INRIA Saclay Ile-de-France

ActiveXML documentation

version 2.1.4

Anca GHITESCU (anca.ghitescu@inria.fr)
Evaldas Taroza (taroza@gmail.com)

July 2008

Contents

1 Introduction

ActiveXML is based on XML and Web services. An active document is an XML document with embedded service calls to Web services, which enrich the document upon activation. In a peer-to-peer setting, this technology relies on the fact that peers communicate solely by means of Web services. The ActiveXML language has a predefined set of Web services for every peer, which enables distributed data management. Still, any peer can also provide other services, which actually causes the network to become a distributed database.

We can distinguish among several kinds of service calls:

- Service calls to the Web services that are known to be in the peer-to-peer network. These Web services represent the interface of a specific peer, and information about it can be meaningfully used, for example, by optimizers.
- Service calls to any Web service on the Web (e.g. Amazon Web Services).
- Continuous service calls. These are the calls to Web services that know how to stream back data, and notify about the end-of-stream. Basically, it is a matter of the communication protocol between a service call and the Web service. For instance, current implementation ships information about a service call (with the endpoint where to reply) to the Web service as a SOAP header.

There are different ways of calling a Web service: synchronously, asynchronously, one-way or two-way and different technologies that support the calling of web services. However, the promise of ActiveXML is to be able to call Web services declaratively. For instance, instead of creating software for calling some SOAP Web service, one could simply create an active document with a service call to that Web service and pass it to the engine to evaluate. Moreover, instead of creating software for piping Web services one could create an active document where one service call has another service call as a parameter. This shows that the correct usage of ActiveXML lies in creating, transforming, sending, receiving active documents and materializing service calls inside them.

1.1 Examples of ActiveXML documents

1.1.1 A document with a service call to a non-axml service

The document defines a service call to a service that converts the temperature from Celcius to Fahrenheit. When the document is activated by the ActiveXML system, the response is added to the axml document.

Definition of an axml document (a xml document that follows the axml syntax):

```
<example xmlns:axml="http://futurs.inria.fr/gemo/axml/">
 <axml:sc axml:id="temperature">
  <axml:return>
    <axml:append/>
  </axml:return>
  <axml:ws-soap endpoint="http://webservices.daehosting.com/services/
                                              TemperatureConversions.wso">
   <CelciusToFahrenheit xmlns="http://webservices.daehosting.com/temperature">
    <nCelcius>10</nCelcius>
   </CelciusToFahrenheit>
  </axml:ws-soap>
 </axml:sc>
</example>
```

The results after activation of the axml document:

```
<example xmlns:axml="http://futurs.inria.fr/gemo/axml/">
 <axml:sc axml:id="temperature"  activated="2008-04-08T11:27:01.656+02:00">
  <axml:activation status="TERMINATED"/>
  <axml:return>
   <axml:append/>
  </axml:return>
  <axml:ws-soap endpoint="http://webservices.daehosting.com/services/
                                              TemperatureConversions.wso">
   <CelciusToFahrenheit xmlns="http://webservices.daehosting.com/temperature">
    <nCelcius>10</nCelcius>
   </CelciusToFahrenheit>
  </axml:ws-soap>
 </axml:sc>
 <m:CelciusToFahrenheitResult axml:origin="temperature"
  axml:timestamp="2008-04-08T11:27:03.312+02:00">
  50
 </m:CelciusToFahrenheitResult>
```

```
</example>
```

The result of the service call is appended to the document, as a sibling of the $< sc/ >$ node. The status of activation is marked as TERMINATED and some extra axml-specific attributes are added to the document. More details can be found in the following sections.

1.1.2 A document with a service call to an axml service

The document defines a service call to a service that executes a xQuery over an xml document. When the document is activated by the ActiveXML system, the response is added to the axml document.

Definition of an axml document (a xml document that follows the axml syntax):

```
<example xmlns:axml="http://futurs.inria.fr/gemo/axml/">
 <axml:sc axml:id="simple">
  <axml:return>
    <axml:append/>
   </axml:return>
   <axml:ws-soap endpoint="http://localhost:6969/MyPeer/services/GenericQueryService">
    <q:executeGenericQuery xmlns:q="http://futurs.inria.fr/gemo/axml/service/Query">
     <q:declaration>
      for $x in doc('/db/bookstore3.xml')/*/* return $x
     </q:declaration>
    </q:executeGenericQuery>
   </axml:ws-soap>
 </axml:sc>
</example>
```

The results after activation of the axml document:

```
<example xmlns:axml="http://futurs.inria.fr/gemo/axml/">
 <axml:sc axml:id="simple"  activated="2008-04-08T11:30:01.656+02:00">
  <axml:activation status="TERMINATED"/>
   <axml:return>
    <axml:append/>
   </axml:return>
   <axml:ws-soap endpoint="http://localhost:6969/MyPeer/services/GenericQueryService">
    <q:executeGenericQuery xmlns:q="http://futurs.inria.fr/gemo/axml/service/Query">
     <q:declaration>
      for $x in doc('/db/bookstore3.xml')/*/* return $x
     </q:declaration>
    </q:executeGenericQuery>
```

```
   </axml:ws-soap>
 </axml:sc>
 <book axml:origin="simple" axml:timestamp="2008-04-08T12:40:42.765+02:00"
                                                          category="COOKING3">
  <title lang="en">Everyday Italian2</title>
  <author>Giada De Laurentiis3</author>
 </book>
 <book axml:origin="simple" axml:timestamp="2008-04-08T12:40:42.765+02:00"
                                                          category="CHILDREN3">
   <title lang="en">Harry Potter2</title>
   <author>J K. Rowling2</author>
 </book>
</example>
```

2 Architecture

The ActiveXML (shortly axml) system is a collection of ActiveXML peers. Each peer has a repository of axml documents. A peer knows how to activate axml documents, meaning how to call the services declared inside these documents and how to receive the results. It also provides axml services like algebra, materialization, query, continuous etc.

2.1 Software components

There are 4 software components that make up an ActiveXML peer (see figure 1):

- Web server. We use Tomcat 5.5
- XML database. Currently we use eXist
- Axis2 Web service engine
- ActiveXML service calls execution engine

Axis2 and ActiveXML make up a Web application that is deployed in the Tomcat 5.5 Web server. Then the Web application is configured to connect to eXist database where each peer has its own repository of documents (a separate collection).
Actually the components can be organized in several ways. In the simplest (and most expensive) case everything is in the Web container (Tomcat) as Web applications. Hence the peer has its own Web server and its own database instance. It is, nevertheless, possible that several peers share one Web server and the same database instance. Possible configurations are depicted bellow (see how to create new peers). As the picture shows there can be several peers residing in one Web server while the database can be inside or outside the

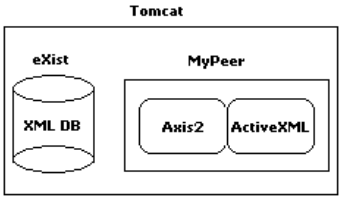

Figure 1: Software components

Web server. The peer-to-peer network is a network of such possible configurations where, as already mentioned, the peers communicate only by the means of Web services.

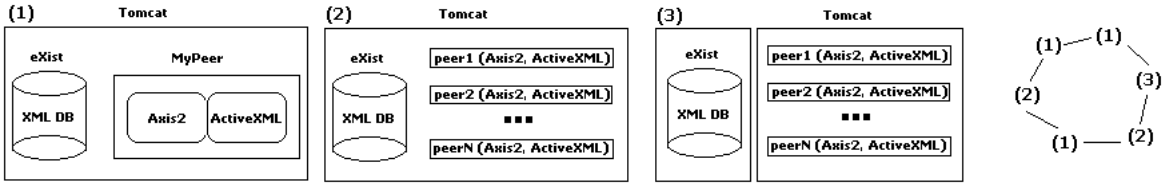

Figure 2: Components configurations

2.2 Peer architecture

An axml peer could be divided in three parts: client, engine, server (see figure 3). As a client, it provides a web interface for accessing, evaluating and optimizing axml documents, and a SOAP client, which is an applet for visualizing the SOAP messages that are passing in and out of the axml peer. As a server, it provides predefined web services, like query, algebra and materialization services (see more details below). As an engine, it contains a layer for database access, a document manager for the management of axml documents, and materializers that know how to evaluate axml documents. These documents are stored into a XML database.

As mentioned above, AXML system provides some predefined web services: algebra, query, continuous and materialization. The three algebra services SendOperator, ReceiveOperator and NewNodeOperator are used by optimax, a module that optimizes and evaluates axml documents. The query service (GenericQueryService) is used to execute queries over documents from database (one-shot or continuous execution). The continuous service is a wrapper for calling web services with intensional XML parameters. The materialization service provides an entry point for the activation of active xml documents that are stored into the repository.

