The VLDB Journal manuscript No.
(will be inserted by the editor)

Serge Abiteboul · Omar Benjelloun · Tova Milo

The Active XML project: an overview

the date of receipt and acceptance should be inserted later

Abstract This paper provides an overview of the Active XML project developed at INRIA over the past five years. Active XML (AXML, for short), is a declarative framework that harnesses Web services for distributed data management, and is put to work in a peer-to-peer architecture.

The model is based on *AXML documents*, which are XML documents that may contain embedded calls to Web services, and on *AXML services*, which are Web services capable of exchanging AXML documents. An *AXML peer* is a repository of AXML documents that acts both as a client by invoking the embedded service calls, and as a server by providing AXML services, which are generally defined as queries or updates over the persistent AXML documents.

The approach gracefully combines stored information with data defined in an intensional manner as well as dynamic information. This simple, rather classical idea leads to a number of technically challenging problems, both theoretical and practical.

In this paper, we describe and motivate the AXML model and language, overview the research results obtained in the course of the project, and show how all the pieces come together in our implementation.

Keywords Data exchange, Intensional information, Web services, XML

The first and third authors were partially funded by the European Project Edos.

S. Abiteboul
INRIA-Futurs & LRI
serge.abiteboul@inria.fr

O. Benjelloun
Stanford University
benjello@db.stanford.edu

T. Milo
Tel-Aviv University
milo@cs.tau.ac.il

1 Introduction

Since the 60's, the database community has developed the necessary science and technology to manage data in central repositories. From the early days, many efforts have been devoted to extending these techniques to the management of distributed data as well, and in particular to its integration, e.g., [42, 78, 71]. However, the Web revolution is setting up new standards, primarily because of (i) the high heterogeneity and autonomy of data sources, and (ii) the scale of the Web. The goal of this work is to propose a new framework for distributed data management, which addresses these concerns, and is therefore suitable for data management on the Web.

XML and Web services Our approach is based on XML and Web services. XML [82] is a data model and representation format for semi-structured data [9], which raised a considerable interest among the data management community as a standard for data exchange between remote applications, notably over the Web. Most importantly for data management, XML comes equipped with high level query languages such as XQuery [84], and flexible schema languages such as XML Schema [83]. The publication of XML data and the access to it are facilitated by *Web services*, which are network-accessible programs taking XML parameters and returning XML results. The WSDL [79] and SOAP [69] standards enable describing and calling these remote programs seamlessly over the Internet. In our approach, XML and Web services are effectively leveraged for complex data management tasks.

Peer-to-peer architecture Centralized architectures for data integration somehow contradict the essence of the Web, which is based on a loose-coupling of autonomous systems. Furthermore, centralized architectures hardly scale up to the large size of the Web. Peer-to-peer architectures propose a credible alternative, and are already spreading, notably in the context of file-sharing (e.g., see

[52,40,21]). Peer-to-peer architectures capture the autonomous nature of the participating systems, and their ability to act both as producers of information (i.e., as servers) and as consumers of information produced by others (i.e., as clients). In our approach, a peer-to-peer architecture forms the basis for scalable distributed data management.

We present *Active XML* (AXML in short) [15], a language that leverages Web services for distributed data management and is put to work in a peer-to-peer architecture. *AXML documents* are XML documents with embedded calls to Web services. Such documents are enriched by the results of invocations of the service calls they contain. Since service invocations rely exclusively on Web services standards, AXML documents are portable and can be exchanged. The AXML model also defines *AXML services*, which are Web services that exchange AXML documents. The ability of participating systems to exchange AXML documents leads to powerful data-oriented schemes for distributed computation, where several systems dynamically collaborate to perform a specific data management task, possibly discovering new relevant data sources at run-time.

One should note that data with embedded calls to operations is not a new idea, and has even already been considered in the context of XML and Web services. For instance, in Microsoft Office XP, SmartTags within Office documents can be linked to Microsoft's *.NET* platform for Web services [66]. However, to our knowledge, AXML is the first proposal that actually turns calls to Web services embedded in XML documents into a powerful tool for distributed data management. By seamlessly combining *extensional data* (expressed in XML) and *intensional data* (the service calls, which provide means to get data) our approach notably captures different styles of data integration, such as warehousing and mediation, and allows for flexible combinations of both.

We give here a comprehensive presentation of a many-year project, which developed and used Active XML as its foundation. We thoroughly discuss motivations for AXML. Although such motivations appeared in shorter forms in previous papers, they evolved greatly with our understanding of the issues. So, we thought it was useful to include them here in detail. We also present the key technical issues raised by the management of AXML documents and services. While previous papers (e.g., [57,5,14]) addressed these issues, none of them explained how the various pieces fit together to form the *big picture* of data management using AXML. The presentation will not be detailed and the reader is referred to previous papers for detailed descriptions of particular aspects. We also discuss a variety of applications that we developed using AXML, and lessons we learned from them.

A first contribution of the paper is therefore a comprehensive presentation of Active XML, the language and the general project, insisting on motivations. To a certain extent, the aforementioned material is not new. However, only facets of it were previously presented in conference articles. We believe that it is important to now present the general picture.

AXML documents and services fit nicely in a peer-to-peer architecture, where each system is a persistent store of AXML documents, and may act both as a client, by invoking the service calls embedded in its AXML documents, and as a server, by providing services over these documents. Building on this paradigm, we implemented a system (called the *AXML peer*) that is devoted to the management of AXML documents and services. The second contribution of the paper is a detailed presentation of the AXML Peer system, both in terms of its functionality and its implementation. We cover extensions to the system. In particular, we mention bridges to a workflow system (namely BPEL4WS) and P2P protocols (namely JXTA) that considerably extend the practicality of AXML. We also discuss some AXML peers with persistent storage (provided by the eXist [37] and Xyleme [86] systems), that we developed to support the management of large volumes of AXML documents.

The paper is organized as follows. The AXML model is presented and motivated in Section 2, where techniques for schema-based exchange control and lazy query evaluation are also overviewed. The functionality of the AXML peer, both as a client and a server is the topic of Section 3. The implementation of the AXML peer is discussed in Section 4. Our experience with the AXML peer in terms of building applications, and the lessons we learned are presented in Section 5. Systems of AXML peers are considered in Section 6. After a study of related work in Section 7, we conclude in a last section.

2 The Active XML model

In this section, we introduce the fundamental components of the Active XML language, namely AXML documents and AXML services. We illustrate them through examples, and give motivations for this new model. We overview at the end of this section two important issues raised by the AXML model, namely AXML data exchange and the optimization of queries over AXML documents.

2.1 Active XML documents

Active XML documents are based on the simple idea of embedding calls to Web services inside XML documents. More precisely, we define an XML syntax to denote service calls, and allow elements conforming to this syntax to appear anywhere in a document. Intuitively, these calls represent some (XML) information that is not given extensionally, but *intensionally*, by providing means to get the corresponding data. Service calls can be *materialized*, which means that the associated Web service is invoked (using the SOAP protocol), and the results it returns enrich the document, at the location of the service call.

Figure 1 is a simple AXML document, which represents a local newspaper homepage. The document consists of (i) some extensional information: the name of the newspaper and the current date, and a news story about Google (given in text with some HTML mark-up as mixed content), and (ii) some intensional information: a service call to get the weather forecast, and another one to get a list of current exhibits from a local guide.

```
<?xml version="1.0" encoding="UTF-8" ?>
<newspaper xmlns="http://lemonde.fr"
           xmlns:rss="http://purl.org/rss"
           xmlns:axml="http://activexml.net">
  <title>Le Monde</title>
  <date>2-Apr-2003</date>
  <edition>Paris</edition>

  <weather>
   % service call
    < axml:call service="forecast@weather.com" >
      <city>Paris</city>
      <unit>Celsius</unit>
    < /axml:call>
  </weather>

  <exhibits>
   % service call
    < axml:call service="getEvents@TimeOut.com">
      exhibits
    < /axml:call>
  </exhibits>

  <stories>
    <rss:item id="cx_ah_0_218">
      <rss:title>Google goes Blog-Crazy</rss:title>
      <rss:pubDate>
        Feb 18, 2003 10:36:03 GMT
      </rss:pubDate>
      <rss:description>
        Google just acquired <b>Pyra labs</b>,
        the company that makes <b>Blogger</b>.
      </rss:description>
    </rss:item>
    ...
  </stories>
</newspaper>
```

Fig. 1 An Active XML document

An AXML document is a syntactically valid XML document, where service calls are denoted by XML elements labeled `call`. Service calls, represented in bold face, use the namespace `http://activexml.net` to unambiguously differentiate them from the rest of the data. We generally associate the prefix `axml` with this namespace. For each service call, the Web service that needs to be called is identified by the `service` attribute of the `axml:call` element. For instance, the service `getEvents@TimeOut.com` means that we want to invoke the service `getEvents` provided by `TimeOut.com`. For the sake of clarity, the `service` attribute syntax that is used in the paper, is actually a simplification of the full syntax, which accounts for all the information needed to invoke Web services using the SOAP protocol.

The subtrees below service calls form the *parameters* of the calls. The call to `forecast@weather.com`, for instance, uses the name of the city `Paris` and the `Celcius` unit as parameters. In general, parameters can be arbitrary XML data, and we will see further on that they may even contain service calls themselves.

To illustrate the meaning of service calls, we can make an analogy with HTML documents. Just like hyper-links use a special mark-up in HTML (`<a href="...">`), which browsers are able to interpret, AXML service calls use a special mark-up that provides enough information for the AXML system to invoke the corresponding Web services. It is important to note that calls to *any* Web service can be inserted in AXML documents, thus taking advantage of the wealth of already existing services (such as those provided by Google, Amazon, e-bay, etc.), and leveraging them as useful data sources. Moreover, it is often straightforward to wrap an existing data source as a Web service, thereby making it interoperable with other systems.

Materialization A service call embedded in an AXML document can be activated, by invoking the corresponding Web service, and the result of this invocation is used to enrich the document. This process is termed *materialization.* Technically, the service invocation is done using the SOAP protocol, based on the attributes of the *call* element. The sent SOAP message also contains the values of the parameters (i.e., the children of the *call* element). The service answers this request with a SOAP message, which contains the result of the invocation, or an error message. If no error occurred, the result is inserted in the document, *in place of* the service call element. Figure 2 shows the modified part of our sample document after the materialization of the call to `forecast@weather.com`.

```
...
<weather>
  <temp>16</temp>
</weather>
...
```

Fig. 2 After invoking `forecast@weather.com`

We will see in Section 3 that other strategies of materialization are possible:

– The `call` element does not have to be *replaced* by the result of the invocation. In particular, it can be kept for future reuse, such as a data refresh.

– Invocations can be viewed as calls to remote procedures. However, an invocation can also be a subscription to some service. In such a case, a single invocation typically results in a stream of answers.

But for now, we assume that materialization simply consists in replacing a piece of intensional information (the service call), by its corresponding extent (the result of the invocation of the referenced Web service).

Tree representation Just like XML documents, AXML documents can be viewed as trees. AXML trees have two types of nodes: *data nodes* and *function nodes*. Data nodes correspond to the standard nodes in XML data, whereas function nodes correspond to service calls. Materialization corresponds to replacing a function node by the result of the corresponding service invocation. Figure 3 gives the tree representation of our example AXML document, before and after the invocation of *GetTemp*.

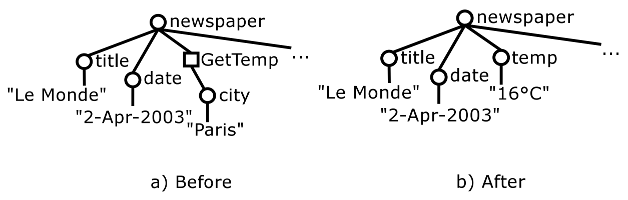


Fig. 3 An AXML document before/after a materialization.

Motivations for AXML documents

We now give motivations for representing information as AXML documents. It is important to observe that since the calls embedded in AXML documents refer to Web services, the corresponding extensional information can be retrieved by virtually any system. Thanks to this portability, a user that receives some intensional information is given the means to obtain the corresponding extensional information autonomously, if and when she decides to. This is best illustrated by the following quotation attributed to an ancient Chinese philosopher (and sometimes to Maimonides):

> *Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime.*

More concretely, AXML documents provide the following advantages:

Dynamicity Data may change with time. For instance, in our newspaper example, the temperature in Paris is updated daily, and the news headlines possibly every hour. We can call several times the same service and obtain different answers (e.g., because the corresponding data changed), so the same document at different times will have different semantics, thus reflecting world changes. An important related question is when to activate each service call. We describe means to control service call activations in Section 3.

