet är ett prototytpsystem där websidan använder sig av Periodic refresh för att hämta data från WS-Notifcation webservicar.

Keyword: Middleware, AJAX, Webservices, WS-Eventing, WS-Notifcation

Preface

Mikael Frosthage studies Computer Science at Chalmers University of technology in Gothenburg Sweden. This thesis is part of his master's degree. The work was performed at Manodo's office in Gothenburg during autumn 2005 and spring 2006.

The author would like to thank Tobias Ahnoff and Olof Zanden at Manodo and Marina Papatriantafilou at Chalmers. Furthermore the author would like to thank his mother, father, Thina and his daughter Minja for all the support before, during and after the thesis work. He would also like to thank Petter Bergström for giving valuable advice and motivation. Most of all he would like to thank Edward A. Cerullo who put a huge effort into proof reading the thesis and raised it to another level.

Contents

Chapter 1

Introduction

Background

The majority of all modern desktop applications are implemented with a framework supporting the publisher subscribe pattern such as .Net or Java Swing. For example, when a user is operating an application and clicks a button on a window form, an event is fired that executes the functionality of the button. In other words, that piece of code subscribes to the click event and performs some logic when the event is fired. This is a stark contrast to the way pure HTML based web pages behaves, where all the input elements are posted to the web server which basically processes the whole software logic for that page.

With the advent of dynamic HTML, web pages have become more and more complex. Buttons can trigger an event which executes JavaScript code locally on the web browser. This has gradually made the difference between desktop applications to start to disappear where web pages perform more and more complex task. One might even consider these web pages as much of an application as a desktop application, but with a browser based front end.

One difference between web based and desktop applications is still the fact web based applications, without embedded software, can only communicate with the HTTP protocol. Furthermore it is not possible to establish a connection with a web browser and send data to it. Therefore it is not possible for a web browser to actually subscribe to events on another process.

However the important thing is the user perceives it as if the web page is subscribing to events from an external data source, even if the events are not actually pushed to the web page.

Manodo

Manodo is a software company which builds and customizes web based software systems for individual measurements of heat, water and electrical consumption targeted towards real estate and energy companies. All the monitored data updates in real time, and the software is flexible and able to monitor a wide range measuring devices.

Problem description

Manodo's current web portal relies on an embedded Java applet running in the clients' web browsers. The purpose of the java applet is to fetch real time data, without post backs, from a server which is reading the current states of the measuring devices. The actual server does not communicate with the HTTP protocol, rather the server and the applet communicate via intermediary software.

This implies the java runtime libraries have to be installed on all the client computers. Since Windows XP however, java runtime is not shipped with the installation, and some costumers refuse to install extra software for security reasons. This forces Manodo to develop a new solution that relies on more generic components.

Furthermore the intermediary software is written by a third party company and Manodo does not have access to the source code. Besides that obvious problem, the current intermediary software does not scale geographically, as it is just an adapter between one server and several clients. It would be preferable if there were a many to many relationship between clients and servers.

Goal

The goal is to build a publisher subscriber prototype system where dynamic HTML web pages act as subscribers. Events should be fired on a regular basis, signaling different state changes, and the events are supposed to be propagated to the subscribers. The users watching the pages should see the state changes almost immediately, and the web page code must only be constructed with standard components. No embedded objects, such as Java applets, can be used.

Rather than being a simple adapter between subscribers and publishers, the prototype system should be able to support a many to many relationship between them. By doing this, a subscriber should be able to subscribe to several events produced by different publishers.

Overview

This thesis describes the publisher subscriber prototype system which was developed.

In chapter 2 the core concepts are described, these include middleware, webservices and publisher subscriber specifications for webservices.

Chapter 3 analyses different approaches to implement the concepts described in chapter 2.

Chapter 4 covers the work method during the thesis.

Chapter 5 describes the general design of the prototype.

Chapter 6 describes in more detail how the prototype was developed.

Chapter 7 discusses the result, makes a conclusion and presents future work.

Chapter 2

Concepts

2.1 Client Server Model

The client server model consists of two parts, the server and the client. The server is a process which implements a service such as a file server or a database. The client makes requests to the server, which in turn, returns data. This client server interaction is sometime called "request-reply behavior". [1]

If the underlying network is a relatively reliable network, such as a LAN for example it might be a good idea to implement a "connectionless protocol" since these protocols are more efficient. [1] The disadvantage with a connectionless protocol is twofold. It does not guarantee a package reaches the destination, and it does not guarantee the package has become corrupted.

An alternative to the connectionless protocols are the more reliable "connection oriented protocols". The principle behind these protocols is the client first establishes a connection to the server before a message is sent. Later on the server, the same connection is used to send the reply. It is unfortunately very time consuming to open and close connections in this manner, in particular when it comes to cases where the request and reply messages are small. [3]

2.1.1 Tier Levels

In contrast to what one might believe, clear definitions of clients or servers really are not as obvious as it first seems. There have actually been debates and controversies regarding the substantive differences between them. For example, a server for a distributed database might act as a client and forward requests to file servers, which stores the physical tables. Instead of dividing functionality into server and client components, it is easier to divide them into three different levels which define functionality. The three levels are:

1. The user interface level
2. The processing level
3. The data level

The user interface level handles all the interaction with the users, the processing level is the actual application and the data level contains the data. [1]

2.1.1.1 The User Interface Level

Normally the user interface is implemented on a client. This level acts as the front end, and it contains the software users use in order to interact with distributed systems. The variance in sophistication between different user interfaces is today huge, and spans from console application to advanced graphical user interfaces. [1]

2.1.1.2 The Processing Level

As previously mentioned, the processing level is the actual core of the system. One example might be a search engine where the user interface level is the web browser and the data level is a database with preindexed web pages. In this case the processing level transforms the user input to one or more queries which fetch data from the level. After that, the result is ranked and returned as a list with addresses to different web pages.

Another example is a system for stock brokers where the user interface level is implemented in some desktop application, and historical stock data is stored in the data level. Here the processing levels might perform advanced computations on financial data. [1]

2.1.1.3 The Data Level

The data level, or resource management level, is the level which contains the data used by the distributed system. Usually the data level is a fully fledged database, however there are exceptions. For example the database might be a file system or something similar. [1]

In traditional business oriented environments, the database is organized as a relational database, with a keyword that is data independent. The data is organized independently of the application, and changes to either level do not change the other. In some cases it might be unsuitable to save data within a relational database. These cases are characterized by the fact the data is saved in the form of various complicated data types, and they are therefore better suited to be saved in ordinary objects. For example, CAD systems, with their complex graphical objects such as polygons, are clearly unsuitable to be saved in a relational database. Another example is multimedia, where it is problematic to save data in the form of tables and relations. [1]

In the cases where the data is more suitable for object manipulation, the data level is ideally implemented as an object oriented database. Such databases not only contain the objects, but also the operations which are executed upon them. Some parts of the logic that could have been implemented in the processing level could thus be moved to the data level. [3]

2.2 Client Server Architectures

By distributing a system into several levels, it is possible to physically distribute a client server architecture across several computers in different ways. The simplest form is to just distribute the system to two different computers.

1. A client computer that contains the entire interface level or a part of the interface level.

2. A server computer that contains the processing and the data level.

The problem with this architecture is the system is not really distributed, since everything is processed by the server, while the clients are just dumb terminals. There are multiple alternatives to this architecture. [3]

2.2.1 Multitiered Architecture

In the previous example, a distinction is made between two different computers that contain the client and the server. This is a two tiered architecture. One way to organize the clients and the server in a two tired architecture is to distribute the software logic over the different levels across different computers. For example the entire user interface level can be implemented on the clients, however in some cases; some part of the processing logic can also be implemented on the clients.

Furthermore even the data level can be distributed to the client computer. Modern web browsers have in practice implemented a part of the data level in the form of the cached memory. [2]

In a multitiered distributed system however, computers might act both as a client and as server. For example, one computer might host a part of the processing level which administers transactions against the data level. Here the interface level perceives this computer as a server with which it interacts. The data level on the other hand is serving the processing levels request. This example describes a three tiered architecture. [2]

2.2.2 Horizontal Architecture

So far all the examples have been demonstrated from a vertical perspective. In this perspective, different kinds of software components have been placed on different machines. Another way to balance the load is to distribute in a horizontal perspective. Instead of distributing different parts of the system, equivalent parts are distributed, and they work with the same data, thus sharing the work load. An example of where horizontal distribution is appropriate is where a webpage is overloaded and needs extra computing power. Each request can then be processed by a component which forwards the request to a server in server farm according to a round robin policy. When updates are to be made, the updates are made to all the servers concurrently. [2]

2.2.3 Communication in an Information System

In an information system the software components communicate via synchronous or asynchronous calls. Synchronous communication is blocking; when a thread makes a call, the thread must wait for a reply before it might proceed. The advantage with synchronous communication is that it is much easier to follow what is happening logically in the code. Due to this fact synchronous communication dominates almost all forms of middleware. A middleware is a software that acts as bridge between two or more kinds of software by translating and changing information between them, without the other applications knowing of each other. [2]

The disadvantage with synchronous communication is the threads are not able to perform other operations while waiting for a reply. This might be extremely noticeable, resulting in long waiting times, if there are different levels where the middleware has to wait on each operation on each level. [2]

In the cases where a sequence must be performed in a certain order, there is no choice but to accept this limit. But in some cases synchronous calls are not required, and this is where asynchronous calls come into the picture. In contrast to synchronous call the asynchronous call thread does not wait for a reply; rather it fetches the reply at a later stage. During the time a synchronous thread would have waited, an asynchronous thread can perform other tasks. This therefore makes asynchronous communication suited for cases where the communication is not of the typical request response type. An example is a system where a server periodically sends information to its clients via publication of events or signals, rather than explicit calls or explicit exchange of messages. [2]

The publisher subscriber paradigm principally works in this manner, where certain components make information available by publishing information, while other components indicates they are interested of the information by subscribing to it. The system is thus responsible for matching published information with the subscriptions, and delivers the information to the subscribers in some form of queue system. [2]

As previously mentioned, threads might be hampered because they have to wait for different operations on different levels. Sometimes asynchronous calls might solve this type of problem, but this is not the only advantage of asynchronous calls in a multitier system. Asynchronous calls make it possible to move message processing from different wrappers or components to the queues. The advantages are obvious, since the message processing, for example filtering, can be modified without having to change the components which generate or fetch the messages. [2]

2.3 Middleware

The term middleware is broadly defined, and middleware is used in many different ways. Most middleware is based upon some form model or paradigm which describes the distribution.[3] This chapter describes a couple of different models or middleware infrastructures.