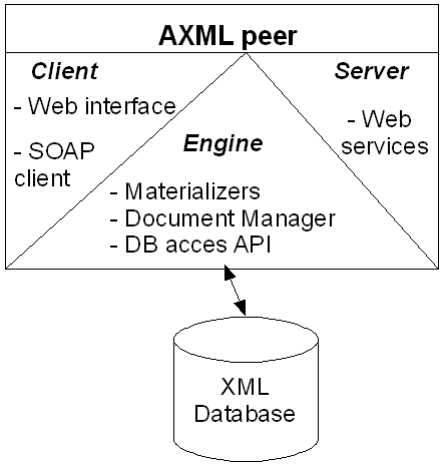

Figure 3: AXML peer architecture

2.2.1 Streams in ActiveXML system

A stream is a channel of data between a service call and the Web service it is calling. The important thing is that data keeps arriving to the service call continuously (and asynchronously) until the end of stream. The procedure is as follows (see figure 4):

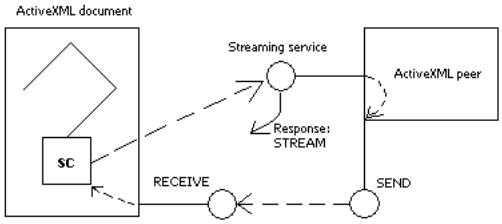

Figure 4: Streams

1. A request comes to materialize some service call
2. The service call invokes the Web service and together with the message attaches a header with information about itself
3. The Web service responds with a message marked as STREAM (if it has what to stream and if it recognizes the header)
4. The service call stays active if the Web service response was marked as STREAM

5. Then it depends on the Web service when it sends data to the requesting service call.

In the picture above SEND is shown as a separate service. Still, the streaming service can directly call RECEIVE. It is then said as behaving like SEND. So to put it simple, a stream is a channel between SEND and RECEIVE. A streaming service is responsible for SEND whereas a service call is the RECEIVE'er.

Worth noticing that a stream can be seen as a communication protocol, nothing else. Current implementation works as described (using markers). However, other implementations or extensions are appropriate. For example, it looks natural to use WS-Addressing for addressing the service call. Also a mechanism of handling sessions can be fruitfully utilized, because a service call is simply a client for a specific Web service.

Actually the concept of streams also captures non-streaming communication between service calls and Web services. In this case the stream consists of one item that is sent directly as a response to the service call and is implicitly marked with END_OF_STREAM. Therefore, materializing a service call ends up in 1) service call termination if the Web service responds with the END_OF_STREAM (it is implicitly assumed), or 2) it stays active if the Web service responds with the STREAM marker and terminates only when its RECEIVE detects the END_OF_STREAM. Note that in the next version of the system, the length of stream should be added (see branch ActiveXML2_2 on SVN).

2.3 Interaction between peers

The figure 2.3 shows the peer-to-peer network of ActiveXML peers. The peers are extensible by plugged-in Web services. Note that Web services are only the interface to the underlying software, which can vary from a very complex system to a simple function. The plugged-in systems are supposed to make use of ActiveXML. This means that the exposed Web services:

- can stream data. For instance, it can send a message on some event
- can try to materialize some service calls before responding to produce more results ("lazy") (ActiveXML version 1.0)
- can transform the requested query into an optimized one in order to bring results from the whole peer-to-peer network quicker ("optimax")
- can evaluate an active document in a special way
- can serve as an integrator of results from several Web services, i.e. by specifying several service calls inside an active document and doing a query over it

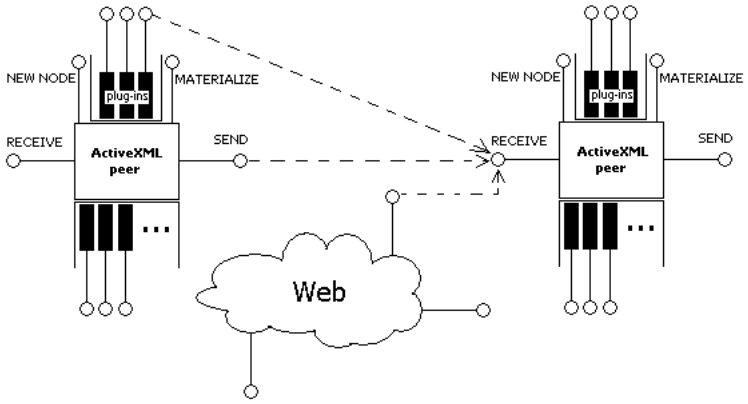

Figure 5: Peers interaction

In essence using ActiveXML means 1) using special syntax to declaratively specify calls to Web services and 2) using the engine for materializing those service calls. Hence ideally service consumers would write active documents and materialize service calls inside them. While service providers would develop systems and expose them as Web services potentially internally also using ActiveXML.

Besides the peer-to-peer network, XML and Web services allow ActiveXML to enter the Web in general. As it is shown in the picture RECEIVE can get data from everywhere, that is, an active document can contain service calls to any Web service, be it on another peer or somewhere on the Web. Similarly it is not forbidden for external systems to call Web services inside the peer-to-peer network if it makes sense to them (because the returned data may be active). Moreover, since streaming is just a communication protocol nothing stops Web services outside the peer-to-peer network to stream data back to a requesting service call through its RECEIVE.

3 User guide

In this section we describe the ActiveXML system from a user point of view. We explain how an user could write axml documents and activate them.

3.1 Usage of ActiveXML system

In order to use the AXML system, the user has to download the latest release from ObjectWeb (http://forge.objectweb.org/project/showfiles.php?group_id=140). If the user wants to develop new functionalities for ActiveXML, please refer to the section 4, Development Guide.

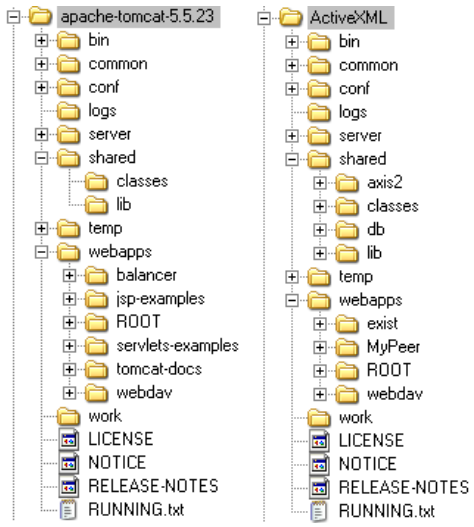

Figure 6: Changes done to Tomcat distribution

The distribution is created in such a way that it includes everything that is needed for the start (except Java Runtime Environment), namely the 4 components: Tomcat, eXist, Axis2 and ActiveXML (see the components representation).

First thing to know is that the distribution is simply the Tomcat Web server with several Web applications included, specifically: eXist and some ActiveXML peers: MyPeer, peer1, peer2, peer3. The peers are actually integrated into Axis2 Web application in order to make use of the Web service repository management.

The changes done to Tomcat distribution are the following:

- the root folder was renamed into ActiveXML
- unneeded Web applications under ***webapps/*** were removed, added 'exist' and 'MyPeer'
- some libraries were put to the shared classpath under ***shared/***. The most important ones are Axis2 libraries and database libraries

The following configuration changes were performed for Tomcat:

- in ***conf/catalina.properties*** paths to new shared folders were added (shared/db/*.jar, shared/axis2/*.jar)
- in ***conf/server.xml*** ports were changed into 6969 (for HTTP requests) and 6960 (for Web server shutdown request)

For more details on changing a port, starting, stopping or anyhow configuring the Web server please follow Tomcat's documentation (http://tomcat.apache.org/).

3.1.1 How to start the application

In order to run the distribution:

- JAVA_HOME (or JRE_HOME) environment variable must be set. It must point to the JDK or JRE root directory. Usually for this purpose one can use set command line tool. This point implies that you have a JVM on your computer.
- use bin/startup.bat or bin/startup.sh to start the Web server
- use bin/shutdown.bat or bin/shutdown.sh to stop the Web server

NOTES: on Linux be sure to make bin/*.sh files executable and if you have another instance of Tomcat running on your machine, you will need additional steps, for example, setting the CATALINA_BASE variable.

3.1.2 Web Application: MyPeer [ActiveXML/webapps/MyPeer]

As it was mentioned ActiveXML is merged with Axis2 Web application. This choice is made in order to utilize the internals of Axis2 when dealing with Web services, which is crucial for ActiveXML. For deeper information please read Axis2 documentation (http://ws.apache.org/axis2/).

There are some important things to know about Axis2 in ActiveXML distribution:

1. *MyPeer* is an extension of axis2.war
2. It has a Web services repository by default configured under WEB-INF/
   - *conf/axis2.xml*, the repository configuration file. Here one can specify in/out phases in order to engage modules for a message flow, also one can configure transports, ports and everything else related to Web service mechanics
   - *modules/*.mar*, a module is a collection of handlers that are used to process an incoming/outgoing message. The modules are pluggable and engageable during a specified phase of a message flow. *.mar (module archive) is simply a zipped collection of resources, like class files, libraries and module.xml.

