Ecole Polytechnique,
PROMOTION X2001
MARINOIU Bogdan-Eugen

RAPPORT DE STAGE D'OPTION SCIENTIFIQUE

Development environment for dynamical Web data

Non - confidentiel

Option	:	Informatique
Champ de l'option	:	Bases de données
Directeur d'option	:	Monsieur Gilles Dowek
Directeur du stage	:	Monsieur Serge Abiteboul
Dates du stage	:	12 Avril – 12 Juillet 2004
Adresse de l'organisme	:	INRIA Futurs Parc Club Universitaire Orsay, 91400, France

Abstract

English

During my internship in the “GEMO” Team at INRIA Futurs Research Unit in Saclay, I worked in the context of the ActiveXML, a declarative framework for data management and integration. The goal of my internship was to develop a powerful tool for interacting with ActiveXML Peers and handling ActiveXML documents.

The first result of my internship is the existence of the **AXML Browser** – a Mozilla based application which permits the interaction with an AXML document in the local space: instant activation of the service calls in the document and some modifications of the attributes and the parameters of the service calls.

The second result is the **AXML Commander** – a client which permits opening connections on several peers at the same time, searching for documents on the peer and importing them. After being handled locally, they can be moved to other peers or saved on the peer of origin. The local handling of documents includes editing them and opening them with the **AXML Browser** described above.

The report consists in 5 chapters. The first one is an introduction, the second one presents the various technologies and standards with which I interacted during my internship, the third one is a presentation of the **Active XML Project**, the fourth is about the **AXML Browser** and **AXML Commander**. The fifth opens the way to new possible developments of the subject.

Français

Pendant mon stage dans le projet “GEMO” de l’Unité de Recherche INRIA Futurs de Saclay, j’ai travaillé avec le Active XML, une structure déclarative pour la gestion et l’intégration des données. Le but de mon stage a été le développement d’un outil performant pour interagir avec les Active XML Peers et pour la gestion des documents Active XML.

Le premier résultat de mon stage c’est l’existence de l’ **AXML Browser** – une application Mozilla qui permet l’interaction avec un document AXML dans l’espace local : l’activation des appels de services dans le document et quelques modifications des attributs et des paramètres des appels de services web.

Le deuxième résultat est le **AXML Commander** – un client qui permet l’ouverture des connections sur plusieurs peers au même temps, la recherche des documents sur le peer et de les importer. Après la manipulation locale, ils peuvent être transférés sur les autres peers ou sauvés sur le peer d’origine. La manipulation locale des documents inclue leur édition et leur ouverture avec le **AXML Browser** décrit plus haut.

Le rapport a 5 chapitres. Le premier est une introduction, le second présent les différents technologies et standards que j’ai rencontrés pendant mon stage, le troisième est une présentation du projet **Active XML** mon travail pendant le stage en étant une partie, le quatrième portent sur **AXML Browser** et **AXML Commander**. Le cinquième chapitre est une ouverture sur les possibles développements du sujet.

Contents

Chapter 1 Introduction

1.1 Context

I am an Ecole Polytechnique student in the 3rd year. From April until July 2004 I was an intern at INRIA Futurs, in the **Gemo Project** Team and I worked on developing instruments for human-**Active XML** interaction.

1.2 Environment

1.2.1 The ***Gemo*** Project
Distributed Web Data and Knowledge Integration

Gemo follows the **Verso** project of INRIA Rocquencourt and its team is made of ex-members of the team of that project and of members of the IASI Team of the *Laboratoire de Recherche en Informatique* (UMR 8623 CNRS) of Paris South University.

The objective of the group is to study the fundamental problems of the modern *Data and Knowledge Management Systems* and to develop suitable new solutions to them. The goal is to obtain systems more open towards richer and more network oriented information, more precisely discover pertinent information or services, understand their semantics, integrate them and study their evolution The Gemo central theme is the integration of those data. By combining approaches such as mediation and warehousing the project investigates how to integrate web services exchanging XML data. One main theme of research is focused on **Active XML**, a model in which XML documents incorporate web service calls.

1.2.2 People

The director of my internship was Serge Abiteboul. Tova Milo, Serge Abiteboul, Omar Bengelloun and Jerome Baumgarten provided me with priceless guidance in my researches. I am most grateful towards them.

1.2.3 Active XML

The **Active XML** was the framework with which I interacted. It was developed in Java because this programming language is portable and provides many instruments for XML parsing, XSLT transformations and XPath evaluations. This project uses third party modules that came under free license and are all part of the Apache project: Tomcat servlet engine, Axis SOAP engine, and Xerces Java XML parser.

Chapter 2 Technologies and Standards

In the first section we compare *Peer-to-Peer* versus *Client/Server* architectures while in the second we describe some technologies that form the context of our work.

2.1 Peer-To-Peer vs. Client/Server Architectures

P2P computing is actually not a new concept. The name is new but the model of computing underneath exists since the beginning of the Internet. The Internet started as a network of computers (peer nodes) which were communicating directly to one another. Computing models, software and network architectures evolved and most of them embraced the major notions of client and server, roles which were performed by different machines.

Presently, the old fashion seems to turn back, encouraged by the important technological gains in the past years resulting in cheap computers, cheap bandwidth, cheap storage and idle processor cycles. P2P computing can be defined as direct collaboration between nodes which no longer rely on a network server. The fundamental concept is sharing: data, processing cycles, resources such as storage and printers. Inside an organizational network this increases overall performance by relieving some of the burden off the servers.

Having one single role is quite restrictive for a machine. Thus, the system is rigid, incapable to evolve and to adapt itself to the changing needs. In P2P architecture, a node can be, at the same time, server and client for various services on the network.

2.2 XML and data

2.2.1 A little history

W3C (World Wide Web Consortium) started to work on eXtensible Markup Language in 1996. XML 1.0 has been released on February 10, 1998 and responds fully to IT industry's need to develop a simple and efficient mechanism for the textual representation of the structured and semi-structured data. It has been influenced by the Standard Generalized Markup Language **(SGML)** and HTML. SGML is a meta-language, that is a language that offers the possibility to specify any given markup language.

XML is a declarative language that, like all the generalized markup (GM) languages, uses tags to identify pieces of information. This way, the content of such a document is both human- and machine- easily understandable.

HTML is, perhaps, the most successful *SGML application*. But because it uses a fixed set of tags with standard meaning it is not very easy to extend. So, there was a need to control the evolution of HTML and create a simple generalized markup language for use on the web. XML has taken the place of SGML as a "light-weight language". Nowadays, XML has become the *de facto* standard for representing structured and semi-structured information in textual form. There are many *XML languages* – in fact, specifications built on top of XML to extend its capabilities for broader range of applications. One of the areas of intense use of XML is the Web Services.

2.2.2 XML properties

If we compare XML with HTML, the main difference we found is that HTML uses predefined tags (that is tags with specific meaning for a browser for example) while XML uses tags to represent data. HTML is document-centric while XML is data-centric. A HTML document contains some (unstructured) data as well as the way to display it (tags represents specific instructions for the browsers) while XML represents (semi-) structured data without any specification of displaying the data.

XML has some features that imposed it as an emerging model for data management and exchange.

- ***Portability***

The XML documents are, actually, text documents, so, they are portable.

- ***Transformability***

A XML document can be presented in several ways, using transformation languages like XSL (*eXtensible Stylesheet Language* – a XML language itself). They can transform XML in HTML for representation, but they can transform XML in XML too – which means XML processing is enabled

- ***Easy querying.***

Since XML documents have a tree-like structure, they are very suited for querying. Many querying languages emerge in the field. One that I used intensely during my internship is XPath.

2.2.3 XML Basics

Let's consider an example for XML which I'll use to outline the major feature of the language. The example shows a XML document that could be used for ticket registration storing ticket registration information.

```
<ticket-order
       xmlns=http://www.eugenm.home.ro
        id = "74564353432"
 >
   <passenger number="1">
      <name> Bogdan </name>
      <lastname> Marinoiu </lastname>
      <address>
          <apartment-number> 273 <apartment-number>
          <building-number> 5 </ building-number>
           <street>  Pacaterie </street>
           <town>  Orsay </town>
           <postal-code> 91400 </postal-code>
       <country> France </country>
       </adress>
  </passenger>
  <passenger number="2">
         …
    </passenger>
<travel-information>
  <from> Paris-Galieni </from>
  <to> Bruchsal </to>
</travel-information>
<price>
```

```
    <currency> EUR </currency>
    <value> 76.00 </value>
</price>
</ticket-order>
```

XML Elements

The term *element* is used for the pairing of a start and end tag in an XML document. The "ticket-order" element has the start-tag "<ticket-order>" and the end-tag "</ticket-order>". Every start/end tag must have an end/start counterpart and everything between the two tags is the content of the element. This includes nested element, text, comments.

According to XML specification, elements can have three different content types: *element-only* content, *mixed* content or empty content. XML documents can be seen as tree-like structures but one constraint is that these trees have order (the set of children for a certain element node is ordered) ex: for *passenger,* the set of child-elements: *name, lastname, address* is always in this order.

As with semi-structured data we may use repeated elements with the same tag to represent collections (i.e. *passenger* element).

XML Attributes

XML allows us to associate *attributes* with elements. An attribute is a name-value pair that appears in the start tag. As with tags, users can define arbitrary attributes but their value is always a string and must be enclosed in quotation marks.

In the above example, "number" is an attribute of the elements "passenger". We assume that several passengers can travel in a group and have a group ticket.

There is a special attribute the elements generally can have and that is "id" attribute. The value must be unique in the document and permits the rapid identification of the element in the document. We can also use the "id" attribute to eliminate duplicate information: we write it only once and we can refer to it later using an "id/idref" mechanism.

XML Namespaces

The namespaces are a way to reuse XML. The mechanism consists in associating document elements with a specific URI and represent a (possible) response to the problem of collision that arises in composed XML documents because of the likelihood of common name elements(ex: item) to be reused in different document types. The problem is addressed by using a *qualified name* the element which is likely to be unique in the composed document:

QName = Namespace Identifier + Local Name.

XML Namespaces uses URIs (Uniform Resource Identifiers) as namespace identifiers which are described in RFC 2396. One type of URIs are the well-known URLs. In our example I used the URL http://www.eugenm.home.ro in order to qualify all the elements in the document: the *default* namespace for that document is, as a consequence, http://www.eugenm.home.ro. So, the element *passenger* has actually, the qualified name:http://www.eugenm.home.ro: passenger

2.3 Queries on XML: the XPath language

The interrogation of databases and of semi-structured data in XML documents has been an important research topic in the past years and this research gave birth to some languages of which I will present the one I used during my internship.

XPath was released as a W3C Recommendation 16. November 1999 as a language for addressing parts of an XML document. XPath was designed to be used by XSLT, XPointer and other XML parsing software.

It's in a way an extension to the paths in Operating Systems. The location path can be absolute or relative. Contrary to a relative path, an absolute location path starts with a slash (/) . In both cases the location path consists of one or more location steps, each separated by a slash.