Knowledge Intensional information conveys more knowledge than extensional information. The receiver becomes independent and can obtain the information directly. She may then be able to generalize the use of the service. For instance, a user (or system) may learn from using the *GetTemp* service how to obtain directly the temperature in *any* city in France.

We implicitly considered, in the previous examples, that systems/users could not only represent information locally as AXML documents, but that they could also *receive* such information from other systems. Indeed, the fact that systems can exchange intensional information is a central aspect of our approach. And since Web services are the natural way for systems to communicate in Active XML, we use them as the main vehicle for the exchange of intensional information. We next introduce AXML services, which are Web services that are able to exchange AXML documents.

2.2 Active XML services

As mentioned earlier, AXML documents are portable (a) because they are syntactically valid XML documents, and (b) because the service calls they contain can be materialized simply by following the SOAP and WSDL specifications. A natural consequence of this portability is that AXML document may be exchanged between systems. This observation naturally leads to introducing AXML services, i.e., Web services that accept AXML documents as input parameters, and return AXML documents as results. Because AXML documents are valid XML documents, AXML services are standard compliant Web services: they exchange SOAP messages and can be described using WSDL.

Observe that, when AXML services are involved, materialization becomes a recursive process, since calling an AXML service may return some data that may contain new service calls that also need to be materialized, and so on recursively. This is illustrated in Figure 4, which gives the modified portion of our sample document after the invocation of the `getEvents@TimeOut.com` service. Here, the service answered with a list of exhibits, plus a service call to get more exhibits from `Yahoo`.

Now, an issue that arises is the one of termination: Can we "fully" materialize a document? Can we know in advance if the materialization of a document can be performed in a finite time? It turns out that the termination problem is undecidable even in some rather restricted setting [57]. This should not come as a surprise in our large scale, highly dynamic Web-like context, where systems are autonomous and loosely-coupled. We will mention in Section 6 some restrictions on both the data model and the allowed services under which termination of document materialization and closely related problems become decidable.

```
...
<exhibits>
  <exhibit>
    <name>Naive Painting in Ancient Greece</name>
    <location>Le Louvre</location>
    <from>25-Apr-2003</from>
    <to>25-May-2003</to>
  </exhibit>
  < axml:call service="getExhibits@Yahoo.com">
    <city>Paris</city>
  < /axml:call>
</exhibits>
...
```

Fig. 4 After invoking `getEvents@TimeOut.com`

Motivations for AXML services

We now focus on benefits brought by AXML services. Convenient means to define AXML services, e.g., using queries or updates over AXML documents will be presented in Section 3. All motivations we gave for AXML documents are of course important motivations for services exchanging AXML documents. We next discuss advantages that are specific to the usage of AXML services, and which come, on the one hand, from intensional results and, on the other hand, from intensional arguments. We then discuss related performance gains potentially brought by AXML services.

Intensional results A service invocation result with intensional data represents a partial description of the answer. It may typically provide only part of it or some form of summary. This is in the style of search engines that returns, for instance, the best 10 answers and a link to get more. In some cases, the computation may be on-going and some results may be provided, possibly in push mode, before the computation is complete. In other cases, the receiver may have to invoke some service calls embedded in the result in order to complete the answer, e.g., apply a decompression, decryption service, personalize the answer, or adapt it to a geographical context. Intensional information in the answer may also be used to "enrich" it, e.g., by providing some provenance data if requested by the receiver.

Intensional arguments When the input to the Web service contains intensional information, the server is asked to perform the extra work needed to obtain the extent of this input before proceeding to its regular task. Note that this leads to a simple kind of service composition, basically by connecting the outputs of some services to the input of others. For instance, to obtain the phone number of Google's CEO, it is possible to call a White Pages service, with as intensional parameter, a call to Google's Company description service, which gives the name of the current CEO. So, although AXML does not address directly the issue of Web service choreography, it does provide some limited form of workflow. We will mention the issue of adding more elaborate forms of workflows as Web services in Section 3.

Performance motivations The motivations we presented so far can be qualified as "logical" in the sense that they deal with the way information is obtained and used, i.e., with the semantics of applications. There are also important performance incentives for using AXML documents and services (which are often counterparts of the logical motivations we described). To give an example, suppose some Web merchant S publishes its electronic catalog, and a comparative shopping site S' downloads the catalog of S periodically. Information such as product prices, which may change rapidly, is often stale. However, if S decides to represent prices intensionally, then S' is able to refresh price information by invoking the corresponding service call, without reloading entire product descriptions. Hence, the *dynamicity* of intensional results induces substantial communication savings. Indeed, we will see in Section 3 that S' will even be able to *subscribe* to price changes and be notified of such changes without the need of polling S. As a consequence, S' will have a more accurate copy of the catalog at a much lower cost.

AXML documents and services, which we just introduced and motivated, form the core of the AXML model. In the rest of this section, we examine two novel issues that arise as a direct consequence of this model, namely the AXML data exchange problem, and the evaluation of queries over AXML documents, summarizing results from [57,14] and [5] respectively.

2.3 Exchanging AXML data

As motivated in the previous section, a key feature of the AXML model is the ability to exchange AXML data (e.g., as intensional parameters or results of AXML services). The important issue we consider now concerns the decision whether to send some piece of data extensionally or intensionally. To see an example, suppose that one asks for the work phone number of an INRIA employee, say Jean Dupond. The system may answer extensionally (and provide that phone number) or may prefer to answer with the following intensional data:

```
< axml:call service="directory@inria.fr">
  <employee>Jean Dupond</employee>
< /axml:call>
```

More precisely, given a document to exchange, and some specification about the nature of the exchange, the system has to control which parts of the document should be materialized, and which should not. We first mention some considerations that may guide the choice of materializing intensional data or not:

– *Capabilities:* One of the two participants (the sender or the receiver) may not be able to call a particular service, for instance by lack of access rights or because she is missing some needed functionality (e.g., no support for calls to Web services).
– *Security:* One of the two (most typically the receiver) may prefer not to perform some particular service call for security reasons.
– *Performance:* Deciding where a particular service is called allows some form of load balancing between the sender and the receiver of the service.
– *Functionality:* Last but not least, the choice may be guided by the application. As we saw already, a service call in place of some materialized data provides the receiver with knowledge such as the provenance of the information. It also gives her the means to reuse it and for instance, refresh the data without using the site that provided her with this information.

For (extensional) XML data, *Schemas* (such as a DTD or an XML Schema) are the natural tool to specify the format of exchanged data. In particular, the data exchanged by Web services is controlled by schemas for their input and output, specified within a WSDL description. Similarly, we use schemas to control the exchange of intensional data, essentially by using them as a declarative specification that guides the materialization of exchanged AXML documents. A key novelty of our setting is that schemas may also entail information about which parts of the data are allowed to be intensional and which service calls may appear in the documents, and where.

A typical data exchange scenario is depicted in Figure 5. The sender and the receiver have agreed on a specific data exchange schema. Now, consider some particular data t to be sent (represented by the grey triangle in the figure). In fact, this document represents a set of equivalent, increasingly materialized, pieces of information, i.e., the documents that may be obtained from t by materializing some of the service calls q, g and f embedded in it. Observe that the data returned by a service may itself contain some intensional parts, which may have to be materialized, so the decision of materializing some information or not is inherently a recursive process. Among these documents, the sender must find at least one that conforms to the exchange schema (e.g., the dashed one) and send it.

To specify the schema of the exchanged data, we use a simple but flexible XML-based syntax based on an extension of XML Schemas. The extension consists in adding new type constructors for service call nodes. In particular, the schema distinguishes between accepting a concrete type, e.g., a *temperature* element, and accepting a service call returning data of this particular type.

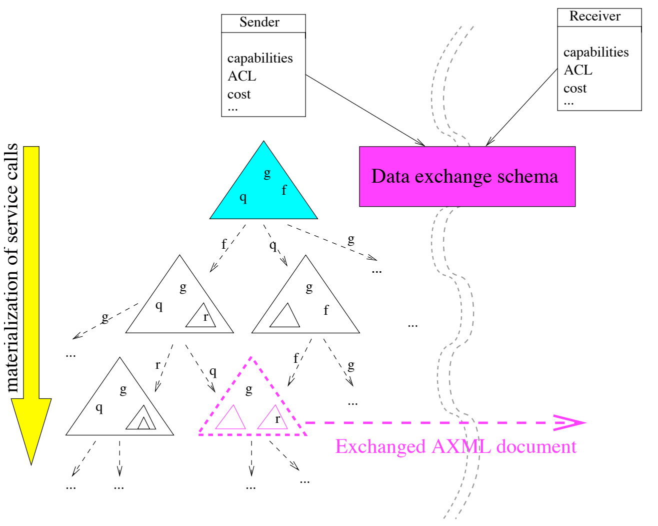


Fig. 5 Data exchange scenario for an AXML document

data :

$$\begin{aligned}
\tau(newspaper) &= title.date.(GetTemp \mid temp) \\
&\quad .(TimeOut \mid exhibit^*) \\
\tau(title) &= data \\
\tau(date) &= data \\
\tau(temp) &= data \\
\tau(city) &= data \\
\tau(exhibit) &= title.(GetDate \mid date)
\end{aligned}$$

functions :

$$\begin{aligned}
\tau_{in}(GetTemp) &= city \\
\tau_{out}(GetTemp) &= temp \\
\\
\tau_{in}(TimeOut) &= data \\
\tau_{out}(TimeOut) &= (exhibit \mid performance)^* \\
\\
\tau_{in}(GetDate) &= title \\
\tau_{out}(GetDate) &= date
\end{aligned}$$

Fig. 6 An AXML data exchange schema (simplified syntax)

An example of such a schema τ is given in Figure 6. To simplify, we use in the example a DTD-like syntax. (The actual syntax we use in the system is an extension of XML Schema.) An AXML exchange schema specifies the structure of each element allowed in a compliant document, describing the data elements and service calls each element may have as children. Service call names here start with capital letters. Observe that the schema also details, for each service, the type of its input parameters (τ_{in}) and of its output (τ_{out}).

In practice, these service input/output signatures are given separately from the exchange schema, as part of the WSDL description of each service. In our actual schema language, whenever a service is referenced in the schema, a URL for its WSDL description must be provided, which allows the system to fetch the corresponding service signature. We include service signatures here within the exchange schema for compactness and ease of presentation. It is important to note that the WSDL descriptions of AXML services remain standard compliant: They may use our extended schema language to describe their AXML input and output types, since the WSDL standard does not impose the use of a specific schema language.

In a nutshell, given a document t and a data exchange schema s, the goal of the sender is to rewrite t by materializing some of its service calls, in order to cast it into an instance of s. We provide in [57] algorithms to find such a rewriting (if it exists), with various levels of confidence:

– First, given a document, the sender may try to find a sequence of calls that *surely* rewrites it into an instance of the exchange schema. Surely means that the sender is guaranteed to find such a sequence and never activates any service call unnecessarily.
– Sometimes, such a guarantee cannot be provided. The sender may then try to find a sequence that *possibly* rewrites the document into one of the correct schema. If the attempt fails, the server has to “backtrack” and try another sequence. Observe that this may lead to calling unnecessary services if a successful sequence is not selected first or is not found.
– Finally, we provide algorithms to verify, at compile time, whether *all* data that complies with some given schema may be (surely or possibly) rewritten into an instance of the exchange schema.

The core technique of this work is based on “alternating” automata. These are automata that alternate existential and universal states. Intuitively, an existential state corresponds to the server selecting the next service call to perform. A universal state corresponds to the arbitrary (but type correct) choice of a response by the server of this particular service. This work motivated a nice fundamental study [61] of “context-free games”.

The general problem of whether a document can be rewritten to match an exchange schema turns out to be undecidable. To overcome this difficulty, our algorithms use simplifying assumptions. Namely, we search for a rewriting that can be done in one left-to-right pass on the document, with a bounded nesting of function calls. Under these assumptions, the search space (i.e., the space where we attempt to find a successful rewriting) remains infinite. The algorithms are indeed quite complex and in particular involve the construction of automata that are potentially exponential in the size of the documents. These automata are built in a lazy manner and the algorithms turn out to be rather efficient in practice.