     There are two modules that belong to ActiveXML:
     - *ServiceCallWrapper.mar*, responsible for creating a valid message for the Web service referred in the service call. For instance, a user does not write a SOAP messages, it is left to this module. It also attaches information about the service call in headers

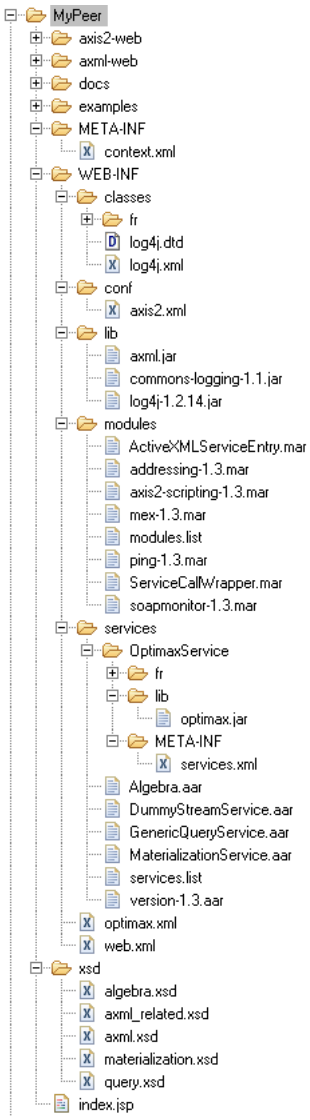

Figure 7: Sources organization of an axml peer

  - *ActiveXMLServiceEntry.mar*, makes the ActiveXML context available for Web services. Without it, neither the engine nor the repository would be easily accessible.
- *services/*.aar*, *.aar (axis archive) is a zipped collection of resources that consist a Web service. It can also be kept unzipped (the same holds for modules) like it is in case of services/OptimaxService/. So a Web service is indeed an independent piece of software, that has its own implementation, libraries and configuration, which enables pluggable architecture of ActiveXML

3. it has a user interface for managing Web services under axis2-web/; normally it will be accessible at http://localhost:6969/MyPeer/axis2-web (login: admin/axis2). Consult it for service endpoint, WSDL, XSD and other information about web services of the peer

4. all the Axis2 related libraries (from WEB-INF/lib/) are shared

The important files related only to ActiveXML are:

- *WEB-INF/lib/axml.jar*, this is the engine
- *WEB-INF/web.xml*, this is the standard Web application configuration file, it mostly has specification for servlets (both Axis2 and ActiveXML)
- *WEB-INF/classes/log4j.xml* is the configuration of the logging system. The logs can be found under ActiveXML/logs/ together with standard Tomcat logs
- *META-INF/context.xml* contains context specific configuration. Currently it only has the connection specification to the XML database. This connection is then accessed using JNDI (the procedure is kept to be very similar to connecting to SQL data sources). Note that other Web containers do not necessarily have this file, so the JNDI resource must be specified elsewhere
- *index.jsp*, axml-web/ contains the GUI for the user that wants to evaluate a document, browse the database, etc. However, it is very modest. This is due to the fact that it is still not clear what the user interface should look like. It is clear that one thing is the administration tools, still there also should be user tools. One of possible solutions could resemble the one proposed for Web mash-ups.
- *docs/* contains this documentation with some additional resources
- *examples/* have a collection of example active documents that are a good start to playing with the peer

- *xsd/*.xsd* are the XML Schema files that declare the types known to ActiveXML engine. For instance, axml.xsd declares the grammar for service calls. The schemas can be used for validating active documents. JAXB is used for doing marshalling from XML to Java and vice versa

For *context.xml*, pay attention when using another Web container. Use logs for debugging.

3.1.3 Shared Libraries [ActiveXML/shared]

Beside MyPeer there can be more peers configured to live in the same Web container (and possibly using the same database instance). Therefore it makes sense to share the heaviest libraries among all of them.

- ActiveXML/shared/axis2/, ActiveXML/shared/classes/ contain everything that initially could be found under axis2.war/WEB-INF/lib/ and axis2.war/WEB-INF/classes/ (17M when unzipped)
- ActiveXML/shared/db/ contains libraries that enable uniform access to an XML database, i.e.: database access API and the implementations for a certain providers
- ActiveXML/shared/lib/ keeps anything related but not compulsory to ActiveXML engine. For instance, it can contain libraries related to GUI or for some Web services

3.1.4 Creating a New Peer

A peer consists of a collection of (active) documents and a set of Web services. In the installation and configuration section it was shown, how the pieces are put together to have MyPeer up and running. If one wants to add more peers to his/her Web container the procedure is the following:

1. make sure the Web server is stopped (use ActiveXML/bin/shutdown.*)
2. copy MyPeer inside ActiveXML/webapps with a given name (e.g. AnotherPeer)
3. change the following resources:
   - ActiveXML/MyPeer/META-INF/context.xml, it was mentioned that here connection to the database is configured. In case of eXist it looks as follows:

     ```
     <Context>
         <Resource name="axml/repository"
                   factory="fr.inria.gemo.axml.db.DataSourceFactory"
                   type="fr.inria.gemo.axml.db.IDataSource"
     ```

```
                driverClassName="fr.inria.gemo.axml.db.exist.DataSourceImpl"
                url="xmldb:exist://localhost:6969/exist/xmlrpc/db/MyPeer"
                userName="admin" password="" />
</Context>
```

It is enough to change the url of peer's repository into: url="xmldb:exist://localhost:6969/exist/xmlrpc/db/AnotherPeer". What actually happens is that MyPeer and AnotherPeer will have their own document repositories (collections) on the same instance of eXist (see the picture)

- ActiveXML/MyPeer/WEB-INF/classes/log4j.xml, replace all occurances of MyPeer into AnotherPeer in order to get separate log files for AnotherPeer
- ActiveXML/MyPeer/WEB-INF/web.xml, replace $< display{-}name > MyPeer < /display{-}name >$ into $< display{-}name > AnotherPeer < /display{-}name >$ because currently it is the only peer identification method. Usually in a peer-to-peer environment peers have unique ids.

4. start the Web server (use ActiveXML/bin/startup.*). Now AnotherPeer should be accessible at http://localhost:6969/AnotherPeer

After creation of AnotherPeer it lives totally in its own context: it has its own repository of documents, and its own repository of Web services.

3.2 Web Interface

The distribution contains also a web interface for the manipulation of axml documents. Once the distribution is running, the peers interfaces are available at the following address: http://localhost:6969/MyPeer. Change the hostname depending on the local settings. If you want to access the other peers, change *MyPeer* with *peer1* etc.

The interface contains a link to eXist interface; use this interface to manage documents from the repository. You can browse eXist database at http://localhost:6969/exist/admin/admin.xql (login: admin/exist).

The dropdown lists from AXML interface contain the list of axml documents installed into eXist. The release comes with some examples of axml documents. In order to install them, one could use eXist interface or ANT scripts (see a template on SVN). There are some constraints regarding these files: algebra1.xml and algebra2.xml should be activated; for *optimax* use planX.xml and strategyX.xml (replace X with the number of file) files for the document and respectively the strategy. Each peer has a predefined collection into eXist (e.g. MyPeer, peer1 etc.) These examples should be installed into these collections in order to be retrieved by the interface and activated by the system. These examples use

some additional files bookstore*.xml that need to be installed into db/ collection.
The interface provides the following functionalities:

- *Reload page*: for the moment, the reload is not done automatically, so one should do it after a document is released or optimized.
- *Suspend document*: when a document is activated, its evaluation could be suspended if something went wrong. If you suspend, do not forget to reinstall the document before a new activation, due to possible remaining ACTIVATED tags.
- *Activate service call*: it activates only one service call inside the document. Its descendants are not evaluated.
- *Evaluate node*: it activates a node inside an axml document, including its descendants.
- *Evaluate document*: it evaluates an entire axml document.
- *Optimize document and evaluate*. It returns the path into database to the optimized document. The optimized document is evaluated and the results are appended to the original document.
- *Optimize document*. It returns the path into database to the optimized document. The optimized document is not evaluated, so no results are appended to the original document.
- *Activation history*: shows the activation history of the documents. It's pretty rudimentary, thus it contains no order or timestamps for the moment. If a document is activated, it also displays the status of service calls.

3.3 Activation of an axml document

In order to activate an axml document, the user has to have the distribution running and the samples installed into eXist repository.

