Active XML Primer *

The Active XML team

1 Introduction

Distributed data management is dramatically changing because of XML and Web services. XML [12], a self-describing semi-structured data model, is becoming the standard format for data exchange over the Web. Web services started providing an infrastructure for distributed computing at large, independently of any platform, system or programming language. Together, they provide a framework for distributed management of information over the Internet.

This paper focuses on XML documents where some parts of the data are given explicitly, while other parts consist of calls to Web services that generate data. We can see such documents as *intensional*, since some parts in them are defined by programs that obtain data when needed. We can also view them as *dynamic*, in the same sense as dynamic Web pages, that possibly return different, up-to-date documents when called at different times. We call such documents *Active XML documents*, AXML for short. We term a system responsible for storing and managing such documents an *AXML peer*. It acts as a client because activations of the calls inside documents use Web services provided by other systems. It is a server in the sense that it provides querying facilities on its repository of documents. These are exposed as Web services.

When a service call is activated, the data it returns is inserted in the document. Therefore, documents evolve in time as a result of service calls. This process of *materializing* some calls plays a crucial role in our approach. It leads to the central issue of deciding when to activate a particular service call. In some cases, this activation is decided by the peer hosting the document, in order to refresh data (e.g. daily). But the Web service provider may also decide to send updates to the client, for instance because the latter registered to a subscription-based, continuous service.

It is important to stress that Web services rely on open standards, namely SOAP [8], WSDL [11] and UDDI [10], which are becoming the standard means of accessing, describing and advertising valuable, dynamic, up-to-date sources of information over the Web. Therefore, AXML documents can be universally understood, used and exchanged. To understand the problem, let us first highlight an essential difference between the exchange of regular XML data and that of AXML data. In frameworks such as Sun's JSP [5], or php [7], dynamic data is supported by programming constructs embedded inside documents. Upon request, *all the code* is evaluated and replaced by its result to obtain a regular, fully materialized HTML or XML document. But since Active XML documents embed calls to Web services, one does not need to materialize all the service calls before sending some data. Instead, a more flexible data exchange paradigm is possible, where the sender sends an XML document with embedded service calls (namely, an AXML document) and gives the receiver the freedom to materialize the data if and when needed. In general, one can use a hybrid

*This project is partially supported by EU IST project DBGlobe (IST 2001-32645)

approach, where some data is materialized by the sender before the document is sent, and some is materialized by the receiver.

The first contribution is the AXML language, an XML idiom that introduces a syntax for embedding Web service calls into an XML document. It also provides sophisticated means to control the activation of these calls. A large portion of this paper is devoted to this issue. A second contribution is the AXML system, a first implementation of an AXML peer we developed and that is briefly described here.

The paper is organized as follows. We first present the basics of Active XML. Next, the client capabilities of Active XML peers are detailed, which introduces the main features of the AXML language. We then present AXML peers as servers, and see how Web services can be declaratively defined. The crucial problem of controlling call activations is studied next, and is followed by an overview of the semantics. Eventually, we briefly describe the implementation, and conclude.

2 Preliminaries

Information is already available on the Web in numerous forms (relational databases, Webpages, spreadsheets, etc.). This diversity primarily comes from the variety of autonomous and heterogeneous devices and platforms: from large volume (massive warehouses, databases), to small (personal computers) and very small (PDA, mobile phone, house or car appliances). The goal of Active XML is to facilitate the management of distributed information for these different settings in a peer-to-peer context. It builds on two core components:

XML , the extensible mark-up language, is becoming the standard for data exchange. The XML model is based on labeled ordered trees. XML comes together with query languages such as XPath or XQuery, and a number of essential tools, like namespaces.

Web services SOAP, the Simple Object Access Protocol and WSDL, the Web Service Definition Language are already very popular, and their standardization in underway by the W3C. Active XML relies on the SOAP protocol to provide/access Web services and on WSDL to describe them.

