Active XML User's Guide

The Active XML team

Contents

1 Introduction

Active XML (*http://activexml.net*) is a declarative framework for distributed data management based on XML and web services (SOAP, WSDL). An Active XML document is an XML document that may contain calls to web services.

An Active XML peer is mainly a repository of Active XML documents. It is both a client (as a web services consumer) and a provider of declarative web services. A service can be defined, for example, as a query over a stored XML document or an update of the document. The use of calls to web services embedded in documents allows to dynamically update data and also to control

frequency of these updates and durability of the information. The peers can exchange intensional data, i.e. data defined as calls to Web services. Thus, each Active XML peer acquire and provide dynamic information, in a decentralised peer-to-peer architecture.

This user's guide presents the basic steps of installation of the Active XML framework, the main concepts of the system, and describes the definition, usage and management of Active XML documents and different kinds of Web services.

2 Instalation

The Active XML framework is provided as a web application for a Servlet 2.3 / JSP 1.2 container like Tomcat 4.1 (*http://jakarta.apache.org/tomcat/*). You should also be able to get it working with other containers like Jetty (*http://jetty.mortbay.org/*), Orion (*http://www.orionserver.com/*) or Resin (*http://www.caucho.com/*).

This section describes its installation with the Tomcat 4.1 container. The steps needed to install and use the Active XML framework are as follows.

2.1 Download and install a Java Development Kit

The first step is to download a Java Development Kit (JDK) release (version 1.4 or later) from http://java.sun.com/j2se/. Make sure it is a JDK and not a Java Runtime Environment (JRE). Install the JDK according to the instructions included with the release.

2.2 Download and install the Tomcat 4.1 container

The second step is to download a binary distribution of Tomcat 4.1 from http://jakarta.apache.org/tomcat/ and install it according to the instructions included with the release. Make sure to test it before going on with the installation of the Active XML framework.

2.3 Download and install the Active XML framework

ActiveXML is now an Open Source project. The latest CVS version of the project, containing Source code, binaries and a sample *webapp*, is available for download at *http://forge.objectweb.org/projects/activexml/*, in the CVS part. There are several directories in this package:

- "Active XML" and "Gemo Utilities", containing the source code of Active XML,
- "XOQL" and "CDQA", containing the source code of the XML repository and query language that are mainly used in Active XML,
- "Copy of Active XML webapp", which contains a sample *webapp* with documents, services and examples for all the main features of Active XML,
- "Java Libs", containing jars that are used by the various projects,
- "ActiveXML Light Client", the Mozilla plugin of the client for ActiveXML peers.

To install an Active XML peer on a machine, copy the directory "Copy of Active XML webapp/axml" in the "webapps" directory of Tomcat. Restart Tomcat and your peer will be available via a user interface at http://localhost:8080/axml. The same can be achieved by using the Eclipse environment and its plugin for Tomcat. In this case, you just need to declare that "Copy of..." is a Tomcat project (if by default it is not the one).

If this part fails and the peer is not available at the mentioned address, you can consult the logs of Tomcat to see what exactly went wrong. You can consult them either from the "logs" directory of Tomcat or from the logging window of Eclipse (if Eclipse was used to start/restart Tomcat).

Finally, if you want to use continuous services (see 3.4.4) as a client, you will need to modify the value of the *axml.peer.url* parameter in the *web.xml* file in the *Copy of Active XML webapp/axml/WEB-INF* directory by entering the correct address of your peer instead of "localhost".

3 Using Active XML

3.1 Basic concepts of Active XML

In Active XML (AXML for short), parts of data are given explicitly, while other parts consist of calls to Web services that generate more data. AXML is an XML dialect. Let us consider an example where an AXML document invokes a service call which consists of a query on another XML document.

First, an XML document representing a short ATP list (*ATPList.xml*) of tennis players is specified:

```
<?xml version="1.0" encoding="UTF-8"?>
 <ATPList date="18042005">
    <player>
        <name>
            <firstname>Roger</firstname>
            <lastname>Federer</lastname>
        </name>
        <points>475</points>
    </player>
    <player>
        <name>
            <firstname>Rafael</firstname>
            <lastname>Nadal</lastname>
        </name>
        <points>313</points>
    </player>
 </ATPList>
```

Next, a Web service including a X-OQL query is defined. X-OQL is a query language for XML developed at INRIA. The following service (named *GetATP*) uses a query which returns the *firstname* and the *lastname* for every *player* element in the *ATPList.xml* document.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
    <serviceDefinition type="query">
        <definition>
            <query>
                <![CDATA[
                        select p/name
                        from p in ATPList//player;
                    ]]>
            </query>
        </definition>
    </serviceDefinition>
```

See more details on defining Web services in section 3.4, with more examples of X-OQL statements given in 3.4.1.

A call to the service *GetATP* is included into the following AXML document (named *test-GetATP.xml*). Generally, an AXML document is a valid XML document where some particular elements (labeled *sc* and associated with the *axml* namespace) are interpreted as service calls.