2.3.1 RPC - Remote Procedure Call

The purpose of the RPC model is to hide the network communication when a process is supposed to call a procedure which is implemented on another machine. The parameters are sent from the process to the host computer, which in turn executes the procedure and returns the result. In practice though, it appears as if the procedure is executed locally. [2]

2.3.2 Transaction Processing Monitors

Transaction processing monitors, or TP monitors, can be considered as RPC middleware with the possibility to perform transactions. Depending on if they are implemented in a 2-tier or 3-tier system, TP monitors are classed as either TP-lite or TP-heavy. TP-lite systems tend to supply a RPC interface to databases. TP-heavy monitors conversely are fundamental middleware platforms with a wide range of functionally and tools, which often matches or surpasses those supplied by the operating systems. [2]

2.3.3 Object Brokers

When object oriented programming matured, it was obvious the RPC model could be extended to distribute objects by the same principle as distributed procedures. The essence of the distributed objects, or object brokers, is the object implements an interface which hides all the internal details from the user. More specifically, the interface is implemented on the client, and the object is created on the server. [2]

2.3.4 Object Monitors

Object monitors originate from the demand that object brokers should support transactional calls and that TP monitors should be extended to support object orienting. The result from this demand was that object brokers and TP monitors were merged into hybrid systems called object monitors. For the most part, object monitors are extended TP monitors with object oriented interfaces. The developers usually found it was easier to extend a TP monitor to an object broker rather than implementing an object broker with the performance demands and features of a TP monitor. [2]

2.3.5 Message Oriented Middleware

In the past, RPC and TP monitor systems only offered support for synchronous communication which did not fill all the needs. At first, this was simply solved by implementing RPC middleware with support of asynchronous communication. On the basis of this, TP monitors were extended to support persistent message queuing systems. It was then realized that queue processing could be classed as a unique form of middleware, message oriented middleware or MOM. Such platforms offer transactional calls to different queues and different sorts of operations to read or write on local or remote queues. [2]

2.3.6 Message Brokers

Message brokers are sort of a variant of a message oriented middleware, where the difference is the software logic, which can be attached to queues and dynamically filter and transform messages. Furthermore the choice of recipients can be selected based upon the content of the message. The biggest difference between message brokers and traditional middlewares though, is that instead of acting as a static and inflexible point to point link between applications. Message brokers act as a communication infrastructure, and they deal with all the routing. This functionality combined with asynchronous communication is just what is needed in dynamic EAI (Enterprise Application Integration) systems. This is the main reason message brokers are the dominating commercial EAI tools used today. Thanks to the possibility to define application specific routing logic, message brokers are able to support a wide range of different message based interaction models such as the publisher subscriber paradigm. [2]

2.4 Web Application Servers

As opposed to traditional middleware, web application servers are middleware where the clients connect to the server via the web. Using the web however, causes several complications. The most significant complication is the presentation level has a much more significant role than conventional middleware. This is a direct consequence from the way the web and HTTP protocol is constructed, where all the information exchange takes place with documents. In order to dynamically create these documents, the presentation level needs to be situated in the application server. This tends to merge the presentation level with the application level. The connection to the data level is then handled via standard architectures and APIs such as JDBC, ODBC or ADO.Net. [2]

2.4.1 Application Level

The goal of application servers is to offer an environment for all sorts of application logic, web based or not. For example, an application server makes automatic functionality available as a transaction, when the application is installed on a server. In this way the developers do not have to implement this by themselves, but instead use the functionality which is offered by the application server. [2]

In this way, the application server makes it easier for the developers when they are supposed to develop middleware, since they do not have to implement functionality from scratch. There is unfortunately a trade off when it comes to performance Application servers can never perform as well as a customized TP monitor; however, it is far more likely the development time is shorter. [2]

2.4.2 Presentation Level

A modern application server supports several types of clients:

- Web browsers, including browsers which are running applets
- Applications
- Pocket computers, mobile telephones etc.
- Email clients

Web browsers are today the most common type of client. They interact with the application server, via a web server, with the HTTP or HTTPS protocol, and they fetch static and dynamically created web pages. However, if an application such as a Java applet is running in a web browser, there are no limits to which communication protocol to use. The applet can, for example, communicate via RMI (Remote Method Invocation), CORBA/IIOP (Inter-Operable Internet Object Protocol) or standard CORBA remote procedure call protocols on TCP/IP. Furthermore, ordinary desktop applications can interact with the application server in the same manner. The only difference is that in the applet case, the application is downloaded via a web browser and it runs with tighter security settings because the applet is by default not trusted. [2]

2.5 Webservices

According to the World Wide Web consortium, the definition of a webservice is the following: “A software application identified by a URI, whose interface and binding are capable of being defined, describe and discovered as XML artifacts. A webservice supports direct interaction with other software agents using XML based messages exchanged via internet based protocols”. This means that webservices should be services like conventional middleware. Not only are they supposed to be up and running, but also being described and announced so that client can interact with them. In this way, webservices can be integrated into more complex distributed applications. [2]

2.5.1 Why Webservices are Needed

Conventional middleware functions very well when the communication is limited to a LAN, or at least within the same organization. However, when it comes to business to business (B2B) integration, the situation immediately becomes more complicated. First, it is not at all obvious where the actual middleware should be placed. This would, for example, require that the organizations were to agree upon which message broker to use and which third party company was to host. Even though it is possible that a small amount of companies, who are cooperating closely, might find a particular solution acceptable.

In practice though, this is fairly rare; companies generally do not trust each other and want to be independent [2]. Their transactions should be confidential, and only the recipients are supposed to be able to read the information. Every company wants to control their own business operations and the way they are performed. This is, of course, not possible if a third party is controlling the message broker. Even if every company hosted their own message broker, this would eventually lead to a situation where every company needed to support lots of heterogeneous middleware systems. Furthermore the security aspect has not been mentioned at all and this is also a large problem. During EAI integration, the information exchange occurs within the same trust domain. However when it comes to B2B integration there is no such thing as trust, even if the communication formally speaking were to happen in the trust domain. Transactions in particular have been complicated to implement, since the company which hosts the server must limit and control which resources should be locked, and not give away the locking access to potentially harmful external entities. [2]

There are of course companies which have successfully implemented successful B2B integration. However the web has offered standard protocols, such as HTTP, and data formats, such as XML, which have created a common middleware infrastructure in which the heterogeneity among the interfaces have been reduced. HTTP and XML are enough to offer application integration which, for example, demands transaction protocols and other abstractions to work. The goal of webservices is to fill this gap between the web (HTTP, XML) and what application integration requires. [2]

2.5.2 B2B Integration with Webservices

Webservices resolve the limits of conventional middleware from three perspectives:

- First, the webservice is exposed as service which in middleware terminology means procedures, methods or objects. The service publishes an interface which can be invoked by clients. Webservices are programs which call other programs; the difference is calls occur across the internet.
- Second, communication protocols are not dependent on communication which only occurs within the organization. What previously required a centralized platform is now replaced by protocols which work in a decentralized environment and across several trust domains.
- Third, the webservice technology is standardized. It does not matter if new protocols and languages solve problems if only few use them. [2]

It is worth mentioning that even though the main purpose of webservices is to solve problems regarding business to business integration, you can just as well use them locally within one organization or LAN. In fact, if all programs were to have webservice interfaces, integration of programs would be much easier since all the components would be homogenous. [2]

2.6 Webservices Infrastructure

The fundamental components of webservices are SOAP, WSDL and UDDI. SOAP is the message protocol webservices use when messages are sent. WSDL is a specification used to describe the webservice. Finally UDDI is used by service providers to publish their webservices so clients can find them. [2]

2.6.1 SOAP

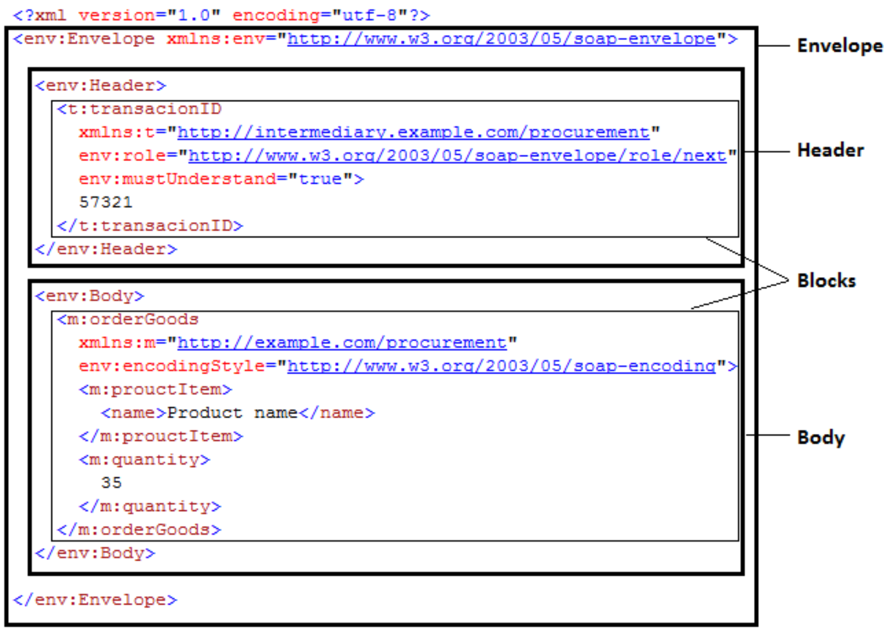
Since business to business integration has been problematic because of firewalls, the lack of standardized protocols and so forth, this is the first issue webservices needs to tackle [2]. The answer for these problems is in the SOAP specification which specifies the following: [2]

- A message format for one way communication that describes how to package data within a XML document.
- A set of conventions on how to implement the RPC interaction pattern by using SOAP messages. That is how client can invoke a remote procedure by sending a SOAP message and how services can send a new SOAP message back as response.
- A set of rules which all SOAP messages needs to follow, in particular the XML elements they have to be able to read and understand. If the content is not understood, measures are also defined on how to deal with these messages.
- A description that states how messages are supposed to be sent with the HTTP or SMTP transport protocol.

As a communication protocol, SOAP is one way and stateless. This means the interaction needs to be encoded within the SOAP document. In order to implement a standard RPC call, the message thus needs to be created on the client side, and then the services create a new SOAP message which is sent back. Synchronous messages like these are usually sent with the HTTP protocol. For asynchronous calls, a possible scenario might look like this: First, the client sends a SOAP message to the service and then the client receives the SOAP message reply via SMTP. [2]

2.6.1.1 Structure and content of a SOAP message

SOAP exchanges information with the help of messages. These messages are used as envelopes, where the application wraps the information which is about to be sent. Each envelope contains two parts: the header and the body. The body is mandatory however, the header is optional. [2]



```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <t:transacionID
      xmlns:t="http://intermediary.example.com/procurement"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
      env:mustUnderstand="true">
      57321
    </t:transacionID>
  </env:Header>
  <env:Body>
    <m:orderGoods
      xmlns:m="http://example.com/procurement"
      env:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
      <m:prouctItem>
        <name>Product name</name>
      </m:prouctItem>
      <m:quantity>
        35
      </m:quantity>
    </m:orderGoods>
  </env:Body>
</env:Envelope>
```

Figure 2.1: SOAP envelope example

The reason that SOAP messages are constructed according to these principles is because the SOAP protocol should follow the same approach as standard communication protocols. A SOAP message assumes that each message has a sender, an ultimate receiver and an arbitrary amount of intermediaries. The actual data which is meant for the ultimate receiver is contained within the body while the data that the intermediaries need is in the header. The data the intermediaries need might, for example, be transaction id, security information and so forth. In the cases where there are no intermediaries at all, there is no need for a header and for this reason the header is optional. [2]

There are no requirements on how the SOAP structure is supposed to be in the header or the body, but one of the most common one is the RPC Style. In RPC style, the request is encapsulated within one message and the reply in another message. The request body contains the data about which procedure is to be executed and various amounts of input parameters in the request message. The response message contains the output parameters. [4]