If you want to activate a service call (*sc*) inside a document, you call *activate*. If you want to evaluate an entire document (all service calls are activated), you call *evaluate*. In order to see the results of the evaluation, check the evaluated document in eXist (refresh or reload the documents from eXist). Usually the evaluation takes some time, and data is appended to the document continuously, so each time one refreshes the activexml document, one should see new results for the service calls.

An evaluation is terminated when all *sc* nodes of the axml document have the status TERMINATED.

3.4 SOAP Alerter

In order to intercept the SOAP messages that are passing through a peer, the system contains an Axis2 module that publishes these messages. An applet application connects to a predefined port (defined into web.xml for each peer) and displays the messages. The applet is accessible at http://localhost:6969/MyPeer/SOAPAlerter.

3.5 AXML Components

The ActiveXML system contains some components that are important to be explained for the users: service, materializers and document manager.

3.5.1 AXML services

There are several Web services that are inherent from ActiveXML (shown as buttons in the picture):

- Three algebra Web services, that enable distributed data management. Algebra Operators are used by optimAX.

    1. *ReceiveOperator* [shown as RECEIVE], through this Web service results come to a service call. Roughly speaking, when a Web service is invoked from within an active document, the response is redirected to this Web service. It is responsible for preparing the incoming result to be correctly merged into the document. For example, it assigns the timestamp and origin attributes, ensures that the ids of incoming and existing service calls don't overlap, etc.
    2. *SendOperator* [shown as SEND], this Web service is sending data to an address specified. More specifically, it calls the respective ReceiveOperator and passes data to it. Normally this Web service is not used, as the ReceiveOperator can be called directly, therefore any Web service that would call RECEIVE can be titled as SEND. Although SendOperator usually is not invoked, service calls to this Web service are indispensable since they bare special semantics
    3. *NewNodeOperator* [shown as NEW NODE], this is a utility Web service that allows to install (active) data in some peer's repository. It can install new document or append data as children for a specific identifiable node. The semantics of this Web service impose that the newly installed node is evaluated

- *MaterializationService* [shown as MATERIALIZE] has the following operations:

    1. *evaluate*, will materialize the document in a depth first manner. This means that when a service call has other service calls as parameters, the parameters

will be materialized earlier. The ordering of execution can be explicitly overridden, which makes the active document look like a graph

2. *evaluateNode*, behaves much like evaluate but materializes the document starting at a specified node
3. *activate*, will activate a specified service call and will follow all the explicitly defined constraints. This operation will not follow a default (depth first) execution order

- *GenericQueryService*, this Web service is a relatively simple implementation of stream processing engine. As an input it gets a query declaration and parameter streams. It can also be used to answer XQuery queries over the database; the method *applyQuery* applies a query that is defined into another axml document; it could be used to push data into another document. For example, document A has a query that updates the document with data coming from a source. Document B calls GQS/applyQuery by replacing the parameters of the query with its own stream of data. Therefore data of document B are pushed to document A. See more example of axml files on SVN, path: svn/ActiveXML2Ext/tests/.../aqTests/*.xml.

- *DummyStreamService* is useful for testing. When called, it streams back result of a specified query

- *OptimaxService*, distributed query optimizer. Given an active document the Web service transforms it into a distributed plan. The documents that have service calls to GenericQueryService are candidates for optimization with this Web service

- *ManagementService* provides the operation *notifyFromTrigger* and it is used for notification tests.

- *ContinuousService* calls a web service (axml or non-axml) in a continuous way. It opens a channel for each triple (peerId, docId, nodeId) and each time a new parameter arrives, calls the specified web service and returns the result. It should know the EOS protocol (v2.2 length of stream). See more examples of axml documents on SVN, path: svn/ActiveXML2Ext/.../functTest/ctemp.xml NOTE: For the moment, the called ws should be of type request-reply, not continuous; the service has one parameter.

3.5.2 AXML materializers

A service call is materialized (call Web service, consume result) by materializers. In an active document, one can set the materializer class for a service call (sc). This class is instantiated when creating a service call object. All the materializer related classes are in fr.inria.gemo.axml.model.sc.materialization:

- *DefaultOutOnlyMaterializer* is used for simple service calls that do not have $<$ *return* $>$, or have $<$ *return* $><$ *void* $><$ */return* $>$
- *DefaultOutInMaterializer* is used for simple service calls that have merge function specified inside $<$ *return* $>$.
- *MaterializerForRECEIVE* is assigned for sc that are RECEIVE. This materializer does not call any Web service but simply makes the service call active
- *MaterializerForSEND* is assigned to SEND service calls. According to SEND semantics it sends everything underneath it until the end of stream (EOS). This means, that it is listening to the service calls bellow it and on new result, this materializer calls the respective RECEIVE Web service.
- *MaterializerForContinuousQuery* behaves similarly to SEND, i.e. it is watching the service calls underneath it, but instead of calling a RECEIVE on the server, it calls the respective *GenericQueryService*.
- *MaterializerForContinuousSC*. It calls continuously a web service, each time a new parameter value arrives. It uses continuous_call.xsd
- *MaterializerForSC* eliminates the sc declaration from active parameters of a service call and calls the parent web service (one-shot call). See more examples of axml files on SVN, path: svn/ActiveXML2Ext/tests/... /functTest/ctemp.xml

3.5.3 Document Manager

The document management layer is responsible for marking service calls as active/terminated. It only keeps references to the interesting parts of active documents using XPath and XQuery.

3.6 Continuous Service Calls

A service is called with parameters coming from a stream. A stream might represent the results of another service call. ReceiveOperator is called to stream back the results.

AXML example: continuous query (see the xml document below and the representation in figure 8).

```
<example xmlns:axml="http://futurs.inria.fr/gemo/axml/">
 <axml:sc axml:id="continuousFILTER">
  <axml:return materializer=
  "fr.inria.gemo.axml.model.sc.materialization.MaterializerForContinuousQUERY">
     <axml:append/>
```

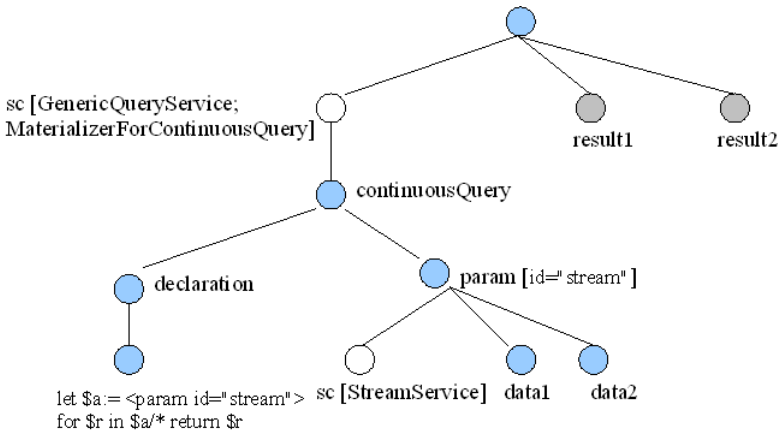

Figure 8: GQS declaration

```
  </axml:return>
  <axml:ws-soap endpoint="http://localhost:6969/MyPeer/services/GenericQueryService">
   <q:continuousQuery xmlns:q="http://futurs.inria.fr/gemo/axml/service/Query">
    <q:declaration>
    let $book := <q:param name="book"/>
     for $b in $book/*
     return if(xs:integer($b/year) gt 2003)
            then $b else ()?
    </q:declaration>
    <q:param name="book">
     <axml:sc axml:id="bookStreamer">
      ...
     </axml:sc>
     <STREAM_DATA/>
    </q:param>
   </q:continuousQuery>
  </axml:ws-soap>
 </axml:sc>
</example>
```

3.7 ActiveXML syntax

ActiveXML documents (or simply active documents) are XML documents with embedded service calls. Embedding a service call is just a matter of following the syntax developed for service calls so that the ActiveXML documents engine would be able to recognize them

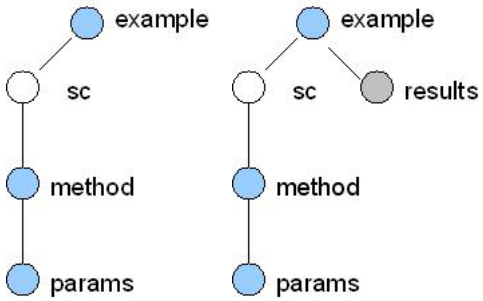

Figure 9: Evolution of an activated axml document

and apply semantics. Without an engine active documents are normal XML documents. The syntax for service calls is given in the XML schema document *axml.xsd* (see all schema files on SVN, path ActiveXML2/resources).