Path expressions represent the bricks of any query language for semi- structured data. They allow expressing a form of constraint navigation in the set of nodes of a labeled tree. The idea is that for an edge-labeled directed graph

A sequence of edge labels *l*1.*l*2...*in* is called a *path expression*. The location steps are evaluated in order one at a time, from left to right. Each step is evaluated against the nodes in the current node-set. If the location path is absolute, the current node-set consists of the root node. If the location path is relative, the current node-set consists of the node where the expression is being used.

Location steps consist of:

- an axis (specifies the tree relationship between the nodes selected by the location step and the current node). This is actually optional, the implicit value is: child
- a node test (specifies the node type and expanded-name of the nodes selected by the location step)
- zero or more predicates (use expressions to further refine the set of nodes selected by the location step) . This one is optional too since it does a filtering.

The syntax for a location step is: **axisname::node[predicate]**

There are twelve axes along which a location step can move. Each selects a different subset of the nodes in the document, depending on the context node. These are: self, child, descendant, descendant-or-self, parent, ancestor, ancestor-or-self, preceding, preceding sibling, following, following-sibling, attribute, namespace.

An XPath expression can return : nodes, set of nodes, numbers, character arrays.

2.4 Processing XML: DOM & SAX

The processing of the XML documents can be done in two ways:

- a declarative way by using XSL, XSLT
- a procedural way by writing procedures:
  - syntactic analysis (parsers)
  - build an internal representation of the XML document
  - the processing of that representation

- automatic generation of the document

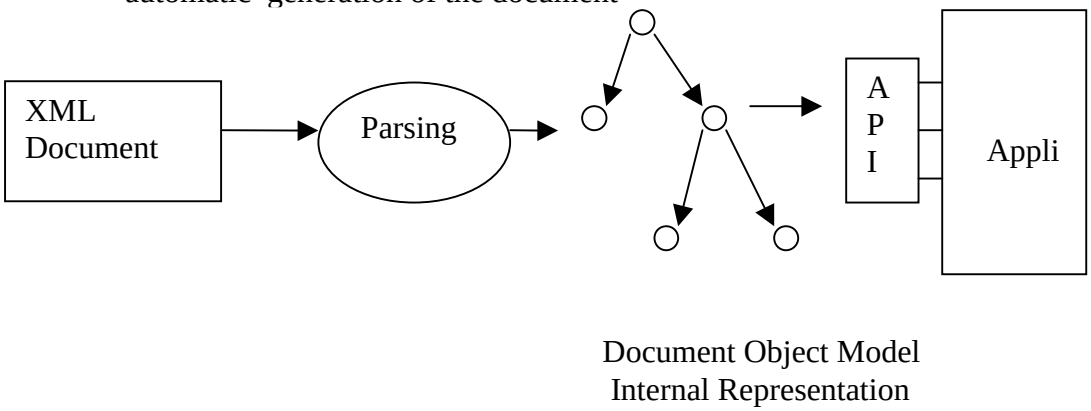

Figure 2.1 The DOM way of operating

The so-called "parsers" which implement the steps 1, 2 and 4 are already written and for the 3rd step, there is an API which is made available to the user.

For an application to be able to work with a XML document, it should first be able to parse it. *Parsing* is a process that involves breaking up the text of an XML document into small identifiable pieces (nodes). These pieces are fed into the application using a well-defined API implementing a particular parsing model. The parsing models commonly used include:

Ø *One-step parsing*

The parser reads the whole XML document and generates a data structure (parse tree) describing its entire content. The data structured obtained is deeply nested and corresponds as hierarchy to the nesting of elements in the parsed XML document. The W3C has defined *Document Object Model (***DOM***)* for XML. It has become so popular that the one-step parsing is usually referred to as: *DOM* parsing. The DOM is a language- and platform- independent *API.* Its biggest drawback is the huge memory-cost. The DOM architecture is the one in the figure 1.

Ø *Push Parsing*

The application receives notifications from the parser about the XML document pieces it encounters during the parsing process. The notifications are typically implemented as event callbacks in the application code. The XML community created a *de facto* standard for push-parsing called the *Simple API for XML (***SAX***).* The architecture for that kind of parsing is shown in the picture below:

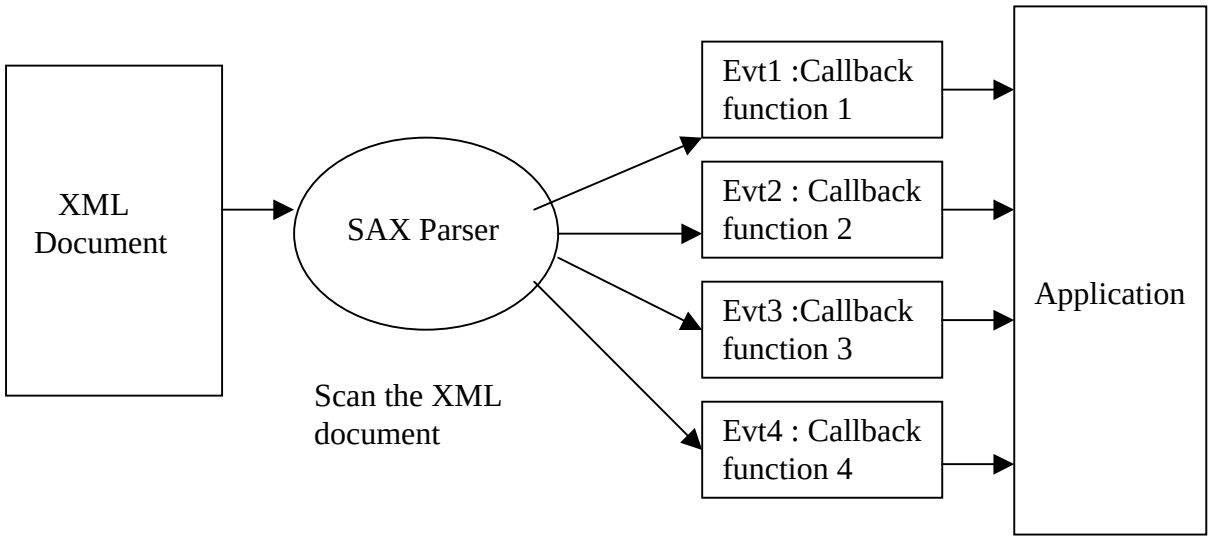

Figure 2.2 The SAX Parser way of operating

If we analyze those two approaches in the same time we can see that the sequential model that SAX provides does not allow for random access in an XML document while the DOM approach can be slow and costly in terms of resources so choosing one or another depends on the type of the application.

2.5 Transforming XML documents : the XSL language

In order to provide the user with more flexibility in styling his XML and HTML documents, the CSS (Cascading Style Sheets) were adopted as an emerging technology. Using it, the developer can change attributes (like the ones for the font) and can even make animations.

XSL is a XML based language (any XSL document is an XML valid document with ".xsl" extension) that provides the user with the possibility of programming besides the ones offered by the CSS. It is used for styling and for XML documents transformation (XSLT = XSL Transform). XSLT is a declarative programming language (a program = collection of rules like in CLIPS). Because it is declarative, it supports extensions easily (programming by examples)

An XSL document starts with:

```
<? xml version="1.0"?>
<xsl:stylesheet xmlns:xsl=http://www.w3.org/1999/xsl/Transform>
```

because the special namespace should be used as an indication for the XSLT processor.

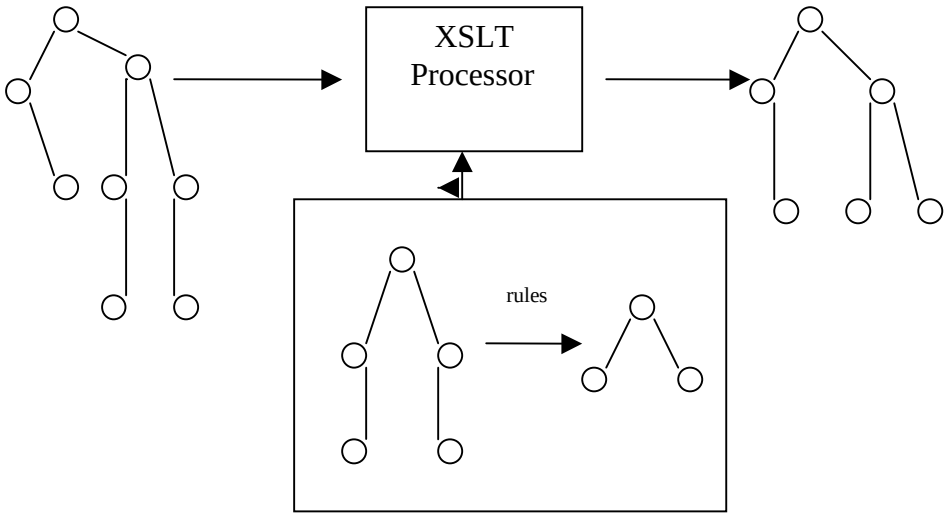

Figure 2.3 XSLT Processing Schema

The XSLT rules have a conditional part (pattern matching: generally XPath expression) and an action part (tree fragment result).

The rules are bounded in “templates”:

```
<xsl:template match = “XPath expression”>
   … tree fragment
</xsl:template>
```

One can explicit invoke templates with *apply-templates / call –template* instructions. A match can be done on multiple rules: the XSLT processor has a strategy for conflict resolution:

- the most specific rule is applied
- priorities can be assigned to templates

2.6 WEB Services

This term has seen multiple interpretations. Many organizations are involved in the refinement of the Web Services standards and therefore, although it seems to be a slow convergence towards a common understanding of the term, there is no single definition for it.

The definition found on the W3C official site, states the following:

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.]

Several main characteristics of the web services need to be outlined:

- Web Service need not necessarily exist on the World Wide Web. This is a naming issue, because, actually, the Web Services can exist anywhere on the network and others can be used using only a simple method invocation in the processes running on the same computer.
- Web Service’s implementation and deployment platform details are not relevant to the program invoking the service. The Web Service is available through its invocation mechanism (network protocol, data encoding schemes etc.)

Web services provide a standard mean of interoperation between applications of different types running on different platforms.

There are many protocols/languages involved in this architecture. HTTP and SMTP which are standard network protocols are only a part.

XML imposed itself as the standard for data transfer and data representation. But it is complemented by other languages. Several XML languages/ protocols can be found around the Web services. The WSDL (Web Services Description Language) is a XML language. The basic unit of communication in the Web Services Architecture (WSA) is the SOAP message and SOAP is an XML language too.

Web Services technologies can be factored into three stacks of which I will detail only the first:

- The wire stack
- The description stack

- The discovery stack

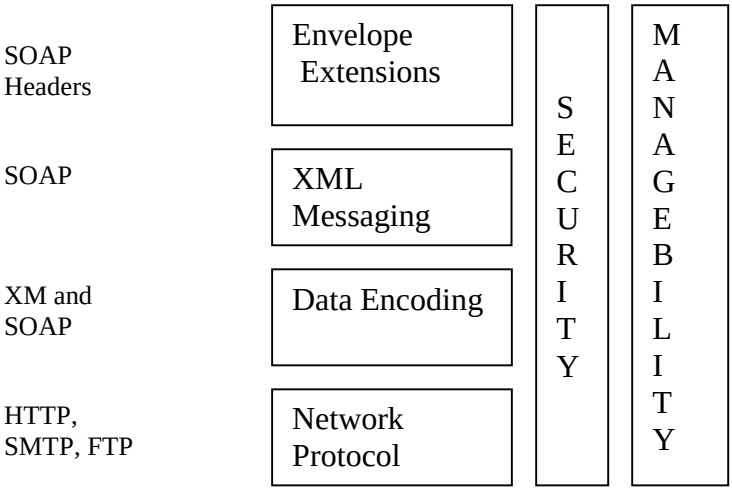

Figure 2.4 The Web Services wire stack.