With XML and Web services, seamless access to heterogeneous, distributed information on the Web becomes feasible. This is crucial for many applications, e.g. in the context of B2B, B2C or B2G, in particular those who perform data integration using Web information. Let us therefore consider data integration in more detail, and highlight some of the key benefits of the AXML approach for it.

Typically, the goal is to integrate information provided by a number autonomous, heterogeneous sources and to be able to query it uniformly. One usually distinguishes between two main alternatives in a data integration scenario: warehousing vs. mediation. The former consists in replication the data of interest from the external sources, and working on this data locally, whereas the latter relies on query rewriting mechanisms to fetch just the necessary data from the sources at query time.

AXML allows to introduce flexibility in various ways. First, it assumes a peer-to-peer architecture, so in particular, the integrator needs not be one particular server, i.e. many peers may participate to this task. Second, it allows to follow a hybrid path between mediation and warehousing. More precisely, it allows to warehouse only part of the information. Finally, it considers Web

services as first class components of the system, hence any new Web resource that uses Web services standards may be discovered and integrated. Thus, in some sense, AXML also allows "service integration". It should however be observed that AXML is not a technique for data integration (such as e.g., Information Manifold), but a language and a system to facilitate data integration.

Beyond data integration, the ambition of AXML is to facilitate the deployment of distributed applications based on distributed data sharing at large. It is thus a relevant technology for a wide range of applications such as comparative shopping or cooperative editing.

The essence of AXML is very simple. An *AXML document* is an XML documents with embedded Web service calls. More precisely, it is a valid XML document (so it may benefit from all existing software tools for XML) where some particular elements (the ones labeled *sc* [1]) are interpreted as service calls. The presence of these elements make the document intensional, since some of the data of is given explicitly, whereas for some of it, a definition (i.e. the means to acquire it when needed) is given. AXML documents may also be seen as dynamic. The same service called twice may give a different answer (e.g., because the external data source changed), so the same document at different time will have a different semantics, possibly reflecting world changes.

Although AXML is based on this very simple idea, we will see that this takes us quite far in terms of modeling power, and that it raises a number of challenging technical issues. It should be noted that the idea of embedding calls into data is not new in databases: (i) relational systems support procedural attributes, i.e., columns in relations that are defined by some code, and (ii) objects in object databases have attached methods. This idea is also not new on the Web: Sun's JSP and php may be seen as means to mix data and calls. Even closer to what we propose here are Macromedia MX or Apache Jelly, that allow to include Web service calls into documents.

Let us consider a sample AXML document, representing a portal for ski resorts in Colorado. Part of it could be:

```
<resorts state="Colorado">
   <resort>
      <name> Aspen </name>
      <altitude scale="feet">8000</altitude>
      <scond> <sc>Unisys.com/snow("Aspen")</sc> </scond>
      <hotels ID="AspHotels" >
      <sc>Yahoo.com/GetHotels(<city name="Aspen"/>)</sc>
      </hotels>
   </resort>   ...
</resorts>
```

where *scond* stands for snow condition. The *sc* elements represent service calls embedded in the document. Elements that have an *sc* child are seen as defined intensionally. Their content changes as a result of the activation of the call. More generally, one could consider elements whose content would be partly extensional and partly defined by one or more service calls. To simplify, this will not be considered here.

Note that some of the information is extensional (e.g., the altitude), whereas some is intensional (e.g., the temperature). The document specifies how to obtain the intensional information from servers such as Unisys or Yahoo.

[1] A special namespace is used to differentiate them. The full syntax of *sc* elements is omitted here.

Suppose that we decide to call Unisys to get the temperature. Then the snow condition element may become:

```
<scond> <sc>Unisys.com/snow("Aspen")</sc> <depth
unit="meter">1</depth> </scond>
```

Observe that the new data is inserted as sibling elements of the *sc* element, which is *not* removed from the document. Therefore, calls to Web services modify (typically enrich) the document. Indeed, the re-activation of this call will allow to refresh the information. We will consider in the next section the means to control the activation and re-activation of calls, and the lifespan of the data that is thereby obtained, as well as some essential issues that are raised. But first, we conclude this section with two essential aspects of this approach.