For example, a simple active document looks like this **before** activation (see figure 12):

```
<example xmlns:axml="http://futurs.inria.fr/gemo/axml/">
    <axml:sc axml:id="version">
        <axml:return>
            <axml:append />
        </axml:return>
        <axml:ws-soap endpoint="http://localhost:6969/MyPeer/services/Version">
            <v:getVersion xmlns:v="unknown" />
        </axml:ws-soap>
    </axml:sc>
</example>
```

As shown in the example, the *sc* element belongs to a special namespace and it has the following features:

- *id* that represents current nodeID (current peerID and current docID are the peer and document where this example lies in the repository) of the service call node. The id must be unique within the document, therefore incoming data may require fixing to conform to this requirement
- *return* statement that specifies what to do with results. In the example it is shown that the results should be appended. One can also specify other things, e.g., for how long the results are valid.
- the specification of how to reach the Web service. In the example it is shown that it will be called using a SOAP message with the provided payload (the root normally

corresponds to the operation name). Besides one can call a Web service REST style, which roughly corresponds to submitting an XForm via GET or POST. Note that the payload can contain active data.

Beside the listed features, service call can be setup to execute after other service call(s).

The active document *after* activation (see figure 12):

```
<example xmlns:axml="http://futurs.inria.fr/gemo/axml/">
    <axml:sc axml:id="version" activated="2007-10-22T19:17:33.921+02:00">
        <axml:activation status="TERMINATED"/>
            <axml:return>
                <axml:append/>
            </axml:return>
        <axml:ws-soap endpoint="http://localhost:6969/MyPeer/services/Version">
            <v:getVersion xmlns:v="unknown"/>
        </axml:ws-soap>
    </axml:sc>
    <ns:return xmlns:ns="http://axisversion.sample" axml:origin="version"
        axml:timestamp="2007-10-22T19:17:35.484+02:00">
        Hello I am Axis2 version service , My version is 1.3
    </ns:return>
</example>
```

After materializing the service call in the example the document changed in the following way:

- a result arrived. It was marked with the origin and timestamp. Note that the origin refers to the id of service call and this reference should not be broken when data is traveling around the network.
- the service call was marked as TERMINATED and was assigned the activation timestamp

For information on where to put a document, how to activate a service call, evaluate the whole document, etc. please see the demos (http://forge.objectweb.org/docman/?group_id=140). Worth mentioning that it is done by calling peer's Web services.

The project contains XML schemas for the syntax of axml documents, the definition of axml services, the communication headers for SOAP messages used for asynchronous calls. A sc is uniquely identified by (peerID, docID, nodeID). The activation order is by default depth first, and additional tags could be added: afterActivated, afterTerminated.

The activation status of a service call has multiple possible value:

- ENABLED is always set by default, you don't have to specify it.

- ACTIVATED is set when a sc is activated
- TERMINATED is always when a sc has finished.
- FAILED is set when the service call execution fails, for instance, when the materializer catches an exception while calling the Web service.
- DISABLED is used to disable a service call, it means that this sc is not activated. If a disabled sc breaks the activation order (e.g. child of a sc or there are nodes waiting for its activation, meaning they have the tags $afterActivated$ or $afterTerminated$), the document is not evaluated anymore and an error is raised. Its descendants are activated.

AXML system adds axml attributes to the responses added to the active document, e.g. axml:timestamp, axml:origin etc. In version 2.1, these attributes are removed before the results are sent as parameters to another service calls.

In order to set the way the system processed the results, three options are available: Append, Replace or No-merge.

4 Developer guide

This section is meant for developers that want to extend the functionality of ActiveXML system. It contains a description of the source code organization, a guide to building the project and to testing the code, and a description of the system from development point of view.

4.1 ActiveXML project setup

4.1.1 Brief description

In order to setup the Activexml project, checkout the source code from SVN (http://forge.objectweb.org/plugins/scmsvn/index.php?group_id=140). The sources are organized as an Eclipse project. Pay attention to keep the same project settings, because the ant build files copy the binary files (.class files) from each *bin* directory of the source folders (e.g. the components/web source folder has as output folder /components/web/bin and excludes bin/;resources/;resources/services/).

After the setup of the eclipse project, one should rename the file *build.properties.tmpl* to *build.properties* and replace the path to the local *libs* directory. All the libraries specified in this directory should be downloaded (see subsection 4.1.2).

In order to build the project, use *build.xml* file (see subsection 4.1.3). In order to create a distribution, use distribution/build.xml. The distribution is created into $distribution/ActiveXML$. For developers, the additional *buildDev.xml* and *buildDev.*

properties files could be useful for building the project, deploying it, starting the application (tomcat), installing samples into eXist, create a release etc. For *svn* : *ignore*, load the values of *svn:ignore* property from *svnignore-list.txt*. Right-click on the project -> Team -> Show Properties -> double click on svn:ignore or add property svn:ignore -> Use a file -> Choose *svnignore-list.txt*.

4.1.2 How to setup the project

ActiveXML depends on the following libraries:

- AXIS2 - contains all the .jar files from axis2.war/WEB-INF/lib/. ActiveXML uses Axis2 internally to provide web services
- EXIST_12 - contains three .jar files (exist.jar, xmldb.jar, xmlrpc-1.2-patched.jar). These contain the classes needed to implement a driver for the active documents repository using eXist XML database
- GWT_14 - contains gwt-dev-windows.jar (for Windows), gwt-user.jar. These libraries are used for creating AJAX Web application (servlets and GUI) using Google Web Toolkit.
- OPTIMAX - optimax.jar. Actually the dependency should be other way, i.e. Optimax should depend on ActiveXML because it is a plug-in.

How to setup the libraries?

- Unzip axis2-1.3.zip, exits-12.zip and gwt-windows-1.4.60.zip or gwt-linux-1.4.60.zip.
- Modify the *build.properties* with the correct paths.
- Add the corresponding jars to AXIS2, EXIST_12 and GWT_14 libraries. ActiveXML2 -> Properties -> Libraries -> Select User library (e.g. EXIST_12) -> Add external jars ->¿ exist/WEB-INF/lib/exist.jar, xmldb.jar etc.

The code (end the whole software) can be divided into the following parts:

- *core* - it is the implementation of the ActiveXML engine. It does not (and should not) depend on other parts.
  - *src* contains all the interfaces that represent the general concepts of ActiveXML, such as a service call, materialization, merge function, result, etc. Changing an interface means possibly breaking the contract causing compatibility issues.

  - *impl* is an implementation of what's inside *src*. The most important aspect of the implementation is a heavy usage of XQuery: in order to avoid loading all the documents into memory tiny skeleton objects are created; when data is needed a query is posted to the repository.
  - *peer* realizes the concept of a peer that has documents and services
  - *jaxb* contains the generated code with some meta information. The code is generated out of ActiveXML related .xsd files using ActiveXML/binding.xml (it is an Ant file with tasks).

- *components* - these are loosely coupled components that could have different implementations.
  - components/web component is the Web application that helps to interact with the peer, in other words this is a GUI
  - components/db component contains the DB access API and eXist driver

- *services* - are the ActiveXML Web services (plug-ins). They are deployed into Axis2 repository

- *modules* - are the collections of handlers (in Axis2 terms)
  - *modules/sc_wrapper* is responsible for dealing with the outgoing message when a service call is activated. Currently the main purpose of it is to enrich the SOAP message with information headers about which service call is being called
  - *modules/svc_entry* deals with an incoming message. Its main purpose is to initialize the ActiveXML context so that plug-ins (Web services) are able to reach information about documents and services available on the peer. It gets the axml context from servlet context and attaches it to the MessageContext to be accessible into ws method.
  - *modules/alerter* monitors the soap messages and publishes them on a predefined port. For each peer, an applet listens on this port and displays the messages. (e.g. http:// localhost:6969/peer1/SOAPAlerter). For each peer the port is defined into build.properties, variable peerX.alerter.port.

Every listed item have associated build.xml in a respective folder. There are separate output folders (where the .class files go) for every component, service, module, and the core. So development goes like this:

1. Decide where the source should go. In most of the cases a new component or a service will need to be developed

2. Create a folder in respective place and make it as source (Eclipse: [right click]->Build Path...->Use as Source Folder)

3. Configure the output folder for the new sources (Eclipse: [right click]->Build Path...->Configure Output Folder). Normally it should be a *bin* folder inside

4. Update the respective build.xml to get a .jar, .aar or .mar out of the sources (see how to build in subsection 4.1.3).

Inside ActiveXML/resources there are all the .xsd files that define syntax for different things. The schemas are quite documented so they will not be covered here:

- axml.xsd defines syntax for service calls
- axml_related.xsd defines some types that are used by the core of the engine. For instance communication headers for SOAP messages
- algebra.xsd defines syntax of SEND, RECEIVE, NEWNODE algebra services
- materialization.xsd defines syntax for materialization services, e.g. activation of a service call, evaluation of a document
- query.xsd defines syntax for query services, e.g. generic query, declarative query
- continuous_call.xsd defines syntax for ContinuousService

Since the most difficult is axml.xsd it is advisable to use this schema when creating active documents from the scratch. For instance, Eclipse shows the auto-complete and documentation for elements and types when the the .xsd is included using xsi:schemaLocation attribute (see figure 10). Any other XML editing tool should also provide that.