The wire stack represents the technologies that are involved in sending a message from the service requestor to the service provider. The network protocols at the base of the stack can be standard Internet wire-protocols like HTTP, HTTPS, SMTP, FTP or sophisticated enterprise-level protocols.

For data encoding, Web Services use XML. The XML messaging layers which are on top of this use SOAP in all the data encoding, interaction style and protocol binding variations. SOAP is as an XML wrapper in an envelope. On top of the SOAP enveloping mechanism is a mechanism for envelope extensions called SOAP headers which allow the association of orthogonal extensions such as digital signatures with XML digital signatures and XML cryptography.

The description stack involves mainly WSDL while the discovery stack involves UDDI (Universal Description Discovery and Integration).

W3C defines WSDL as following:

WSDL is ***an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints(services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate****.*

It describes several critical pieces of information a service client would need:

• the name of the service, including its URL.
• the location the service can be accessed at.
• the methods available for invocation.
• the input and output parameter types for each method.

2.6.1 The Web Services Architecture

The Web Services Architecture can be described by the figure 2.5.

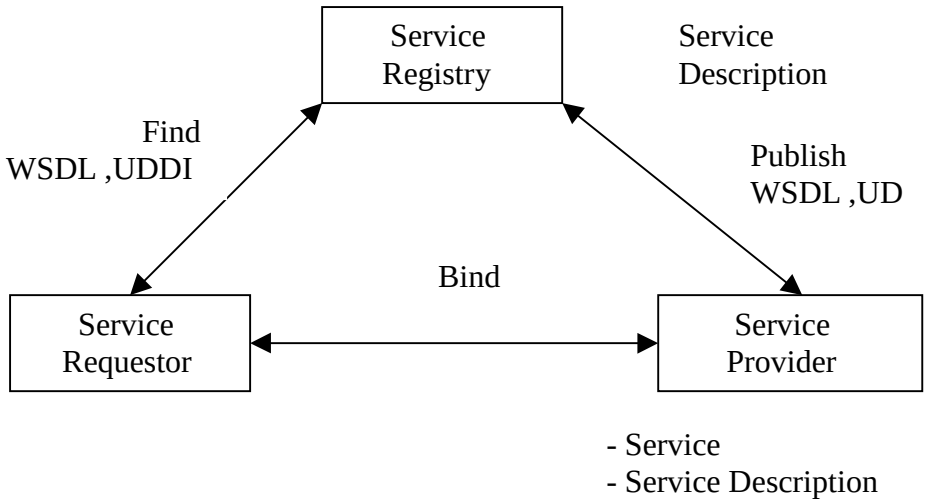

Figure 2.5 The Web Services Architecture

Any service-oriented architecture contains three roles:

- a service provider, who creates a service description, publishes it and receives web services invocation messages.
- a service requestor who has to find a service description published to one or more service registries and use it in order to bind to or invoke the web services.
- the service registry is responsible for advertising Web service descriptions published by the service providers and allowing the service requestors to use the service registry for finding the descriptions. The above figure outlines all of these.

Three operations are involved in this architecture:

- the publish operation – act of service registration or service advertisement.
- the find operation – the service registry matches its collection of descriptions against a search criteria stated by the service requestor and returns a list of service descriptions to the service requestor.
- the bind operation can take two forms: an on-the-fly generation of a client-side proxy based on the service description used to invoke the Web service or a very static model where the developer specifies the way the client invokes a Web service.

2.7 Simple Object Access Protocol (SOAP)

The official W3C definition of the SOAP Protocol is the following:

SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses.

Basically, SOAP provides the following:

- a mechanism for defining the unit of communication. In SOAP all the exchanged information is packaged into SOAP messages. The SOAP envelope encloses all the information and the SOAP body can contain arbitrary XML.
- a mechanism for error handling which allows the identification of the source and of the error nature through the SOAP Fault notion
- a mechanism for introducing extensions via SOAP headers
- flexible mechanism for data representation : data already serialized in some format (XML, but text too) and convention for representing abstract data structures such as programming languages data types
- a convention for representing Remote Procedure Calls (RPCs – the most common type of distributed computing interaction) and responses as SOAP messages
- a binding mechanism between SOAP and the underlying HTTP protocol

Example:

Let's say we want to access a service which, through the method getTime which takes two arguments: *locale* and *timezone*, provides us with a text response representing the date and time in a town. The answer is in the country's language.

The SOAP request:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><s
oapenv:Header/>
   <soapenv:Body>
     <a0:getTime xmlns:a0="Time">
      <locale> it </locale>
      <timezone> Europe/Rome </timezone>
     </a0:getTime>
   </soapenv:Body>
</soapenv:Envelope>
```

The SOAP response:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
```

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
   <ns1:getTimeResponse
   soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:ns1="Time">
      <ns1:getTimeReturn xsi:type="xsd:string">
       martedì, 22 giugno 2004 15:41:25 Ora estiva Europa centrale
      </ns1:getTimeReturn>
    </ns1:getTimeResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Actually the two examples above are true examples from some of the tests of my application. The values for the parameters were *it* and *Europe/Rome.* Those parameters specify that the requested answer should be in Italian and that the town is the European city of Rome.

The response is: *"martedì, 22 giugno 2004 15:41:25 Ora estiva Europa centrale"*

2.8 The Mozilla framework

2.8.1 Mozilla is everything you need.

For my application I chose to use the Mozilla framework which provides a good support for such applications. The Mozilla Application Framework allows easy development of cross-platform applications.

To benefit from the **ActiveXML Commander** one has to have installed only Mozilla on his computer. The application can be downloaded from a web server and then it will be installed automatically using XPI.

The **Mozilla Project** was started in March 1998 with the goal of developing the successor to Netscape's Communicator 4.x browser suite. Today Mozilla is used by developers as a platform for creating applications that can be installed locally or run remotely over the Internet.

For a company committed to creating an application that runs on a wide range of different systems, using platform-specific code was a big waste of time. **XPFE**, Mozilla's cross-platform front end, was designed to solve this problem by enabling engineers to create one interface that would work on any operating system.

Perhaps the biggest advantage Mozilla has for a developer is that Mozilla-based applications are cross-platform, which means that these programs work the same on Windows as they do on Unix or the Mac OS. It's possible to have applications run across different platforms because Mozilla acts as an interpretation layer between the operating system and the application.

As long as Mozilla runs on a given computer, most Mozilla-based applications also run on that computer, regardless of what operating system it uses. Not all Mozilla applications are cross-platform however, since it is possible to create an application with platform-specific code that runs only on certain operating systems.

XPFE uses a number of existing web standards, such as Cascading Style Sheets, JavaScript, and XML (the XML component is a new language called XUL, the XML-based User-interface Language). In its most simple form, XPFE can be thought of as the union of each technology. Viewed together, these technologies can be seen forming the XPFE framework in the figure 2.6.

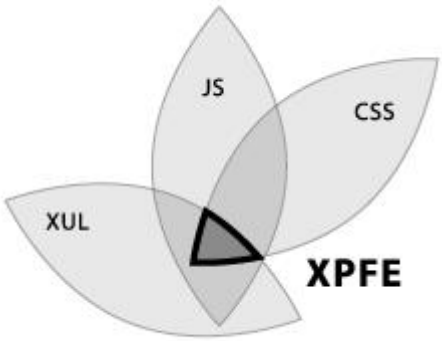

Figure 2.6 The XPFE framework

To understand how XPFE works, we can look at how these different components fit together. JavaScript creates the functionality for a Mozilla-based application, Cascading Style Sheets format the look and feel, and XUL creates the application's structure.

Instead of using platform-specific code in languages like C or C++ to create an application, XPFE uses well-understood web standards that are platform-independent by design. Because the framework of XPFE is platform-independent, so are the applications created with it. Since the framework is also made up of some of the technologies used to create web pages, people familiar with creating a web page can learn how to use XPFE to create a cross-platform application.

So what is this framework made up of?

- **XUL**: a comprehensive, cross-platform UI (User Interface) toolkit that provides the structure and the content of the application
- **Gecko**: a performance web content rendering/editing engine with support for standards that can drop into the application with a single line of XUL;
- **XPCOM/XPCONNECT** : used to allow JavaScript, or potentially any other scripting language, to access and utilize C and C++ libraries
- **JavaScript**: Used to create the functionality of an application, although other scripting languages, such as Python, Perl, or Ruby, can be used in place of JavaScript.
- **Web Services**: support for XMLHttpRequest, XML-RPC, SOAP, and WSDL
- **XPInstall**: the cross-platform installer for trivial packaging and installation of the Mozilla framework applications;

2.8.2 Web Services Calls in Mozilla

2.8.2.1 Mozilla SOAP API

Until the Mozilla 1.0, the browsers had nothing to do with the SOAP clients, since composing SOAP messages and connecting with Web services was better suited to server-based applications designed specifically for those tasks. A Web application running in Mozilla (or in a client using the same scripting engine, such as Netscape 7.0) can now make SOAP calls directly from the client. The data returned from a SOAP operation can be accessed via the same DOM methods used to traverse any XML document.

Mozilla's SOAP API is a JavaScript interface for a series of objects designed to create, send, and receive SOAP messages. These messages are encoded as XML, but the developer does not need to know much about the XML part of SOAP to use the Mozilla API. SOAP is used as a wire protocol. Construction of a SOAP message is as easy as creating any other JavaScript object.

In this context although, the security becomes an issue. A secure browser will not allow, normally, SOAP calls to a service in a foreign domain. So, special requests to the user have to be made by the script in order to be granted with proper rights.
A SOAP message can be created just as an array or image object with scripting would be.

The four important objects that need to be created are the following:

- `SOAP Call`: the heart of the SOAP operation, providing the means to encode and send your message.

- `SOAPParameter`: a single parameter to be passed as an argument to the service method. Multiple parameters are stored in an array to be given to the `SOAPCall` object.

- `SOAPResponse`: the response from the service. It contains the results of the method invoked by the `SOAPCall`.

- `SOAPFault`: an object that represents an error or warning from the service.

The `SOAPCall` and `SOAPParameter` objects are the only ones needed to create with JavaScript. `SOAPResponse` and `SOAPFault` are automatically generated when the service sends back a response.

A basic SOAP message contains some key information: the URI of the target service, the name of the service method you want to invoke, and a series of parameters to be passed as method arguments. The `transportURI` property of the `SOAPCall` object is a string indicating the endpoint URI of the service.