Data exchange in AXML An AXML document can call a regular Web service, in which case the parameters and results are "plain" XML. But it can also call another AXML-aware system, in which case the exchanged data can be AXML documents. Note that this allows for some form of distributed computing. By sending data containing service calls (as parameters of a call), one can delegate some work to other peers. Also, by returning data containing service calls, one can give to the receiver the control of these calls and the possibility to obtain directly the information from its provider.

AXML Peer As we just saw, AXML is first a language for intensional/dynamic documents. But it is also a framework built around the notion of AXML *peers*. An AXML peer has essentially three facets:

- Repository: This is similar to an XML database server. An AXML peer has a repository of AXML documents and a query engine to process them. In some cases, such as an AXML peer running on a PDA, the repository may be small and the query language very limited, e.g., a subset of XPath.
- Client: An AXML peer may have to evaluate some service calls embedded in the documents of its repository, by calling Web services.
- Server: An AXML peer may provide a number of Web services, that are typically described using parameterized queries or updates over its repository.

In the next section, we focus on the client facet of an AXML peer, whereas we will look at it as a server in the following one.

3 AXML peer as a client

The main novelty in AXML is the evaluation of calls. It entails a number of issues: (i) When to activate a call; (ii) Where to find its arguments; (iii) What to do with its result; (iv) How long with the returned data remain valid; (v) What exactly to exchange. These are considered next in turn.

3.1 When to activate a call ?

Consider a service call in a document of a peer c (for client) that calls a service on a peer s (for server). One can first distinguish between calls that are activated from the client side (pull mode), which are regular function calls, and those activated from the server side (push mode). For the pull mode, we will further distinguish between what is called explicitly (explicit pull) and what is called implicitly (implicit pull), because of some other operation that requires this call to be activated. The particular form of activation control that is used is specified using some attributes of the *sc* element: frequency, mode, etc. such as in:

```
<sc frequency="daily">...</sc>
```

Let us consider these alternatives in more detail.

Explicit pull The most standard type of explicit pull is based on time frequency, e.g., activate this particular call daily, weekly, etc. For instance, we may want to call Unisys daily to get the snow conditions. More generally, one can request the call to be activated after some particular event, e.g., when some other call is completed. This aspect of the activation problem is very related to the field of active databases.

Implicit pull This means that we want to call the service only when the data it returns is needed. As will be seen later, this correspond to the fact that the peer – as a server – has to answer some service request. Since services are implemented as queries on top of the repository, this amounts to checking whether a query uses some information that this service call returns. Implicit pull is also called the *lazy* mode, because it is in the spirit of lazy evaluation of calls in functional programming. For instance, we may decide to obtain the list of hotels in Aspen in lazy mode, i.e. only when someone requests information about hotels in Aspen.

The main issue for managing implicit calls is the following: Given a document and a query, we need first to find the portions of the document that may have an impact on the result of the query. Then we need to activate all the lazy calls that may contribute to these portions of the document. Note that because of this lazy mode, a call may activate a call on another server, and so on recursively. Unsurprisingly, this leads to similar techniques as the ones used in deductive databases. More precisely, in this recursive context, we need to generate only the "relevant data", which is in the spirit of the goals achieved by query optimization techniques for deductive databases such as Magic sets or Query-SubQuery.

Push In push mode, the server decides to push information to the client, generally because the client subscribed to some continuous service it provides. It is then the choice of the server to decide when to send information to the client. A stream of data from the server will then be flowing asynchronously to the (subscription) service call node of the client document. Additional parameters for the *sc* are available, to control how the continuous service should operate. These will be considered in Section 4.

The ski resort site may for instance choose to subscribe, in its tourist guide, to some promotional packages offered by AspenAlive.com. Push services are also useful for managing change control, and in particular for synchronizing replicas of the same data.

3.2 What to do with the result ?

We first consider a local insertion, and then a more global fusion. As before, attributes of the *sc* element are used to specify what to do with the result.