4.1.3 Building and Creating a Distribution

When everything compiles in Eclipse still specific build procedure must be held in order to properly deploy ActiveXML. To understand the building process it is enough to go through all the build.xml files in the project. Before doing any builds or bindings ActiveXML/build.properties must be updated as explained in subsection 4.1.1. There is a description of the main build file.

- ActiveXML/build.xml used for building everything into .jar, .aar, .mar and web components. By default the resulting resources are put under ActiveXML/build/. This build file is actually calling targets from several other build files:
  - ActiveXML/services/build.xml used for creating Axis archives of the services that are being developed

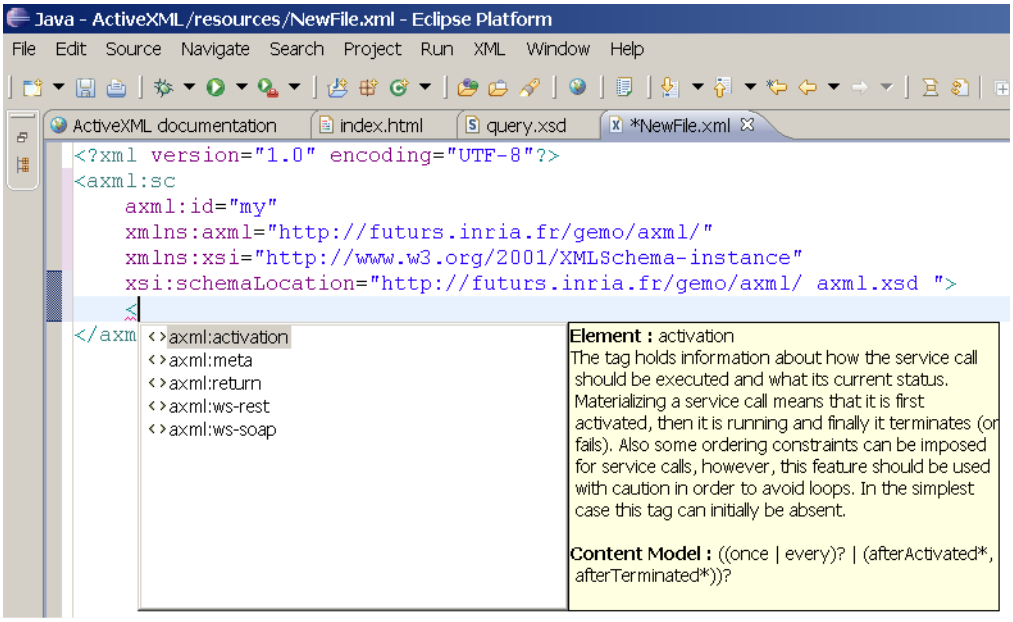

Figure 10: XML schema auto-completion in Eclipse

  - ActiveXML/modules/build.xml used for creating Axis module archives of the modules (collections of handlers) that are being developed
  - ActiveXML/components/build.xml used for building other independent components. Moreover this build file creates actual peer webapps. It puts all the .jar, .aar, .mar files into right places of a webapp and this is how MyPeer is created

- ActiveXML/distribution/build.xml used for creating the whole distribution that can be zipped and shipped for the end users. It transforms Tomcat into ActiveXML distribution by including certain built resources from ActiveXML/build/ and eXist into right places (see installation and configuration for details)

- ActiveXML/buildDev.xml used by developers.

4.2 Main components

The main components of the source code, as depicted in figure 11, are *services*, *components*, *modules*, *core*. The *services* contain the axml services. The *components* contain the code for connection to the database, web for web interface(GWT implementation). The *modules* are plugged-in components of axis2, and are executed for each service request

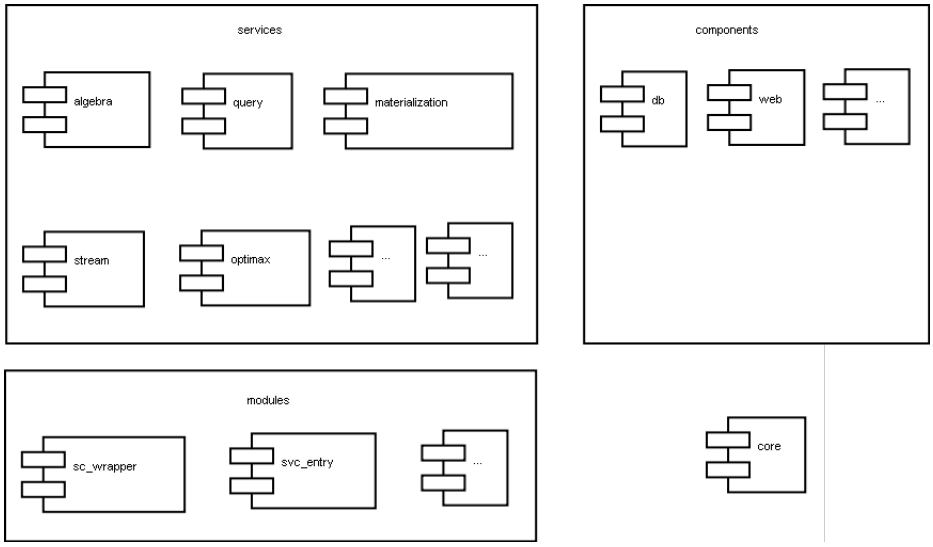

Figure 11: Main code components

reply (settings found in axis2.xml). The *core* represents the main functionality of axml engine.

4.3 Materializers

AXML materializers call different services and process the results. They might need to be specified into axml documents depending on called services. The materializers are described in user guide subsection 11. DefaultOutOnlyMaterializer / DefaultOutInMaterializer are not visible to the user. MaterializerForContinuousQuery calls GenericQueryService with parameters coming as results of other service calls (streams). MaterializerForContinuousSC calls continuously a web service, each time a new parameter value arrives. MaterializerForSC eliminates the sc declaration from active parameters of a service call and calls the parent web service. It could be replace by a new axis2 module that eliminates the service calls declaration from a request. MaterializerForRECEIVE / MaterializerForSEND are used by optimAX.

MaterializerUsingConstraints and DeepTreeMaterializerUsingConstraints are helpers for MaterializationService. These materializers are not designed for service calls.

4.4 Services

An active xml document contains service calls. They could be calls to any web service or to specific ActiveXML web services (e.g. OptimaxService, GenericQueryService). We

would like to explain in this section their functionality.

We define two parties:

- Client: It contains an active axml document with service calls. It represents the place where a service call(sc) is materialized (the corresponding web service is called) and where the results are collected and saved into the document.
- Server: The place where service calls are executed;

4.4.1 Generic Query Service (GQS)

GQS provide the following methods:

- *executeGenericQuery* : It gets a query, executes it and returns the result.
- *continuousQuery*: execute a continuous query (stream parameters). It gets a query whose parameters come from a stream. It executes the query each time a new value of a parameter arrives and sends back the results in a stream. When there are no more parameters to be received, the GQS sends EOS (end of stream) to the client and the service call is terminated. It uses a channel for subscription. This subscription is based on currentID.
- *applyQuery*: Applies a public query defined in another document (as a function). It contains a reference to query declaration. The parameters are: a query defined in a *sc* inside an axml document. It could be called continuously by combining it with ContinuousService.

Below one can find an explanation of the protocol between Client (MaterializerForContinousQuery) and Server (GenericQueryService) when a service that has a query with continuous parameters is activated. Example (the syntax is simplified):

```
<sc="GenericQueryService">
 <continuousQuery>
  <declaration>
    let $a:= <param id="stream1">
    let $b:= <param id="stream2">
    for $a1 in $a/*, $b1 in b/*
     where $a1/data = $b1/data --join condition
    return $a1
  </declaration>
  <param id="stream1">
    <sc="StreamService" />
    <data1 />
```

```
  </param>
  <param id="stream2">
    <sc="StreamService" />
    <data2 />
  </param>
 </continuousQuery>
</sc>
```

Attention, the parameters return results as a tree, so they are not a list.

```
<param>
<data1 />
<data1 />
...
</param>
```

The implementation of *continuousQuery* is the following: there is a document with a service call to GenericQueryService, it takes an XQuery with parameters as an input. Parameters can have other service calls. When the service call is activated the server receives the input and registers a channel for the service call on the client. The channel has buffers for every parameter of the given XQuery, when the parameter changes on the client side, the service call again refers to the GenericQueryService, where the query is re-executed with respect to actual parameters and what is inside the buffer. And the result goes back to that service call. So the service call to the GenericQueryService is actually a query on one or more streams (depending on the number of parameters), where a stream is a service call with incoming data.