Generally, the asynchronous invocations of methods should be used in order not to block the applications. But that enforces the use of a particular style of programming: event-driven programming.A name of a callback function must be provided. That function will be called when a response to the service invocation becomes available.

Example:

For the example presented earlier the code needed for making a SOAP call from inside the browser and after that parsing the response would be the following:

```
function callMethod(method,parameters,callbackFunction)
{
  try {
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowser
Read");
  } catch (e) {
      alert(e);
    return false;
  }

  var soapCall = new SOAPCall();
  soapCall.transportURI = "http://staros.futurs.inria.fr:8080";
```

```
  soapCall.encode(0, method, "", 0, null, parameters.length,
parameters);
  soapCall.asyncInvoke(
    function (response, soapcall, error)
     {
        var r = handleSOAPResponse(response,soapcall,error);
        callbackFunction(r);
     }
  );
}

function handleSOAPResponse (response,call,error)
{
    if (error != 0)
    {
        alert("Service failure");
        return false;
    } else
    {
        var fault = response.fault;
        if (fault != null) {
            var code = fault.faultCode;
            var msg = fault.faultString;
            alert("SOAP Fault:\n\n" +
                "Code: "  + code +
                "\n\nMessage: " + msg

            return false;
        } else
        {
            return response;
        }
    }
}

// the main function
function main() {
      // create the parameters array
      var params = new Array();

      params[0] = new SOAPParameter(“it”, “locale”);
      params[1] = new SOAPParameter(“Europe/Rome”,"timezone");

      // now call the getTime service
      callMethod("getTime",params,parseResult);
}
```

This works fine when the arguments you need to provide are of simple type: text / integer etc, types that are easy to map to JavaScript types or when they are JavaScript arrays or objects. And that because the Mozilla SOAP API was designed with one purpose in mind: allowing the developer to made SOAP calls once he has the parameters as JavaScript objects. When the arguments are more general: XML arbitrary content, this method will not work. At least this is the conclusion I reached after trying to do it for some time.

2.8.2.2 XMLHttpRequest Object

The other method is actually constructing a SOAP message with DOM with the parameters in XML format that I want to send, serializing it as text message, encapsulating it

in a HTTP message and sending it on the wire. This method of sending complex XML parameters actually worked.

Internet Explorer on Windows and Mozilla provide a method for client side javascript to make HTTP requests. This allows you to make HEAD requests to see when a resource was last modified, or to see if it even exists.

Whilst the object is called the **XML HTTP Request object** it is not limited to being used with XML, it can request or send any type of document.

The creation of the XML HttpRequest object is very simple:

```
try {
   xmlhttp = new XMLHttpRequest();
   xmlhttp.overrideMimeType("text/xml");
}
catch(e) {
  alert("XMLHttpRequest Message not built");
}
```

After that , the endpoint URI has to be specified as well as the way of sending the HTTP message (GET/ POST):

Example:

```
xmlhttp.open("POST",http://coulos:8180/axml/services/WebServicesXMLTest);
```

When a response will be ready (*readyState = 4),* it will be processed by a callback function (asynchronously).

```
xmlhttp.onreadystatechange = function() {
 if (xmlhttp.readyState == 4) {
  callbackFunction(xmlhttp.responseText);
}
```

Some non-standard headers could be appended to the HTTP message:

```
xmlhttp.setRequestHeader("SOAPAction"," ");
xmlhttp.setRequestHeader("Content-type","text/xml");
```

and the SOAP message (serialized in the JavaScript string) is sent:

```
xmlhttp.send(SoapMessage);
```

2.8.3 The XUL Language

The XML User Interface Language (**XUL**) is a markup language for creating rich dynamic user interfaces. It is a part of the Mozilla browser and related applications and is available as part of Gecko. It is designed to be portable and is available on all versions of Windows, Macintosh as well as Linux and other Unix-based operating systems. With XUL and other Gecko components, there is possible to create sophisticated applications without special tools.

Like HTML, in XUL an interface can be created using a markup language, use CSS style sheets to define appearance and use JavaScript for behaviour. The developer also have access to programming interfaces for reading and writing to remote content over the network,

calling web services, and reading local files. Unlike HTML however, XUL provides a rich set of user interface widgets for creating menus, toolbars, tabbed panels, and hierarchical trees to give a few examples.

XUL is an XML language and you can use numerous existing standards including XSLT, XPath and DOM functions to manipulate a user interface, all supported directly by Gecko. In fact, XUL is powerful enough that the entire user interface in the Mozilla application is implemented in XUL.XUL applications may be either opened directly from a remote Web site, or may be downloaded by the user and installed. Mozilla's XPInstall technology allows an application to be placed on a remote site and installed easily. The benefit of installing an application is lowered security restrictions so that applications may read and write files, user preferences and system information.

XUL may also be used to create standalone applications that embed the Gecko engine or may be used as part of the browser. Gecko also supports various Web Services technologies such as SOAP and WSDL.

Example:

```
<?xml version="1.0"?>
<window
xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul
">
<box align="center">
  <button label="Push Me" onclick="alert (‘Hi’);" />
</box>
</window>
```

We notice the special namespace:

(http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul)

that is used to specify to the browser that this is a XUL application. This simple application represents a window with a button in it which, when pressed, opens a JavaScript alert window with some text in it. The XUL documents have to be saved with the “.xul” extension.

2.8.4 XPCOM/ XPCONNECT

JavaScript in XUL can be used for other things besides the manipulation of various interface elements and scripting the DOM. It can be used for example for accessing the services provided by Mozilla. Browsing the Web, reading email, and parsing XML files are examples of application-level services in Mozilla. They are part of Mozilla’s low-level functionality which is usually written in platform-native code (C++). This functionality is organized in modules which are known as *XPCOM components*.

XPCONNECT is the bridge between the JavaScript and **XPCOM.** The XPCONNECT wraps the natively compiled components with JavaScript objects. Using JavaScript, instances of these components can be created and used the same way a regular JavaScript object is created and used. The relationship is shown in the figure 2.7.

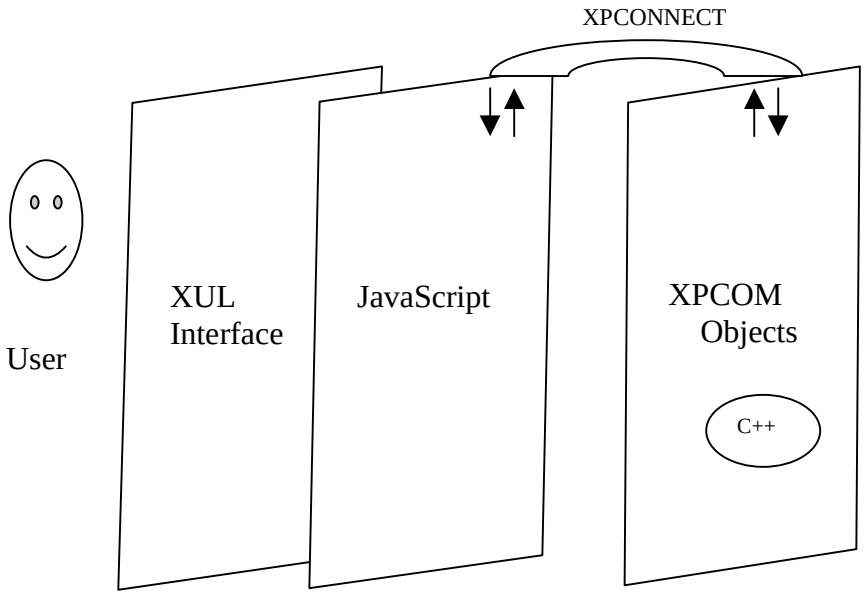

Figure 2.7 The Mozilla Framework

The example below shows the way a file is removed from a directory whose path is provided (*localpath).*

Example (from my application):

```
function removeFile(fileName)
{
 var directory =
      Components.classes["@mozilla.org/file/local;1"].
      createInstance(Components.interfaces.nsILocalFile);

      directory.initWithPath(localPath);
      var file;
      if (directory.isDirectory())
      {
          var found = false;
          var listFiles = directory.directoryEntries;
            while (listFiles.hasMoreElements())
            {
                              file = listFiles.getNext();
        file.QueryInterface(Components.interfaces.nsILocalFile);
                        if (file.leafName == fileName)
                        {
                                      found = true;
                                      break;
                         }
            }
            if (found == true) {
                     file.remove(true);
                     alert("The delete phase was successful!");
            }
      }
}
```

2.8.5 XPInstall

XPInstall is a technology used for cross-platform installation. Mozilla uses it to install new packages, but as a general-purpose technology, it can be used to install anything on your computer. The XPI file format is the basic unit of exchange in an XPInstall and the installation script manages the installation.
Installation scripts -- whether they appear within the XPI, on a web page as an external "trigger script," or elsewhere in JavaScript -- use the XPInstall API to do all the heavy lifting. The XPInstall API includes installation functions organized into such high-level objects as the Install object, the InstallTrigger object, and the File object, which can be used to install new Mozilla packages.

The Common XPInstall functions

```
initInstall( )      // initializes every installation
getFolder( )        // creates a new folder on the target system
addFile( )          // adds files and directories to the install
performInstall( )   // executes the installation
cancelInstall( )    // cancels installation if there are errors
```

In addition to the XPInstall API installation functions, installation scripts often call methods on the chrome registry itself.

Chapter 3. The Active XML Project

3.1 Concept presentation

Active XML (http://purl.org/net/axml) is a declarative framework for distributed data management based on XML and web services. An **Active XML** document is an XML document that may contain calls to web services. These calls to web services can produce more data as result, data that can be used to update the document. So, these documents are not static, they evolve in time, since only a part of data they contain remains the same and, besides it, they contain *intentional* information.

The system which for stores and manages this kind of documents is called **Active XML Peer**, since, as we will see, it is working in a decentralised Peer-To-Peer environment. This is the system responsible for the automatic activation of the service calls and for updating the **AXML Documents**.

An **Active XML peer** can have three roles:

- Repository for **AXML Documents**. It has a Query Engine which can process queries over the documents in repository.

- Client. A number of service calls in the documents in the repository needs to be activated and the **Active XML Peer** needs to call the correspondent Web Services.

- Server. A number of services can be provided by the Peer under the form of queries over the documents in its repository.

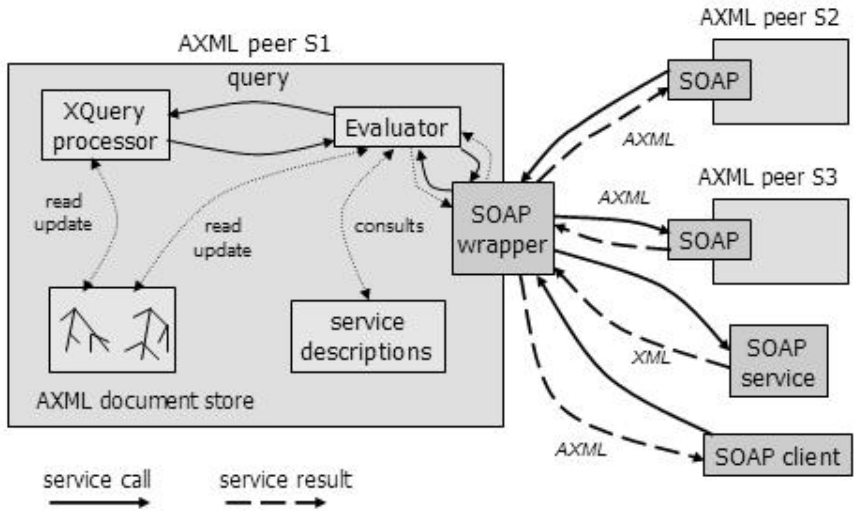

Figure 3.1 (from the official site)

The internal architecture of an AXML peer (shown in the figure above) relies on the following modules:

- The **repository**, which provides persistent storage for AXML documents,

- The **evaluator**, whose role is to trigger the services calls embedded inside AXML documents and to update them accordingly.