2.6.1.2 Binding SOAP to a Transport Protocol

Even if the SOAP specification states how to bind the protocol to HTTP and SMTP, SOAP is not tied to a specific transport protocol. A SOAP envelope can be sent on any kind of transport protocol. Thus when referring to a SOAP binding, one is actually referring to a specification on how to package a SOAP message within a transport protocol. For example, the HTTP binding is the specification on how to wrap a SOAP message within a HTTP package. Depending on what is supposed to be done, SOAP can be transmitted via GET, POST or other HTTP primitives. [4]

2.6.1.3 A Simple SOAP Implementation

An example of how to implement RPC functionality with SOAP interaction can be described as follows:

In the client code, a local method call is made to a method, which is a proxy method created during compile time. From an external perspective, the method behaves as if it runs locally, however, in fact it actually re-routes the message to SOAP engine that transforms the method parameters to a SOAP message. When the SOAP message is created, a HTTP engine wraps the envelope within a HTTP post request message. The HTTP message is then sent to, and received by the server. This process is then done in a reversed order on the server. On the server the SOAP envelope is extracted from the HTTP call, and forwarded to a SOAP router. This router then forwards the messages to a server stub, which in turn calls the target method. [4]

2.6.2 WSDL

The acronym WSDL stands for Webservice Description Language and the main purpose of WSDL is to describe the interface of the service. From this perspective, the role of WSDL is the same as that of an IDL in a conventional middleware. What separates them is WSDL needs to define the mechanism on how to access the webservice. This is done implicitly in traditional middleware, since the access mechanism is identical on the various middleware platforms. For webservices, this is not the case because webservices can be accessed via different kinds of transport protocols, and for this reason, it is of utmost importance this kind of information is a part of the service description. [4]

2.6.2.1 Structure of a WSDL Interface

A WSDL specification can be divided into two parts: an abstract part and a concrete part. The abstract part is conceptually like the conventional IDL, while the concrete part contains the protocol binding and so forth. [2]

The abstract part of a WSDL specification consists of four different parts: types, messages, operations and port types. The type part is the part which defines the different data types that both of the parties must be able to interpret. By default WSDL specifications uses the same type system as XML schemas but this is by no means a requirement. XML schemas have built in datatype primitives such as integers and strings, and these primitives can then be used in order to construct structures, etc. Thus, the first step in building a WSDL interface is therefore to identify and define all data structures which are supposed to be exchanged between both of the parties. [2]

The second step is to define the actual message to be sent. Each message is divided into different parts where each part has a name and a datatype. The parts therefore represent the in and out parameters of the method of the webservice. [2]

The third step in defining the WSDL interface is to define operations, also called transmission primitives or interactions. There are four kind fundamental operations: [2]

- One-way means the webservice is called without the expectation of any reply.
- Request-response corresponds to a traditional RPC or method call, where clients expect some form of result in return immediately after a call.
- Solicit-response is the opposite of request-response where the webservice is doing the calling and expects a reply in return.
- Finally, notification is the opposite of one-way, i.e. the service calls the client without expecting a reply

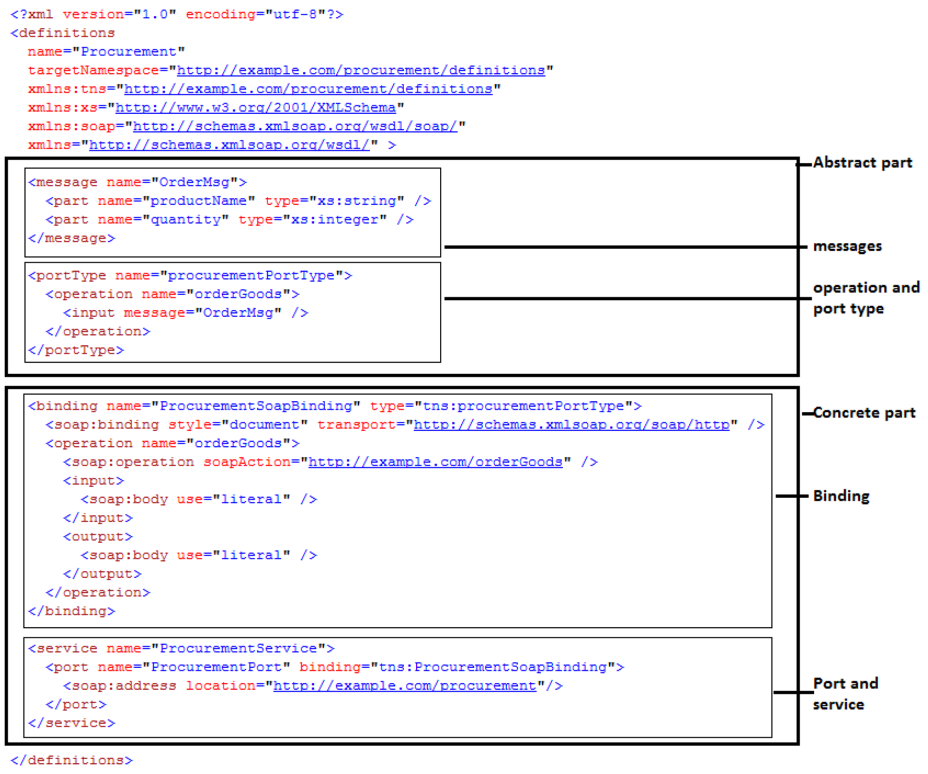
Of these, request-response and solicit-response operations are classed as synchronous while one-way and notification are classed as asynchronous. [2]

The last step in defining the abstract part of the WSDL interface is to group operations into port types. A port type is the counterpart of an interface in traditional IDLs. Each porttype is a logical collection of related operations. [2]

The reason the parts above are classed as abstract is because the definitions miss concrete binding or an encoding specified for these constructs. They also miss a definition of a service that implements a set of port types. For example, in order to define a real instance of a webservice one must define:

- The exact set of ports which it implements
- The transport binding which is used when implementing the port types
- The addresses used by the clients to invoke the services.

The absence of these parts makes the port definitions abstract, because port types and messages can be implemented with different kind of transport bindings or encoding. Furthermore, even data types can be serialized according to different rules. The other part of the WSDL definition is thus to define a concrete service by specifying all of these aspects. [2]



```
<?xml version="1.0" encoding="utf-8"?>
<definitions
  name="Procurement"
  targetNamespace="http://example.com/procurement/definitions"
  xmlns:tns="http://example.com/procurement/definitions"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/" >
<message name="OrderMsg">
  <part name="productName" type="xs:string" />
  <part name="quantity" type="xs:integer" />
</message>
<portType name="procurementPortType">
  <operation name="orderGoods">
    <input message="OrderMsg" />
  </operation>
</portType>
<binding name="ProcurementSoapBinding" type="tns:procurementPortType">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
  <operation name="orderGoods">
    <soap:operation soapAction="http://example.com/orderGoods" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
<service name="ProcurementService">
  <port name="ProcurementPort" binding="tns:ProcurementSoapBinding">
    <soap:address location="http://example.com/procurement"/>
  </port>
</service>
</definitions>
```

Figure 2.2: WSDL document example

The concrete part of a WSDL interface is defined using the following three constructs. [2]

- Interface bindings. The binding specifies the message encoding and protocol binding for all operations and messages which are defined within a specific port type. For example, an operation can be defined as a RPC call. An interface binding can also define that the message being sent is bound to the HTTP transport protocol. This is by far the most common approach. Finally, the message encoding used for serializing the message to XML is defined. When messages are encoded using SOAP, the translating is done by translating the WSDL data types to XML based upon the encoding rules according to the rules of a SOAP specification, such as version 1.2.
- Ports. Also known as EndPoints, ports combine the InterfaceBindning information with a network address (specified by an URL) where the implementation of the port type can be accessed.
- Services. Services are a logical group of ports which can be deployed on different addresses. This means the actual webservice can be deployed on different addresses. Furthermore this implies that the same functionality can be accessed with different kinds of transport protocols and interaction styles.

2.6.3 UDDI

There is a third technology worth mentioning called UDDI which is an initialism for Universal Description, Discovery and Integration. UDDI is a registry and an API where the registry is simply a registry containing the addresses to the webservice. The API conversely defines how to publish webservices, what is needed to register webservice and how to query them. There is much more to UDDI than this, but this will not be covered because it is out of scope. [4]

2.7 Publisher Subscriber Architecture

Previous chapters cover specifications on how webservices are supposed to be implemented, and even though the specification offers the possibility for webservices to communicate in ways other than ordinary request reply communication. They do not explicitly treat how to implement a webservice architecture based upon the publisher subscriber paradigm. The principal behind the publisher subscriber paradigm is that different software components subscribe on different kinds of state changes. This means when a state change happens, the code which is supposed to react on the state change is executed.

As of now there are two propositions on extending standards on how to do that. One of them is called WS-Eventing and was originally designed by BEA systems, Microsoft and TIBCI. The other is called WS-Notification and contains three sub specifications. WS-BaseNotification, WS-BrokererdNotification and WS-Topics [6]. WS-Notification was originally designed by Akamai, The Globus Alliance, Hewlett-Packard, IBM, Sonic software and TIBCO software. [10]

WS-Eventing and WS-Notification are unfortunately only suggestions, and according to Gartner they should be considered as rough drafts rather than stable standards. If one is supposed to implement the publisher subscriber paradigm between webservices, it is recommended to review both of the specifications and choose the one which is best suited. Furthermore it is recommended to revise the application if a commercial middleware appears which has implemented a converged standard. [5]

2.7.1 WS-Eventing vs WS-notification

This part of the paper is based upon the report “A comparative study of Web Services-based Event Notifications Specifications” [11] and describes the differences from different perspectives such as architecture, functions, message delivery, message formats and broker supports. Since the publications of this report, WS-Eventing has formally been delivered to W3C, reviewed and accepted as a formal W3C standard. The submitted specification is however identical to the one examined in ”A Comparative Study of Web Services-based Event Notification Specifications”

In general terms, WS-Eventing is a simpler standard than WS-Notification. WS-Notification however, has more features than WS-

Eventing and can be used in a full-fledged publisher subscriber system. Since webservice specifications are composable, the WS-Eventing and WS-Notification specifications only contain key publish subscriber related functions. Functions such as security, reliability and transaction handling are dependent on other WS-* specifications.

2.7.1.1 Architecture Comparison

The architecture of WS-Eventing and WS-BaseNotification has almost identical webservice based architecture. They both follow the publisher subscriber paradigm and both define subscriber and subscriber manager units. Event sink, which is defined in WS-Eventing, is equivalent to WS-BaseNotification's notification consumer. Both of the specifications separate subscribers from notification consumers, such that notification consumers only need to deal with received messages. They do not need to know about the broker locations and how to create subscriptions. WS-Eventing does not separate the publisher from the event source. The event source in WS-Eventing both has the notification function and the publisher role defined within the WS-BaseNotification.