```
<?xml version="1.0" encoding="UTF-8"?>
 <atp xmlns:axml="http://www-rocq.inria.fr/verso/AXML">
    <players>
        <axml:sc frequency="every 3600000"
            serviceNameSpace="GetATP"
            serviceURL="http://localhost:8080/axml/servlet/AxisServlet"
            methodName="GetATP"
            mode="replace">
           <axml:params/>
        </axml:sc>
    </players>
</atp>
```

In this example, the service call will be activated every hour (*frequency* is set to 3600000 miliseconds) and the previous results will be replaced by the ones returned by the current service invocation (*mode="replace"*). For more details on service definition, see section 3.5.

After the service invocation, the end of the AXML document, below the *axml:sc* element, will look like this:

```
                    ...
            <name>
                <firstname>Roger</firstname>
                <lastname>Federer</lastname>
            </name>
            <name>
                <firstname>Rafael</firstname>
                <lastname>Nadal</lastname>
            </name>
```

```
        </players>
</atp>
```

For the sake of simplicity, some attributes that are automatically generated during the service invocation (namely *origin* and *timestamp*) were not shown in the last XML document.

3.2 Creating your Active XML peer

As said in the introduction of this document, an Active XML peer is mainly a repository of Active XML documents. An Active XML document is a well-formed XML document that may contain calls to Web services.

The repository of the Active XML peer is located in the *repository* directory in your Active XML installation. In order to publish a Web service that contains a query and to call the service from an AXML document, the *repository* directory should contain:

1. an XML document that will be queried (such as *ATPList.xml*),
2. a Web service (like *GetATP*),
3. an AXML document that invokes the Web service (such as *testGetATP.xml*).
4. an *docRepository.xml* file, which consists of references to Active XML files (in this example, to *testGetATP.xml* and *ATPList.xml*):

```
<?xml version="1.0" ?>
 <mxbase>
    <forest name="TestGetATP" replicationIndex="0">
        <file name="" href="testGetATP.xml" />
    </forest>
    <forest name="ATPList" replicationIndex="0">
        <file name="" href="ATPList.xml" />
    </forest>
 </mxbase>
```

5. an *svcRepository.xml* file should contain the reference to Web services, such as:

```
<?xml version="1.0" ?>
 <mxbase>
   <forest name="GetATP" replicationIndex="0">
        <file name="" href="GetATP.xml" />
   </forest>
 </mxbase>
```

As shown above, the structure of *docRepository.xml* and *svcRepository.xml* files is quite simple. The root element is named *mxbase* and the items of the repository are represented by a *forest* element.

In *docRepository.xml*, a forest is a logical document that may be composed of several physical XML or Active XML documents. The *name* attribute of the *forest* element identifies the forest. The relative path of each physical document can be specified by using the *file* element and its *href* attribute. In *svcRepository.xml*, a forest is related to a Web service.

Consider now another example of the *docRepository.xml* file, where the forest named *ATPList* consists of the two files, *ATP1.xml* and *ATP2.xml*:

```
<?xml version="1.0" ?>
 <mxbase>
    <forest name="ATP" replicationIndex="0">
        <file href="ATP1.xml" />
        <file href="ATP2.xml" />
    </forest>
 </mxbase>
```

If *ATP1.xml* is written as:

```
<?xml version="1.0" encoding="UTF-8"?>
 <atp1>
    <player>
            <firstname>Roger</firstname>
            <lastname>Federer</lastname>
            <points>450</points>
    </player>
 </atp1>
```

and *ATP2.xml* is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
 <atp2>
    <player>
            <firstname>Rafael</firstname>
            <lastname>Nadal</lastname>
            <points>313</points>
    </player>
 </atp2>
```

then the following query (notice that we use the name of the forest here):

```
        ATP;
```

will return:

```
<result>
 <atp1>
    <player>
            <firstname>Roger</firstname>
            <lastname>Federer</lastname>
            <points>450</points>
    </player>
 </atp1>
 <atp2>
    <player>
            <firstname>Rafael</firstname>
```

```
            <lastname>Nadal</lastname>
            <points>313</points>
    </player>
 </atp2>
</result>
```

3.3 Accessing your Active XML peer

You can access the Web interface of your AXML peer by pointing your browser to *http://localhost:8080/axml/*. You can then interact with your Active XML peer:

- access, query and modify your AXML documents,
- manually activate AXML service calls,
- see and test your AXML web services.

One can also use the provided Java Web Services (such as *PeerAccessPoint* or *DocumentRepository*) to:

- query AXML documents,
- retrieve, save or remove AXML documents/services,
- activate service calls.

via Web Service invocation.