Local merge In the example, we assumed so far that the result of a service call was inserted at the place of the call (as a sibling of the *sc* element). More precisely, a call returns an AXML forest and when the call is reactivated, we need to know what to do with its result and with the existing forest (the former siblings of the service call). To do that, we use merge functions (particular services). Some are provided by the system but more may be defined. The basic ones are:

- append: just append the new forest to the previous one. This is the default mode;
- replace: replace the old forest by the new one;
- fusion: Based on IDs or key specifications (such as those of XML Schema), identify the "same" elements among the old AXML forest and the new one and merge them. Clearly, there are various ways of merging two elements [2]. The system provides a particular one (details omitted). Others could be used by introducing customized merging operators.

Global merge The same mechanisms as for local merge are used, except that the new forest is merged with the whole document and not only with the *sc* element siblings. It is important to observe that, with global fusion, it is much harder to control which portions of the document are affected by a service call. So to simplify, we currently forbid the use of these calls together with the lazy mode.

To conclude this section, we should note that service calls may lead to errors. This is because some merges may fail, due to key specifications that do not match the data, or to a duplicate attribute inside an element that makes the document not well-formed anymore. In such cases, the the call that is responsible for the error is simply rejected and its result ignored.

3.3 How long will the returned data remain valid ?

Another issue is how long some data remains valid, once it has been obtained. This is also guided by an attribute of the service call, called `valid`, which can take several values:

1 day, 1 week, 1 month, etc. This may be viewed as some controlled form of caching.

Validity is zero This means that the data remains valid just the time needed to serve the request that asked for it. This is in the spirit of mediators that do not store any information but simply obtain it when needed. It is generally used together with a lazy mode.

[2] For instance, one could append elements with the same tag and for attributes, one could take the value of the new one.

Unbounded The information will remain forever unless explicitly deleted. This can be used in particular for archiving purposes, where data is never deleted. However, unbounded validity may also be of interest in other settings. For instance, when used together with a replace mode (the information is erased when a new version is obtained), it allows to keep a single version of the result, the last one. Also, it may serve as the basis for some form of caching, if used together with other services that explicitly delete data to reclaim space when needed.

Note that various portions of the document may follow different policies for validity based on the nature of the information. For instance in comparative shopping applications, one often needs to combine a mediator style for some products that change price very rapidly (e.g., plane tickets) and a warehousing style for other products (e.g., housing).

To illustrate the use of attributes controlling the execution of service calls, the sky resort document may be:

```
<resorts state="Colorado">
  <resort> <name> Aspen </name>
  <scond> <sc valid="1 day" mode="lazy" >
  Unisys.com/snow(../name)> </scond>
  <hotels ID=AspHotels > <sc valid="1 week" mode="immediate">
  Yahoo.com/GetHotels(<city name=../name/>) </sc>
  </hotels>
  </resort>
</resorts>
```

Here, the service call that gets snow conditions is called when the data is needed, only if the data is older than 1 day, whereas the one that gets the list of hotels is called regularly, every week.

3.4 Where to get its arguments

The arguments of a service call are specified as children of the *sc* node. In the simplest case, an argument is an XML value. More generally, it can be any AXML element, therefore it may itself contain some service calls. Indeed a particular case of service calls turns out to be very useful in this context, namely XPath queries. For instance, instead of stating explicitly that we want the snow condition of Aspen, we may request the snow condition of "../name". Note that this is a regular call to a local service, defined using XPath, which is a subset of XQuery. The fact that we allow to specify the XPath expression directly is just useful syntactic sugar.

Now, we cannot pass the XPath expression to Unisys because it only makes sense in the context of this particular node. Thus, such context dependent calls have to be resolved before calling the service. In general, when the parameters of a call contain some unresolved service calls, it is unclear whether we should evaluate them or not before sending. Some have to, e.g., path expressions that make sense only locally. For others, a particular choice may influence the semantics of the application. This is what we coin the "to call or not to call" problem. We will devote an entire section to that question of controlling the invocation of service calls.