For example, given the following declaration of a query for a generic query service

```
<q:declaration>
    let $book := <q:param name="book"/>
    for $b in doc('/MyPeer/bookstore1.xml')/*/*
    where $b/title=$book/*/title
    return $b
</q:declaration>
```

For each new parameter that arrives, the query is rewritten on the server side as:

```
<q:declaration>
    let $book := <q:param name="book">ACTUAL VALUE OF THE PARAMETER</q:param>
    for $b in doc('/MyPeer/bookstore1.xml')/*/*
    where $b/title=$book/*/title
    return $b
</q:declaration>
```

4.4.2 Dummy Stream Service (DSS)

It behaves like a stream service. It executes a query and sends back the results one by one, in a stream.

4.4.3 Optimax Service

Optimax Service uses the optimizer for AXML documents, that rewrites the documents with the help of Algebra operators (Send, Receive, NewNode). The rewriting is based on strategy and statistics.

4.4.4 Continuous Service

It calls a web service in a continuous way, each time a new parameter arrives. It works now for one continuous parameter. When called from an active xml document, MaterializerForContinuousSC should be specified, as it knows how to call this service. An example that calls a non-axml web service in a continuous way; it uses MaterializerForContinuousSC and ContinuousService; it generates the request for temperature service from a query inside DummyStreamService.

```
<axml:sc axml:id="query3" >
 <axml:return
   materializer="fr.inria.gemo.axml.model.sc.materialization.MaterializerForContinuousSC">
  <axml:append/>
 </axml:return>
 <axml:ws-soap endpoint="http://localhost:6969/peer1/services/ContinuousService">
  <c:continuousCall xmlns:c="http://futurs.inria.fr/gemo/axml/service/ContinuousCall">
   <c:endpoint>
    http://webservices.daehosting.com/services/TemperatureConversions.wso
   </c:endpoint>
   <c:data>
     <axml:sc axml:id="gettemp3">
      <axml:return>
       <axml:append/>
      </axml:return>
      <axml:ws-soap endpoint="http://localhost:6969/peer2/services/DummyStreamService">
       <s:streamToMe xmlns:s="n/a">
        <s:max-timeout>5</s:max-timeout>
        <s:query>for $i in (1 to 3) return
         <CelciusToFahrenheit xmlns="http://webservices.daehosting.com/temperature">
          <nCelcius>{$i}</nCelcius>
```

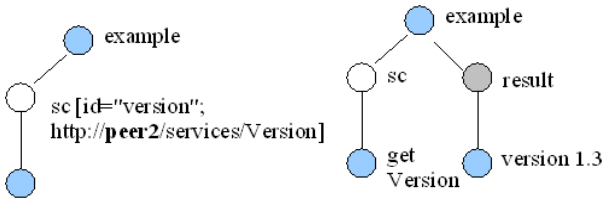

Figure 12: *version.xml*@*peer*1 before and after activation

```
        </CelciusToFahrenheit>
       </s:query>
      </s:streamToMe>
     </axml:ws-soap>
    </axml:sc>
   </c:data>
  </c:continuousCall>
 </axml:ws-soap>
</axml:sc>
```

4.5 Document Manager

The document manager (DM) is responsible for the management of documents being evaluated by the system. It only keeps references to the interesting parts of active documents using XPath and XQuery. For example it marks service calls as actived/terminated. For evaluation, a document is loaded into the Document Manager.

To check if a document is being evaluated, call *isBeingEvalued*(). In this case, a new evaluation is not possible until the current one has finished. Some documents may be loaded into the document manager by Optimax and then evaluated. In this case, the Materialization Service gets the document from DM and evaluates it. If the document does not exist in DM, then it is loaded by *MaterializationService* (manage document) and then evaluated.

4.6 Activation: chain of execution

We would like to present the chain of execution in the case when a document is activated. Basically we describe the main actions that are performed in the system during the evaluation of an active xml document.

Let's take as example a document, *version.xml* that is installed on *peer1* and gets data from *peer2*. The figure 12 depicts the representations of the xml document before and after activation.

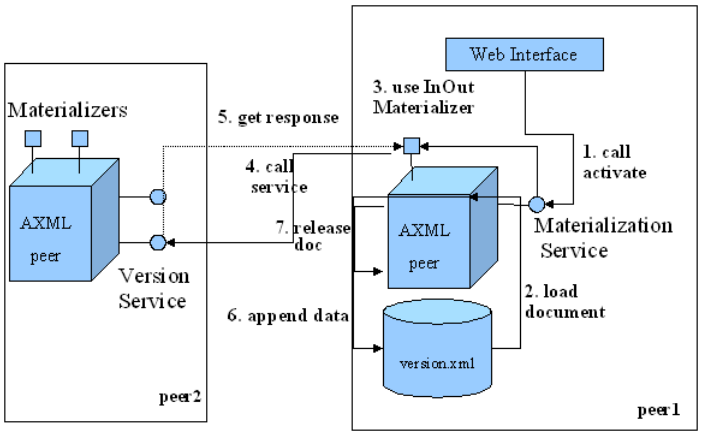

Figure 13: Activation: chain of execution

For example, from the web interface, the user asks for the activation (evaluation) of an active document. The Materialization Service is called by sending an activation message. The service loads the document from the repository and parses it. It uses the appropriate materializer, in this example the $InOutMaterializer$, that calls the service that is defined by *sc* tag. The materializer gets also the response and calls $Receive@peer1$ in order to append data into document. At the end the document is released from Document Manager.

Let's us explain now more detailed what happens when a document with multiple service calls is evaluated:

- The peer should already be running;
- The document should already be uploaded into eXist;
- Evaluate the document: the MaterializationService is called. This service activates the axml document.
- Load document into DocumentManager (IActiveXMLData);
- A batch is used to materialize in parallel the sc, taking into the consideration the activation order and the specified materializers (see subsection 4.7).
- Check if the batch has cycles and is consistent (no scs that depend on DISABLED scs, e.g. sc1 is DISABLED and sc2 waits for sc1)
- Parallel materialization: each materializer calls the specified service

- The request to any Web service goes through ServiceCallWrapper (Axis2 module). It looks at what's inside the service call and creates a SOAP message that will travel to the Web service. Besides the message body, a header is attached which has info about what service call has been activated. The info is currentID and originalID. So the Web service that understands this header can reference that service call
- The message comes to the Web service, and depending on implementation it does something.
- When the result comes back to the place where the call to the Web service was issued, it is passed to the local RECEIVE. Since Materializer is issuing the call, therefore it is also calling the local RECEIVE because it knows the EPR to the RECEIVE Web service. All the responses pass through Receive because if one wants to process the incoming message of every service call, then an Axis2 module could be created and attached it to the RECEIVE Web service.
- Release the document, if all service calls have been evaluated

Example for GenericQueryService / continuousQuery:

```
Client:                                              Server:
1)call gqs.cq   ------> 2)install buffers, create channel
4)service call
  stays active  <------ 3) respond with the marker STREAM
6)get data into
        RECEIVE <------ 5) execute cq, send data
                                  (directly calls RECEIVE)

Now the service call is still active. The Materializer is listening
to the changes in the parameters.
...(When p11 or p12 change):
1)call gqs.cq(p1x) ------> 2) execute cq(p1x) on data+buffers
4)the same         <------ 3) the same
6)the same         <------ 5) the same

This continues until server sends END_OF_STREAM in the step (3)
```

4.7 Activation batch

During the activation, a batch of materializers is created, to activate in the correct order the service calls, depending on activation order tags (afterActivated, afterTerminated).

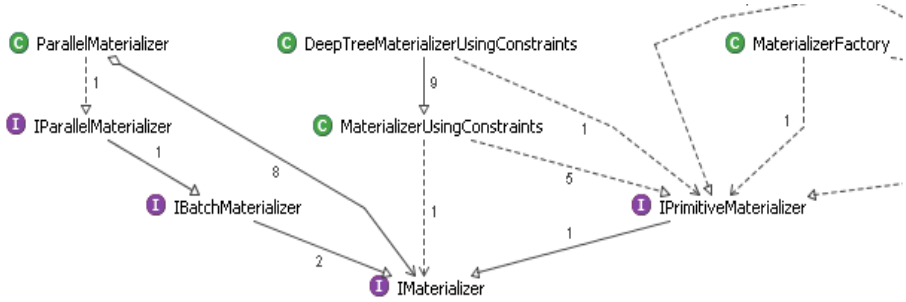

Figure 14: Materialization batch

A batch is a collection of service call materializers (e.g. DeepTreeMaterializerUsingConstraints elements). A ParallelMaterializer materializes the elements inside the batch taking into consideration the order constraints. The diagram of the available materializer classes used by the batch is depicted in figure 14.