3.4 Active XML web services

An Active XML peer is also a repository of Active XML web services. An Active XML web service is defined in a declarative manner and can be of type:

- query,
- update,
- xslt,
- continuous,
- written in Java.

3.4.1 X-OQL queries

In order to explain some basic elements of X-OQL syntax on different examples, the *ATP.xml* file will be slightly modified:

```
<?xml version="1.0" encoding="UTF-8"?>
 <ATPList date="18042005">
    <player rank=1>
        <name>
            <firstname>Roger</firstname>
            <lastname>Federer</lastname>
        </name>
        <citizenship>Swiss</citizenship>
        <points>475</points>
    </player>
    <player rank=2>
        <name>
            <firstname>Rafael</firstname>
            <lastname>Nadal</lastname>
        </name>
        <citizenship>Spanish</citizenship>
        <points>313</points>
    </player>
</ATPList>
```

The basic syntax of X-OQL can be illustrated by the following query:

```
select p/name/lastname
from p in ATPList//player
where p@rank=1;
```

The query selects the lastname of the player who is at the first place of the ATP list. The FROM clause binds the *p* variable to each *player* element in the document. The WHERE clause selects the elements that satisfy the given filter. In this query it is the value of *rank*, which is an atribute of the *player* element. The result of the query is:

```
<result>
    <lastname>Federer</lastname>
</result>
```

In the following query, the WHERE clause is composed of two conditions considering the values of elements *citizenship* and *points*:

```
select p/name/lastname, p/points
from p in ATPList//player
where p/citizenship = Spanish and p/points > 300;
```

The result consists of the *lastname* and *points* of the player satisfying both conditions of the given filter:

```
 <result>
    <lastname>Nadal</lastname>
    <points>313</points>
 </result>
```

The next example shows the construction of elements in X-OQL. The *player* element will be constructed, having *points* and *rank* as attributes and *firstname* and *lastname* by using the following statement:

```
select <player points ={p/points}
            rank={p@rank}>
         p/name/firstname,
         p/name/lastname
        </>
from p in ATPList//player;
```

The result is:

```
<result>
   <player points="475" rank="1">
       <firstname>Roger</firstname>
       <lastname>Federer</lastname>
   </player>
   <player points="313" rank="2">
       <firstname>Rafael</firstname>
       <lastname>Nadal</lastname>
   </player>
</result>
```

For more information about X-OQL, see *http://activexml.net/xoql/documentation/index.html*.

3.4.2 Update services

These Active XML services can be used to perform updates on documents of the peer's repository. We define them by using a simple update language implemented on top of X-OQL.

The definition of an update service will have two parts, both consisting of X-OQL queries:

- a "location" part, defining the document nodes on which an update will be performed,
- a "data" part, along with an action ("replace", "delete" or "insert").

Depending on the specified action, one of the following activities will take place at each of the nodes selected by the "location" part (as a relative root):

- deleting the piece of data selected by the "location" part,
- replacing it by the piece of data returned/constructed by the "data" part,
- inserting the piece of data returned/constructed by the "data" part.

Example 1. A replace service that changes the value of the social security number for a certain patient in a medical file:

```
<serviceDefinition type="update">
    <definition>
    <action type="replace">
        <data>
          <SSN>123-555-1234</>;
        </data>
        <location>
          select y
          from   x in PatientFiles/patientData,
                 y in x/SSN
          where x@patientID='5';
        </location>
    </action>
    </definition>
</serviceDefinition>
```

Example 2. A service that removes a patient:

```
<serviceDefinition type="update">
    <definition>
    <action type="delete">
        <data>
        </data>
        <location>
          select x
          from  x in PatientFiles/patientData
          where x@patientID='5';
        </location>
    </action>
    </definition>
</serviceDefinition>
```

Example 3. A service that constructs a *visit* element using data from one document and inserts it in another document:

```
<serviceDefinition type="update">
  <definition>
   <action type="insert">
    <data>
       select <visit patientID='5'>
                y,
                {<notes>Some notes</>}
              </>
       from x in PatientFiles/patientData,
            y in x/visit/*
       where x@patientID='5';
```

```
     </data>
     <location>
          MedicalFiles/patientVisits;
     </location>
    </action>
   </definition>
  </serviceDefinition>
```

3.4.3 XSLT

X-OQL is not the only data management language that can be used to define Active XML service. Similarly to X-OQL query services, we can define XSLT services, specified by a template. The service will return the piece of data that is constructed by the template. The template will be applied on data is passed as parameter, and not on repository data as in the case of services defined as X-OQL services.