4 AXML peer as a server

An AXML peer may call arbitrary Web services. It may also support and publish Web services, as illustrated in this section. In short, an AXML service is defined by a parameterized query or update over the peer's AXML documents, and we also support push services, that are specified as continuous queries.

Pull: query and update services The basic way to define a service in an AXML peer is as a parameterized XQuery query over its repository of AXML documents. Here is a sample service definition:

```
let service Get-Hotels($x ) be
  for $a in document("resorts.axml")/resorts/resort,
        $b in $a//hotels/hotel
  where $a@name=$x
  return <h> {$b/name} {$b/price} </h>
```

This query uses the `resorts.axml` document from the repository. When a call to this service is received by the peer, the $\$x$ variable is bound to the transmitted parameter value (the resort name), then the query is evaluated, and its result is returned as an answer.

We also support update services, that have side effects on the repository documents. These are defined declaratively, using the update extension to XQuery proposed in [9].

Providing such services assumes of course that the peer has an XQuery/update engine. More limited AXML peers, such as those running on PDAs, will of course provide less complex services, defined using XPath queries for instance.

Push: continuous services A server may work in pull mode, as described above, but it may also work in push mode: The client subscribes to a particular push service, by issuing service call, then the server asynchronously sends a stream of messages as an answer.

These services may also be also specified declaratively, using parameterized queries. Additionally, a number of parameters specify how such a service operates:

- possibly a frequency of the messages (e.g., daily);
- possibly some limitations, such as the duration of the subscription or some size limit on the data that is transmitted in each message;
- a choice of representation for the changes: send consecutive versions, send a delta or an edit script for the changes, see e.g., [4], send a notification and publish the changes.

Typically, the server supports a limited subset of these choices among which the client chooses, by specifying the corresponding parameters in the subscription call.

5 To call or not to call ?

One of the key aspects in our approach is the exchange of AXML data. As mentioned before, when sending a piece of AXML data, one has to decide whether to evaluate the service calls embedded

in it or not. This is what we call the "to call or not to call" problem. In other words, this is the issue of controlling the invocation of the calls present in the data that is exchanged.

As an example, suppose someone asks for the phone number of the CEO of AXML-software. As a server, we can answer: look in the yellow pages (a Web service) under the name John Doe, in Orsay, France. But we can also look up his name and address in the yellow pages, and answer "(33 1) 32 45 45 45". Exactly how far we need to resolve service calls when returning an answer is the issue that is considered in this section. The same question arises on the client side, for the services calls present in the parameters of a call that we want to activate.

To see a more precise example, suppose we want to send the data:

```
(1) <temp>
      <sc> Unisys.com/GetTemp(
       <city> <sc> Webster.com/GetCapital("Colorado")</sc>
       </city>)</sc>
     </temp>
```

that is, the temperature of the capital of Colorado. Note that this piece of data could be the result of a service call to be returned, but may also be a parameter passed to another service call, e.g., one that transforms Celsius to Fahrenheit.

We can decide to simply send the previous element. But we have two other alternatives, obtained by activating some or all of the service calls:

```
(2) <temp>
      <sc> Unisys.com/GetTemp(
        <city> Denver </city>)
      </sc>
    </temp>
```

```
(3) <temp>
      <celcius>
        25
      </celsius>
    </temp>
```

Exactly what should be sent depends on the application and needs to be controlled. Choosing (1) provides more information to the recipient who now knows the name of a service for obtaining capitals and one for getting their temperatures. Both (1) and (2) let the recipient refresh the temperature by directly calling Unisys. On the other hand, one needs to use (3) if the client doesn't know the AXML language, or cannot call Unisys directly, e.g. by lack of access privileges.