In [14], we considered even further restrictions to get simpler and more efficient algorithms, while still covering most practical cases. There, we assume that exchange schemas obey unambiguity conditions that naturally extend the ones imposed on XML Schema for regular XML data. We show that the optimal rewriting strategy (i.e., the one that succeeds in rewriting the maximal set of documents) is unique, and regular. Documents can be rewritten in one pass using simple automata that are easy to derive from the target schema.

The algorithms of [57] have been implemented and tested in the context of the AXML Peer system presented in Section 3. The rewriting algorithms are used in the module in charge of controlling the dialogue between an AXML peer and the rest of the world. They serve to cast the data that is exchanged (if possible) in order to meet the interface requirements of the services the peer provides and/or invokes.

To conclude this section, it should be observed that controlling service calls by enforcing a schema on the exchanged is also useful (i) for customizing services to particular client preferences [3] and (ii) for building secure applications [2]. For security, it fits nicely with other security and access models that were proposed for XML and Web services, e.g., in [33, 56].

2.4 Optimizing queries over AXML documents

Optimizing queries over AXML data involves an array of techniques, including “traditional” query optimization techniques developed for XML data. Our focus is on the novel optimization challenges brought by the presence of embedded service calls in AXML documents. We present here a local form of optimization, which considers the evaluation of a query on a single AXML document present at a given peer. We will see in Section 6 that more global forms of optimizations can be considered, taking advantage of the fact that data and services are spread across several peers in order to distribute the evaluation of queries.

The evaluation of a query (e.g., expressed in XPath, or XQuery) over an AXML document naturally involves standard query processing over the extensional part of the document. However, the result of a query may also be influenced by service calls present in the document. If the data they provide possibly modifies the query answer, such calls should be materialized. By analogy with the notion of laziness in programming languages, we term this approach “lazy evaluation”. This principle brings a new dimension to query optimization, and is the essence of the work in [5], which we briefly discuss next.

Consider, for instance, a site about a city’s night-life [1] described by an AXML document and containing information about, say, movies and restaurants. Now, suppose someone asks the query:

```
/goingOut/movies//show[title= "The Hours"]/schedule
```

Then, there is no point in materializing any call that would appear below the path:

```
/goingOut/restaurants
```

[1] In the style of http://www.timeout.com

since the data they return would be set in a place where they cannot contribute to the result. One would also like to avoid materializing a call found below:

```
/goingOut/movies
```

with a signature indicating that the returned data is irrelevant to the query, e.g., a service calls returning movie reviews.

The above desiderata rule out a naive approach that would consist in materializing *all* the calls in the document recursively, until a fixpoint is reached, and finally running the query over the resulting document. On the other hand, a query processor that would process the query and materialize "on the fly" service calls found to be relevant would be very inefficient, as the processing may often block waiting for the results of calls. In [5], we present an alternative efficient technique that is based on identifying in advance a tight superset of the service calls that should actually be invoked to answer a query. The idea is to first materialize these calls and then move to classical query processing. Observe that while the problem we address is related to the classical mediation paradigm of data integration, where data sources are called when queries are asked on a mediated schema, things are substantially different in AXML since service calls may appear *anywhere in the data*, and *dynamically* in results of previously materialized calls. Also, the integration of data sources is not defined through mappings expressed at the schema level, but through service calls embedded in the queried AXML documents.

More precisely, the technique of [5] is based on:

– *Computing the set of relevant service calls:* Given a query, the algorithm generates a set of auxiliary queries that, when evaluated on a document, retrieve all service calls that are relevant to the query.
– *Pruning via typing:* The return types of services are used to rule out more irrelevant service calls.
– *Service calls sequencing:* The relationships among the calls are analyzed to derive an efficient sequence of call invocations appropriate to answer the query.
– *F-guide:* A specialized access structure in the style of data-guides [41] is used to speed up the search for relevant calls. The structure acts as an index, summarizing concisely the occurrences of functions (service calls) in the documents (hence its name, F-guide).

Note that there is an essential difference between this technique and the work on data exchange presented in Section 2.3. The goal there was to identify the calls that need to be materialized in order to make a document match an AXML Schema, and the solution relied on schema and data analysis via automata-based algorithms. In contrast, the goal here is to find the sequence of call invocations needed to evaluate a query over AXML data, and the solution is based on query analysis and query rewriting.

In this section, we presented the AXML language, which is based on AXML documents and AXML services. We also examined two important issues that arise in the AXML setting: the exchange of AXML documents, and the lazy evaluation of queries. In order to effectively support the AXML language, we adopted a distributed architecture based on the *peer-to-peer* paradigm, where each participant may act both as a client and as a server. We developed the *AXML peer* system, to serve as an individual component in this distributed architecture. The next two sections are respectively devoted to the description of the functionality of the AXML peer and of its implementation.

3 The AXML peer: Functionality

In this section, we present the AXML peer from a functional viewpoint. The implementation of the AXML peer is the topic of Section 4. AXML peers have essentially three facets:

– *Repository:* The first task of an AXML peer is to manage persistent information, very much like a traditional database management system.
– *Client:* In order to take advantage of the intensional data present in the AXML documents of its repository, an AXML peer may invoke the corresponding Web services. By doing so, it acts as a client of the peers that provide these services. We extend the AXML language with features to control the activation of service calls in persistent documents (Section 3.1).
– *Server:* An AXML peer may also provide Web services for other peers to use. These services can be defined using queries, transformations or updates over the documents of its repository, and are indeed AXML services (Section 3.2).

Since the AXML language is centered on the use of Web services, our architecture relies on invocations of Web services for all communications between participating peers. AXML peers exchange AXML data with each other through the Web services they provide (as server) and invoke (as clients). Note that existing Web service providers (such as Google) and clients (such as the Firefox browser) naturally fit in this architecture, since all exchanges follow standards for Web services. However, the exchange of AXML documents may only occur between systems that understand the AXML language.

3.1 The AXML peer as a client

The main roles of an AXML peer, seen as a Web client, is the management of *persistent* AXML documents, and the invocation of the service calls embedded in the document it manages. For instance, for the newspaper document of Section 2, the peer may want to enforce the following policy:

Temperature information is refreshed daily. New exhibits are fetched every week and archived for 6 months.

The main novelty in managing AXML documents is the control of the invocation of embedded service calls. In a nutshell, we incorporate in the AXML language, simple constructs to support specifying when service calls are invoked. For instance, the language will enable specifying the policy above. More precisely, we enrich the syntax of service calls in AXML documents. Each individual call will indicate its activation and persistence policy. It will be the responsibility of the peer to enforce these policies.

Note that in this setting, service calls should generally be *kept inside* AXML documents, for future reuse. Therefore, materialization will not replace service calls by their results anymore, but will append the results of each call next to it. While this is a slightly different semantics than the one used in previous sections, it is necessary in the context of an AXML peer managing persistent data. We should mention that this change to the model does not affect the algorithms of lazy evaluation presented in Section 2.4 (calls are marked as executed instead of being removed). For the document rewriting algorithms of Section 2.3, calls are indeed replaced by their results (since the main goal is to hide the intensional definition of data), but recall that these algorithms are mainly used in the context of data exchange, and therefore only affect the exchanged messages, and not the persistent data.

The space of persistence options for service calls in AXML documents can be decomposed along three axes: (i) where to find the arguments to be passed to a call, (ii) when to activate a call, and (iii) what to do with its result. These issues, and the language constructs that support them are considered next in turn.

Where to get the arguments of a call?

Recall that the arguments of a service call are specified as children of the *call* element. In the simplest case, an argument is plain XML. More generally, arguments can be AXML data, and therefore may themselves contain service calls. Indeed, a particular case of services turns out to be very useful in this context, namely those defined as XPath queries. For instance, as shown on the example document of Figure 7, instead of stating explicitly that the city we want the temperature of is Paris, we may request the temperature of `"../../edition/text()"`.

This construct actually denotes a regular call to a local Web service, which is implicitly defined by the specified XPath query. More precisely, a relative XPath expressions (evaluated starting from the *call* element) translates to a query that refers to the unique ID of the corresponding service call in the document. This ID is passed as a parameter of the actual invocation. The fact that we allow specifying the XPath expression directly in the document is just useful syntactic sugar.

Such XPath parameters (now viewed as service calls) are evaluated in lazy mode, whenever the service call they bring parameters for, needs to be invoked. The algorithms of Section 2.4 apply in this setting, to find and invoke any calls (marked as lazy) that may be relevant to answer the XPath query that retrieves the value of the parameter. If calls retrieved by this analysis also have XPath parameters, then (recursively) lazy evaluations of these parameters are performed.

Note also that an XPath parameter cannot be passed intensionally to the invoked Web service, because it only makes sense in the context where it appears. Therefore, 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 materialize them or not before sending. Some have to be materialized, e.g., path expressions that make sense only locally. For others, a particular choice may influence the semantics of the application. Observe that this is an instance of the AXML data exchange problem we discussed in Section 2.3, with the service call parameter as the exchanged document and the Web service input type (specified in the WSDL description of the service) as the exchange schema. In the system, we use the schema-based rewriting algorithms of Section 2.3 to control the materialization of service call parameters.

When to activate a call?

Consider a service call in a document of a peer c (a client) that calls a service on another peer s (a server). One first distinguishes 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 further distinguish between those calls that are activated explicitly (*explicit pull*), i.e., because of a user request, and those that are activated implicitly (*implicit pull*), because some other operation might use data they retrieve. The particular form of activation control that is used is specified using a special attribute of the *call* element: the `mode`, such as in:

```
<axml:call service="..." mode="daily">
  ...
</axml:call>
```

Let us now consider these alternatives in more detail.

Explicit pull The most common 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 the

Yahoo Weather service daily to get an updated forecast. More generally, one can request the call to be activated after some particular event occurred in the system, e.g., when some other call has completed. This aspect of the activation problem is closely related to the field of active databases, where triggers are used to update the database whenever some particular events occur, and/or some conditions are satisfied.

Implicit pull Sometimes, we want to call a service only when the data it returns is relevant for a computation the system needs to perform. Implicit pull is also called the *lazy* mode. Indeed, when the computation to perform is a query, the lazy evaluation algorithms mentioned in Section 2.4 are used to determine and activate the calls that are relevant to answer that query. In the system, these algorithms apply only to a subset of the service calls embedded in the documents of the repository: those that are specified as lazy, again using the `mode` attribute of the *call* element.

Push In push mode, the server pushes information to the client. This starts by a client issuing a subscription to a *continuous* service from a server. It is then the server that sends information to the client, in the form of a stream of data from the server to the (subscription) service call node of the client document. Additional parameters can be specified by the client and/or the server to control how the continuous service operates, e.g., when the server decides to send some data. These will be considered in Section 3.2.

In our example, the local newspaper may subscribe to exhibit announcements provided by the TimeOut local guide, and be notified each time a new exhibit opens in the city. Push services can also be useful for managing change control, and in particular for synchronizing replicas of the same data. This aspect of the problem is closely related to stream management and subscription queries.

What to do with call results?

Another important issue when managing persistent AXML documents is what to do with service calls results. This question has two facets: First, a policy must be determined for each invocation of a service call, to decide how long the returned data should be kept. Second, when a service call is invoked multiple times, the resulting consecutive results need to be integrated. These two aspects are considered next.

Managing individual call results The first issue we consider is controlling how long data returned by a service call remains valid (i.e., persists) once it has been obtained. In the AXML system, the persistence of results is also guided by an attribute of the service call, called `valid`, which may take a number of values:

– *1 day, 1 week, 1 month, etc.* This may be viewed as a controlled form of caching.
– *Validity is zero* This means that the data remains valid only for the time needed to serve the request that asked for it. This is in the spirit of mediators, which do not store any information but simply obtain it when needed. The zero validity is generally used together with the lazy mode, where calls are activated when the data they retrieve is needed to answer a query.
– *Unbounded* The information remains present forever, unless explicitly deleted. Unbounded validity is particularly useful 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, see further), it allows keeping a single version of the result, that is, the last one. Also, it may serve as the basis for more advanced forms of caching, if used together with other services that explicitly delete data to reclaim space when needed.

Note that various portions of an AXML document may follow different policies for validity and mode, 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 newspaper document may be written as in Figure 7. In this example, the service call that gets the temperature is called when the data is needed, and only if this data is older than 1 day, whereas the one that gets exhibits is called weekly, and the data it returns remains valid for 6 months.