We will show an example to clarify these type of AXML services. First, see the following XML document containing *book* elements:

```
<bib axml:docName="BIB" xmlns:axml="http://www-rocq.inria.fr/verso/AXML">
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>
    <book year="1992">
        <title>Advanced Programming in the Unix environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>
    <book year="2000">
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>39.95</price>
    </book>
</bib>
```

The XSLT service below takes the document describing books. It filters out the books published before the year 1994. It also does some restructuring: it transforms *price* elements into attributes and keeps only book titles (author and publisher data is left out).

```
<serviceDefinition type="xslt">
```

```
  <parameters>
     <param name="bib"/>
  </parameters>
  <definition>
   <xslt>
    <![CDATA[
     <xsl:stylesheet version="1.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:output method="text"/
      <xsl:template match="/">
         <xsl:apply-templates/>
      </xsl:template>
      <xsl:template match="bib">
        <xsl:text>&lt;bib since="1994"></xsl:text>
            <xsl:for-each select="book">
                <xsl:if test="@year &gt;= 1994">
                    <xsl:apply-templates select="." />
                </xsl:if>
            </xsl:for-each>
        <xsl:text>&lt;/bib></xsl:text>
      </xsl:template>
      <xsl:template match="book">
         <xsl:text>&lt;book year="</xsl:text>
         <xsl:value-of select="@year"/>
         <xsl:text>" price="</xsl:text>
         <xsl:value-of select="price/text()"/>
         <xsl:text">></xsl:text>
         <xsl:text>&lt;title></xsl:text>
         <xsl:value-of select="title/text()"/>
         <xsl:text>&lt;/title></xsl:text>
         <xsl:text>&lt;/book></xsl:text>
      </xsl:template>
   </xsl:stylesheet>
                ]]>
   </xslt>
  </definition>
</serviceDefinition>
```

After calling that XSLT service, the result will be the following:

```
<bib axml:origin="2270466F-89C2-0834-016B-B7D952B36E3A"
   axml:timestamp="1125859312619" since="1994">
 <book price="65.95" year="1994">
      <title>TCP/IP Illustrated</title>
 </book>
 <book price="39.95" year="2000">
```

```
      <title>Data on the Web</title>
 </book>
</bib>
```

3.4.4 Continuous services

Continuous services enable the clients to subscribe to some information, which will be periodically pushed to them by a server. The functionality of the continuous services can be described by the following basic steps:

1. The client sends a subscription request to the service provider which registers it.
2. Periodically, the service provider will do the following actions:
   - Evaluate the subscription.
   - Call a possible post-processing service (if it is specified to do so in the subscription).
   - Call the client's callback service and send it the results.
3. The client will periodically integrate the results by means of the callback service.

By using the continuous services, the clients do not have to call the same service over and over again, as often as they wish to receive the results. On the other hand, in a pull system, the same request would be periodically sent over the network and generate an overhead. Also, it could happen that the data is not available for clients every time the service is called. Thus, the peer would be forced to evaluate the calls that it cannot respond to. Another important limitation that is overcome by the continuous services system is the one given by the timeout of connections. In our system, a connection is open only when the client is demanding a subscription and every time the service result is available for the client. The connection will be automatically closed after reaching the maximum allowed inactivity time interval.

For more details and some examples on continuous services, see Appendix A.

3.4.5 Services written in Java

Users can also create their own services written in Java and those services can be invoked from AXML documents. The services can be anywhere in the *CLASSPATH*. Let us create, for instance, a a simple class "Salut" and store it in the "Active XML" project.

```
package fr.inria.gemo.axml.service.webservices;
 public class Salut
{
    String hello = "Salut!";
    public Object greeting(){
        return hello;
    }
}
```

After creating this service, the "axml.jar" file should be updated. Then the "server-config.wsdd" file in "Copy of ActiveXML webapp/axml/WEB-INF" should be modified by adding a new *service* tag for the *Salut* service:

```
<service name="Salut" provider="java:RPC">
   <parameter name="allowedMethods" value="greeting"/>
   <parameter name="className"
       value="fr.inria.gemo.axml.service.webservices.Salut"/>
   <parameter name="scope" value="application"/>
</service>
```

To invoke this service written in Java, we need an AXML document like the one below:

```
<?xml version="1.0" encoding="UTF-8"?> <salut
axml:docName="TestGreeting"
xmlns:axml="http://www-rocq.inria.fr/verso/AXML">
    <sayHello>
        <axml:sc methodName="greeting"
            serviceNameSpace="Salut"
            serviceURL="http://localhost:8080/axml/servlet/AxisServlet">
        </axml:sc>
    </sayHello>
</salut>
```

3.5 Active XML Service Call

The main Active XML specific XML element is the Service Call (*sc*) element. It is defined in the *http://www-rocq.inria.fr/verso/AXML* namespace and usually bound to the *axml* prefix. The *sc* element defines the behaviour of an Active XML Service Call. It is composed of a set of attributes and children XML elements defining :

- the Web Service to call,
- how and when to call it,
- what to do with the results,
- the call parameters.