To decide how far some piece of data should be evaluated before being sent, we use a type system, which is an extension of XML Schema. This approach is particularly relevant in the context of Web services, because their input and output parameters are generally controlled by XML Schema types, which are declared in their WSDL description. Our extension of XML Schema consists in introducing, besides the classical extensional types for data, some new types to describe service calls. For instance, the type system distinguishes between:

$$\text{(a) Hotel}^* \quad \text{and} \quad \text{(b) } () \rightarrow \text{Hotel}^*$$

where (a) describes a list of hotels, and (b) a service call returning a list of hotels. As an example, we could specify the following type τ for transmitting the temperature element defined above:

```
<temp> <sc> () -> temperature </sc> </temp>
```

This amounts to imposing option (2).

Let us now consider in more detail what happens. Suppose a server publishes a service providing temperatures of capitals and specifies that it returns some data of type τ. This will force the server to call the Webster service but prevents it from calling the Unysis one. Now, a client who expects "pure" XML (that is, without service calls) will not be able to use this particular service. In general, one can imagine some more sophisticated services that allow a client to specify the kind of result it expects (as an extended schema). This will not be presented here.

From the server viewpoint, a particular piece of AXML data, say t, has to be sent, and a type τ should be matched by the transmitted data. The server must find a *rewriting* of t (by resolving some of the service calls in t) into some data of type τ. In this case, it is highly desirable that the algorithm which guides the rewriting be guaranteed to succeed. That is, no matter what the calls it decides to activate do return (assuming of course they fit their declared type definitions [3]), t will be rewritten to a document of the proper type τ. If this is the case, we will say that the rewriting is *safe*. Even, when this is not possible, the server may find some rewriting that *possibly* produces a document of the proper type, but may also fail.

This problem turns out to be quite non-trivial. It can be reduced to problems of alternating tree automata. The general case is proven to be undecidable. An efficient solution that covers the cases that are useful in practice is given in [6]. We conclude this section by showing that this control of typing may be essential for security reasons.

Security A call may return some AXML document which contains service calls. This is potentially a very serious security hole, that allows for Trojan horse style attacks. Suppose that I want to use a service that gets me "quotes of the day" in my homepage:

```
<quote><sc freq="daily">qod.com/QuoteOfTheDay </sc></quote>
```

On April 1st 2003, the server may return me some "nasty" data, e.g.,

```
<quote date="2003/04/01">
  <sc freq="daily">qod.com/QuoteOfTheDay </sc>
  My heart was bumping
  <context>Nikoloz Tskitishvili, picked 5th in the NBA
    draft by the Denver Nuggets on July 8th, 2002
  </context>
  <sc>buy.com/BuyCar("BMW")</sc>
</quote>
```

that makes me order a car that I probably cannot afford.

Via typing, we can control the services we are willing to serve and to use. We may refuse to call *QuoteOfTheDay* because we are able to detect that it possibly makes us call some services we don't know of or don't trust. More generally, we may refuse documents with embedded service calls that do not satisfy some particular predicate (e.g., be in some approved list). At the extreme, we may reject all service calls in the data we receive from the rest of the world.

[3] We extend WSDL to allow the type declarations for the input and output types of Web services to use our extended schema language.

6 Semantics in brief

What is the exact semantics of a set of AXML peers interacting together and also possibly with standard SOAP clients and services. They can be modeled as state machines interacting with each other via asynchronous communications. The state of the system is the vector of states of all its components. To capture state transitions, let us focus on one AXML peer. To simplify, let us assume w.l.g. that it consists of a single document. As we saw, service calls in this document may be activated for a number of reasons: (i) because some event occurred (e.g., a day passed since the last activation for a daily activated call), (ii) some external peer has activated it (push), (iii) the call is in lazy mode and the data it provides is requested by some service call. At an instant of time, there are thus possibly many service calls in a document that should be activated. If this is the case, one is chosen non-deterministically and activated.

On the other hand, the peer may have received several calls from other peers that it is being servicing simultaneously (again, in a non-deterministic order). When the peer has computed the answer to a service, and has evaluated the service calls in the answer it needs to evaluate, the answer is sent to the caller and will (eventually) be received. We assume each message is eventually received but there is no guaranty on the delay.