- The **XQuery processor** deals with the service requests, by evaluating the corresponding queries.

Peers communicate with each other *only by the mean of web service invocations*, through their **SOAP wrapper** modules. They can exchange XML data with any web service client/provider and AXML data with AXML peers.

3.2 Active XML documents

The **Active XML documents** are valid XML documents. They contain static XML data and might contain intentional data which is represented under the form of service calls. Let's consider as example a part of a document I tested my application on:

```
<cities axml: docName="CITY" xmlns:axml=
"http://www-rocq.inria.fr/verso/AXML">
   <city locale="it" name="Rome" timezone="Europe/Rome">
     <time>
       <axml:sc doNesting="true" frequency="every 3600000"
        id="076A1B60-C353-D4A4-0056-5E9DC88450B9"
        lastCalled="1087387299111" methodName="getTime"
        mode="replace" serviceNameSpace="Time"
        serviceURL="http://staros.futurs.inria.fr:8080/axis
        /servlet/AxisServlet">
            <axml:params>
               <axml:param name="locale">
                   <axml:xpath>
                        ../../@locale
                    </axml:xpath>
               </axml:param>
               <axml:param name="timezone">
                    <axml:xpath>
                        ../../@timezone
                    </axml:xpath>
               </axml:param>
             </axml:params>
       </axml:sc>
     </time>
   </city>
    ….
</cities>
```

The first important element is the namespace declaration. "***http://www-rocq.inria.fr/verso/AXML"*** is required as a namespace for all the Active XML Documents and is usually bound to the ***axml*** prefix.

The central element is the Service Call (***sc*)** XML element. It is defined in the special namespace mentioned before and has a set of attributes and children XML elements defining:

- The Web Service to call
- Its parameters
- How and when to call it
- What to do with the results

The most important attributes of this element which define the web service to call are: **serviceURL, serviceNameSpace** and **methodName.**

The ***serviceURL*** attribute specifies the end-point of the WebService to call, in our case: "*http://staros.futurs.inria.fr:8080/axis/servlet/AxisServlet*".

The ***serviceNameSpace*** attribute specifies the namespace to use for the body element of the SOAP message (*soap:body*), more simply the method namespace URI. In our case this is *"Time"*.

The ***methodName*** attribute defines the name of the operation to invoke on the Web Services, in our example *"getTime"*. For the Active XML Web Service (a declarative Web Service provided by the Active XML Peer), it is not important: *"invoke"* can be used or anything else.

Other less important attributes are: signature and *useWSDLDefinition* attribute. The first one set the URL of the WSDL file defining the Web Service and is optional since it's used only when type validation on that particular service call is wanted. The other one specifies if the file indicated by the first should be used, is optional and its default value is false.
A Service Call element has also attributes that provide information on how and when to activate the service call.

The ***frequency*** attribute is the most important in this category. It can have several attributes:

- Once
- Lazy
- On Date
- Every X

If no frequency is defined, the Service Call will not be activated by the **Active XML Peer**.
Once means that the service call will be activated only once at the start-up of the peer. Every time the peer is restarted, the service call is activated.
Lazy means that a service call will be activated only when its result is useful to the evaluation of a query or when the instantiation of a Service Call parameter, defined through a XPath expression is necessary.
On Date indicates exactly when the service call will be activated. If the format is incorrect or the date is in the past, the service call will not be activated. Example: frequency = "on 02/04/05 14:22"
Every X means the service call will be activated periodically and the interval is given in milliseconds.

Other important attributes are:

- ***id*** – identifies uniquely in time and space the Service Call. It is automatically generated by the Active XML System.
- ***Name*** – has no particular meaning and is optional
- ***Callable*** – its value is *true* by default and if it's set to *false* causes the Service Call not to be activated, not even the frequency is correct.
- ***lastCalled*** – used to keep track of the last activation of the Service Call
- ***followedBy*** – allows chaining the evaluation of the Service Calls inside the same Active XML Document in a simple manner. Its value has to be a valid XPath expression returns a single ***sc*** element or the id of the Service Call.

3.2.1 Service Call Parameters

The parameters of the service call are specified by a child element of the ***sc*** element which has the name ***params*** and is bounded to the Active XML namespace. It has to be present even if the service call has no parameters. Every parameter has a correspondent element ***param*** in the representation (bounded to the same namespace). An attribute **name** is required for this element. The parameters can be expressed directly as a value or through an XPath expression. If the service call has several parameters their order must match exactly the one from the WSDL that describes the Web Service.

The above statements can be resumed through the following schema representations:

```
<element name = “params”>
    <complexType>
      <sequence>
         <element ref = “axml:param” minOccurs=”0”
maxOccurs=”unbounded”/>
       </sequence>
    </complexType>
</element>

<element name = “param”>
   <complexType>
       <choice>
           <element ref=”axml:xpath”/>
           <element ref=”axml:value”/>
       </choice>
       <attribute name = “name” type = “xsd:NCName” use =
“required”/>
</complexType>
</element>
```

3.2.1.1 The *Value* Parameter

This value can be any well-formed XML fragment. It conforms to the following representation:

```
<element name = “value”>
  <complexType mixed = “true”>
  <sequence>
   <any namespace=”##any” minOccurs=”0” maxOccurs=”unbounded”/>
</sequence>
</complexType>
</element>
```

3.2.1.2 The *XPath* Parameter

The second way of writing an Active XML Parameter is through an XPath expression and conforming to the following schema component representation:

```
<element name=”xpath”>
  <simpleType>
    <restriction base=”xsd:string”>
      <whiteSpace value = “collapse”/>
      <minLength value = “1”/>
    </restriction>
  </simpleType>
</element>
```

Every time a Service Call needs to be activated, the XPath parameter is evaluated. The evaluation of the XPath is made starting from the corresponding ***sc*** element.
So, in our example, the Service Call has two XPath parameters: ../../@locale and ../../timezone. Those parameters evaluate to *it* and *Europe/Rome* respectively.

3.2.2 Service Call result handling

The **AXML Document** author might provide the system with the information regarding what to do with the results of an Active XML Service Call. The two attributes influencing this behavior are: ***mode*** and ***doNesting***

The ***mode*** attribute has possible values: *merge* and *replace.*

The *merge* value means that the results will be added as the immediate right sibling of the **sc** element, considering the AXML Document as an ordered labeled tree. The previous results will be kept in the AXML Document like in the figure 3.2

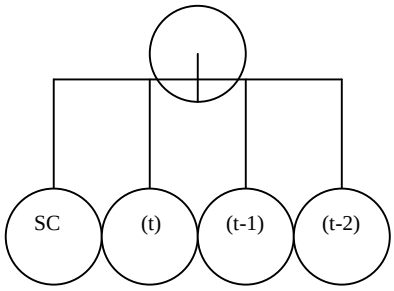

Figure 3.2 The service call and its responses in the tree hierarchie.

Replace means that the previous results will be replaced by the current invocation of the Active XML Service Call. To keep track of the results, the Active XML system will add a special attribute to the top-level elements of the results named ***origin*** and bound to the Active XML namespace having as value the ***id*** attribute of the Active XML Service Call.

The ***doNesting*** attribute is used to keep track of the TEXT nodes which resulted from a service call. That special kind of nodes has to be handled differently: if keeping track is desired, they have to be wrapped in a special element named **text** and bound to the Active XML.

Example: Since the ***doNesting*** attribute is true, after calling the Service Call from the presented document, the updated document looks like this:

```
<cities axml:docName="CITY" xmlns:axml="http://www-
rocq.inria.fr/verso/AXML">
  <city locale="it" name="Rome" timezone="Europe/Rome">
    <time>
      <axml:sc doNesting="true" frequency="every 3600000"
id="076A1B60-C353-D4A4-0056-5E9DC88450B9" lastCalled="1087387299111"
methodName="getTime" mode="replace" serviceNameSpace="Time"
serviceURL="http://staros.futurs.inria.fr:8080/axis/servlet/AxisServle
t">
      <axml:params>
          <axml:param name="locale">
              <axml:xpath>
                   ../../@locale
              </axml:xpath>
```

```
            </axml:param>
            <axml:param name="timezone">
                <axml:xpath>
                ../../@timezone
              </axml:xpath>
              </axml:param>
         </axml:params>
        </axml:sc>
        <axml:text
           axml:origin="076A1B60-C353-D4A4-0056-5E9DC88450B9">
            mercoledì, 23 giugno 2004 16:23:38 Ora estiva Europa
            centrale
        </axml:text>
         </time>
        </city>
</cities>
```

3.3 Active XML Services

The **Active XML** peers can offer services as queries over the documents in their repository. The example below illustrates the definition of such a service in the X-OQL syntax:

```
<serviceDefinition type="query">
             <definition>
                   <query>
                        <![CDATA[
<results>
      select
            <result>
                  t,
                  a
            </>
from b in BIB/bib/book, t in       b/title, a in b/author
</>;
                             ]]>
                  </query>
            </definition>
</serviceDefinition>
```

The result returned by calls to this kind of services can contain themselves other service calls. The question is how should the result be returned to the caller. Should the peer return a response with service calls in it (an AXML document) or should it try to call the service calls itself and return only simple XML content. The response to that question is given by using schemas in a way I am not going to detail here.

Chapter 4

A light-weight Active XML client

Goal: **Active XML** is a very flexible language and framework, but it needs a user interface to be easily used/managed by application developers and end-users. The goal was to develop such a client. Its main characteristic should be the few pre-requisites in terms of infrastructure: Mozilla is the only thing needed to be installed on the user's machine.