3.5.1 Web service information

Attribute	Type	Default Value	Status
serviceURL	anyURI		required
serviceNameSpace	anyURI		required
methodName	NCName		required
signature	anyURI		optional
useWSDLDefinition	boolean	false	optional

serviceURL attribute This attribute specifies the endpoint URL of the Web Service (see the SOAP 1.1 TR : 2.6 Ports; 3 SOAP Binding; 3.8 soap:address; 4.3 http:address) to call, for example *http://api.google.com/search/beta2* or *http://localhost:8080/axml/servlet/AxisServlet*.

serviceNameSpace attribute This attribute specifies the namespace to use for the body element of the SOAP message (see the SOAP 1.1 TR : 3.5 soap:body), more simply the method namespace URI. This information is provided in the WSDL of the Web Service. For example : *urn:GoogleSearch* or *GetMatchingFeeds*.

methodName attribute This attribute defines the name of the operation to invoke on the Web Service (see the WSDL 1.1 TR : 2.4 Port Types), for example *doGoogleSearch*. Note that for an Active XML Web Service (a declarative Web Service provided by an Active XML Peer), the value of the *methodName* attribute is not important. By convention we use "invoke", but any other name can be used.

signature attribute This attribute sets the URL of the WSDL file defining the Web Service. It is optional and is only used if you want to do type validation on that particular Service Call. For example : *http://api.google.com/GoogleSearch.wsdl*.

useWSDLDefinition attribute This attribute specifies if the WSDL file defined by the signature attribute should be used for type validation. It is an optional boolean attribute and its default value is false.

Examples The following example shows how to define the Web service related information to call the *GoogleSearch* Web service and especially the *doGoogleSearch* operation. Note that it specifies type validation. If you do not want to validate types, you can just omit the *signature* and *useWSDLDefinition* attributes.

```
<axml:sc xmlns:axml="http://www-rocq.inria.fr/verso/AXML"
  serviceURL="http://api.google.com/search/beta2"
  serviceNameSpace="urn:GoogleSearch"
  methodName="doGoogleSearch"
  signature="http://api.google.com/GoogleSearch.wsdl"
  useWSDLDefinition="true"
  ...>
        [children elements]
</axml:sc>
```

This other example shows how to define the Web service related information to call an Active XML Web service. As the previous example it specifies type validation.

```
<axml:sc xmlns:axml="http://www-rocq.inria.fr/verso/AXML"
  serviceURL="http://localhost:8080/axml/servlet/AxisServlet"
  serviceNameSpace="GetMatchingFeeds"
  methodName="invoke"
```

```
  signature="http://localhost:8080/axml/services/GetMatchingFeeds?wsdl"
  useWSDLDefinition="true"
  ...>
        [children elements]
</axml:sc>
```

3.5.2 Service call information

DESIREDFREQUENCY, FORWARDINGLIST]

A *sc* element also provides information on how and when to activate the service call through the following attributes:

Attribute	Type	Default Value	Status
id	ID		generated
name	string		optional
frequency	string		optional
callable	boolean	true	optional
lastCalled	unsignedLong		generated
followedBy	string		optional

id attribute The *id* attribute identifies (uniquely over time and space) a Service Call. You should not define it manually to insure its uniqueness, the Active XML system will generate it automatically and make it persistent.

Its value is restricted to the pattern:

```
[0-9 A-F]{8}\-[0-9 A-F]{4}\-[0-9 A-F]{4}\-[0-9 A-F]{4}\-[0-9 A-F]{12}
```

name attribute This attribute specifies the name of a Service Call. It has no specific meaning and thus is optional. It is kept to ensure backward compatibility but may be totally ignored in future releases.

frequency attribute This attribute defines when the Service Call will be activated unless the *callable* attribute is set to false. It can have several values :

- once,
- lazy,
- on Date,
- every X.

If no frequency is defined the Service Call will not be activated by the Active XML Peer unless you explicitly evaluate it through the Active XML Web Interface.

Once means that the Service Call will be activated only once at start-up time. Note that every time you start or restart your Active XML Peer, Service Calls with a frequency of once will be activated.

Lazy means that a Service Call will only be activated when its results may be useful to the evaluation of a X-OQL query or the instantiation of a Service Call Parameter defined through an XPath expression.

On Date specifies when exactly the Service Call will be activated. The date format pattern is dd/MM/yy HH:mm. If the date has an incorrect format or if it is in the past, the Service Call won't be activated. For example *frequency="on 25/12/05 14:36"* will activate the Service Call on the 25th December of 2005 at 14:36.

every X means that the Service Call will be activated every X milliseconds. For example *frequency="every 1800000"* will activate the Service Call every 30 minutes.

callable attribute By setting the value of this attribute to "false" you can make a Service Call not being activated by the Active XML Peer even though it defines a correct call frequency.

lastCalled attribute This attribute is used by the Active XML Peer to keep track of the last activation of a Service Call to enforce the frequency when the Active XML Peer is restarted or the Active XML Document reloaded after modifications. It is set by the system and you should not specify it unless you are sure of what you are doing. For implementation reasons, the value should be less than 9223372036854775807.

followedBy attribute This attribute allows to chain (a very simple kind of workflow) evaluation of Service Calls inside the same Active XML Document. The value of this attribute must be an XPath expression returning a single *sc* element or the *id* attribute of a Service Call. For example,

```
followedBy="//axml:sc[@serviceNameSpace='XSLT_BIB_QUERY']"
```

specifies that right after the activation of the current Service Call is completed, the Service Call with the serviceNameSpace attribute set to

```
XSLT_BIB_QUERY
```

should be activated.