Managing results of multiple invocations When the same service call is invoked multiple times, one needs to define how the answers returned by an invocation interact with those previously obtained. We first consider a local insertion, where new results only interact with previous results of the same service call. We then mention a more global fusion that may involve the whole document. As before, attributes of the *call* element are used to choose between available options.

Recall that the result of a service call, a forest, is inserted as siblings of the *call* element. When the same call is materialized again, we need to know how to combine its result with the existing forest (the former siblings of the service call). To do that, the AXML peer relies on *merge functions*. Some are provided by the AXML peer but more may be defined, locally or as Web services. Those provided by default are:

– *append:* just append the new forest next to the previous one. This is the default behavior;

```
<?xml version="1.0" ?>
<newspaper>
  <title>Le Monde</title>
  <date>06/10/2003</date>
  <edition>Paris<edition>

  <weather>
     < axml:call service="Yahoo.GetTemp"
                 mode="lazy" valid="1 day">
       <city>
         <axml:xpath>
           ../../edition/text()
         </axml:xpath>
       </city>
     < /axml:call>
  </weather>
  <exhibits>
     < axml:call service="TimeOut.GetEvents"
                 mode="every Monday morning"
                 valid="6 months">
       exhibits
     < /axml:call>
  </exhibits>
</newspaper>
```

Fig. 7 A persistent AXML document

– *replace:* replace the old forest by the new one;
– *fusion:* based on IDs or key specifications (similar to the key mechanism of XML Schema), we identify the "same" elements among the old AXML forest and the new one and combine them, so that the resulting forest still has unique IDs or satisfies the key specification. Clearly, there are various ways of combining two elements that have the same ID or key value into a single one. The AXML peer provides simple strategies, such as appending new children to old children of the "same" element, or replacing the old children by the new ones. It also supports nesting the application of ID or key-based rules.
– Custom merge functions may also be used.

Now consider a more global merging mechanism, where the new forest is merged with the whole document and not only with the siblings of the *call* element. Again, the merging is driven by IDs. It is important to observe that global fusion makes it much harder to control which portions of the document are affected by a service call invocation. Therefore, the system currently forbids the use of global merge together with the lazy mode.

In this section, we presented the client features offered by AXML peers for managing the service calls embedded in persistent AXML data. A variety of choices are available to control (i) where to get call arguments, (ii) when to invoke calls, and (ii) what to do with their results. These choices can be specified at the granularity of each embedded call, through simple dedicated attributes, which leads to a great flexibility in combining different styles of data integration (e.g., mediation, warehousing) or different kinds of service invocations (e.g., push, pull). We believe this flexibility is a main strength of our approach.

3.2 The AXML peer as a server

The previous subsection discussed how the activation of Web services embedded in the AXML documents a peer manages is specified and controlled. So, the peer was then viewed as a client of the Web. As a counterpart, an AXML peer may also support and publish Web services, i.e., be a server. Recall from Section 2 that AXML services are Web services that exchange AXML documents. In an AXML peer, AXML services can be defined as parameterized queries or updates over the peer's AXML documents. The system supports different languages for the definition of services, and allows for push services. Recent extensions to the AXML peer enable services that leverage existing P2P infrastructures. We present these different kinds of services in turn.

Pull: query and update services

The basic way for defining a service in an AXML peer is to write a parameterized query over its repository of AXML documents. Here is a sample service definition [2]:

```
Declare service GetExhibitsByLocation($loc) as {
for $a in document("news.xml")/newspaper/exhibits,
    $b in $a//exhibit
where $b/@name=$loc
return <exhibits> {$b} </exhibits> }
```

Fig. 8 An AXML (Query) Service definition

This query refers to the `news.xml` document from the repository, which may be similar to the one of Figure 7. When a call to this service is received by the peer, the `$loc` variable is bound to the passed parameter value (the location), then the query is evaluated, and its result is returned as an answer. It should be noted that any kind of AXML fragment can be passed as a parameter to the service (e.g., a service call, a forest of AXML trees, or an atomic value).

Again, the lazy evaluation technique of Section 2.4 is leveraged, in order to find and invoke the service calls that are relevant to the query that defines the service, both in the repository documents and in the parameters passed by the invocation.

Also, note that the query answer may naturally contain service calls, since the query is evaluated on the AXML parameters and the AXML documents of the repository. The document rewriting algorithms presented in Section 2.3 are used to make the result match the output type of the service. This output type is generally specified in the WSDL description of the service, that is,

[2] The queries in the standard AXML system are written in the X-OQL language [16], but to simplify the presentation we use here the syntax of XQuery, which is a standard. A version of the AXML peer running with XQuery and eXist has also been developed.

by the service provider. In our implementation, we also allow the client to dynamically specify a desired output type, as an extra parameter of the service invocation.

We also support update services, i.e., services that have side effects on the documents of the repository. These are also defined declaratively, as an extension to X-OQL that is very much in the same lines as the update extension to XQuery proposed in [72].

Pull: Other kinds of services

Other languages are sometimes more appropriate to define services than query/update languages. We integrated the following important ones into the AXML system:

Workflow services AXML peers can provide access to services defined using the Business Process Execution Language for Web services (BPEL4WS) [24], one of the main proposals to define a workflow language for Web services. To execute such services, AXML relies on the implementation provided by IBM, and enriches them with the ability to return AXML answers whose type is controlled using the algorithms of Section 2.3.

XSLT services Services can also be specified using the XSLT transformation language [85]. Such services take a single parameter, which is an XML document, and return the result of transforming the input document using the stylesheet specified in the definition of the service. While XSLT uses XPath as a selection language, to extract information from the document to transform, in the current implementation, such services do not take advantage of the intensional information that may be present in the input AXML document, since we have not integrated the lazy evaluation technique of Section 2.4 within the XSLT processor that we use for the transformation.

Push: continuous services

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

These services may also be specified declaratively, e.g., using parameterized queries. In the AXML peer implementation, any pull service can serve as a basis for a continuous service. This continuous service periodically reevaluates the query and pushes its answer. A number of additional parameters can be specified, to control how a continuous service operates:

– a frequency for the messages (e.g., daily);
– 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 or send a delta or an edit script for the changes (as in, e.g., [30]).
– a choice of a media for the changes: send a notification message, or publish the changes on the Web (possibly as an RSS feed).

Typically, the client chooses between these options by specifying parameters of the subscription call. (Note that a particular server may support only a limited subset of the options.)

The implementation of continuous services in AXML peers emulates an asynchronous behavior on top of the (synchronous) HTTP protocol. Each AXML peer exposes a dedicated "call-back" Web service whose purpose is to receive answers to any continuous services the peer has subscribed to. When a subscription call is issued, the client AXML peer automatically adds to the call some (intensional) information about its call-back service. The peer that provides the continuous service then sends its answers asynchronously to that call-back service, which integrates them at the right place (e.g., next to the corresponding subscription call).

In the Edos project (see Section 5), we need to deal with subscriptions resulting in large volumes of very large messages. In particular, the same set of large messages may have to be sent to many peers. We are currently upgrading the subscription system to include (i) the multicasting of messages and (ii) the use of BitTorrent-like approaches [21]. These techniques aim to optimize resources (bandwidth, machine load) in the distribution of such large messages: multi-casting for intelligent pushing of messages in the network, BitTorrent for parallelizing message transfers.

Peer-to-peer services

We now overview two independent extensions of the AXML peer that leverage the functionality of existing peer-to-peer infrastructures. A first extension uses the JXTA infrastructure essentially as a transport, in order to facilitate finding and invoking AXML services. The second extension provides access to the contents (documents and services) of AXML peers, by indexing them in a distributed hash table. We discuss these extensions next.

JXTA services JXTA [51] is a popular set of protocols that can be used to create a virtual P2P network. It is particularly useful to connect peers that are behind firewalls or on different networks. We essentially extended AXML peers to become JXTA peers. A JXTA-enabled peer can call both standard (AXML) Web services and JXTA services. (The URL of the service specifies the protocol that is used.) With this feature, AXML peers can take advantage of the rich functionality of JXTA. In particular, services that do not have a fixed IP address can be made accessible, as well as "generic" services, i.e., services that are supported by several peers (which involves the selection of a specific peer first). A peer that is not JXTA enabled may reach JXTA services simply by using a JXTA enabled peer as a proxy.

Distributed Hash Tables The KadoP [12] system integrates AXML and Distributed Hash Table (DHT) technology. KadoP is a system for constructing and maintaining a warehouse of resources in a P2P environment. Most importantly, KadoP maintains a distributed index of the resources that have been published up to date. These resources (XML documents, Web services, ontologies) are queried using a subset of XQuery. The system relies on a DHT implementation [38] to keep the peer network connected.

Using KadoP, AXML peers can easily discover global resources, such as Web services in peers they did not previously know. On the other hand, KadoP peers are themselves AXML peers. The dynamicity of AXML turns useful to maintain the indexing fresh. Also, part of the indexing may be intensional in KadoP. To give an example of this feature, consider a very popular word. The maintenance of the index for this word may penalize a peer that would be in charge of it. This peer may delegate some of the entries to other peers, and replace them by a service call.

In this section, we presented AXML peers from a functional viewpoint, both as clients, dynamically enriching their persistent AXML documents by materializing the service calls they contain, and as servers providing AXML services on top of their AXML documents. We now turn to the implementation of AXML peers.

4 The AXML Peer: Implementation

We now overview our implementation of AXML peers. We start with a brief overview of the technical environment, then focus on the main modules of the AXML peer. After that, we consider alternatives for the storage of persistent AXML data, and choices we provide for user interfaces.

Before that, we should note that since the AXML approach mainly relies on standards for Web services, any system that understands/implements these standards can join the AXML world. The AXML peer is one such system. It was built to fully take advantage of the AXML model by supporting both its client-side and server-side features and managing persistent AXML data. It was typically designed to run on a fixed desktop workstation, with a permanent network connection.

Of course, other implementations are possible, on different platforms, or with different objectives. For instance, we developed an AXML "lightweight" peer that is designed to run on PDAs and mobile phones [32]. Due to limited processing power and intermittent connectivity, such a system is more geared towards client-side functionality, and has limited query and update capabilities. (For instance, it supports only XPath queries.) It focuses on a mobile usage, and is able to use a standard AXML peer as a proxy, to which it delegates most resource-intensive and/or bandwidth-consuming computations. In particular, a proxy can carry out some computations (potentially involving service calls) for a lightweight peer that is currently off-line, and send it the results when it is back online.

4.1 Technical environment

We start by briefly presenting the technical environment we chose for implementing the AXML peer. Figure 9 presents the stack of technologies the system is built upon. The AXML peer essentially relies on (1) a set of widely-used public packages, and (2) X-OQL, an open-source XML query engine.

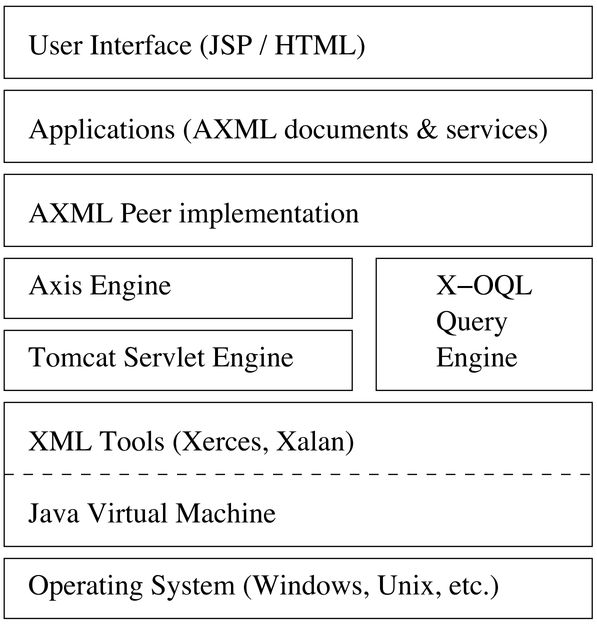

User Interface (JSP / HTML)
Applications (AXML documents & services)
AXML Peer implementation
Axis Engine
Tomcat Servlet Engine
X−OQL Query Engine
XML Tools (Xerces, Xalan)
Java Virtual Machine
Operating System (Windows, Unix, etc.)

Fig. 9 The AXML peer technology stack

The standard tools our implementation is based on are the following:

– *The Java Virtual Machine* The AXML peer is implemented in Java [48], one of the most widely used and portable programming languages.
– *XML tools* The AXML peer relies on the Apache Xerces XML parser to parse documents, and manipulate them through their Document Object Model (DOM) [35] representation. The AXML peer also uses the Apache Xalan processor for XPath queries and XSLT transformations.