Expected functionality

The first part of our work was to precise the functionalities of such a light-client. We consider three functionalities of major importance:

- the user should be able to exploit any received **Active XML document** (in browser, through the e-mail). This means that the developed tool should permit the interaction with the document, and especially with the service calls that are inside it. The user should also be able to view/edit the various attributes of the service call (activation mode and passed parameters) and to activate (from the local computer) the service calls. The returned values should enrich/update the document by replacing the previously obtained results, or being appended to them
- the user should be able to remotely connect to an existing **Active XML peer** and to use it as a proxy to perform some operations that are locally unavailable like: remotely activating a service call on a document that resides on a peer forcing this way an update of that document. It should also be possible to query a peer remotely.
- the user should be able to perform management operations
  - on peers:
    - Ø creating/deleting/modifying **Active XML documents** at the remote peer
    - Ø creating/deleting/modifying **Active XML services** at the remote peer
    - Ø interacting *remotely* with the service calls of these documents (as described above)
  - locally:
    organize/manage the **Active XML documents** on the computer the person uses

4.1 The Active XML browser

An important part of our work was the design of the **Active XML browser**. The first version is, in fact a **Mozilla/JavaScript/XUL application** which has an graphical interface which resembles to the one in the figure 4.1

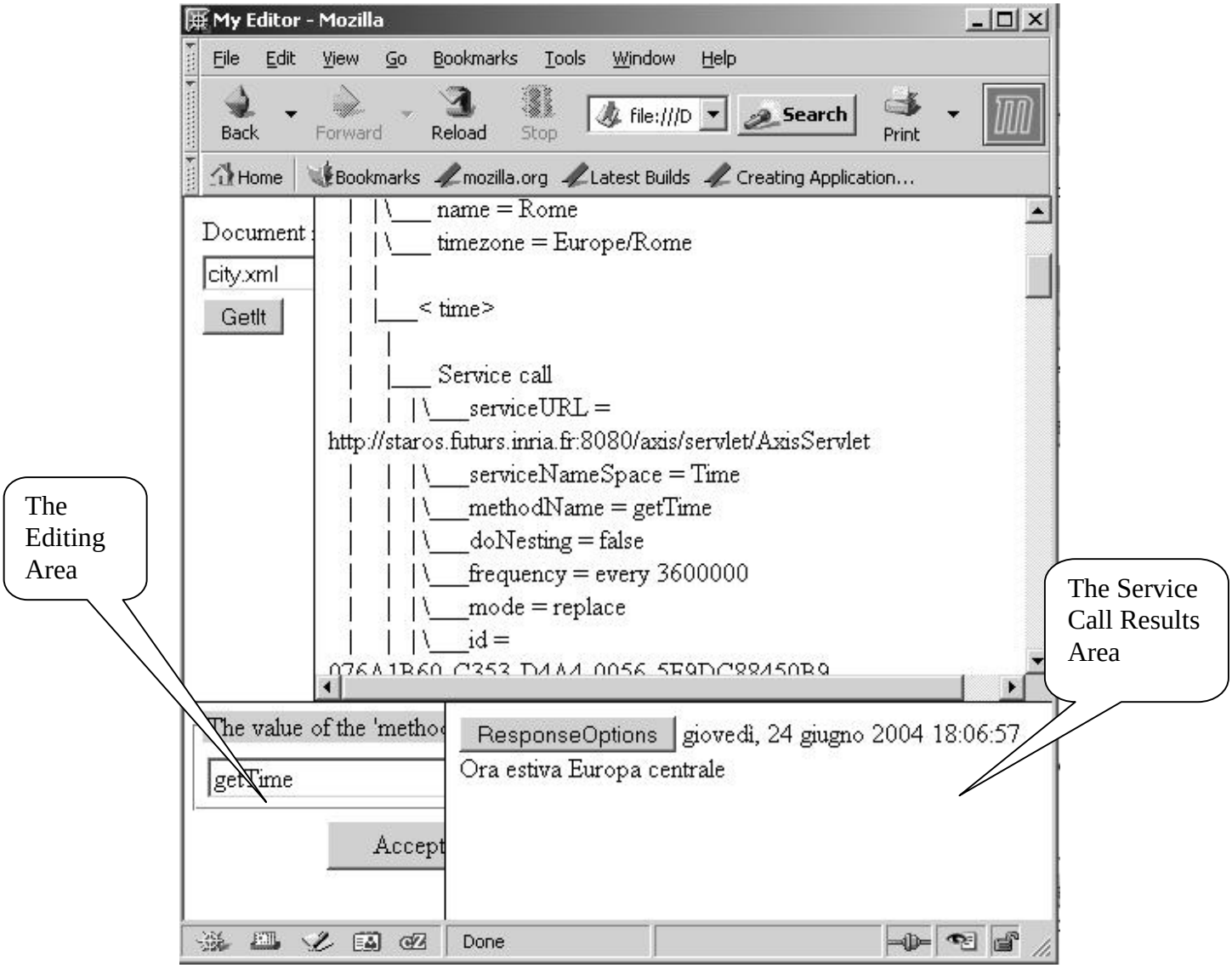

Figure 4.1 A snapshot with the *Active XML Browser*

4.1.1 Interface and functionalities description

As can be seen in the picture above, the graphical interface appears as a web page divided in four areas: the *Load/Save Area*, the *Document Area*, the *Editing Area* and the *Service Call Results Area.*

The *Load/Save Area* permits the I/O interaction with the system. Actually, this part is very little developed in the first version.
The *Document Area* is the zone in which the **Active XML document** is loaded.
The *Editing Area* is where the modifications of the attributes are made.
The *Service Call Results Area* is where the responses from the service calls are displayed.

Possible scenario of utilization :

1. Load a document by filling the area of text in the *Load/Save Area* with the name of the AXML document and by pressing the button “Get It”.

2. The corresponding document is “loaded” in the *Document Area* using a XSL style sheet. Actually in that space it is loaded the “image” of that document which is a HTML document obtained by the XSL transformation. This HTML document contains JavaScript code which actually enables the user to interact with the interface.

3. The user has some choices:

a) modify some attributes of the Service Call Element

This can be done by clicking on the corresponding attribute name. That moment a XUL document is loaded in the *Editing Area*. In some cases the an editing process can be started (in our example for the *serviceURL*, *serviceNameSpace*, *methodName* attributes). In the case the attribute allows only for discrete values, the alternative is displayed in the form of radio buttons (like for *doNesting* and *mode* attributes) and in the last case where the alternative is more complexe the choice is offered by a combination of radio-buttons and areas of text ensures (like for the *frequency* attribute).

b) activate a service call

This can be done very easily by clicking on the image of the corresponding *Service Call* element tag. The result appears in the *Service Call Results Area* and can be further used to enrich the document. This can be done by clicking the *Response Options* button.

The result will be manipulated accordingly to the values of **mode** and **doNesting** attributes. If the **mode** is *replace* all the previously results of the service call (which are *siblings* of the Service Call element in the tree representation of the XML document) are replaced by the last result. If the **mode** is *replace*, the result will be appended as the first sibling of the corresponding Service Call node. So the conventions presented for the Active XML peer are respected. For the text nodes, the convention regarding the **doNesting** respected as well.

c) modify a parameter

There are two types of parameters: *Value* and *XPath* they are treated mainly the same way.

The selection is made by clicking on the images of the corresponding *XPath* and *Value* element-tags. A XUL window is opened in the *Editing Area,* containing a text area and two radio buttons for selection. If the text typed is selected as a normal *Value,* no validation is made and the parameter gets the text typed as value. If not, two validations are made:

- the first one to make sure that the text represents a correct XPath expression
- the second one to make sure that the expression actually evaluates to a pertinent value for the parameter.

I chose to halt the modification operation if at least one of the above validations fail. Another possible choice is to halt it only if the syntactic (the first) validation fails, in order to permit future development. In my opinion, adopting the second would be a wrong choice.

4.1.2 Technical Issues

This is a *Mozilla/JavaScript* application. The front-end is basically *HTML / XUL* and the functionality is ensured by JavaScript code. For XML parsing, the *Mozilla DOM 3 Implementation*, available under JavaScript wrapping functions, was used. Some XPConnect/ XPCOM elements were used in order to ensure the I/O file operations.

The architecture of modules is depicted in the figure 4.2 .

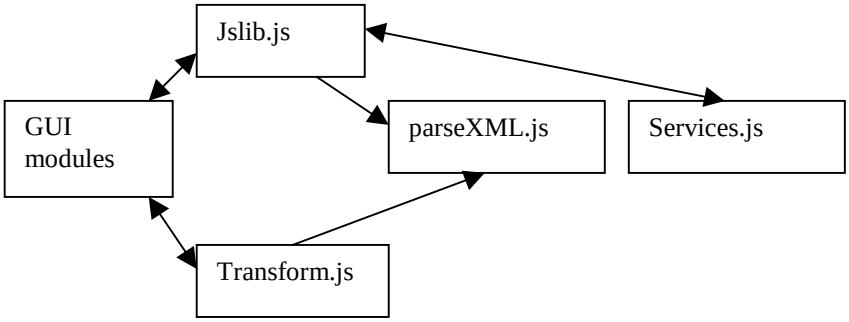

Figure 4.2 The modules architecture in the *AXML Browser* application.

The parsing of the XML document is made mainly in the module *parseXML.js* using the Mozilla implementation of DOM.

The *Jslib.js* module is the one having the central role: it's the main *interface* between the GUI modules and the JavaScript modules and in the same time it is a sort of interface module between the GUI modules themselves because it ensures the dynamic/contextual loading of XUL in the application's front-end window. All the *event-handling functions* are here.

The *Tranform.js* module provides the I/O file operations and contains the XML transformation code which uses a XSL Stylesheet in order to obtain a valid HTML fragment for presentation. Key JavaScript code used for the transformation:

```
var xsltProcessor = new XSLTProcessor();
var xsltProcessor.importStylesheet(xslStylesheet);
var fragment = xsltProcessor.transformToFragment(xmlDoc, document);
```

In this code, *xslStylesheet* is a JavaScript (DOM) object representing the style sheet XSL and the *xmlDoc* represents a XML document.The two objects have been built both using a similar code:

```
xslStylesheet=
document.implementation.createDocument("","",null);
xslStylesheet.addEventListener("load", documentLoaded,
false);
xslStylesheet.load("document.xsl");
```

One important worth mentioning thing is that the "loading" of XML documents is asynchronous which imposes an event-driven style programming (using callback/event-listener functions like the *documentLoaded* function in the previous code).

The *Services.js* module is the one responsible of the interactions with the Web Services: the SOAP messages creation functions, the functions for sending the SOAP message over HTTP and the callback functions handling the responses to the service calls. The creation of the SOAP message is done using the Mozilla DOM implementation and serializing the resulting DOM object in text. The message core is built by the *buildXMLFromParameters* which take as arguments the parameters (names and values as JavaScript arrays) and the wrapping SOAP "envelope" is added by the *buildSOAPMessage* function.

As an example of DOM utilization, I will present below the content of *buildSOAPMessage* function:

```
function buildSOAPMessage(namespace, xmlContent, methodName) {

  var docSOAP =
document.implementation.createDocument('','root',null);
      var root = docSOAP.documentElement;
      var namespaceSOAP =
"http://schemas.xmlsoap.org/soap/envelope/";

      var nodeEnv = docSOAP.createElementNS(namespaceSOAP,
"soapenv:Envelope");
      root.appendChild(nodeEnv);
      var encodingAttr = docSOAP.createAttributeNS(namespaceSOAP,
"soapenv:encodingStyle");
      encodingAttr.value
="http://schemas.xmlsoap.org/soap/encoding/";
      nodeEnv.setAttributeNode(encodingAttr);

      var nodeHeader =
docSOAP.createElementNS(namespaceSOAP,"soapenv:Header");
      nodeEnv.appendChild(nodeHeader);

      var nodeBody =
docSOAP.createElementNS(namespaceSOAP,"soapenv:Body");
      nodeEnv.appendChild(nodeBody);

      var nodeMethod =
docSOAP.createElementNS(namespace,"a0:"+methodName);
      nodeBody.appendChild(nodeMethod);
      var params = xmlContent.documentElement.childNodes;

       var i;
       for (i=0; i<params.length; i++) {
            nodeMethod.appendChild(params.item(i).cloneNode(true));
        }

      var s = new XMLSerializer();
      var str =
s.serializeToString(docSOAP.documentElement.firstChild);
      return str;
 }
```

The *xmlContent* argument represents the DOM object resulted after the encoding of the Service Call parameters. The *namespace* is the Web Service namespace required in order to access it and the *methodName* attribute – the name of the Web Service method called.