Examples The following examples extend the ones presented in the previous section with the Service Call related information presented in this section.

This example defines the *id*, *name*, *frequency*, *lastCalled* and *followedBy* attributes. Remember that the *id* and *lastCalled* are generated by the Active XML Peer and are provided here for information purpose only, you should never define them yourself.

```
<axml:sc xmlns:axml="http://www-rocq.inria.fr/verso/AXML"
  serviceURL="http://api.google.com/search/beta2"
  serviceNameSpace="urn:GoogleSearch"
  methodName="doGoogleSearch"
  signature="http://api.google.com/GoogleSearch.wsdl"
  useWSDLDefinition="true"

  id="038BD50A-C353-D490-0082-D603FA7A81AC"
```

```
  name="GoogleSearch"
  frequency="lazy"
  lastCalled="1082395879693"
  followedBy="//axml:sc[@serviceNameSpace='Crawler']"

  ...>
        [children elements]
</axml:sc>
```

This example only defines the *frequency* attribute. You can see it as a typical Active XML Document that has never been loaded by a peer.

```
<axml:sc xmlns:axml="http://www-rocq.inria.fr/verso/AXML"
  serviceURL="http://localhost:8080/axml/servlet/AxisServlet"
  serviceNameSpace="GetMatchingFeeds"
  methodName="invoke"
  signature="http://localhost:8080/axml/services/GetMatchingFeeds?wsdl"
  useWSDLDefinition="true"
  frequency="once"
  ...>
        [children elements]
</axml:sc>
```

3.5.3 Service call result handling

The way of treating the results of an Active XML Service Call can be specified by users. Considering an XML document as an ordered labelled tree, the results will be inserted as a sibling of the *sc* element. The behaviour is defined by two attributes :

Attribute	Type	Default Value	Status
mode	string	merge	optional
doNesting	boolean	false	optional

mode attribute This attribute specifies what to do with the results of an Active XML Service Call activation. It can take two values (default value is *merge*) :

- merge,
- replace.

Merge defines that the results will be added as a sibling of the *sc* element, while the previous results will be kept in the XML document too.

Replace means that the previous results will be replaced by the ones returned by the current invocation of the Active XML Service Call.

To keep track of the inserted 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 *http://www-rocq.inria.fr/verso/AXML* having as value the id attribute of the Active XML Service Call.

doNesting attribute The tracking of inserted results described previously works well with elements, but cannot be applied to TEXT nodes. If you want to keep track of inserted TEXT nodes, you can ask the Active XML Peer to nest the TEXT nodes in a special element named *text* and bound to the Active XML (*http://www-rocq.inria.fr/verso/AXML* like the following example :

```
<axml:text
    axml:origin="038BD50A-C353-D490-0082-D6033B1BA431">Hello World
</axml:text>
```

[ADD: FUSION DESCRIPTION (SYNTAX, BEHAVIOUR AND EXAMPLES)]

3.5.4 Service call parameters

The parameters of the Web Service that a Service Call references are specified by a child element of the *sc* element. This element must be named *params* and bound to the Active XML namespace *http://www-rocq.inria.fr/verso/AXML* and also must be present even though the Web Service does not have any parameters. This *params* element conforms to the following XML schema representation :

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

If your Web Service has no input parameter, you would just write an empty *params* element like :

```
<axml:sc ...>
  <axml:params />
</axml:sc>
```

An Active XML Parameter is defined by the *param* element bound to the Active XML namespace *http://www-rocq.inria.fr/verso/AXML* and can be expressed as a value or through an XPath expression. The Active XML parameter is either a *value* or a *xpath* parameter, according to the following schema component representation :

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

The *param* element has only one required attribute (named *name*) specifying the name of this parameter. The value of this attribute must conform to the name of a web service parameter referenced by the Active XML Service Call.

Note that if your Active XML Service Call has several parameters, they must be in the correct order as defined by the web service WSDL file.