- *The Tomcat servlet engine* In order to provide services accessible over the Web, the AXML Peer needs to act as a Web server. This role is fulfilled by the Tomcat engine [73], a popular open source Java servlet container.
  Tomcat is also used to provide the basis for the user interface of AXML applications. AXML documents can be turned into a Web application through Java Server Pages (JSP) [50]. A simple template library gives the developer access to the peer's AXML documents and their embedded service calls.
- *Axis* [19] is a Java toolkit that enables Web services functionality both on the server-side and the client-side. The AXML peer relies on the Axis toolkit whenever service calls need to be issued or answered, thereby ensuring that communications strictly follow Web services standards, and that the AXML peer is interoperable with other service providers and consumers.

X-OQL: As mentioned already, the AXML peer relies on the X-OQL engine [16] to execute complex queries on XML documents. The X-OQL query language roughly provides the same functionality as XQuery (with a SQL-like syntax derived from the object query language OQL).

Note that the X-OQL engine did not need to be substantially modified in order to be used by the AXML peer. Because the AXML approach is data-centric, service invocations are issued from the data, not from the programs or queries that operate on the data. As a consequence, the query languages (here, X-OQL) remain the same, and do not need to perform service invocations themselves. This data-centric approach is quite different from other techniques that extend query languages (and processors) with primitives to invoke Web services (see, e.g., [63]). In AXML, embedded service calls can be leveraged at query time through the rewriting-based lazy query evaluation we discussed in Section 2.4.

4.2 General architecture

We now give a high level overview of the AXML peer architecture. Figure 10 illustrates its main components. These can be divided into two categories: those responsible for the client-side functionality of the AXML peer (on the left) and those responsible for its server-side ones (on the right). Components that play a role in both appear in the middle of the figure.

As a server, the AXML peer receives a service request through the SOAP Wrapper, which routes it to the Service Provider. The latter finds the main Service Type to execute, and orchestrates any necessary pre and post processing of the request and the answer, which may involve executing other (local or remote) services through the Execution Engine. In case of a service with side effects, updates may be performed on the persistent AXML documents of the peer by the Persistence Manager. Finally, the answer is returned through the SOAP Wrapper.

On the client-side, a service call is activated by the Document Manager (e.g., based on a `mode` specification), and scheduled by the Execution Engine. In the case of a remote service, the invocation is carried out by the SOAP wrapper. The answer flows back (possibly asynchronously) to the Document Manager, which updates the corresponding AXML document through the Persistence Manager.

The logger module keeps a trace of important events in the AXML peer such as errors, both on the client and the server side. For example, a client-side error arises if a remote service does not respond, and a server-side failure may happen is a service is invoked with a parameter of the wrong type, or if the result of a service computation cannot be cast into the output type schema declared for the service (using the rewriting technique of Section 2.3).

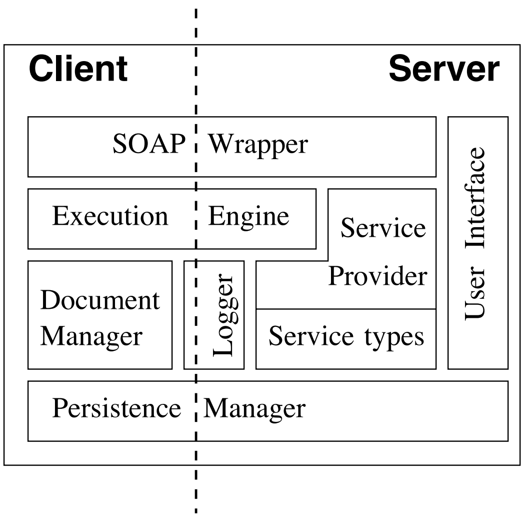


Fig. 10 The AXML peer architecture

We next discuss the main modules of the AXML peer implementation. We start with the AXML Service Provider module, which enables the server-side functionality of the AXML peer, and the extension of Axis with declarative service types. We then consider the AXML Document Manager, which provides the client-side functionality of the peer, and the Execution Engine.

4.3 The AXML Service provider

This module provides the server-side functionality of an AXML peer by answering the service requests the peer receives. This module builds on the flexible architecture of Axis for message processing, which is based on the *chain of handlers* design pattern. A chain is simply a sequence of handler invocations, where each handler is a program that performs some part of the required processing. Handlers share a common message context, which is used to pass information between them. Chains of handlers are easily configurable on a per service basis, without the need of changing the code or recompiling it, but simply by specifying in an XML configuration file which sequence of handlers is needed to provide a given service.

Besides the core functionality of a service, we saw previously that AXML often requires the use of pre or post processing, such as the lazy query evaluation technique of Section 2.4, or the schema-based rewritings of Section 2.3.(The implementation of each of these modules is detailed in [57,5].) These techniques are exposed as handlers, and incorporated in the chain of processing of services.

To illustrate, Figure 11 shows such a chain for a query service. In this example, the `AXMLRequestHandler` creates AXML documents for each parameter of the service invocation, and puts them in the shared message context. It also constructs the `XOQLService` handler corresponding to the service to be answered, or retrieves it from a cache if it has been accessed recently, and *dynamically* inserts it in the chain of handlers. Then, the `AXMLLazyHandler` performs the lazy evaluation of Section 2.4, to invoke any lazy services that are relevant to answering the query, both in the documents of the repository and in the parameters of the invocation. After that, the `XOQLService` evaluates the query corresponding to the service, and puts the result in the message context. Then, the `OutputTypeFinder` sets the target output type of the service invocation, by checking if one of the input parameters was a desired output type specified by the caller, and if not, by using the output type specified in the WSDL definition of the service. Eventually, the `TypeValidator` transforms, if possible, the result of the query into an instance of the type previously set, using the algorithms presented in Section 2.3, and the `AXMLResponseHandler` generates a SOAP response from the resulting AXML document.

Declarative service types AXML focuses on *declaratively* specified services. Therefore, Axis was extended to support such specifications. We defined different service *types*, one for each declarative language we needed to support. Each service description is an XML document which references an existing service type, defines the types of the input and output parameters of the service, and gives its declarative specification (e.g., an X-OQL query, or an XSLT transformation). Service descriptions are stored in the persistent repository, alongside AXML documents (just like stored procedures are stored in the database in a relational system). Declarative services of any service type can be used to provide the core functionality of an AXML service, and incorporated in a chain of processing, as described above.

To implement service types, we introduced a flexible mechanism that enables easily adding new types of services. To define a new service type, one only needs to register and provide classes that implement two simple interfaces:

– `IAXMLServiceFactory`: a factory that generates, from the declarative specification of the service an object complying with `IAXMLService`.
– `IAXMLService`: An object that provides an `invoke` method, to handle Web service requests.

New service types can be added to the AXML peer without recompiling it, through a simple XML configuration file. Additionally, we provide default implementations of these interfaces, which support common functionality (e.g. AXML parameter handling, partial generation of the WSDL description of the service), and can be extended by each service type implementation. The service types currently available in the system are those described in Section 3.2.

4.4 The AXML Document Manager

This module is responsible for managing the peer's persistent AXML documents and their embedded service calls. It interprets the client-side policies for the mode of activation of service calls and the validity of their data discussed in Section 3.1, essentially by registering events corresponding to these policies in the scheduler of the Execution Engine (see further). Additionally, the document manager is in charge of updating the AXML documents by merging the results of service call invocations, following the merge policies also discussed in Section 3.1.

To manage AXML documents, the Document Manager loads them in memory as DOM trees. This representation allows for easy manipulation and straightforward updates, e.g., when a service call is activated. Since most operations on AXML documents have to do with their embedded service calls, the calls are made accessible through a hash table, which uses unique, randomly generated, persistent identifiers as an access key.

Parameters of service calls may consist of arbitrary AXML data, and are recursively represented as AXML document objects. Recall from Section 3.1 that parameters can also be specified as relative XPath queries, which are just syntactic sugar for calls to services defined as queries. To speed up computation, we directly represent such parameters as query objects.

Client-side handlers Just like on the server-side, some additional processing may need to be performed on the client-side of the AXML peer, when service calls are activated. The chains of handlers mechanism discussed above is also provided by Axis on the client-side. Therefore, the same handlers can be reused. For instance, the `TypeValidator` handler is used to ensure, when calling a service, that the parameters that are transmitted match the input type specified in the WSDL description of the service.

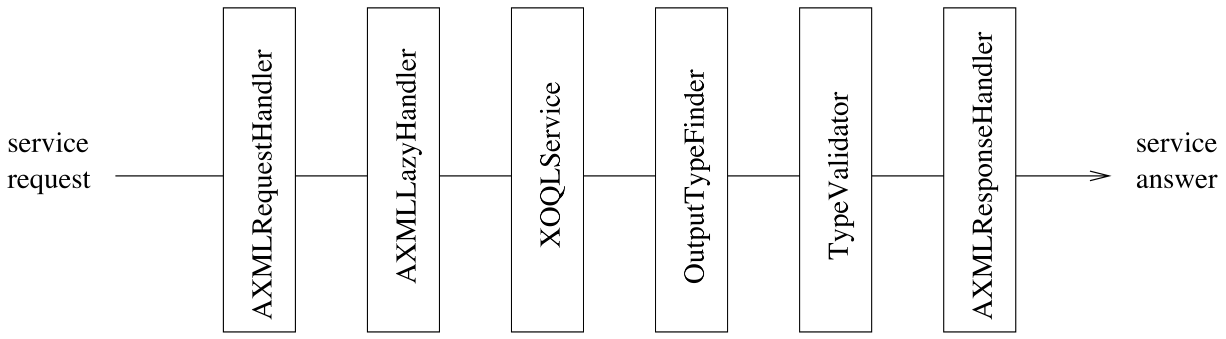


Fig. 11 Chain of handlers for a query service

4.5 The Execution Engine

This module plays a role on both the server and the client side of the AXML peer. It is in charge of scheduling all the service invocations in the AXML peer, which can either be calls to remote Web services, or to locally defined ones. In the latter case, the corresponding functions are invoked directly in Java, avoiding unnecessary (and expensive) SOAP communications.

Since Web services may take quite a long time to respond (sometimes over half a second), calling them in a blocking mode can be a serious performance bottleneck for the AXML peer. To avoid that, the execution engine must support the invocation of several services in parallel, i.e., in a non-blocking mode. The Execution Engine is multi-threaded, and uses a different thread for each service invocations. Thread management in this context is fairly complex, since some service calls may depend on others (e.g., for calls that appear in parameters of other calls) and therefore some threads need to wait for the completion of others before being started.

Moreover, some service calls need not be executed immediately, but at some fixed future time, and possibly repeatedly. This is the case for instance on the client-side, for service calls embedded in documents of the AXML peer that follow a time-based explicit pull strategy. But it is also the case on the server-side, for continuous services defined by the peer that need to be reevaluated periodically. To enable differed and periodic invocations, the execution engine relies on a scheduler, where such operations are registered. When these tasks are due, threads are created to handle them and their execution interleaves with the tasks currently being carried out by the execution engine.

4.6 Storage

The AXML peer provides two main options for making its data persistent: A storage based on a standard file-system and a more scalable mass storage based on a native XML repository. We discuss these options in turn.

Storage on a file-system This simple storage module is the default storage option of the AXML peer. It manages AXML documents, AXML service definitions, and the F-guide summaries of documents used by the lazy query evaluation of Section 2.4. Each of those corresponds to an XML file stored on disk. The persistence manager loads XML files into memory when they are needed, and is responsible for keeping the disk copy in sync with the one stored in memory whenever a document, service definition, or F-guide is updated.

Documents are manipulated through their DOM representation, which is generated by the Xerces XML parser and stored in main memory. Accessing documents this way was mainly a requirement of the X-OQL query processor, which operates at the DOM level. Storing DOM representations of documents in main memory clearly becomes an issue when AXML documents are very large or there are many of them.

XML mass storage To address the scalability issues of the storage with a file-system, we developed a variant of the AXML peer on top of a persistent XML repository. More precisely, we experimented with two mass storage XML repositories: (i) Xyleme Server [86], a native XML repository, and (ii) eXist open source XML database [37]. The work on Xyleme is reported in [34].