The XUL modules used are: *choiceDialog*, *choiceSpecialDialog*, *dialogInput* & *XPathValue*. While the first three provide GUI for human - service call attributes interaction, the last one permits the service call arguments management.

4.1.3 Conclusion

As I mentioned previously, this was just a "side-tool". It is now included in the final version of the **Active XML Commander** as a central feature. I preferred to presented it separately for a better understanding.

4.2 The Active XML Commander

The second main part of our work was the design and implementation of the AXML Commander.

4.2.1 The Global Architecture

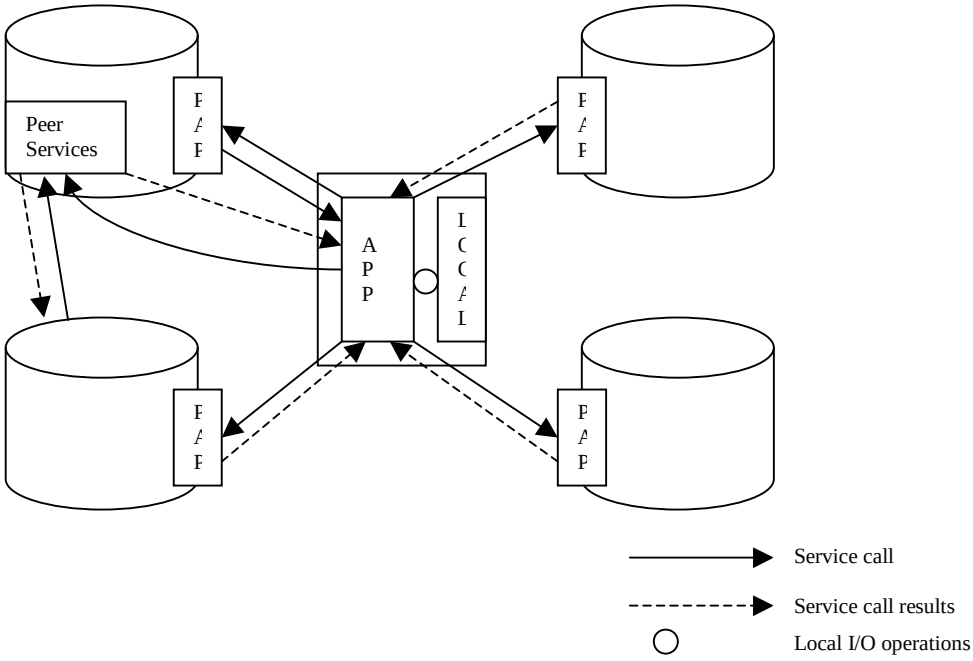

Figure 4.3 The P2P context in which AXML Commander works.

Figure 4.3 above shows the architecture. The **Active XML Commander** is a light-weight XML client that permits the management of a local AXML documents repository.

It can also connect remotely to the **Active XML Peers** at a special web service called **Peer Access Point (PAP)** provided by every peer and run operations on those peers. To install and run this powerful instrument on a local computer, only the Mozilla framework is a pre-requisite.

Of course, the **Active XML Commander** inherits all the functionality of the **Active XML Browser** since the latter is a part of it. As a consequence, a user would be able to access a Web Service (on an Active XML Peer or not) directly from the **Commander**.

In addition, he would also be able to force a service call activation in a document on an Active XML Peer and to make a query on a Peer. So, he would be able to use Peers as proxies perform operations that are not locally available. We could imagine that the light-weight client might not have the proper credentials for accessing a web service and that the user might try to do it using a Peer as a proxy.

4.2.2 The Peer Access Point module presentation

This module is actually a Java language class that is used by the **Axis Engine** as the logic module corresponding to *Peer Access Point* Web Service and was especially designed to ensure the interaction between the Peers and the **Active XML Commander.**

4.2.2.1 The Axis Context

Axis (one of the latest products of the Apache SOAP project) is a "SOAP Wrapper". It can be seen as a thin layer between the logic of the Web Service and the transport layer carrying the data. As depicted in the figure below, Axis is simply the means by which the SOAP message is taken from a transport (such as HTTP) and handed to the Web Service and the means by which any message is formatted as a SOAP message to then be sent back to the requestor.

The components of the Axis architecture are the following:

- Axis Engine – the main entry in the SOAP processor
- Handlers – the basic building blocks inside Axis that link Axis to existing back-end systems (modules that perform specific functions)
- Chain – an ordered collection of handlers
- Transports – mechanism by which the SOAP messages flow into and out of Axis
- Deployment/Configuration – means by which the Web Services are available through Axis
- Serializers /Deserializers – code that will convert native (for ex. Java) data types into XML and back.

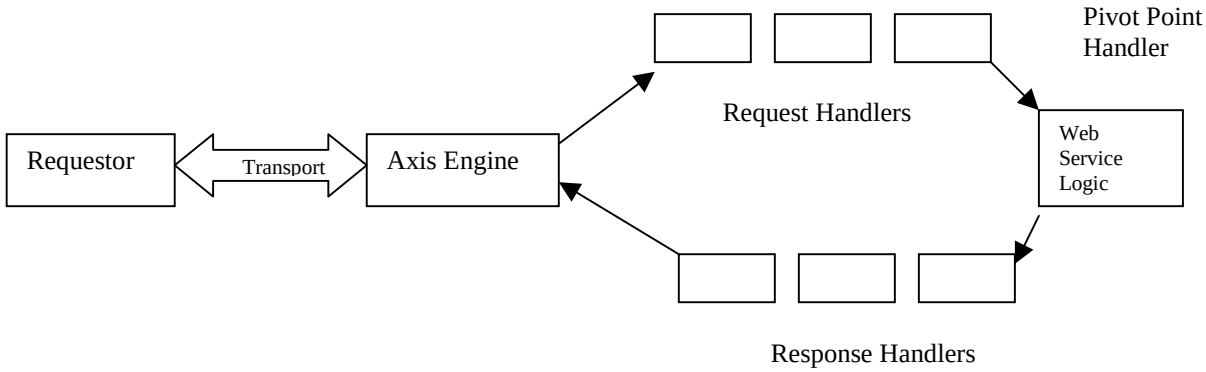

Figure 4.4 The AXIS architecture

One special handler is Pivot Point Handler. This handler is the point where the target of the processing is switched: it's the response message that is processed now and no longer the request message. That is the place where the Web Service Logic resides.

4.2.2.2 The Module

This module is the **Pivot Point Handler** in the corresponding Web Service Handlers Chain, thus having a central role. It's materialization is a Java class with several public methods serving as back-end for the "methods" offered by the Web Service.
The classes imported belong to the following packages :

```
org.w3c.dom
org.xml.sax  -   used for parsing
org.apache.axis – for the Axis Engine
fr.inria.gemo.axml – the classes from Active XML project
```

This module is actually built on top of the classes from the Active XML project, using specific methods to access the documents and the services from Peer's repository, to access the Query Processor and the Service Call Executor.

```
public org.w3c.dom.Element[] getDocument(String documentName) throws Exception
```

-> takes as argument String representing a document name and returns an AXML Document as result (in fact an array containing only the root of this AXML Document) or generates an Exception if there is no document having that name

```
public org.w3c.dom.Element[] getService(String serviceName) throws Exception
```

-> takes as argument String representing a service name and returns an AXML Service Definition as result (in fact an array containing only the root of this AXML Service Definition) or generates an Exception if there is no service on the Peer having that name

```
public String[] getAvailableDocuments ()
```

-> takes no arguments and returns an array of String representing a list of document names

```
public String[] getAvailableServices ()
```

-> takes no arguments and returns an array of String representing the list of names of the Peer services

```
public void activateServiceCall(String documentName, String serviceId) throws Exception
```

-> takes as arguments two String object: one representing a name of document, and the other presenting a Service Call Id. The side effect of calling this method would be forcing the activation of a certain service call in a document on the Peer. This method can raise an Exception if the document with that name doesn't exist.

```
public org.w3c.dom.Element query(String q) throws Exception
```

-> takes as argument a String object representing a XOQL query and returns a DOM Element object representing the response to the query

```
public void saveService(String serviceName, Element serviceElement) throws Exception
```

-> takes 2 arguments:

- a String object representing a service name
- a DOM Element representing the new definition of the service

public boolean saveDocument(String documentName, Boolean update, Element content) throws Exception
-> takes 3 arguments:
- a String object representing a document name
- a Boolean object representing an option (true -> if the document file can be overwritten, false -> if not // the document can be only created if it does not exist)
- a DOM Element representing the new content of the document

The response is a boolean (true -> if the save was successful, false -> if it wasn't).
The effect is saving the document on the peer.

public void removeDocument(String URL) throws Exception
-> takes as argument a String representing a document name and its effect is erasing the document from the peer. If such a document does not exist, an Exception is raised.

4.2.3 The GUI description

A snapshot of the **Active XML Commander** *GUI* in presented in Figure 4.5.
The **AXML Commander** is actually an independent *Mozilla XUL application*. The Graphical Interface appears as a standalone window application composed of several areas:

- The bar of menus – is actually not very "heavy" in the sense that it was preferred, for user-ergonomics reasons, that as many choices as possible be made in contextual menus. It contains two menus: *AXML Peer* and *Local Space*. The first one can be thought of as a Bookmark section in a Browser. The menu- items in the popup menu are names of known (registered) peers. A new peer can be registered by providing its URI (required) and a name (optional). The second one contains 2 items: the first one offers the possibility of creating a blank Active XML document while the second one permits changing the path of "the local repository" directory.

- *AXML Navigator Area*

In this area names/URI of Peers and the special "Local Space" label are presented. The Peers are seen as special "repositories" holding documents and services. The **Local Space** represents actually a directory on the local computer, where AXML Documents are stored. This way the "*Active XML Peer-to-Peer Space*" is seen in a uniform way. Peers can be added by clicking a menu-item representing a registered peer or directly by entering the name and the corresponding URI.

- *AXML Content*

Situated just below the *Navigator Area*, this area can present alternatively (using tabs) lists of AXML Documents and Services names.

- *Display Area*

In this area, AXML Document and Services Definitions can be displayed / modified. For the documents there are two ways of displaying: using a simple text box (the document can then be edited at will) or using an updated version of the **Active XML Browser** previously presented. For the service definitions only the second way is possible. Actually, in this area many documents/services definitions can be displayed in the same time

and this is possible thanks to the idea of using panels. The displayed document can be changed using the tabs.