There is a IMaterializer interface which has one method called materialize(). There is a class implementing this interface which is able to deal with a batch of materializers, i.e. given a batch of IMaterializer instances, it calls in parallel their materialize() method, it returns when all the threads join. In this way one can combine simple and batched materializers to have any order of materialization. There is also another flavor of batch materializers: materializer with constraints. It keeps a buffer of materializers which constrain its execution with afterActivated and afterTerminated, so every time something is activated or terminated it tries to execute or waits till the constraints are satisfied. The materializers with constraints are used by MaterializationService. When one wants to evaluate a document, such a materializer is created by collecting the afterActivated and afterTerminated constraints, then materialize() is called and all the service calls are materialized with the given constraints. So to materialize service calls one simply creates materializer objects for them. The materializers can extend functionality of a service call, as it is with SEND and RECEIVE (therefore you need to specify the special materializers for them), but in general there is a default materializer which looks at the service call, creates the SOAP message, calls the Web service and deals with the result.

4.8 AXML communication headers

In order to see the communication headers, let's take an example of a SOAP request:

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

```
  <soapenv:Header>
    <ns1:sc-communication xmlns:ns1="http://futurs.inria.fr/gemo/axml/">
      <ns1:currentID>
        <ns1:peerID>MyPeer</ns1:peerID>
        <ns1:docID>getversion.xml</ns1:docID>
        <ns1:nodeID>version</ns1:nodeID>
      </ns1:currentID>
      <ns1:originalID>
        <ns1:peerID>MyPeer</ns1:peerID>
        <ns1:docID>getversion.xml</ns1:docID>
        <ns1:nodeID>version</ns1:nodeID>
      </ns1:originalID>
      <ns1:endpoint>
        http://195.83.212.240:6969/MyPeer/services/ReceiveOperator
      </ns1:endpoint>
    </ns1:sc-communication>
  </soapenv:Header>
  <soapenv:Body>
    <v:getVersion xmlns:v="unknown" />
  </soapenv:Body>
</soapenv:Envelope>
```

And the corresponding SOAP response:

```
<?xml version='1.0' encoding='utf-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns:getVersionResponse xmlns:ns="http://axisversion.sample">
      <ns:return>
        Hello I am Axis2 version service , My version is 1.3
      </ns:return>
    </ns:getVersionResponse>
  </soapenv:Body>
</soapenv:Envelope
```

When an data arrives to an axml peer, as a consequence of the activation of an axml document, the system calls the local Receive service. The message sent to receive has the following structure:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
```

```
    <ns2:receive xmlns:ns2="http://futurs.inria.fr/gemo/axml/service/Algebra">
      <ns2:address>
        <ns2:currentID>
          <ns2:peerID>MyPeer</ns2:peerID>
          <ns2:docID>getversion.xml</ns2:docID>
          <ns2:nodeID>version</ns2:nodeID>
        </ns2:currentID>
      </ns2:address>
      <ns2:data>
        <ns:getVersionResponse xmlns:ns="http://axisversion.sample">
          <ns:return>
           Hello I am Axis2 version service , My version is 1.3
          </ns:return>
        </ns:getVersionResponse>
      </ns2:data>
    </ns2:receive>
  </soapenv:Body>
</soapenv:Envelope>
```

4.9 Axis2 handlers

Axis2 handlers are modules that are attached to an axis2 container in order to perform some predefined actions on the in/out transferred messages (see figure 15).

On the client, where a request is done, so the SOAP envelope is constructed, ActiveXML system provides one module, *sc-wrapper*. It adds the *sc-communication* header: endpoint of ReceiveOperator, OriginalID and CurrentID (peerID, docID, nodeID). On the server, where messages are received, there are two modules: *svc_entry*, that sets the axml context and *SOAP alerter*, that monitors the web service requests and responses.

4.10 Interaction with eXist repository

ActiveXML uses the eXist(XML database) as a repository for active documents, but also for temporary xml files generated/used during materialization of active documents.

Every peer has a default working collection, $PeerName$ working collection: $PeerName/$ system. This collection contains the documents that are not for activation. So the peer can keep working data there, also can evaluate documents after registering them as being active.

NEWNODE uses the system collection. This service creates a node in the repository. When the address is specified this node will end up inside some document, when the address is not given it will end up as a "lonely root" in the repository, i.e. a document

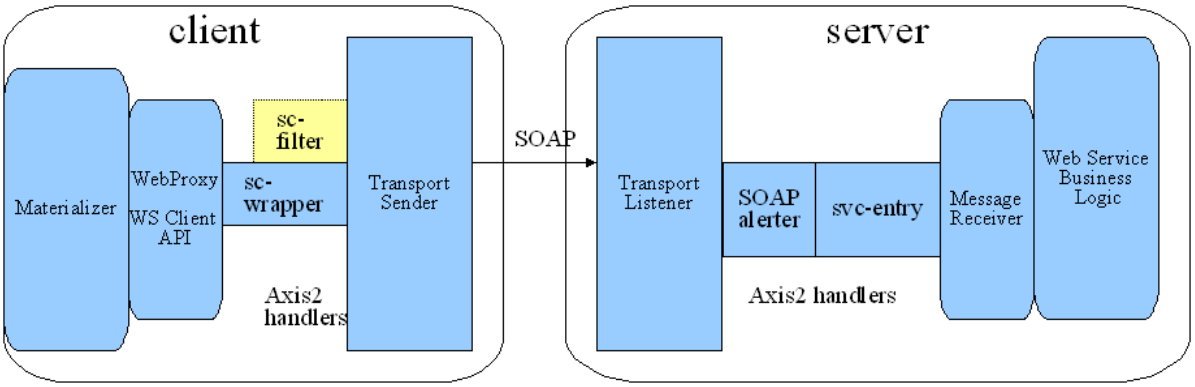

Figure 15: Axis2 handlers

like system/lonely/...xml. Moreover when it ends up under system/lonely it is evaluated (so it is registered as being active and then evaluation service is called).

System/streams collection is used as a working folder of the query service. From the service perspective a stream is identifiable by some service call in the P2P network. So whenever some service call is "subscribing to a channel" the generic query service is registering system/streams/[unique service call identification in P2P], where it keeps the buffer of parameters for a specific service call. Query service (depending on the query) has to know the history of parameters, without the buffer they all would need to be shipped every time.

System/tmp is used on the client side. When a result comes (it comes only through RECEIVE) it is first saved under system/tmp. This can be avoided, but it's put there for easier manipulation (you could do that in memory of course). The manipulations that need to be done on the incoming document consist of:

1. Attaching some meta info, e.g. adjusting the timestamp, signing the service call to which this result belongs

2. Ensuring that this result will not harm the integrity of the document. Inside a document every service call has a unique id (currentID), but when a result comes it can contain a service call with an id that is already in the target document (to be fixed).

4.11 Listeners

When a result arrives for a sc, interested parties (e.g. parent service calls) are informed with the help of listeners. The corresponding materializer will be notified when a new

result arrives, and it selects the data from axml data and sends it to its service.

The listeners follow a basic observer pattern. ServiceCallStateListener's are listening for termination or result arrival of specific service calls. The SendingServiceCallMaterializer needs to know when this happens to the service calls underneath it in order to send fresh data. AbstractActiveXMLServiceCall fires these events. Basically on every result (when RECEIVE asks to consume the result), the service call consumes it and then fires $afterResult$. When the result was EOS it fires $afterTerminate$ as well.

4.12 Regression tests

Before committing to SVN, it is advisable to run the tests from ActiveXML2Ext/tests in order to verify if older functionality is broken. The tests contain basically some axml files that describe different scenarios of ActiveXML usage and some expected behavior. The axml files provide also good examples for users.

These tests contain scenarios for testing:

- algebra services
- dummy stream service
- generic query tests
- distribution tests (calling services from different peers)
- optimax
- continuous service
- activation order

In order to configure and run the tests, the following remarks should be taken into consideration:

1. Implementation: AxmlDocumentTest.java evaluates a document, waits for termination and checks the results.

2. Configuration: Each set of sets has a folder in *config* directory. *test.testSetName.config.properties* contains the configuration values for the current set of tests.

   The files from *peer* folder are installed into exist/db/peerX. Peer name is defined into *test.testSetName.config.properties*.

   The file *testSetName.data.xml* contains the configuration properties for tests. These values are read by AxmlDocumentTest.java. *prepare.tests.data* ant target copies

this document to the default one mentioned into *testDocSrc.properties* (the file *axml-DocumentTest.data.xml*) This file name is hard-coded into AxmlDocumentTest.java before the constructor *(@XMLParameters (/axmlDocumentTest.data.xml)*, so each test should use it. *testSetName.data.xml* could contain more tests, that are run sequentially.

Pay attention that values into *testSetName.data.xml* correspond to the files into *peer*.

3. Running:

   - Set into build.properties the configuration file for the desired test (e.g. *test.gqTest.config.properties*);
   - build and run the tests (tests.all).