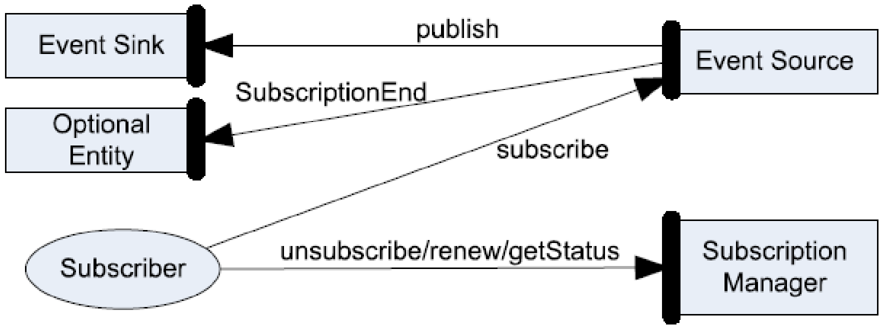


Figure 2.3: WS-Eventing architecture and operations

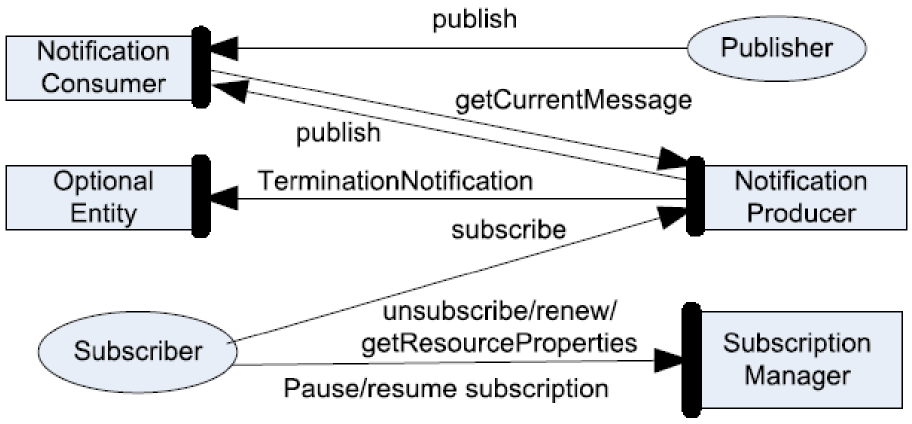


Figure 2.4: WS-BaseNotification architecture and operations

2.7.1.2 Function Comparison

We can also find many similarities within the actual functions of the specifications. WS-Eventing defines the five web service operations:

- Subscribe
- Renew
- GetStatus
- Unsubscribe
- SubscribeEnd.

The Subscribe message is used to create a subscription for an event sink. The Renew, GetStatus and Unsubscribe messages are sent from subscribers to subscription managers in order to handle existing subscriptions. The SubscriptionEnd message is generated when an event source terminates a subscription unexpectedly. The actual SubscriptionEnd message is sent in the address which is specified in the subscription request. In the case where the address is not a part of the subscription request, the message is not generated.

WS-Eventing	WS-BaseNotification
Subscribe	Subscribe
Renew	Renew
Unsubscribe	Unsubscribe
GetStatus	*Not defined, can use GetResourceProperties in WSRF*
SubscriptionEnd	*Not defined, can use TerminationNotification in WSRF*
Not available	Pause/resume subscription
Not available	GetCurrentMessage

Table 2.2: Function comparison

WS-BaseNotification has comparable operations for the five first items in the table above. Even though the specification does not define GetStatus and SubscriptionEnd, these messages can be constructed with the optional WS-ResourceFramework (WSRF) since WS-Notification can handle subscriptions as WS-Resources in WSRF. In addition to the five top most operations, WS-BaseNotification also has three more. These operations defines how to pause/resume subscriptions and how to the current message. (GetCurrentMessage).

2.7.1.3 Message delivery comparison

This sections covers in detail how to specify delivery in subscription requests of WS-Eventing and WS-Notification.

Delivery mode: Both WS-Eventing and WS-Notification can use push, pull and wrapped mode in order to deliver notification messages. The wrapped mode can package several notification messages into one in order to make the delivery more effective. Push mode is the default mode within WS-Eventing. It uses the delivery extension point in the subscription messages in order to support the other delivery modes. Notification message formats are not defined in the specifications.

WS-Notification defines a PullPoint interface, but it is not possible to specify in a subscription message that pull delivery is supposed to be used. A pull point needs to be created before a subscription is created and the subscriber is treated as a regular push event consumer from a publisher perspective.

Filter: WS-Notification defines three types of message filters: TopicExpression, ProducerProperties (if subscription is handled as a WS-Resources) and MessageContent. A subscriber can filter on any or all of these. WS-Eventing conversely only supports one filter expression in the subscription request. No filter topic expressions are specified as in the WS-Topics specification. Both specifications can use any expression (xsd:any) in a specified dialect that evaluates to a Boolean value as a filtering criteria. WS-Eventing does not specify a way to filter messages using the ProducerProperties of publishers.

2.7.1.4 Broker support comparison

The WS-BrokereredNotification specification is the specification in the WS-Notification family that defines brokered support between producers and notification consumers, and the specification acts as an extension of WS-BaseNotification. The notification brokers can handle publisher registration, and has support for demand based publishers. A demand based publisher publishes only messages which consumers are interested in. A notification broker can keep track of each message and it can keep track of the number of consumers of each kind of message. It can also pause or resume subscriptions to publishers based on the demand. WS-Eventing does not define how to use a broker as the intermediary between eventSink and eventSource. However, it is possible to create a broker that implements both the eventSink interface and the eventSource interface. Neither the publisher registrations, nor the demand based publishers are defined in WS-Eventing.

Chapter 3

Analysis

Introduction

This chapter analyses the current system, and the system to be built. Both systems have a multitier architecture, however different types of middleware are used to communicate between them. Furthermore the architecture of the old system is more or less scaled vertically, while the architecture of the new system has the ability to scale horizontally.

3.1 The current system

The current webpage is hosted by an internet information server 6.0 which is implemented with the classic ASP framework. This server side ASP code is the first part of the processing level which renders the HTML and JavaScript code that will act as presentation and processing level on the client side. In order to render the client side code, the server based processing level is dependent on data stored in a Microsoft SQL server. This is the data level on the server; for the clients however, the data level is hosted on a different server.

The other part of the processing level is implemented with JavaScript code, and it runs in a web browser. One might think the code running on the web browser just contains the user-interface level, but the webpage does not merely respond to user input. It also makes regular requests to retrieve the current states of the measuring devices, independent of what the user does. Hence, a part of the processing level is also executed by the JavaScript code.

All calls from the JavaScript code are made by the Java applet mentioned in the problem description. This Java applet is a RPC middleware that makes an asynchronous call to a web page called “sfaxml.asp”, which dynamically renders a SOAP message. Depending on which topics which are subscribed to, the SOAP message returns the latest state changes. However, even though messages are communicated with SOAP, this web page can not be considered a web service since no WSDL description is presented. Rather “sfaxml.asp” is also a RPC middleware part of the data level, which communicates with another server that actually communicates with the measuring devices

3.1.1 The Data Level

The “sfaxml.asp” web page acts as one client that communicates with one server. The server in turn communicates with the measuring devices via either standard Ethernet connection or a modem. More specifically the server is an OPC server.

3.1.1.1 OPC

OPC stands for Object Linking and Embedding (OLE) for Process Control and is a standard defined by the OPC foundation. Over 300 companies such as Microsoft and nearly all of the world's major providers of control systems are represented in the OPC foundation. This representation assures OPC will remain the industry standard for the foreseeable future. The purpose of OPC is to define application interfaces in industrial automation software, which makes interoperability possible between automation/control applications, field systems/devices and business/office applications. Thanks to OPC, it is no longer necessary for software application developers to write a custom interface or server/driver to exchange data with hardware field devices. OPC eliminates this requirement by defining a common interface that allows this work to be done once. [12]

A typical system based upon OPC consists of OPC clients and OPC servers. The OPC server provides real-time data from PLCs (a Programmable Logic Controller is a device used to automate monitoring and control of industrial plant) and other control devices. An OPC client is an OPC data consumer and it is typically a visualization or database application which presents or uses the OPC data provided by OPC servers. [12]

Rather than one specific specification, OPC is actually a series of specifications. The only standard this thesis focused on is called the Data Access Specification or OPC DA. OPC DA is the result of a collaboration of a number of leading worldwide automation suppliers working in corporation with Microsoft. Originally based on Microsoft's OLE COM (Component Object Model) and DCOM (Distributed Component Object Model) technologies, the specification defined a standard set of objects, interfaces and methods to facilitate interoperability in process control and manufacturing automation applications. The COM/DCOM technologies made it possible for clients and servers to remotely interconnect, and they provided the framework for software products to be developed. [12]

3.1.1.2 OPC Data Access Fundamentals

Any OPC client is able to connect to several OPC servers from different vendors. On a higher level the clients contain three kinds of objects: The server, the group and the items. The servers work as a container for the OPC-groups, and the OPC groups in turn contains OPC items. An OPC item, which is a topic, represents the connection to the data source within the server, and for every item a value, a quality value and time stamp is associated with it. The value is the value the OPC servers most recently read from the data source, however, there can be several factors which influence the quality of the value. The quality value which indicates if the value is reliable or not, and the time stamp that is the time when the last measurement was conducted. [13]

An OPC item is not directly accessible by the OPC client, but must always be done via an OPC group. The purpose of an OPC group is thus to organize data. When an OPC client registers itself on the OPC server, it requests the server to continuously read the values of every data item within the OPC clients group or groups. When the groups are registered the OPC server automatically starts to poll each data source and continue doing so until the clients unregister the groups. OPC clients can then retrieve data from server by two means. One means of data retrieval is to manually read the data from server, and the other means is to activate a subscription. During the subscription, a client, rather than manually requesting updates, waits for the server to notify the client by calling a callback function via DCOM. In addition to set which data a group should retrieve, a client can also control how often the OPC server should poll the items in a group, and it defines how much a value should change before update is returned. This naturally implies that an OPC server is a message broker since it has business logic. [13]

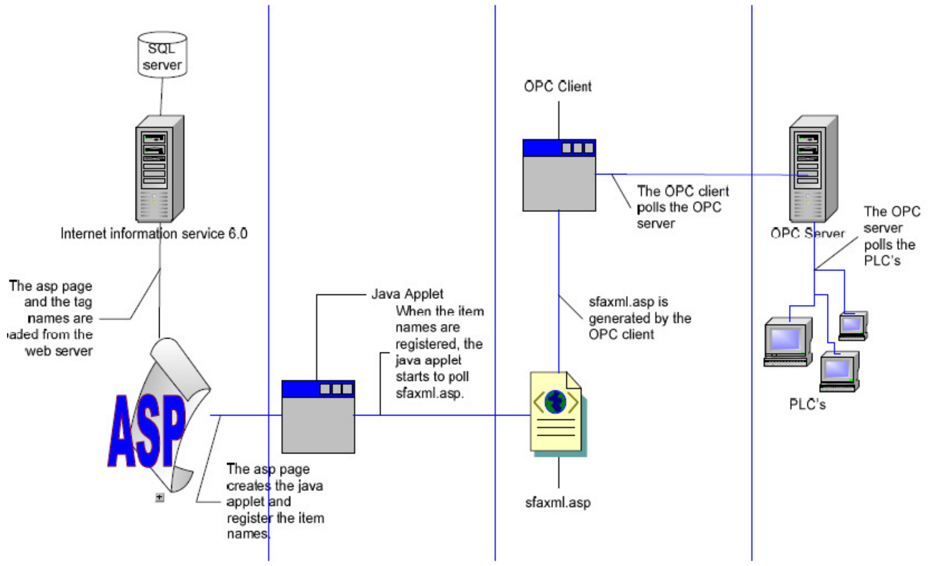

SQL server
Internet information service 6.0
The asp page and the tag names are loaded from the web server
ASP
The asp page creates the java applet and register the item names
Java Applet
When the item names are registered, the java applet starts to poll sfaxml.asp.
OPC Client
The OPC client polls the OPC server
sfaxml.asp is generated by the OPC client
sfaxml.asp
OPC Server
The OPC server polls the PLC's
PLC's

Figure 3.1: Architectural overview of the current system

3.2 The Prototype System

Mostly the prototype system is a replacement for the two middleware software shown above. Thus, the core of the data level is still the OPC server and the previous user interface level does not need to be changed. However, the processing level must be rewritten because the applet middleware needs to be replaced, and a new middleware needs to be implemented to integrate the OPC server with the processing level.