In both cases, the AXML peer is integrated to the repository server. Instead of being systematically loaded as DOM objects in main memory, documents remain in the XML persistent store. Information about service calls embedded in documents is maintained in memory. Queries from the XML repository are used to selectively access portions of the data, and form the core of the Web services provided by the AXML peer. They can also be used to select parameter values for service calls issued by the peer, as in Section 3.1. Similarly, updates of the XML repository serve to modify a document without having to load it entirely in main memory. The AXML peer can thereby scale to millions of AXML documents. It can serve XML or AXML documents and also take advantage of Web services embedded in the documents it stores.

One can view this work as extending the AXML peer with a mass storage functionality. However, one may also see it as introducing more dynamicity and the management of intensional information into a standard XML repository.

The two experiments were promising but the resulting systems do not cover the entire functionality of the AXML peer. The "official" AXML peer is therefore still storing data in a file system.

4.7 User Interface

While Web services are essentially meant for communications between machines, applications are generally meant for users. The AXML peer builds on the Tomcat dynamic Web serving capabilities to let developers easily build Web applications based on the peer's documents and services. Additionally, we developed custom browser extensions to interact with AXML documents and peers respectively. These user interface options are discussed next.

The AXML peer provides the developers of Web applications with a generic JSP template library for generating Web pages from the AXML documents of the peer and turning their embedded service calls into HTML links (or forms, if some parameters of the calls need to be provided by the user). The application developer simply needs to write application-specific XSLT style-sheets to define how the data contained in the AXML documents should be presented to the user.

A particular application that we developed, (and which comes with every AXML peer), is a Web-based management interface. With this interface, a user can access and modify the AXML documents of the peer, and she can access, modify the definitions of AXML services provided by the peer and test these services.

While such Web-based interfaces to the AXML peer are an effective way for users to interact with it, in some situations they are not adequate:

- First, users may receive an AXML document by e-mail, or download one from a Web site. They should be able to visualize the document, and take advantage from embedded service calls, by activating them and possibly changing the values of their parameters. We developed the *AXML browser*, an extension to the cross-platform Mozilla Firefox browser [60] that allows just that. The implementation of this component was done in Javascript and builds on the built-in capabilities of Firefox for XML manipulation (DOM parsing, XPath querying and XSLT transformations), Web services invocation (SOAP API), and custom, feature-rich user interface (XUL). A one-time, Web-based installation of this plug-in enables its use for all subsequently accessed AXML documents.
- Second, a developer or administrator may have to manage several peers at once, and should be able to do so without having to connect to the Web management interface of each individual AXML peer. We developed the *AXML commander* to that aim. This stand-alone Firefox application allows the user to manage several peers at once, by viewing/modifying their documents and services, and copying or moving contents from one peer to another, or to their local disk. Embedded service calls can be activated either locally (from the user's machine), or remotely, from the peer where the document resides. All communications with the managed peers are also performed through service calls. We enriched the AXML peer with basic Web services for downloading/uploading documents and service definitions, and activating embedded service calls. A screenshot of the AXML commander is shown in Figure 12, which illustrates the features mentioned above. For debugging purposes, exchanged SOAP messages are also displayed by the application.

Although these two applications are already usable, they still leave a lot of room for improvement. In particular, the AXML commander is still missing important features such as access control and security mechanisms.

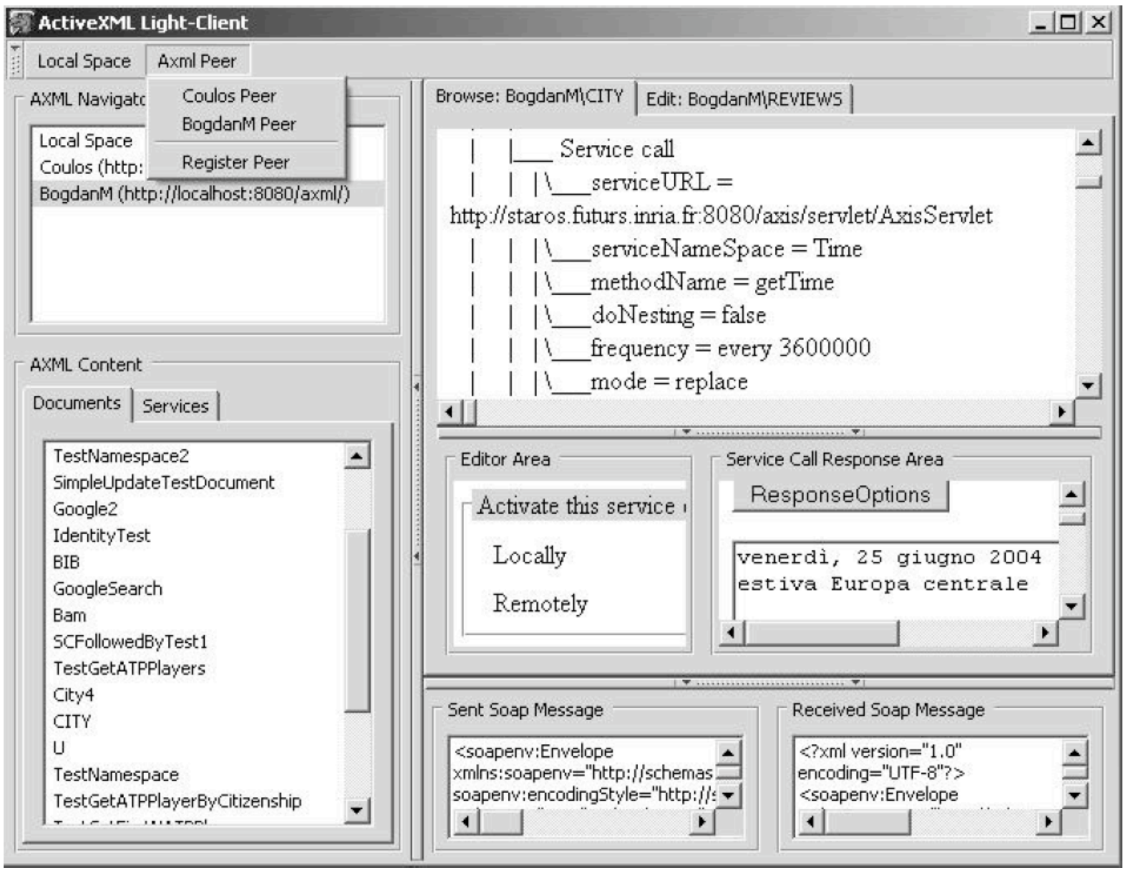
5 The AXML Peer: Experiences

We now describe some of the applications that we developed using AXML peers to validate our approach. All applications presented below were built using the features of the AXML peer we have presented. All are based on fairly small sets of AXML documents distributed among the participating peers, and on AXML services declaratively specified as queries. After describing these applications, we will summarize the main conclusions we have drawn so far from the use of AXML as a development platform.

5.1 Demonstrations

Peer-to-peer auctions This is the first application we built using AXML. It was demonstrated in [7]. The application consists of a decentralized auctioning scenario. Each peer has some items to sell, which other peers can bid on. Each peer knows about some of the other peers, and can search for items they provide. The returned answers consist both of extensional data – items matching the issued query, and intensional data, in the form of service calls to get more items from other peers the requester didn't know about originally. Each item returned by a query answer also contains a service call that can be used to bid on the item. Eventually, when an auction closes, the winner of the auction is notified.

The main goal of this application is to illustrate the flexible discovery mechanism of new peers and auctions which is made possible by the use of intensional answers,



```
|   |___ Service call
|   |   |\___serviceURL =
http://staros.futurs.inria.fr:8080/axis/servlet/AxisServlet
|   |   |\___serviceNameSpace = Time
|   |   |\___methodName = getTime
|   |   |\___doNesting = false
|   |   |\___frequency = every 3600000
|   |   |\___mode = replace
```

```
venerdì, 25 giugno 2004
estiva Europa centrale
```

```
<soapenv:Envelope
xmlns:soapenv="http://schemas
soapenv:encodingStyle="http://s
```

```
<?xml version="1.0"
encoding="UTF-8"?>
<soapenv:Envelope
```

Fig. 12 AXML commander screenshot

and the possibility to perform actions on these auctions, again through service calls which are shipped with the answers.

RSS news syndication This second application was demonstrated in [3]. It further shows the benefit of exchanging AXML data, through a scenario based on the distributed syndication and querying of news headlines. A number of news sources (including actual news Web sites, such as the New York Times, or Yahoo! news) expose their latest headlines in a standard format called RSS [67]. AXML peers act as news brokers, integrating news from several sources and/or from other news brokers. Queries asked to one peer recursively propagate to the sources and brokers it knows.

This application shows how services that aggregate news headlines can be customized, to control how intensional/extensional their answers are. This customization happens without modifying their code (in this case, a simple query on one of the peers' AXML documents). Instead, the customization is performed "declaratively", by varying the schemas used in the signatures of the services. The rewriting algorithms presented in Section 2.3 are used to automatically match the specified schemas, if possible. Such a customization can also be done by the client of the service, since the application allows her to specify a desired schema for the result as an additional input parameter. The application also demonstrates continuous services, corresponding to news subscriptions, These services are made constantly aware of the location of the user, again using intensional information controlled through types.

Electronic patient record management The goal of this application, which was demonstrated in [2], is to show that AXML can seamlessly manage distributed data and the privacy of this data, through the example of sensitive medical information. This is done by combining the AXML language with GUPster [68], a framework developed at Lucent Technologies that unifies the description of both data sources and access control requirements.

In this application, the lazy evaluation technique of Section 2.4 is at the heart of the involved computations. It is used to seamlessly push queries in a uniform way, both to relevant data sources and to Web services in charge of protecting information (by restricting the queries that can be asked).

5.2 Academic and Industrial Collaborations

Content warehousing E.dot [36] was a French (RNTL) project on content warehouse construction. The E.dot system uses an AXML peer as a platform for a warehouse definition, construction and maintenance. A high level warehouse model is specified visually through a dedicated GUI. This model is then compiled into a set of AXML documents and AXML services, which are maintained by an AXML peer. The warehouse also relies on a set of external Web services, such as the Google WebAPI, a Web crawler, a classification and clustering engine, and a PageRank module. The system was tested on a particular application, namely the construction of a warehouse on food risk management. In this application, the warehouse takes advantage of external resources provided in the form of Web services, e.g., relational databases containing measurements and two domain ontologies. The warehouse is dynamically enriched with data discovered on the Web. This data is semi-automatically analyzed and semantically enriched.

The declarative aspect of the approach is most useful in a content warehouse context, because the designers and users of the warehouse are typically non computer scientists. For the particular application on food risk management, they were biologists from the French National Institute of Agronomy, and had some experience in knowledge management. They were very happy with the tools developed in the project although there is clearly a lot of room for improvements. One issue is that tools to sequence the interaction with the warehouse (in the spirit of workflow systems) were viewed as missing, which motivated the work on workflow extension to AXML (see Section 3.2). Another issue was scalability, which motivated the work on mass storage for the AXML peer (see Section 4.6).

Telecom applications This ongoing joint work with telecom specialists at INRIA Rennes finds its motivation in the Swan project between Alcatel, INRIA, and France Telecom. One idea is that software and hardware components get information on how to configure or reconfigure themselves in the form of AXML data, where parts that are likely to change are provided intensionally, in order to seamlessly account for these changes. The use of both push and pull modes of invocation is envisioned, and AXML data should also prove useful to make components dynamically aware of their environment. Also, we have used AXML in the context of telecom network diagnosis. In short, each component of the system is modeled as a communicating Petri net. The possible runs are captured by the unfolding of the Petri nets, constrained by the observed traces of execution. The diagnosis problem may then be stated as a recursive query on a large volume of distributed data. This motivated the use of AXML and a distributed query optimization technique for AXML presented in [1].

Distribution of Mandriva Linux This ongoing work is performed in the context of the Edos European project, which aims at better controlling the production and distribution of Open Source software. Active XML is used for the collective management of software packages and their metadata by the developers.

Package locations in the network and package metadata are indexed and may be queried using the AXML-based KadoP system, built on top of a distributed hash table. Sets of packages located through such queries can be obtained by developers using BitTorrent-based parallel downloading. Also, developers may subscribe to distribution channels. When some update occurs or some new version is released, subscribers are notified and may automatically get new packages. Active XML services provide the main communication functions in the system: subscription, notification, querying, download activation.

In the application that is considered (Mandriva Linux), there are thousands of developers sharing thousands of packages, some of them quite large, in a P2P environment.