When a peer receives the answer of a service call, the document is rewritten according to the mode of merge that is specified. This involves adding new data and possibly removing or updating existing one. Data may also be removed because it becomes invalid.

The system therefore evolves continuously and non-deterministically. This is a complex setting and a number of useful properties of computer systems are not a priori guaranteed:

- termination: A service call is never guaranteed to terminate. Suppose that a service f calls some service g, and that similarly g calls f. A call to one of them never completes.
- finiteness: A document may grow in an unbounded manner. Consider the very simple piece of data t:

  ```
  <a><sc>get-a()</sc></a>
  ```

  Suppose the service call *get-a* returns t and that the attributes of the service call specify that the call should be activated immediately. Then the document is rewritten into:

  ```
  <a><sc>get-a()</sc><a><sc>get-a()</sc></a></a>
  <a><sc>get-a()</sc><a><sc>get-a()</sc><a><sc>get-a()</sc></a></a></a>...
  ```

  This results in constructing a tree with unbounded depth.
- confluence: Clearly, the order of calls may have an effect on the result. This happens, typically, in a scenario of resource allocation. Suppose there is a single hamburger to distribute and that two clients are trying to call the same method that "grabs" the hamburger. Then the first call that will be received will return the hamburger whereas the second one will return nothing.
- Consistency: We assume that the document we start with is well formed and obeys its DTD (or XML Schema) if one is specified for it. An inconsistency arises if some call leads to

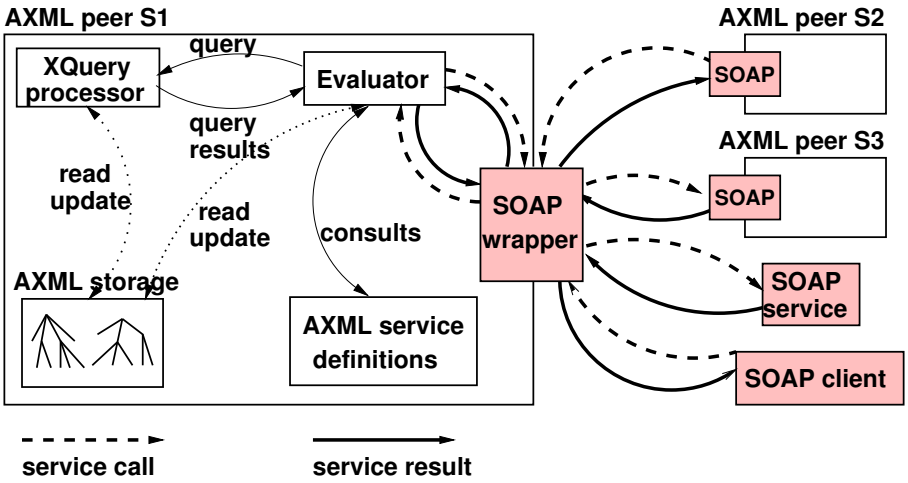

Figure 1: Outline of the AXML peer architecture.

constructing a document that no longer obeys this schema. While some of this may be prevented by consulting the declared signature of the used services static type-checking may rapidly become infeasible or too constraining, specially if one uses services with very liberal type policies.

The fact that such properties are not guaranteed may be acceptable for many Web applications and is certainly a price to pay for the power and autonomy of the peers. On the other hand, it is clearly possible to enforce some of them for particular applications. For instance, to enforce consistency, the hamburger stand above should run each service call inside a transaction to avoid delivering twice the same hamburger. It is also possible to impose systematically restrictions on the model to enforce some of the above properties.

7 Implementation

We implemented an AXML peer. A global architecture of the system is shown in Figure 1. Our implementation uses mostly standard freely available software:

- SUNs Java SDK 1.4 (includes XML parser, XPath processor, XSLT engine)
- Apache Tomcat 4 servlet engine
- Apache Axis SOAP toolkit 1.0
- JSP-based user interface, using JSTL 1.0 standard tag library

The only exception is the XML persistent store and query processor that we use, namely X-OQL [3]. This is because no XQuery processor existed at the time we started the project, and because we were fairly satisfied with the robustness of X-OQL. In principle, it should be quite simple to replace the storage module and its query language by another one.