3.2.1 Replacement of the Applet Middleware

There are several techniques to perform a data fetch without doing a post back. One technique is to read the current status with an invisible Iframe [15] containing a data string that might, for example, represent a XML document. In this technique, a JavaScript loop regularly forces the Iframe to read a page which renders the data, and then reads the data saved within the Iframe. If the GUI is rendered via flash, the flash plugin may also fetch data without doing a post back. However, by far the most popular technique to solve the problem is to implement the Ajax paradigm. [14]

Ajax stands for asynchronous java and XML and it is a paradigm for creating web pages which retrieve data from the web server without refreshing the entire page. The core component of Ajax is the XMLHTTPRequest object, which is defacto standard today, and it is the actual component which makes the asynchronous call. [14]

In this case I have found three different suitable techniques for performing data fetch operations: periodic refresh [15], HTTP streaming[15] and long polling [16].

3.2.1.1 Periodic Refresh

The periodic refresh technique is the most basic pattern where the XMLHTTPRequest object makes requests to the webserver at a regular basis. Simply put, periodic refresh is the same thing as polling. Ideally the interval between the requests should be zero but that is unrealistic. Thus there will always be certain latency between the state change and when the browser updates the user level. This latency might pose a problem if the users are working with volatile server-side data. For example, one user might be editing a data item while another user deletes it. Another problem is the fact each request demands resources at both the server and the client. In addition, each request has some bandwidth which might add up to a substantial amount if refreshes occur every few seconds. [15]

So how long should the time span between refreshes be? The short answer is it depends on the application. For example, a realtime stock trading website should have a very short refresh interval, such as 20-100 milliseconds. While a casual monitoring system can have a much longer refresh interval (minutes), e.g. a system which monitors an rss feed. In between these cases are web applications where the state changes do not need to be propagated instantly, such as website which monitors outdoor thermometer or heat consumption in a building. The interval could be anything from a couple of seconds to minutes. [15]

3.2.1.2 HTTP Streaming

HTTP streaming happens when a request message is retrieved by the web server, and a connection is established between the client and the server. However, instead of the server sending a reply message back and then closing the connection, the server keeps the connection open and flushes new data to client's web browser. Ajaxpatterns [15] mention two kinds of solutions to implement HTTP streaming. One solution is called Page Streaming, and this is where the data is streamed through the first original page response. With Page Streaming, an initial page is created that later on is changed by continuously added script code which in turn changes the page's DOM. DOM stands for Document Object Model, which represents HTML documents as a tree of data objects with methods for manipulating their content. The disadvantage with Page Streaming is the web browsers which sooner or later run out of memory. [15]

An alternative to solve the memory problem is to implement Service Streaming rather than Page Streaming. In contrast to using the first page request message an XMLHTTPRequest or an iframe call is used to keep the connection open. Instead it is up to the XMLHTTPRequest object or the iframe to keep the connection open, which makes it easier to control the connection. Another difference is data is sent as ordinary data rather than script code. The disadvantage with Service Streaming is some browsers such as Internet Explorer requires the response message from the XMLHTTPRequest object or the iframe is completed before the browser can process the data. [15]

3.2.1.3 Long Polling

Long polling can be considered to be a compromise between the periodic refresh and the HTTP streaming technique. Each time a request is made, the server does not send a reply until a response is available or when the request times out. As soon as the connection closes, a new one is initiated. The result is lower latency and less bandwidth consumption than using the periodic refresh technique, and even though this technique has higher latency than HTTP streaming, it is at least possible to implement it on all browsers that supports the Ajax paradigm. [16]

3.2.2 Replacement of the Data Level Middleware

As previously mentioned, the goal of a webservice is to integrate different applications using the internet as the common infrastructure and XML as the data format. Thus one way of solving the problem on how an AJAX webpage could communicate with an OPC server is via a webservice interface. Furthermore if an AJAX webpage were to communicate with either the WS-Notifcation or the WS-Eventing port types, the goal of this thesis would be solved.

Chapter 4

Method

Introduction

The following chapter describes the development process of the prototype. No formal software process was used, however from a wider perspective; the thesis project contained two major iterations. The first iteration was more of an experimental nature where I solved the problems on how to fetch data without post back and how to read data from the OPC server. In the second iteration I focused much more on analysis in order to create a more robust system. Both of these iterations will be explained.

4.1 Iteration One

There are several kinds of development environments that could solve this thesis project. However Manodo has a strict policy to only use Microsoft technologies since Manodo's main customers use Internet Explorer as their default web browser. Because of this policy, I used .Net version 2.0. This is the motivation the iteration one prototype has been implemented with the .Net 2.0 framework. Whether the .Net platform is the best choice from a performance or development perspective is out scope.

The most important problem to solve during the start up phase was to find a good replacement for the Java applet. This replacement needs to retrieve data from the server without post back. Once that problem was solved, I started writing simple console applications that read data from the OPC server. Finally, I used that code to write server side code which would process the calls from the clients.

4.1.1 Client Side

Even though using the XMLHTTPRequest component is not the only solution, it is a de facto standard on all major web browsers [14]. Furthermore it is, as previously mentioned; a part of the Ajax paradigm and therefore using this component would solve this first problem.

When I decided to follow this approach I started to compare different components or software libraries which would ease the development. The ones I found were the following:

4.1.1.1 ASP.Net Client Callback

The client callback feature of ASP.NET 2.0 is a wrapper for the XMLHTTPRequest component. It allows the programmer to asynchronously call a method on the web server. The method on the webserver simply accepts a string as the input parameter and returns a string as the output. [17]

4.1.1.2 Ajax .Net

Ajax.Net is an Ajax Framework for building interactive and responsive web sites that work across many web browsers. The framework is developed by Microsoft. [18]

4.1.1.3 Webservice Behavior

The webservice behavior is an XMLHTTPRequest component that enables a webpage to call methods on webservices via JavaScript code. The calls can be made synchronously or asynchronously and programmers do not need to know anything about SOAP. [21]

4.1.2 Server side

I had two choices when implementing the server side code. Either as an ordinary ASP.Net webpage that would render an XML document instead of HTML code, or as an ASP.Net webservice. I choose to implement the server side code as an ASP.Net webservice because I found it to be simpler. Instead of manually constructing XML code, ASP.Net webservice framework did the object to XML serialization automatically.

Ideally I would at this point implement three kinds of prototypes based upon the three techniques Periodic refresh, HTTP streaming and Long polling. However I did not implement an HTTP streaming webservice due to time constraint. Furthermore it is impossible to implement HTTP service streaming on Internet Explorer [15]. Thus I only implemented webservices based upon the other two techniques.

4.1.2.1 Periodic Refresh Webservice

The period refresh webservices exposed two operations. The first was the Init operation, whose purpose was to initialize the OPC group in order to start the subscription. This operation accepted an array of strings where each string element represented an OPC item the client wanted to subscribe to. After a successful initialization the operation returned a GUID.

As the client's web browser received the GUID key, it started to poll the other operation every five seconds, whose purpose was to retrieve

OPC item values. This second operation accepted the GUID as a parameter, which it then used to retrieve the OPC group from memory. If some item or items had been updated, the corresponding key value pair or pairs would be returned and processed by the client's web browser.

4.1.2.2 Long Polling Webservice

Like the periodic refresh webservice, the long polling webservice essentially exposed the same operations. In fact the first operation was identical, returning the OPC group GUID key. Instead of polling every five seconds though, as soon as the client received the reply, it immediately performed a new request to the data fetch operation.

The main difference between the data fetch operation of the Long polling vs. the Periodic refresh webservice, is the long polling webservice is not manually reading the OPC items via the OPC group. Instead the OPC server will make a remote procedure call to webservice whenever an OPC item has changed value. This call will be processed and the corresponding thread that processes the client data fetch call will be signaled the new data is available. To prevent time outs, the data fetch operation only waits 10 seconds to be signaled or else an empty response is returned to the client web browser. If the current OPC group is called by the OPC server between requests, the data fetch operation thread will deal with this as if it were signaled immediately.

4.1.2.3 Validation and Testing

In order to test that my code communicated correctly with an OPC server I used the following software:

- OPC DA3 Test Client
- Graybox simulator

Graybox simulator is software that simulates an OPC server. The purpose is to use it while developing OPC client software, such as my webservices. Since it is irrelevant my prototypes receive any real world data, the Graybox simulator was adequate for this testing. To validate my data fetch operations returned correctly, I used the OPC DA3 Test Client. OPC DA3 Test client is an OPC client where it is possible to create OPC groups and subscribe to different OPC items. Thus I used it to verify the values returned from the data fetch operation were correct. Furthermore, it was possible to assign values to different OPC items the webpage currently was subscribing to. This way I could also see that the data fetch operation returned a correct value.

Ideally I should have made additional tests such as stress test on both of the webservices, and I tried to use Microsoft's Application Center test application in order to do just that. At first it seemed to be a great application were you could record user input and send multiple requests to webpages, who in turn would call the webservices. However, it was not smart enough to poll the webservices with unique

GUID's, but rather the one used when the record where commenced. Therefore, I did not manage to perform any realistic tests, and I did not have enough time to fix this issue.

4.2 Iteration Two

The first iteration was considered to be over at the point when I had verified that both of the prototypes worked. The first step of iteration two was to scrutinize them both. Was it a correct decision to implement the server side code as webservices? If yes, how could I improve them? What about the Periodic Refresh or Long Polling technique? Which one seemed to be best for this particular situation?

4.2.1 Motivation for Periodic Refresh

I choose to not use the Long Polling technique because I found it to be much more complicated. I thought the odds of succeeding in creating the simpler Periodic Refresh technique were higher. In addition, since the OPC groups themselves poll between certain intervals, the actual gains were not that great. There was no way to fetch the data in real time anyway.

Neither of the Ajax libraries were used because none of them supported the SOAP messages that were supposed to be sent. Thus all the code that used the XMLHTTPRequest object had to be written from scratch.

4.2.2 Motivation for WS-BaseNotification

It was pretty hard to find good information on what Webservices really are about. Even though I immediately found webservices to be a great tool where I could communicate via XML and the HTTP protocol, I questioned whether Webservices were ideal in this case. As I stated in the concepts chapter, the answer is yes for several reasons.

Next step, how could I improve the prototypes I made? What I found were the two specifications WS-Notification and WS-Eventing. Any publisher subscriber system supporting either of these specifications would reasonably be better than the first prototype I constructed. First and formost from a software design perspective, but also from a B2B perspective since other software supporting these specifications could be integrated with my prototype.