Value parameter The first way of writing an Active XML parameter is through a "value". This value can be any well-formed XML fragment. If you need to define an Active XML parameter as an (A)XML document (including the prolog part - see *http://www.w3.org/TR/2004/REC-xml11-20040204/#NT-prolog*), we recommend using a CDATA section to keep your Active XML document well-formed. The *value* parameter conforms to the following schema component representation :

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

For example, to call a currency converter Web Service taking as inputs from and to (as currency element), you would write the following parameters :

```
<axml:params>
  <axml:param name="from">
    <axml:value>
      <currency symbol="EUR">1</currency>
    </axml:value>
  <axml:param>
  <axml:param name="to">
    <axml:value>
      <currency symbol="USD">1</currency>
    </axml:value>
  <axml:param>
</axml:params>
```

XPath parameter The other 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>
```

An XPath parameter is considered by the system as being non-concrete and its evaluation is done upon need. Thus, every time a Service Call is activated, the XPath parameters will be evaluated. The evaluation of an XPath expression may return several values. In that case, a cross-product will be done with all the values of the different parameters of your Service Call and will actually call the Web Service as many times as required.

To better understand this, consider a Currency Converter web service and the following example of AXML document:

```
<?xml version="1.0" encoding="UTF-8"?>
  <currencies xmlns:axml="http://www-rocq.inria.fr/verso/AXML">
  <currency symbol="EUR">1</currency>
  <currency symbol="GBP">1</currency>
  <currency symbol="CHF">1</currency>
  <currency symbol="USD">1</currency>
  <axml:sc serviceURL="http://www.some-url.com/services"
     serviceNameSpace="urn:CurrencyConverter"
     methodName="convert"
     frequency="every 1800000">
    <axml:params>
      <axml:param name="from">
          <axml:xpath>//currency</axml:xpath>
      </axml:param>
      <axml:param name="to">
          <axml:xpath>//currency</axml:xpath>
      </axml:param>
    </axml:params>
  </axml:sc>
</currencies>
```

After the evaluation of the instantiation of the two parameters (i.e. evaluation of the XPath expressions), the following invocations will be performed:

conv(EUR,EUR)	conv(GBP,EUR)	conv(CHF,EUR)	conv(USD,EUR)
conv(EUR,GBP)	conv(GBP,GBP)	conv(CHF,GBP)	conv(USD,GBP)
conv(EUR,CHF)	conv(GBP,CHF)	conv(CHF,CHF)	conv(USD,CHF)
conv(EUR,USD)	conv(GBP,USD)	conv(CHF,USD)	conv(USD,USD)

As can be seen, this can lead to many service calls including redundant ones.

3.6 Customizing your Active XML Peer

[EXPLAIN WEB.XML FILE PARAMETERS]

APPENDIX

A Continuous Services

The process of turning a normal service into a continuous one can be seen as a "wrapping" of that service in a continuous service. After a client sends a subscription request, the behavior on server side consists of the following steps:

- calling the wrapped service from the continuous service,
- retrieving the results,
- sending the results back to the client's callback service.

We can also define services that process data after invoking the wrapped service and before sending the data to the client's callback service. A typical example of such post-processing is a service that returns the differences between a document and its previous version. For the sake of simplicity, the post-processing services will not be further described in this document.

When using continuous services, the repository on the **server side** should contain:

- a continuous service,
- a wrapped service,
- a post-processing service (optional).

Another option is to have a continuous service that contains, for instance, a wrapped query, instead of invoking another service (the wrapped service) to execute the query.

See the continuous service `ContinuousGetPlayerByRank.xml` (that invokes the wrapped service `getPlayerByRank.xml`):

```
<?xml version="1.0" encoding="UTF-8"?> <serviceDefinition
name="ContinuousGetPlayerByRank" type="continuous">
  <!-- Continous service parameters -->
  <parameters>
    <param name="rankNumber"/>
  </parameters>
  <definition>
    <!-- Non continuous service location -->
       <baseService URL="http://localhost:8080/axml/servlet/AxisServlet"
            methodName="getPlayerByRank"
            nameSpace="GetPlayerByRank"/>
    <!-- Result transformation service (Post Processing Service)
        <transformService URL="" methodName="" nameSpace=""/>
    <!-- Frequency Settings -->
```

```
        <frequency default="60000" max="10000"/>
    <!-- Subscription Settings -->
        <subscription maxLifeTime="8640000"
        maxSuspensionTime="86400" maxClientsNo="15000"/>
  </definition>
</serviceDefinition>
```

Note the following details in the continuous service shown above:

- The response frequency is specified by the *frequency* element. In this example, the wrapped service will be invoked by default every minute (60000 ms), while the maximum allowed frequency (i.e. minimum time period) that a client can ask for is every 10 seconds (10000 ms).
- The *subscription* element is used to impose subscription related restrictions:
  - *maxLifeTime* (maximum subscription validity time) - For instance, a provider can allow free subscription for one month. After that the clients that would like to continue using the service would have to pay for their subscription.
  - *maxSuspensionTime* (maximum subscription interruption time) - It limits the time interval for saving consecutive results. In other words, it defines for how long the results of a service will be saved in the case that a client is connected to the server.
  - *maxClientsNo* (maximum number of subscribers).
- In this example, the transform service (i.e. post-processing service) is not specified.
- In the continuous service above, by using the *baseService* tag, the wrapped service named *getPlayerByRank* is called.

The wrapped service `getPlayerByRank.xml` is shown below.