5.3 Lessons learned

Overall, the outcome of using the AXML language and system to develop the applications described above has been very positive. Thinking of distributed interactions at a *data* level, in terms of (i) documents with embedded calls, and (ii) local computations encapsulated as Web services undeniably brings a higher level of abstraction, and enables more declarative thinking than developing the same applications at a *code* level, e.g., in Java.

However, developers are more used to traditional programming paradigms, therefore the shift to AXML requires some practice. To take an analogy with relational databases, using SQL and deciding which parts of the processing should be formulated as queries requires some practice. Analogously, learning how to "design" an AXML application requires specific skills. The development of conceptual modeling tools suitable for the AXML paradigm could help address this issue. The visual warehouse model developed for the E.dot project can be viewed as a first step in that direction.

Another lesson we learned is that for some applications, the AXML approach was just not adequate. Intuitively, this always happened when the task to perform were not "data-oriented" but "process-oriented", e.g. when a precise sequence of operations needs to be carried out, or branching is involved. When such a behavior is well isolated in a single peer, the natural solution is to code the corresponding functionality in a program, and to expose this program as a Web service, which can be invoked in AXML documents. In case of a more global process-oriented interaction, approaches in the spirit of workflows, such as BPEL [24] are more adequate. As a first step, we already mentioned that service compositions defined in BPEL can be specified in an AXML peer. Clearly, more work is needed to fully understand the necessary interaction with workflow systems.

This discussion of the applications we built and the lessons we learned from them concludes our presentation of the AXML peer system. In the next section, we take a more theoretical point of view to study the interactions between *several* AXML peers. We consider global issues such as termination, and distributed query evaluation.

6 AXML systems

A peer-to-peer *AXML system* involves several AXML peers communicating with each other via service calls. It is important, in such a distributed environment, to understand how the peers interact and to be able to verify the properties of the overall system. For instance, we may want to test for the termination of some global task or globally optimize the usage of resources.

To analyze the behavior of such a distributed system, we proposed a formal foundation for AXML systems. Clearly, the general problem is complex, so we have limited our theoretical study to a restricted, yet powerful enough model. We next briefly review this formal model and theoretical results obtained in [6]. We then illustrate how important issues like query optimization in a peer-to-peer context can be revisited in the light of this formal model.

6.1 Positive AXML

In [6], we consider a simple model where AXML documents are modeled as unordered labeled trees with two kinds of nodes, data nodes and function nodes (the latter represent calls to Web services). The semantics of a document is defined as the tree obtained at the limit of an arbitrary *fair* sequence of service invocations, i.e., a sequence where any service call that may be materialized will eventually be. Once fired, each service call remains active. This is meant to capture a peer-to-peer data management based on streams of data (both in pull and push mode).

In that work, we consider only monotonic Web services. The documents containing the calls are monotonically enriched by the answers. This is in the style of peer-to-peer computations a la Kazaa [52], where data (e.g., the results of a search) is incrementally collected in a distributed network of peers. Observe that non-monotonic service calls do occur in practice. However, they are hard to envision in such a setting: one can never assume that a fact is false, since this fact may be stated in some parts of the network not investigated yet. Ignoring order in XML documents and non-monotonic services are serious limitations and more powerful models are also clearly worth investigating.

We showed in [6] that if services are monotonic, then the semantics of a document is possibly infinite but unique (up to some equivalence relation), i.e., that this semantics is independent of the order of service invocations. So, monotonicity guarantees some form of confluence. On the other hand, monotonicity does not guarantee termination. A call to a service may activate a call to another service, and so on, possibly recursively. Also, a service may return as an answer some data including new service calls, which may in turn return more data including more calls, etc. Therefore, monotonic computations may never terminate.

We went further in the analysis of AXML systems where the semantics of service calls is known at analysis time. Because of this knowledge, the analysis of the systems may tell us more about their properties. In particular, we considered a class of AXML documents and services, which we call *positive AXML*, where Web services are defined using a monotonic query language that corresponds to a core tree-pattern fragment of XQuery. Because of the recursion between documents and services, we still obtain an important expressive power. In particular, a large class of Turing machine computations can be simulated, and therefore termination is undecidable for such systems. We have isolated a subclass of these systems based on simple queries, with interesting properties. In short, *simple queries* are obtained by disallowing the use in tree pattern queries of variables ranging over subtrees of documents. Intuitively, such variables can be used to copy subtrees of arbitrary complexity. We showed that a number of interesting properties are decidable for positive systems using simple queries only. For instance, termination is decidable when AXML services are defined by simple queries. The proofs are based on tree automata.

6.2 Distributed query evaluation

We focused in Section 2.4 on the problem of optimizing a single query in isolation (in a single peer). In fact, the evaluation of a query in an AXML system may involve the evaluation of many service calls, and therefore of many queries at different peers. Optimization now becomes a global issue, i.e., we want to optimize (for the entire system of queries) a number of resources such as communication, total execution time or response time. This is taking us to the general setting of distributed database systems [71] and many techniques developed there are relevant to our context.

Of particular importance (as in distributed database systems in general) is the replication of data and service calls on various peers, to speed up the evaluation of query services. This issue is investigated in [8] where in particular, the selection of which services and data to replicate is considered, given some workload and some cost model. To be able to support such distributed query optimization, we rely on one main primitive, namely the possibility to install a document or a service at a particular peer. This will be best illustrated with an example.

Suppose we want to perform at peer P_1 a join of three relations, say R_1 stored locally, and R_2, R_3 stored, respectively, at peers P_2 and P_3. P_1 can get R_2, R_3 and perform the join locally. Alternatively, P_1 could, for instance, install at P_2 a service that would compute the join of the two remote relations and call that service. To compute the same join with a semi-join technique, one would have to install more services on the various peers. This is illustrating the installation of services. It is often interesting to also be able to store data at remote peers, e.g., to cache results for re-use. This is achieved by installing AXML documents.

The AXML peer was extended to allow the installation of documents or services at remote peers (assuming appropriate access rights). This capacity, together with query rewriting, transforms AXML into a middleware system appropriate for supporting distributed query processing. Details on the use of replication of data and services to speed up the evaluation of a query in an AXML system may be found in [8].

To conclude this section, we mention two works around distributed query optimization that are founded on AXML. The first is an extension of the query-subquery technique from deductive databases [75], whereas the second is a proposal of an algebra for distributed data management that extends AXML.

AXML and deductive databases

We developed a distributed query optimization technique for positive AXML systems, namely AXML-QSQ [1]. The core of the technique consists in pushing queries to other peers (acting as servers) and passing information sideways between peers. A difficulty here is that recursion is built in the system via service calls. This is clearly reminiscent of issues encountered in (distributed) deductive databases and query optimization techniques developed in those contexts such as query-subquery (QSQ) [75] and Magic Sets [74]. In these two last works, the focus is on materializing only “relevant” tuples to answer a query over a datalog program. Similarly, AXML-QSQ generalizes QSQ so that only relevant service calls are activated (in the entire AXML system) and furthermore, that queries are pushed when possible to also limit the quantity of information that is materialized.

To illustrate the AXML-QSQ technique, suppose that we have some peers P_i $(i \geq 1)$. Each peer P_i stores locally a list of songs that can be accessed via a service $MySongs_i$. Peer P_i exports a list of songs with a service $Songs_i$. $Songs_i$ is defined as the union of $MySongs_i$ and $Songs_{i+1}$. Now suppose someone at peer P_1 wants to know about songs by Carla Bruni. To do so, she needs to evaluate $\sigma_1(Songs_1)$ where $\sigma_1 = \sigma(\text{singer="Carla Bruni"})$. The selection is pushed to $MySongs_1$, which is a simple local selection, and to $Songs_2$, i.e., it leads to evaluating $\sigma_1(Songs_2)$ at P_2. Next, we are lead to evaluate $\sigma_1(Songs_3)$ at P_3, etc., so σ_1 is pushed to more and more distant peers. Observe also that this is a place where continuous queries (mentioned in Section 3 and supported by the formal model described above) are necessary because P_2 can start sending answers to P_1 before she is done with computing $\sigma_1(Songs_2)$. Indeed, in case of cyclic computation the use of continuous queries is compulsory. Otherwise, the computation may never terminate. The issue of detecting termination is critical in this setting; see [1].

6.3 A distributed algebra based on AXML

As data management applications grow more complex and distributed, query processing also becomes more complex and distributed. Furthermore, the juxtaposition of different computing models prevents reasoning on the application as a whole, and wastes important opportunities to improve performance. It is argued in [13], that distributed query evaluation should be based on the evaluation of queries expressed in a common *algebra*, performed by collaborating query processors, installed on different peers and exchanging streams of XML data. We would also like query optimization to be performed in a distributed manner by algebraic query rewriting.

Relational calculus or SQL provides a logical language for centralized table data. By analogy, AXML can be seen as providing a *logic for distributed tree data.* [13] proposes an extension of AXML as an *algebra for distributed tree data.* They introduce two main extensions to AXML: (i) generic data and services and (ii) a more explicit control of the distribution. *Generic* data and services are data and services available on several sites, an essential feature to capture replication and the fact that a query service may be evaluated by any peer with query processing facilities. The AXML algebra also provides the capability to explicitly control the shipping of data and queries (explicit *send* and *receive* operators), an essential feature to specify the delegation of computations.

7 Related work

It should be observed that the AXML approach touches upon a number of important areas of the database field. Clearly it is based on semi-structured data. We already mentioned connections with deductive databases. The activation of some calls contained in a document essentially leads to active databases. AXML services may be in push mode, which relates to subscription queries and stream databases. We consider previous works from these areas, and mention previous works on workflow and peer-to-peer systems.

Data integration AXML was primarily developed as a framework for the integration of distributed, autonomous sources. Data integration systems typically consist of data sources, which provide information, and of mediators or warehouses, which integrate it with respect to an integration schema, typically using mapping rules, see, e.g., [39,54,55] In fact, AXML takes an hybrid path between mediator systems (the integration is virtual) and warehouses [43] (all data is materialized). Significant attention has also been devoted to "semantic mediation" [17,44], where the integration involves an ontology. By contrast to these schema-based or ontology-based approaches, mappings between data sources are captured in AXML by service calls embedded in the data, with new relationships discovered at run-time, in the answers of service calls, somewhat in the style of the dynamic discovery of mappings in [44]. The AXML approach is, in some sense, complementary to schema-based approaches. For instance, tools developed for schema/data translation and semantic integration can be wrapped as Web services and used within an AXML framework.

Deductive databases The presence of service calls that can be materialized, and this recursively, makes AXML documents *intensional* in the sense of deductive databases. Intensional information was mostly studied in a relational setting, in the context of Datalog, a language for deductive databases [11,58]. An important question in the context of intensional information is that of the optimization of query evaluation. We explained how some techniques from deductive databases have been adapted to the AXML context.

Data with embedded calls As already mentioned, the idea of embedding function calls in data is not new. Embedded functions are already present in relational systems [74] as stored procedures. They are indeed proposed as first-class datatypes in [70]. Of course, method calls form a key component of object databases [28]. In the Web context, scripting languages such as PHP [65] or JSP [50] have made popular the integration of processing inside HTML or XML documents. Combined with standard database interfaces such as JDBC and ODBC, functions are used to integrate results of queries (e.g., SQL queries) into documents. A representative example for this is Oracle XSQL [64].

XML with embedded calls Service calls in semi-structured data are considered in the context of Lore and Lorel [46, 53]. Our work is also related to Microsoft Smart Tags [66], where service calls can be embedded in Office documents. In Smart Tags, this is meant mainly to enrich the user experience by providing contextual tools, while our goal is to provide means of controlling and enriching the use of Web service calls for data management purposes, in a distributed setting. Other systems, proposed languages based on XML with embedded calls to Web services, like Macromedia MX [31], or Apache Jelly [49]. Our results can be used in such settings.

Active databases and triggers The present work is, in some sense, a continuation of previous works on *ActiveViews* [4]. There, declarative specifications allowed for the automatic generation of Web applications where users could cooperate via data sharing and change control. The main differences with ActiveViews are that (i) AXML promotes peer-to-peer relationships vs. interactions via a central repository, and (ii) the cornerstones of the AXML language are XPath, XQuery and Web services vs. object databases [28]. The activation of service calls is also closely related to the use of triggers [74] in relational databases, or rules in active databases [77]. Active rules were adapted to the XML/XQuery context [22] and the firing of Web service calls [23] was considered. Our work goes beyond those by promoting the exchange of AXML data.