Due to the lack of time, I was only able to implement one of them. I choose to implement a distributed webservice system using the WS-BaseNotification specification, and the motivation for this was the fact it had Pull Point interface. Although the WS-Eventing specification does not explicitly prevent pulling delivery, WS-BaseNotification seemed like a better choice in this case.

In order to implement a WS-BaseNotification webservice I had to be able to customize the soap messages it sends and receives. Unfortunately I was not able to implement such a webservice using the ASP.Net 2.0 framework. Therefore I searched for alternatives and the alternative I found that gave me the possibility was the Windows communication foundation framework, which is a part of the .Net framework from version 3.0. I choose to implement the prototype with the 3.5 framework but I could have also chosen 3.0.

Chapter 5

Design

Introduction

This chapter describes the design of the webservice port types within a WS-BaseNotification prototype system. Since I blindly followed a previously written specification, there was not really any design planning using a design process or the like, rather I just implemented webservices using these specifications [7, 9]. Though the prototype does follow the specification, due to time constraints the whole specification was not implemented.

5.1 Architecture Overview

The prototype system consists of webservices exposed by the following porttype interfaces:

- NotificationConsumer
- NotificationProducer
- SubscriptionManager
- PullPoint
- Create PullPoint

In an environment that is not restricted from pushing delivery, webservices only have to expose the three topmost porttype interfaces. The NotificationConsumer porttype acts as the client web service, which subscribes to topics and receives events. Furthermore the NotificationProducer porttype acts as an event producer, which sends events to the notification consumer web services. Finally, the SubscriptionManager provides operations for controlling and regulating the subscriptions.

Environments such as the one investigated in this thesis also need the webservices exposing the PullPoint and the Create PullPoint porttype interfaces. In this case, clients fetch the data via a webservice exposing the PullPoint porttype interface. Before data can be fetched via the pull point it has to be created with pull point factory.

The rest of this chapter will cover the design of each porttype and what part of the WS-Notification specification is supported. Because of time constraints the whole specification has not been implemented in the final prototype, however all messages sent from and to webservices follow the specification correctly except for error messages. According to WS-Notification incorrect data input and other errors have to return certain error messages. Unfortunately, there was not time enough to implement these in the prototype.

These XML namespaces are used in the prototype:

Prefix	Name	Namespace
wsnt	WSNotification	HTTP://docs.oasis-open.org/wsn/b-2
wsa	WSAddressing	HTTP://www.w3.org/2005/08/addressing
wstop	WSTopic	HTTP://docs.oasis-open.org/wsn/t-1
pn	Prototype	HTTP://PrototypeNamespace
s	SOAP	HTTP://www.w3.org/2003/05/soap-envelope

Table 5.1: Prototype namespace

5.2 The SOAP Message Envelope

All messages sent to and from the webservices are sent via a SOAP envelope. This is how a basic SOAP message looks, where the body contains the message specific data and each message has a unique action-URL.

```
<s:Envelope ...>
  <s:Header>
    <wsa:Action>
      HTTP://action-url
    </wsa:Action>
  </s:Header>
  <s:Body>
  </s:Body>
</s:Envelope>
```

Even though the header is not mandatory from a SOAP perspective, each and every message specified in the WS-Notification uses the WS-Addressing action property. According to the WS-Addressing specification [19] an action represents “An identifier that uniquely (and opaquely) identifies the semantics implied by this message”, or “the verb or intent of the message”.

The XML specified in the rest of this paper is the data defined within the body element.

5.3 Create PullPoint Porttype

In order to fetch data via a pullpoint, the pull point resource has to be created by the Create PullPoint interface. If a requester wishes to create a pull point the following request message has to be sent to the Create PullPoint port type.

```
<wsnt:CreatePullPoint>
</wsnt:CreatePullPoint>
```

Action:
HTTP://docs.oasis-
open.org/wsn/bw2/CreatePullPoint/CreatePullPointRequest

If the CreatePullPoint request message is successfully processed, the following message is returned:

```
<wsnt:CreatePullPointResponse>
  <a:Address>url-to-consumer</a:Address>
</wsnt:CreatePullPointResponse>
```

Action:
HTTP://docs.oasis-open.org/wsn/bw-
2/CreatePullPoint/CreatePullPointResponse.

The PullPoint endpoint reference (a:Address) returned with the CreatePullPointResponse is not only the address to the PullPoint; it will also be used later when creating a subscription on the SubscriptionProducer.

5.4 NotificationProducer Porttype

The purpose of the NotificationProducer is to produce a sequence of zero or more notifications. A subscriber can register the interest of a subset of this sequence. In order to do that, the subscribers sends a subscribe message to the NotificationProducer.

No matter which order subscribe requests are made, the amount of subscribers or if NotifcationConsumers subscribes to the same topic multiple of times. WS-BaseNotifcation is never restricted in which order the notifications are produced. The NotificationConsumer can never be guaranteed to receive the notification in a particular order.

The following request is sent to NotificationProducer to create a subscription:

```
<wsnt:Subscribe>
  <wsnt:ConsumerReference>
    <a:Address>url-to-consumer</a:Address>
  </wsnt:ConsumerReference>
  <wsnt:Filter>
    <wsnt:TopicExpression
      Dialect=
"HTTP://docs.oasis-open.org/wsn/t-
1/TopicExpression/Simple"
      xmlns:pn="HTTP://PrototypeNamespace" >
      pn:root-topic
    </wsnt:TopicExpression>
    <wsnt:InitialTerminationTime>
      time-of-termination
    </wsnt:InitialTerminationTime>
  </wsnt:Filter>
</Subscribe>
```

Action:
HTTP://docs.oasis-open.org/wsn/bw-2/NotificationProducer/SubscribeRequest

The components of the subscribe request are as follows:

/wsnt:Subscribe/wsnt:ConsumerReference:

This is the url to the NotificationConsumer. In this case the url will be the one returned from the CreatePullPoint operation

/wsnt:Subscribe/wsnt:Filter:

The purpose of the filter component is to allow the subscriber to express the subset of notifications which the subscriber should receive. WS-BaseNotification defines the filter TopicExpression, ProducerProperties and MessageContent, however the thesis prototype only supports TopicsExpression.

/wsnt:Subscribe/wsnt:Filter/wsnt:TopicExpression:

The TopicExpressions filters which topic or topics the NotificationConsumer should be notified of. The grammar of the topic expression is defined by the Dialect attribute. It can either be one of the three defined in the WS-Topics specification (simple, concrete or full TopicExpression) or some other. The value of the Dialect attribute is a URL string.

This prototype supports the simple Simple TopicExpression dialect. The value of the wsnt:TopicExpression element is a QName where QName is defined as:

QName ::= PrefixedName | UnprefixedName
PrefixedName ::= Prefix ':' LocalPart
UnprefixedName ::= LocalPart

In this case the prefix is the topic namespace prefix and the LocalPart is the root topic. A topic namesspace is a collection of topics. Topics can either be root topics, where there is no parent topic, or child topics, where there are zero or more child topics. The Simple TopicExpression can filter on a single root topic which implies one SubscribeRequest has to be made per topic.

The components of the response are as follows:

```
<wsnt:SubscribeResponse>
  <wsnt:SubscriptionReference>
    <a:Address>
      url-toSubscriptionManager
    </a:Address>
  </wsnt:SubscriptionReference>
```

```
</wsnt:SubscribeResponse>
```

/wsnt:SubscribeResponse/wsnt:SubscriptionReference
    The URL to the SubscriptionManager

5.5 NotificationConsumer Porttype

The NotificationConsumer may receive notifications “raw”, i.e. as application specific content, or as a Notify message defined in the WS-BaseNotification. The prototype sends notification messages as the latter. The advantage of the Notify message is the NotificationConsumer may receive a wide range of Notifications without having to publish each Notification type in the WSDL document. It also allows a physical message to contain a batch of notifications. However, since the prototype only supports Simple TopicExpressions only one notification per Message is sent.

The following request is sent from the NotificationProducer to the NotificationConsumer:

```
<wsnt:Notify>
  <wsnt:NotificationMessage>
    <wsnt:SubscriptionReference>
      <a:Address>
        SubscriptionManager-URL
      </a:Address>
    <wsnt:/SubscriptionReference>
    <wsnt:Topic
      Dialect=
"HTTP://docs.oasis-open.org/wsn/t-
1/TopicExpression/Simple"
      xmlns:pn="HTTP://PrototypeNamespace">
      pn:root-topic
    </Topic>
    <wsnt:ProducerReference>
      <a:Address>
        NotificationProducer-URL
      </a:Address>
    </wsnt:ProducerReference>
    <wsnt:Message>
      <pn:NotifyContent>
        value
      </pn:NotifyContent>
    </wsnt:Message>
  </wsnt:NotificationMessage>
</wsnt:Notify>
```

Action:

HTTP://docs.oasis-open.org/wsn/bw-2/NotificationConsumer/Notify.

/wsnt:Notify/wsnt:NotificationMessage/wsnt:SubscriptionReference:

The URL to the SubscriptionManager

/wsnt:Notify/wsnt:NotificationMessage/wsnt:Topic:

The content of the Topic element is a TopicExpression. In this case it is a Simple TopicExpression expressing which root topic has been updated.

/wsnt:Notify/wsnt:NotificationMessage/wsnt:ProducerReference:

The URL to the NotificationProducer

/wsnt:Notify/wsnt:NotificationMessage/wsnt:Message:

This is message payload sent, according to WS-BaseNotification it can literary be anything but in this prototype it is a pn:NotifyContent element where the inner text contains a value

This operation is one way only, and does not send a return message.

5.6 PullPoint porttype

The PullPoint interface is an extension of the NotificationConsumer which allows NotificationProducers to send Notification Messages to the PullPoint. When the PullPoint receives new notifications, it can either dispose of older Notification Messages or accumulate them. This prototype saves the latest notification of each topic, and disposes of the rest.

This porttype contains two operations; the first is GetMessages where the request message looks like the following:

```
<wsnt:GetMessages>
</wsnt:GetMessages>
```

Action:

HTTP://docs.oasis-open.org/wsn/bw-2/PullPoint/GetMessagesRequest.

The response is the following:

```
<wsnt:GetMessagesResponse>
  <wsnt:NotificationMessage>
    <wsnt:NotificationMessage>

    <wsnt:NotificationMessage>
    .
    .
  </wsnt:NotificationMessage>
```

```
<wsnt:GetMessagesResponse>
```

Action:
HTTP://docs.oasis-open.org/wsn/bw-2/PullPoint/GetMessagesResponse .

/wsnt:GetMessagesResponse/wsnt:NotificationMessage/wsnt:NotificationMessage:

> The response is a simple zero or more NotificationMessage elements, where the content is exactly the same as defined in NotificationConsumer.

The second operation is DestroyPullPoint where the purpose is to terminate the PullPoint resource. The request and response messages are simply:

```
<wsnt:DestroyPullPoint />
```

Action:

HTTP://docs.oasis-open.org/wsn/bw-2/PullPoint/DestroyPullPointRequest.

```
<wsnt:DestroyPullPointResponse />
```

Action:

HTTP://docs.oasis-open.org/wsn/bw-2/PullPoint/DestroyPullPointResponse.