```
<?xml version="1.0" encoding="UTF-8"?> <coreServiceDefinition
type="query">
      <parameters>
        <param name="rankNumber"/>
      </parameters>
      <definition>
        <query><![CDATA[
          select <player>
              p/name,
              p/citizenship,
              p/atp/rank
              </>
            from p in ATPRace//player
            where p/atp/rank = rankNumber;]]>
        </query>
      </definition>
</coreServiceDefinition>
```

After writing the wrapped service and the continuous service, the references to them should be added to `svcRepository.xml` :

```
<forest name="GetPlayerByRank" replicationIndex="0">
        <file name="" href="GetPlayerByRank.xml" />
    </forest>
    <forest name="ContinuousGetPlayerByRank" replicationIndex="0">
        <file name="" href="ContinuousGetPlayerByRank.xml" />
    </forest>
```

After subscribing to the continuous service, the client will start receiving periodical results. The entry point on the client's peer for the data sent by the server is the callback service. It deals with the integration of received data in the client application. The result of the service will be inserted as a sibling of a service call's node.

When using continuous services, the repository on the **client side** contain:

- a callback service,
- an AXML file that calls a continuous service.

The callback service `CallbackService.xml` :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<serviceDefinition
    name="CallbackService" type="callback">
  <definition>
    <transformService URL="" methodName="" nameSpace=""/>
    <integratorService URL="" methodName="" nameSpace=""/>
  </definition>
</serviceDefinition>
```

In this simple example, no transformation (post-processing) service and no integrator service is specified.

The callback service should be referenced in `svcRepository.xml` :

```
    <forest name="CallbackService" replicationIndex="0">
        <file name="" href="CallbackService.xml" />
    </forest>
```

The continuous service is called from an AXML file: `testCompilInLineCont.xml`

```
<?xml version="1.0" encoding="UTF-8"?> <test_compil_inline_cont
axml:docName="TestCompilInLineCont"
    xmlns:axml="http://www-rocq.inria.fr/verso/AXML">
  <axml:sc callable="true"
```

```
        desiredFrequency="60000"
        frequency="once"
        methodName="doSubscribe"
        mode="merge"
        serviceNameSpace="ContinuousGetPlayerByRank"
        serviceURL="http://localhost:8080/axml/servlet/AxisServlet"
        subscription="true" useDiff="false"
        xmlns:axml="http://www-rocq.inria.fr/verso/AXML">
    <axml:params>
      <axml:param name="rankNumber">
        <axml:value>1</axml:value>
      </axml:param>
    </axml:params>
  </axml:sc>
</test_compil_inline_cont>
```

Note the following details in the AXML file shown above:

- The value of the *subscription* attribute should be set to "true".
- The *frequency* should be set to "once" (a subscription is started only once)
- SOAP parameters of the continuous service we wish to subscribe to are specified.
- The way of inserting the result into the document is defined.
- A set of optional parameters can be defined. Atribute *useDiff* specifies whether we want to use the post-processing service or not by setting the value for *useDiff* to "true". The *desiredFrequency* attribute specifies the desired time interval for receiving the results from the continuous service.
- Finally, the AXML file should be added to `docRepository.xml` :

```
<forest name="TestCompilInLineCont" replicationIndex="0">
    <file name="" href="testCompilInLineCont.xml" />
</forest>
```

After subscribing, the AXML will look like this:

```
<axml:sc callable="false"
    desiredFrequency="60000"
    frequency="once"
    id="175F6799-C353-D4DB-014B-03EAA1F6AB06"
    lastCalled="1125673561153"
    methodName="doSubscribe" mode="merge"
    serviceNameSpace="ContinuousGetPlayerByRank"
    serviceURL="http://localhost:8080/axml/servlet/AxisServlet"
    subscription="true" useDiff="false"
```

```
    xmlns:axml="http://www-rocq.inria.fr/verso/AXML">
    <axml:params>
      <axml:param name="rankNumber">
        <axml:value>1</axml:value>
      </axml:param>
    </axml:params>
  </axml:sc>

  <player axml:origin="F379673B-C353-D4DB-00B7-CD92ACD5E127"
    axml:timestamp="1125317811997">
    <name>
            <firstname>Roger</firstname>
            <lastname>Federer</lastname>
    </name>
    <citizenship>Swiss</citizenship>
    <rank>1</rank>
  </player>

</test_compil_inline_cont>
```

The *id* and *lastCalled* attributes are generated by the system, while the *player* element is included in the document as an output of the wrapped service. The system set the *callable* attribute to "false".

For more details on continuous services, see:
http://activexml.net/reports/internships/emanuel_report.pdf