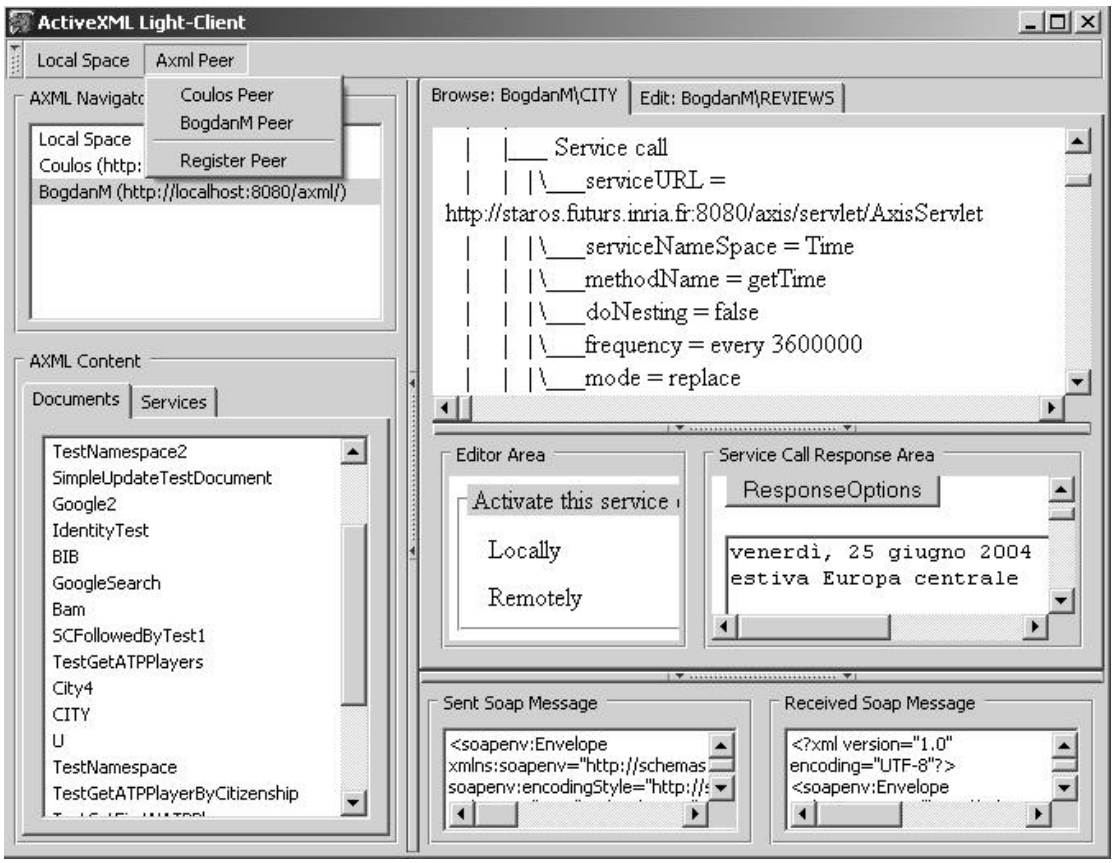

Figure 4.5 A snapshot with the *Active XML Commander*

- *SOAP / Debug Area*

This is the area where the SOAP messages exchanged by the ActiveXML Commander with the Peers are presented. This area is divided in two sections: one for the ongoing messages and the other for the just received ones. This instrument has the same functionality as the **Axis TCP Monitor Tool** but the approach is a bit different, our application working on the AXML Client side.

4.2.4 A new Active XML Browser version

The version of the **Active XML Browser** used by the **Active XML Commander** was updated. All the I/O file operations area was removed since its place was taken by the global I/O modules of the Commander.

An add-on is the possibility of forcing a service call in an AXML Document remotely on a peer. As it can be seen in the picture above, when the activation of a service call in a document is requested, an alternative is offered: activate it locally (from the imported document copy) or activate it remotely on the peer in the original document.

4.2.5 Technical Issues

This application is a *Mozilla/JavaScript* application. The front-end is basically *XUL and* the functionality is ensured by JavaScript code.

For XML parsing, the *Mozilla DOM 3 Implementation*, available under JavaScript wrapping functions, was used. XPConnect/ XPCOM elements were used in order to ensure the I/O file operations. The architecture of modules is presented below.

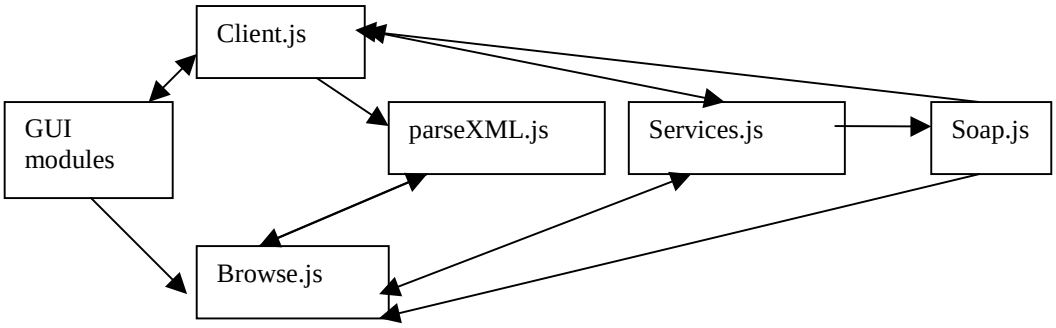

Figure 4.6 The modules architecture in the AXML Commander application

The *Client.js* module is the *script entry point*: it is the main *interface* between the GUI modules and the JavaScript modules and in the same time it is an interface module between the GUI modules themselves ensuring the dynamic loading of XUL interface modules in the application's front-end window. The most of the *event-handling functions* are in this module.

The *Browse.js* module is the one grouping the main of the updated Active XML Browser functionality.

The *parseXML.js* module uses the JavaScript API offered by the *Mozilla 3 DOM implementation* for parsing any needed XML documents in order to extract the useful information.

The *Services.js* module is now grouping the PAP module's counterpart functions (the functions responsible for the **remote interaction** with the module on the Peer).

The *soap.js* module contains the functions involved in messages SOAP coding.

4.2.6 AXML Commander's Functionality

Active XML Commander is a flexible and powerful tool which meets most of the requirements stated in the project specification:

- The first requirement is fulfilled: the local I/O file operations are fully implemented, so the user is able to have access to any document on the local system (not only Active XML actually, but the tentative of loading a non-XML file will generate an error because the application would try parsing that file) by setting the directory path for the local repository as being the directory path of that document. An important issue being the portability, the application works fine on the Unix and Windows System since it's using the Mozilla back-end to figure out the platform Mozilla is running (the application has access to the version compilation type through a JavaScript navigator object). Documents can be edited (using a

simple text editor) and browsed (using **the Active XML Browser** built on a special-purpose written XSL style sheet). Using the same Browser, various attributes of the service calls in documents can be manipulated (activation mode and passed parameters) and service calls can be activated from the current compute, the possibility of modifying documents using the results of these service calls being provided too.

- The second requirement is partially fulfilled: the user is able to connect to remote peers and view the *Active XML content* of those peers (documents and service calls through their definitions). Using the updated version of the Active XML Browser, the user can force the remote activation of a service call and view the corresponding results by reloading the document in the application. The option of *querying* an Active XML Peer from the **Commander** context is not yet implemented.
- The third requirement is partially fulfilled as well. The user is able to perform all the desired management operations in the case of the documents: documents on the Peer locally available documents can be created/deleted/modified at will. Transferring a document from a Peer to another, from the Local Space to a Peer or from the Peer to the Local Space is possible with a simple **drag-and-drop** from the corresponding "label" representing the document name in the *AXML Content Area* to the label representing the destination Peer name in the *AXML Navigator*. For the services, only the possibility of viewing/modifying the Service Definition is currently available. It is not possible to create (install) a service on a Peer using the **Active XML Commander**.

Chapter 5
Future extensions

As it can be noticed, the work on the **Active XML light-weight Client** project is far from being completed.

Functionalities such as creating/ installing services on the Active XML Peer and of remote querying this Peer should be introduced as well.

One important, very important issue of such architecture would be the security. There are two possible options I consider here:

- providing a username and a password for every single operation on the Peer.
- using a session mechanism which would allow a user to authenticate only at the beginning of such a session. The session would have to be managed on the Peer.

Another issue would be the encryption of the SOAP messages (one reason would quickly come to mind in this context: parts of Active XML documents are exchanged over SOAP and they would have to be confidential, the other being the fact the authentication to a peer requires an username and a password which would have to remain secret).

As the graphical interface is concerned we could optimize it by adopting inline modifications of attributes/ parameters, a unique space for displaying the interchangeable views of one document and a simplified-tree view aside which could be seen as providing shortcuts to the elements of interest of Active XML documents.

Bibliography

[1] S. Abiteboul, P. Buneman, D. Suciu **Data on the Web: From Relations to Semistructured Data and XML**, Morgan Kaufmann Publishers, San Francisco, 1999
[2] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, F. Ngoc **Exchanging Intensional XML Data**, SIGMOD 2003
[3] S. Abiteboul, T. Milo, O. Benjelloun **Web Services and Data Integration**, International Conference onWeb Information Systems Engineering 2002
[4] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, R. Weber **Active XML: Peer-to-Peer Data and Web Services Integration (demo)**, VLDB 2002
[5] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, R. Weber **Active XML: A Data-Centric Perspective on Web Services**, Conference sur les Bases de Donnes Avances 2002
[6] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, T. Milo **Dynamic XML Documents with Distribution and Replication**, INRIA Futurs 2003
[7] S. Abiteboul, O. Benjelloun, T. Milo **Towards a Flexible Model for Data and Web Services Integration**, proc. Internat. Workshop on Foundations of Models and Languages for Data and Objects 2001
[8] The Active XML Site (http://www.purl.org/net/axml)
[9] Steve Graham, Simeon Simeonov and others, **Building Web Services with Java**, SAMS Publishing 2001
[10] **Creating Applications with Mozilla,** O'Reilly & Associates http://books.mozdev.org/html/index.html
[11] Scott Andrew LePera, **Using the Mozilla SOAP API,** O'Reilly 2002 http://www.oreillynet.com/lpt/a/2677
[12] Trausan-Matu, Stefan, **Advanced Interfaces** Course**, *Politehnica University***, Bucharest
[13] The **JavaScript Central** Site, http://devedge.netscape.com/central/javascript/
[14] Doron Rosenberg (Netscape Communications), **The XSLT/JavaScript Interface In Gecko**, Published 09 May 2003 http://devedge.netscape.com/viewsource/2003/xslt-js/
[15] W3C Working Group, **Document Object Model Core**, http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/core.html
[16] W3C Working Group, **Document Object Model XPATH**, http://www.w3.org/TR/2004/NOTE-DOM-Level-3-XPath-20040226/xpath.html
[17] **Using the XML HTTP Request Object,** http://jibbering.com/2002/4/httprequest.html
[18] **The *XUL Planet* Site**, http://www.xulplanet.com/