5.7 SubscriberManager Porttype

The purpose of the SubScriptionManager is to manipulate the subscription resource. WS-BaseNotifications define operations such as pausing and resuming subscriptions. The SubscriptionManager implemented in the prototype only supports the Unsubscribe operation, which destroys the subscription resource. In this case, a subscription resource is an individual topic. The request and response messages are simply:

```
<wsnt:Unsubscribe />
```

Action:

HTTP://docs.oasis-open.org/wsn/bw-2/SubscriptionManager/UnsubscribeRequest.

```
<wsnt:UnsubscribeResponse />
```

Action:

HTTP://docs.oasis-open.org/wsn/bw-2/SubscriptionManager/UnsubscribeResponse.

5.8 A Subscription from Beginning to the End

The picture below is a sequence diagram describing the interaction between the different kinds of components within the prototype system. Here we have a webpage which subscribes to different topics, receives the notification and finally finishes the subscriptions.

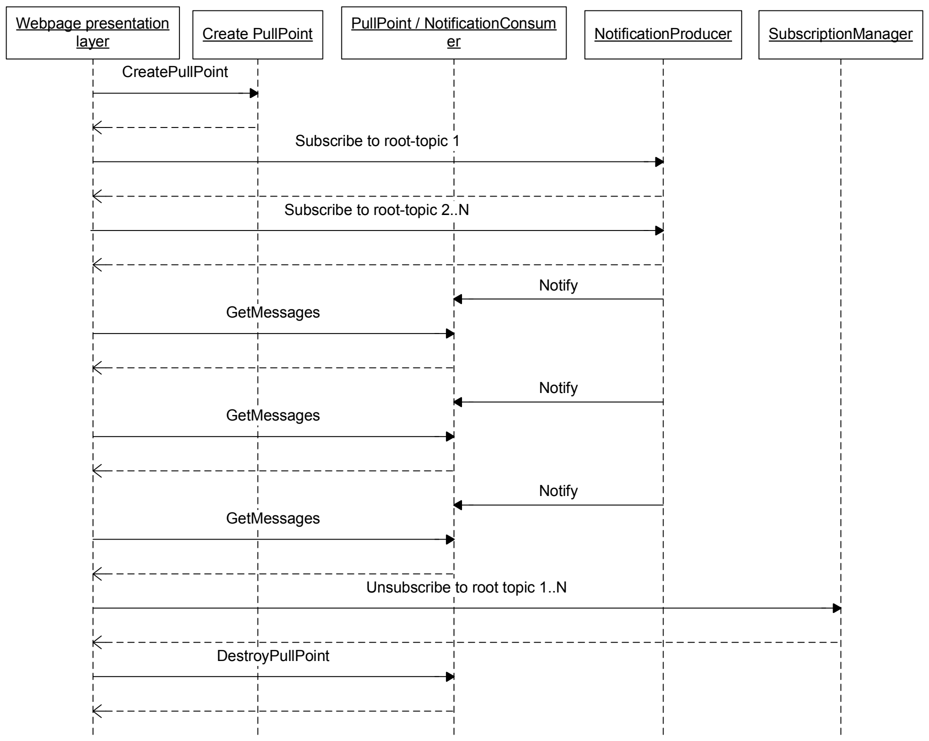

Webpage presentation layer
Create PullPoint
PullPoint / NotificationConsumer
NotificationProducer
SubscriptionManager
CreatePullPoint
Subscribe to root-topic 1
Subscribe to root-topic 2..N
Notify
GetMessages
Notify
GetMessages
Notify
GetMessages
Unsubscribe to root topic 1..N
DestroyPullPoint

Figure: 5.1: Sequence diagram describing a subscription lifetime

Chapter 6

Implementation

Introduction

This chapter describes the main parts of the prototype system and the interaction between them. All of the server side code is, as mentioned in the method chapter, implemented with the .Net 3.5 framework. From a broader perspective the system can be divided into 4 components:

1. The server side application which renders the HTML and JavaScript for the user web browser. Additionally this page performs the create pull point and the subscribe operation for the client.
2. The PullPoint webservice which exposes the Create PullPoint, the PullPoint and the NotificationConsumer.
3. The NotificationProducers which exposes the NotificationProducer and the SubscriptionManager port types
4. The webpage.

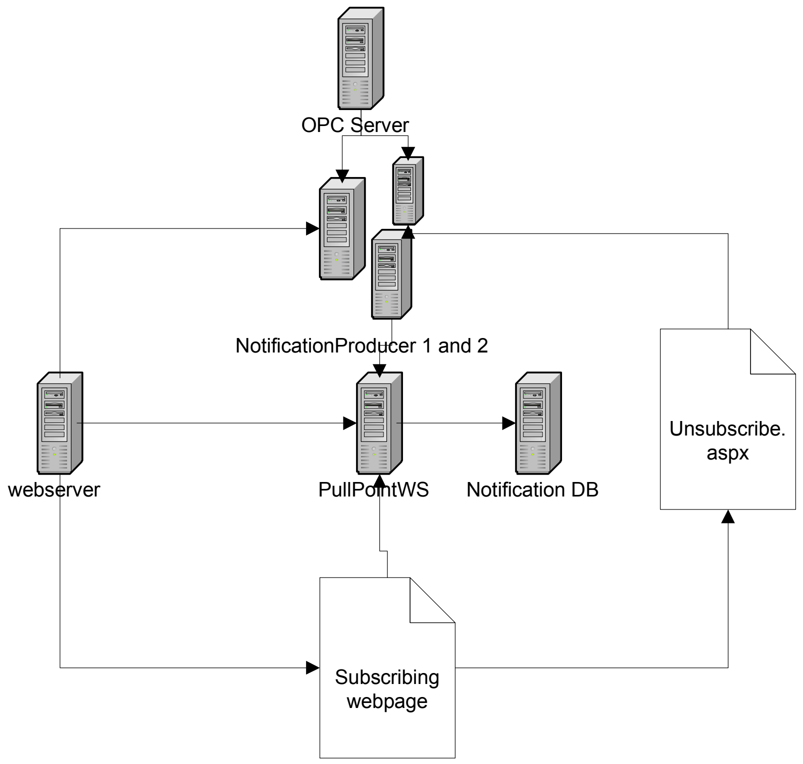


Figure 6.1: Architectural overview of the prototype system

6.1 Subscription Initiation and Page Render

When the web server starts to process the request from a web browser, it first makes a request to the PullPoint webservice to create a PullPoint. The PullPoint message returned contains the URL which the web page will poll to receive topic updates.

As the PullPoint has been created, several subscribe requests are sent to each of the NotificationProducers, with one request per topic to subscribe to. Half of the subscription requests are sent to one NotificationProducer and the other half are sent to the second NotificationProducer.

For each subscribe request, the NotificationProducers expects the URL to a NotificationConsumer. However since a PullPoint is used, the URL returned from the create pull point response is used. Thus, this is the URL the NotificationProducer sends the notifications to. Each subscribe request to the NotificationProducers is returned with the URL to the SubscribtionManager.

All the XML used when performing the Ajax calls on the web page is generated during this phase.

6.2 The PullPoint Webservice

The purpose of the PullPoint webservice is to act as a node between the web page and NotificationProducer. During the initialization phase the create pull point port type is called by the web server when the page is created. After the subscription has been created and the web page has been loaded, both the web page and the NotificationProducers will start to interact with the PullPoint webservice. The web page will, on a regular basis, call the PullPoint port type: More specifically, the GetMessages operation and the NotificationProducers will send Notification messages to the NotificationConsumer port type. Finally, when the user is finished observing the data, the DestroyPullPoint operation is called on the PullPoint port type.

6.2.1 Creating the PullPoint

When the create pull point operation is executed, a GUID is generated. This GUID will represent the ID of the subscription.
The pull point URL returned with the create pull point response is a concatenation of the URL to this webservice and the GUID. No resources or anything are created.

6.2.2 Pulling and notifying data

In order for the webservice to keep track of each subscription, it saves all the notifications in a Microsoft SQL server database. This database contains one table called Notifications:

Column name	Description
Topic	The name of the topic
Value	The topic value
Timestamp	The time when the NotificationConsumer received the notification
PullPointID	The id of the pull point
ID	The primary key of the row
ProducerReference	The URL to the NotificationProducer.
SubscriptionReference	The URL to the SubscriptionManager.

Table 6.1: The columns of the Notifications table

As the web page calls the GetMessages operation, it sends the message to the URL returned by the Create PullPoint operation. This way the PullPoint webservice can determine the pull point id by extracting the GUID from the destination address. When the web service has extracted the pull point id, it selects all the notifications from database having that the same pull point id. Before sending back the reply containing all the new notifications, it deletes them from the database.

At the same time as the web page is polling the PullPoint, the NotificationProducers are sending notification messages when the topic values have changed. When a notification message arrives, there are two possible scenarios: either the database contains an older version of the subscription topic or it does not exist. Depending on which, the row is either updated or inserted.

6.2.3 Destroying the PullPoint

At the point when the web page calls this operation all the collected notification rows for the pull points are deleted from the database.

6.3 The NotificationProducer webservice

The NotificationProducer webservices are the data level of the prototype system and are in practice OPC clients. During the initialization phase, the subscribe operation is called on the NotificationProducer port type. As soon as the NotificationProducer detects new state changes for subscribed topics, it sends notifications to the PullPoint webservice. This will continue until the SubscriptionManagers Unsubscribe operation is called.

6.3.1 Creating subscriptions

Each time the Subscribe operation is called, a new OPC subscription is created. Even though OPC subscriptions may subscribe to several OPC items, the Subscribe operation only supports the simple topic expression. Thus, each subscribe request only contain one topic and therefore each new OPC subscription will only contain one OPC item.

The OPC subscription will be initiated to perform updates every 2000 milliseconds, and the name and the client handle of the OPC subscription will be a newly generated GUID. Finally, a delegate is assigned to the subscription which points to the method on the NotificationProducer, which the OPC server will then remotely call whenever an update occurs.

In order for the NotificationProducer to keep track of which NotificationConsumer subscribes to which operation, a hash map is used; this is called a dictionary in the .Net framework. The hash map key is the subscription handle GUID and the value is the class containing the NotificationConsumer URL and the Subscription object.

The reply message contains the URL, the subscription manager, which is a concatenation of the URL to the NotificationProducer, and the OPC subscription name GUID.

6.3.2 Sending Notifications

Every 2000 milliseconds, each OPC subscription reads the value of its OPC item. Since this prototype communicates with an OPC server simulator, each OPC item will update every time an OPC subscription polls the items, however this would not necessarily be the case in a real world application.

When the OPC server detects that one or more of an OPC Subscription's items has changed value, it will remotely call the method assigned to the OPC subscription, passing the client handle and update OPC items as parameters. The method on the NotificationProducer will in turn use the client handle GUID on the hash map specified in 6.3.1 in order to get the NotificationConsumer URL. Subsequently, the updated value of the topic is sent to the NotificationConsumer's Notify operation with the URL of the NotificationProducer and the SubscriptionManager included. The URL of the SubscriptionManager is the URL of the NotificationProducer concatenated with the client handle GUID.