A demonstration of version V0 of the system has been presented at VLDB02 [1]. It was featuring a P2P auctioning system. V1 is now available, offering primarily more robustness. V0 does not

support continuous services, nor lazy mode and the control of call invocations via typing. A "light" AXML peer that may run with limited resources (e.g., on a cell phone) has also been implemented.

8 Conclusion

We presented the Active XML language through its client-side and server-side features, that naturally meet in AXML peers. On the client side, dynamic/intensional documents are easily defined by including calls to Web service in them. A wealth of features in the language allow for a fine-grained control of the activation of these calls, and of the materialization of data. On the server side, data-oriented Web services are declaratively specified as queries or updates over the documents of the peer. The power of the approach is achieved via the exchange of Active XML data between peers. Work is currently progressing in a number of directions, that are mentioned next.

First, we are working on the foundations of AXML, mainly focusing on the semantics, and issues such as finding useful restricted cases that guarantee termination and/or confluence.

The typing that we use provides a partial solution to the general problem of security. It seems useful as a complement to other techniques, e.g., ACL (access control lists), to protect the access to information and services. We are investigating how to blend our typing-based approach with such techniques.

The construction of a warehouse from Web resources based on AXML is on-going. This work is done in the context of the e.dot project, which aims at building a warehouse for food risk based on Web resources. Another application is considered that deals with Genome data. Service calls are used in that context to model calls to data resources, as well as the processing of some genome data, e.g. using BLAST.

An important direction of research we are pursuing is the replication and distribution of fragments of AXML documents [2]. This is motivated by a number of reasons, and in particular:

- since many devices have limited storage, it is useful to be able to distribute a portion of a document on another peer. At the logical level, the entire document is still seen as residing on one peer, therefore this involves more work than just a clever use of XPointer, and relies on query rewriting techniques.

- replication presents many standard advantages in terms of both the improvement of the global system performance, and the availability of the services when some peers fail. One of the issues considered in this context is the synchronization of AXML data.

In the context of distribution and replication, we are lead to very interesting issues in query optimization (load balancing) and automatic replication of data and services to further improve performances.

Other topics should be considered, such as developing a visual editor for AXML documents, or building AXML peers on mass storage systems (relational or native XML DBMS).

References

[1] Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo, and Roger Weber. Active xml: Peer-to-peer data and web services integration (demo). In *Proc. of VLDB*, 2002.

[2] Serge Abiteboul, Angela Bonifati, Grgory Cobena, Ioana Manolescu, and Tova Milo. Dynamic XML Documents with Distribution and Replication. In *Proc. of ACM SIGMOD*, 2003.

[3] V. Aguilera. The X-OQL homepage. `http://www-rocq.inria.fr/~ aguilera/xoql`.

[4] Grégory Cobena, Serge Abiteboul, and Amelie Marian. Detecting Changes in XML Documents. In *Proc. of ICDE*, 2002.

[5] SUN's Java Server Pages. `http://java.sun.com/products/jsp/`.

[6] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Fred Dang Ngoc. Exchanging Intensional XML Data. In *Proc. of ACM SIGMOD*, 2003.

[7] The PHP Hypertext Preprocessor. `http://www.php.net`.

[8] Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/SOAP`.

[9] I. Tatarinov, Z. Ives, A. Levy, and D. Weld. Updating XML. In *Proc. of ACM SIGMOD*, 2001.

[10] Universal Description, Discovery, and Integration of Business for the Web (UDDI). `http://www.uddi.org`.

[11] Web Services Definition Language (WSDL). `http://www.w3.org/TR/wsdl`.

[12] Extensible Markup Language (XML) 1.0 (2nd Edition). `http://www.w3.org/TR/REC-xml`.