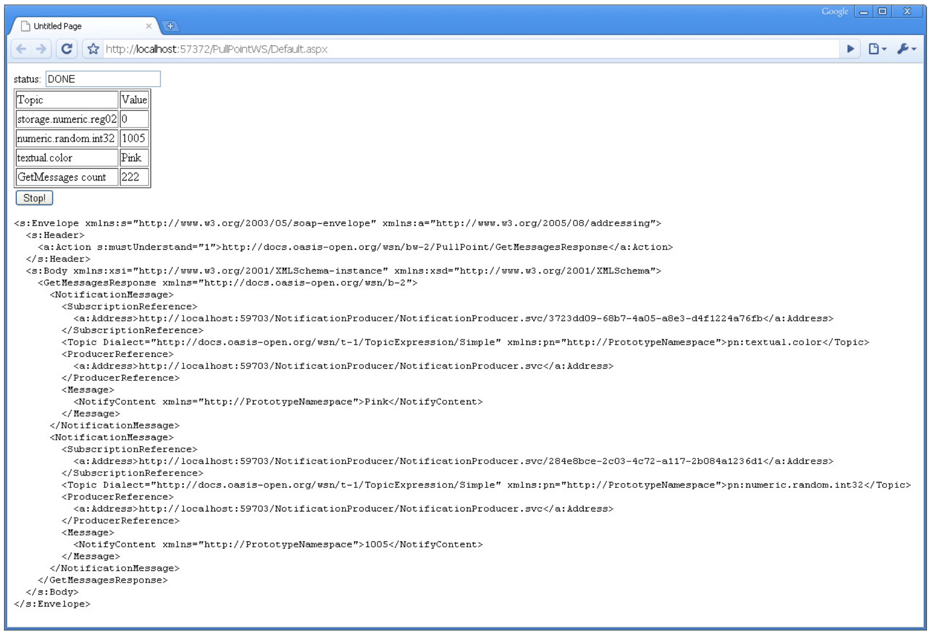
6.3.3 Unsubscribing

Whenever a subscriber wants to cancel the subscription on a topic, it sends a request to the Unsubscribe operation on the SubscriptionManager port type. The GUID attached to the destination URL is used to fetch the OPC subscription object from hash map specified in 6.3.1. This object is then used to instruct the OPC server to cancel the subscription and finally, the item is removed from hash map.

6.4 The Web Page

The purpose of the web page is simply to prove you can subscribe to topics generated by a distributed publisher subscriber system. Thus the only thing it does is to fetch the notifications generated by the

NotificationProducer. The topics and their values are displayed in an ordinary table and the latest GetMessages response is printed.



status: DONE

Topic	Value
storage.numeric.reg02	0
numeric.random.int32	1005
textual.color	Pink
GetMessages count	222

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">http://docs.oasis-open.org/wsn/bw-2/PullPoint/GetMessagesResponse</a:Action>
  </s:Header>
  <s:Body xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <GetMessagesResponse xmlns="http://docs.oasis-open.org/wsn/b-2">
      <NotificationMessage>
        <SubscriptionReference>
          <a:Address>http://localhost:59703/NotificationProducer/NotificationProducer.svc/3723dd09-68b7-4a05-a8e3-d4f1224a76fb</a:Address>
        </SubscriptionReference>
        <Topic Dialect="http://docs.oasis-open.org/wsn/t-1/TopicExpression/Simple" xmlns:pn="http://PrototypeNamespace">pn:textual.color</Topic>
        <ProducerReference>
          <a:Address>http://localhost:59703/NotificationProducer/NotificationProducer.svc</a:Address>
        </ProducerReference>
        <Message>
          <NotifyContent xmlns="http://PrototypeNamespace">Pink</NotifyContent>
        </Message>
      </NotificationMessage>
      <NotificationMessage>
        <SubscriptionReference>
          <a:Address>http://localhost:59703/NotificationProducer/NotificationProducer.svc/284e8bce-2c03-4c72-a117-2b084a1236d1</a:Address>
        </SubscriptionReference>
        <Topic Dialect="http://docs.oasis-open.org/wsn/t-1/TopicExpression/Simple" xmlns:pn="http://PrototypeNamespace">pn:numeric.random.int32</Topic>
        <ProducerReference>
          <a:Address>http://localhost:59703/NotificationProducer/NotificationProducer.svc</a:Address>
        </ProducerReference>
        <Message>
          <NotifyContent xmlns="http://PrototypeNamespace">1005</NotifyContent>
        </Message>
      </NotificationMessage>
    </GetMessagesResponse>
  </s:Body>
</s:Envelope>
```

Picture 6.2: Screenshot of prototype webpage printing notifications

Since the subscription initializations occur on the server before the web page is rendered, the first interaction the web pages makes is to the pull point webservice. The request message used in order to call the GetMessages operation has already been generated, and the web page will keep on sending the same request message until the stop button is clicked.

After the web page receives the response from the GetMessages operation, it checks whether any topics have changed value by parsing the XML document. If one or more topics have changed value, the corresponding rows in the table are updated.

Whenever the user clicks the stop button, the web page starts to send UnsubscribeRequest messages to a web page called Unsubscribe.aspx. This page is a proxy page which forwards the UnsubscribeRequest messages to SubscriptionManager. For every topic subscribed to, a separate UnsubscribeRequest message is sent. All of the UnsubscribeRequest messages have been created during the rendering of the web page, as were the GetMessages request messages.

For security reasons, the XMLHTTPRequest object prohibits any request anywhere except the domain which hosts the web page [20]. In this prototype the NotificationProducer webservice is not hosted within the same domain, and therefore messages needs to be forwarded. The PullPoint webservice is however hosted within the same domain; if that were not the case, a similar GetMessages proxy page would have to exist as well.

Chapter 7

Conclusion

7.1 Discussion

The goal of the thesis was to solve two kinds of problems, the first of which was to find a way to retrieve data from a server without doing any post back, and the second problem was to build a more scalable system.

The first problem was solved by implementing the periodic refresh technique with the XMLHTTPRequest object. Despite the fact the other techniques, HTTP streaming and long polling, makes the topic changes appear faster on the clients. They are also significantly more challenging to implement. Most importantly they require the server to be able to handle multiple idle request threads. I found that the .Net framework by default handles idle request threads quite poorly. This is because the .Net framework can deal with relatively few requests simultaneously, which is not a problem if the request are processed quickly (milliseconds). However in this case when threads idles for several seconds, which is a very long time in this perspective, many more clients can be served by using the periodic refresh technique.

That being said, if I had to select between HTTP streaming and long polling, I would have chosen the long polling technique for the main reason that not all web browser supports HTTP streaming. Though long polling might return updates slightly slower because of the fact it has to establish new connections after a response. I think there are very few cases where the users will notice any big difference, and if it is extremely important the clients receive the updates immediately. The client applications should not depend on web browsers in the first place.

The second problem was solved by implementing WS-BaseNotification specified webservices. In theory, this would allow webservices supplied by different companies to communicate with each other. For example, it would be relatively easy to replace a NotificationProducer from a third party company. Or a third party NotificationConsumer can subscribe to topics from my NotificationProducer. Unfortunately as far as I am aware, there is not much commercial software on the market which exposes this specification or the WS-Eventing specification for that matter. Thus the main success in implementing the WS BaseNotification is the ability to implement any prototype system using a unifying standard, were one to become available.

From a scaling perspective dividing the system into NotificationProducers, NotificationConsumers, PullPoints etc. is a great solution. This way it is possible to scale both horizontally and geographically. If there are multiple servers hosting an OPC server deployed at different locations. Each server would only have to expose a NotificationProducer in order for a NotificationConsumer to communicate with them. Furthermore if there is a huge work load on the PullPoint serving thousands of clients. It would be possible to scale horizontally using several servers which would host the PullPoints.

One flaw with current prototype is the NotificationProducers cannot be added to the system dynamically; all of them have to be known by system during the runtime. One possible solution to this is to implement the WS-BrokeredNotification.

7.2 Future work

An interesting topic would be how to implement a webservice which could process a large number of long polling clients. This is probably a substantial topic on its own.

Moreover I think it could be interesting to investigate how to scale PullPoint's from a horizontal perspective. There are opportunities to choose from, such as investigating if SQL server is an ideal datastore, or how to implement the Create PullPoint operastion to achieve optimal load balance.

Bibliography

[1] Andrew S. Tanenbaum and Martin van Steen. *Distributed systems principles and paradigms.* Prentice Hall, Upper Saddle River, NJ, USA, 2002.

[2] Gustavo Alonso, Fabio Casatio, Harumi Kuno and Vijay Machiraju. *Web services Conceps, architectures and applications.* Springer Verlag. Berlin Heidelberg, Germany, 2004.

[3] Jean Dollimore, Tim Kindberg and George Coulouris. *Distributed Systems: Concepts and Design, Third Edition.* Addison-Wesley, Harlow, England, 2001

[4] Elisa Bertino, Lorenzo D. Martino, Federica Paci and Anna C. Squicciarini. *Security for Web Services and Service-Oriented Architectures.* Springer Verlag, Berlin Heidelberg, Germany, 2004.

[5] W. Roy Schulte and Daniel Sholler, WS-Notification Standard Ratified by OASIS Still Needs Work. Available: HTTP://www.gartner.com/DisplayDocument?doc_cd=144177

[6] Don Box, Luis Felipe, et al., "Web Services Eventing (WS-Eventing)", Available: HTTP://www.w3.org/Submission/WS-Eventing/

[7] Steve Graham, David Hull and Bryan Murray, "Web Services Base Notification 1.3 (WS-BaseNotification)", Available: HTTP://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf

[8] Dave Chappell and Lily Liu, "Web Services Brokered Notification 1.3 (WS-BrokeredNotification)", Available: HTTP://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-os.pdf

[9] William Vambenepe, Steve Graham and Peter Niblett, "Web Services Topics 1.3 (WS-Topics)", Available: HTTP://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf

[10] Steve Graham, David Hull, et al., "Web Services Notification", Available: HTTP://www.ibm.com/developerworks/library/specification/ws-notification/

[11] Yi Huang and Dennis Gannon, "A Comparative Study of Web Services-based Event Notification Specifications", Available: HTTP://www.cs.indiana.edu/~yihuan/research/yhuang-comparativeStudy.pdf

[12] Frank Iwanitz and Jürgen Lange. *OPC, fundamentals, implementation and application*. Hüthig GmbH & Co. KG Heidelberg, Germany, 2006.

[13] OPC Foundation, *OPC Data Access Custom Interface Specification Version 3.0*

[14] "Dynamic HTML and XML: The XMLHTTPRequest Object", HTTP://developer.apple.com/internet/webcontent/xmlhttpreq.html

[15] Michael Mahemoff. *Ajax Design Patterns*. O'Reilly Media, Inc, Sebastopol, CA, 2006

[16] Dylan Schiemann, "The Long-Polling Technique", Available: HTTP://cometdaily.com/2007/11/15/the-long-polling-technique/

[17] "Implementing Client Callbacks Programmatically Without Postbacks in ASP.NET Web Pages", HTTP://msdn.microsoft.com/en-us/library/ms178208.aspx

[18] "What is ASP.NET Ajax", Available: HTTP://www.asp.net/AJAX/what-is-ajax/

[19] Don Box, Erik Christensen, et al., "Web Services Addressing (WS-Addressing)", Available: HTTP://www.w3.org/TR/XMLHTTPRequest/ [21] "About the WebService Behavior", Available: HTTP://msdn.microsoft.com/en-us/library/ms531032(VS.85).aspx