Service choreography The integration and composition of Web services is an active field of research [76]. Before Web services appeared, intensional data was used in Multilisp [45], under the form of "futures", i.e., handles to results of ongoing computations, to allow for parallelism. Ambients [27,26], as *bounded* spaces where computation happens, also provide a powerful abstraction for processes and devices mobility on the Web. Standard languages for service workflow have even been proposed such as IBM's Web Services Flow Language [80] or Microsoft's XLang [81], which converged into the BPEL4WS proposal [24]. An overview of existing works can be found in [47]. In the terminology of this latter work, the focus of AXML is not on workflow or process-oriented techniques [29] but more on *data* flow.

Peer-to-peer Peer computing is gaining momentum as a large-scale resource sharing paradigm by promoting direct exchange between equal peers, see [52] or [59]. We propose a system where interactions between peers are at the core of the data model, through the use of service calls. Moreover, it allows peers to play different roles, and does not impose strong constraints on interaction patterns between peers, since they are allowed to define and use arbitrary Web services. While we do not consider issues such as dynamic data placement, distributed indexing, or automatic peer discovery, solutions developed in the peer-to-peer community for such problems can benefit our system as well as enrich it.

In a p2p context, security is a critical issue. To conclude, we would like to mention some works around AXML in that direction. In [25], a new framework based on the notion of trust (Trusted AXML) is proposed for handling security in Active XML. In [2,10], security and access control are considered in the context of AXML.

8 Conclusion

The relational model and the SQL query language have been important breakthroughs for data management in centralized information systems. With the advent of the Web, new data management paradigms are being considered, to master an inherently distributed planet-scale information system. Semistructured data, and its standard incarnation XML, are being accepted as the suitable model for data exchange over the Web. XQuery, a query language for XML promoted by the W3C, is often advertised as the *SQL of the Web*. However, XQuery only allows asking queries on centralized collections of documents, and does not capture the distributed essence of the Web. Thus, XQuery cannot pretend alone to serve as a comprehensive language for Web data management. Based on XML, Web services and XQuery, we proposed in this paper a first step towards such a language for Web data management.

Active XML is based on XML with embedded calls to Web services. These embedded calls allow AXML documents to integrate external information. With AXML services and the exchange of AXML data, new perspectives open for dynamic collaboration among peer systems on the Web.

We mentioned a number of works that have been performed around AXML. In particular, we showed how to use schemas to control the exchange of AXML data. We mentioned techniques, some related to deductive databases, to efficiently answer queries in this context. We briefly discussed some work on managing distributed and replicated (A)XML documents. Finally, we presented the implementation of a system for supporting AXML in a peer-to-peer environment. The system is now distributed as Opensource software [20] within the ObjectWeb framework [62].

AXML opens up an array of challenging and exciting issues, notably in concurrency control, query optimization, access control, security, workflow management or fault tolerance. Research is needed on these various fronts. Now from an AXML application viewpoint, one would like to consider each such *aspect* separately in the style of aspect-oriented programming [18]. For instance, one would prefer to ignore concurrency control issues when designing a data management application, or security issues when considering concurrency control. Of particular importance is the issue of distributed query optimization. As already mentioned, an extension of AXML has been recently proposed as an algebra for distributed management [13]. Encouraged by experiences in query optimization involving AXML, in particular [8,1,5], we believe that this is a promising direction. Clearly a lot remains to be done.

Acknowledgments The authors would like to thank all the people who participated in the Active XML project: Zoe Abrams, Bernd Amann, Vikas Bansal, Jérôme Baumgarten, Angela Bonifati, Bogdan Cautis, Gregory Cobena, Cosmin Cremarenco, Frederic Dang Ngoc, Eric Darondeau, Florin Dragan, Ioana Manolescu, Bogdan-Eugen Marinoiu, Radu Pop, Nicoleta Preda, Shobhit Raj Mathur, Gabriela Ruberg, Nicolaas Ruberg, Emanuel Taropa, Minh Hoang To, Gabriel Vasile, Dan Vodislav, Boris Vrdoljak and Roger Weber.

In particular, many ideas in the present paper were influenced by discussions with Ioana Manolescu who co-authored several of the papers on AXML that were cited.

References

1. Abiteboul, S., Abrams, Z., Milo, T.: Diagnosis of Asynchronous Discrete Event Systems - Datalog to the Rescue! In: Proc. of ACM PODS (2005)
2. Abiteboul, S., Alexe, B., Benjelloun, O., Cautis, B., Fundulaki, I., Milo, T., Sahuguet, A.: An Electronic Patient Record on Steroids : Distributed, Peer to Peer, Secure and Privacy Conscious (demo). In: Proc. of VLDB (2004)
3. Abiteboul, S., Amann, B., Baumgarten, J., Benjelloun, O., Ngoc, F.D., Milo, T.: Schema-driven Customization of Web Services (demo). In: Proc. of VLDB (2003)
4. Abiteboul, S., Amann, B., Cluet, S., Eyal, A., Mignet, L., Milo, T.: Active Views for Electronic Commerce. In: Proc. of VLDB (1999)
5. Abiteboul, S., Benjelloun, O., Cautis, B., Manolescu, I., Milo, T., Preda, N.: Lazy Query Evaluation for Active XML. In: Proc. of ACM SIGMOD (2004)
6. Abiteboul, S., Benjelloun, O., Milo, T.: Positive Active XML. In: Proc. of ACM PODS (2004)
7. Abiteboul, S., Benjelloun, O., Milo, T., Manolescu, I., Weber, R.: Active XML: Peer-to-Peer Data and Web Services Integration (demo). In: Proc. of VLDB (2002)
8. Abiteboul, S., Bonifati, A., Cobena, G., Manolescu, I., Milo, T.: Dynamic XML Documents with Distribution and Replication. In: Proc. of ACM SIGMOD (2003)
9. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann Publishers, San Francisco, California (1999)
10. Abiteboul, S., Cautis, B., Fiat, A., Milo, T.: Digital signatures for modifiable collections. In: ARES, pp. 390–399 (2006)
11. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley Publishing Company, Reading, Massachusetts (1995)
12. Abiteboul, S., Manolescu, I., Preda, N.: Constructing and Querying Peer-to-Peer Warehouses of XML Resources. Workshop on Semantic Web and Databases (2004)
13. Abiteboul, S., Manolescu, I., Taropa, E.: A framework for distributed xml data management. In: EDBT, pp. 1049–1058 (2006)
14. Abiteboul, S., Milo, T., Benjelloun, O.: Regular and Unambiguous Rewritings for Active XML. In: Proc. of ACM PODS (2005)
15. The Active XML homepage. `http://activexml.net/`
16. Aguilera, V.: The X-OQL homepage. `http://www-rocq.inria.fr/∼aguilera/xoql`
17. Amann, B., Beeri, C., Fundulaki, I., Scholl, M.: Querying XML Sources Using an Ontology-Based Mediator. In: Proc. of CoopIS (2002)

18. The Aspect-Oriented Software Development homepage. http://aosd.net/index.php
19. The Apache Axis soap Engine. http://xml.apache.org/axis
20. Open Source Active XML. http://forge.objectweb.org/projects/activexml/
21. The BitTorrent Homepage. http://www.bittorrent.com
22. Bonifati, A., Braga, D., Campi, A., Ceri, S.: Active XQuery. In: Proc. of ICDE (2002)
23. Bonifati, A., Ceri, S., Paraboschi, S.: Pushing Reactive Services to XML Repositories using Active Rules. In: Proc. of the Int. WWW Conf. Hong Kong, China (2001)
24. Business Process Execution Language for Web Services Version 1.1. Available from http://www.ibm.com/developerworks/library/ws-bpel/
25. Canaud, E., Benbernou, S., Hacid, M.S.: Managing trust in active xml. In: IEEE International Conference on Service Computing (2004)
26. Cardelli, L.: Abstractions for Mobile Computation. In: Secure Internet Programming, pp. 51–94 (1999)
27. Cardelli, L., Gordon, A.D.: Mobile Ambients. In: M. Nivat (ed.) Proc. of FoSSaCS, vol. 1378, pp. 140–155. Springer-Verlag, Berlin, Germany (1998)
28. Cattell, R.G.G.: The Object Database Standard: ODMG-93. Morgan Kaufmann, San Mateo, California (1994)
29. Christophides, V., Hull, R., Kumar, A., Siméon, J.: Workflow Mediation using VorteXML. IEEE Data Engineering Bulletin **24**(1), 40–45 (2001)
30. Cobena, G., Abiteboul, S., Marian, A.: Detecting Changes in XML Documents. In: Proc. of ICDE (2002)
31. Macromedia Coldfusion MX (2004). http://www.macromedia.com/software/coldfusion/
32. Cremarenco, C.: Implementation of the Active XML Peer for the J2ME platform (2003). Internship report, available from http://purl.org/net/axml
33. Damiani, E., di Vimercati, S.D.C., Paraboschi, S., Samarati, P.: Securing XML Documents. In: Proc. of EDBT (2001)
34. Active XML and Xyleme. Mḿoire CNAM (2005)
35. The Document Object Model (DOM). http://www.w3.org/DOM/
36. RNTL Project E.dot, Content Warehouse open to the Web. http://www-rocq.inria.fr/ amann/edot/
37. exist, an open source native XML database. http://exist.sourceforge.net/
38. The FreePastry Distributed Hash Table. http://freepastry.rice.edu
39. Garcia-Molina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., Widom, J.: The TSIMMIS Approach to Mediation: Data Models and Languages. Journal of Intelligent Information Systems **8**, 117–132 (1997)
40. The Gnutella homepage. http://www.gnutella.com
41. Goldman, R., Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: Proc. of VLDB (1997)
42. Gupta, A.: Integration of Information Systems: Bridging Heterogeneous Databases. IEEE Press (1989)
43. Gupta, H.: Selection of Views to Materialize in a Data Warehouse. In: Proc. of ICDT, pp. 98–112 (1997)
44. Halevy, A.Y., Ives, Z.G., Mork, P., Tatarinov, I.: Piazza: data management infrastructure for semantic web applications. In: Proc. of the Int. WWW Conf. (2003)
45. Halstead, R.: Multilisp: A Language for Concurrent Symbolic Computation. ACM Trans. on Programming Languages and Systems **7(4)**, 510–538 (1985)
46. Hugh, J.M., Abiteboul, S., Goldman, R., Quass, D., Widom, J.: Lore: A Database Management System for Semistructured Data. Tech. rep., Stanford University Database Group (1997)
47. Hull, R., Benedikt, M., Christophides, V., Su, J.: E-services: a look behind the curtain. In: Proc. of ACM PODS (2003)
48. The SUN Java Technology. http://java.sun.com/
49. Jelly: Executable XML. http://jakarta.apache.org/commons/sandbox/jelly
50. SUN's Java Server Pages. http://java.sun.com/products/jsp/
51. JXTA. http://www.jxta.org/
52. The Kazaa Homepage. http://www.kazaa.com
53. Lahiri, T., Abiteboul, S., Widom, J.: Ozone: Integrating Structured and Semistructured Data. In: Proc. Int. Workshop on Database Programming Languages (1999)
54. Levy, A., Rajaraman, A., Ordille, J.: Querying Heterogeneous Information Sources Using Source Descriptions. In: Proc. of VLDB, pp. 251–262 (1996)
55. Manolescu, I., Florescu, D., Kossmann, D.: Answering XML queries over heterogeneous data sources. In: Proc. of VLDB (2001)
56. Microsoft and IBM: The WS-Security specification. http://www.ibm.com/webservices/library/ws-secure/
57. Milo, T., Abiteboul, S., Amann, B., Benjelloun, O., Ngoc, F.D.: Exchanging intensional XML data. In: Proc. of ACM SIGMOD (2003)
58. Molina, H., Ullman, J., Widom, J.: Database Systems: The Complete Book. Prentice Hall (2002)
59. The Morpheus homepage. http://www.morpheus-os.com
60. The Mozilla Firefox Browser. http://www.mozilla.com/firefox
61. Muscholl, A., Schwentick, T., Segoufin, L.: Active Context-Free Games. In: Proc. of STACS (2004)
62. ObjectWeb, Open Source Middleware. http://forge.objectweb.org/
63. Onose, N., Simeon, J.: XQuery at your web service. In: Proc. of the Int. WWW Conf., pp. 603–611. ACM Press, New York, NY, USA (2004). DOI http://doi.acm.org/10.1145/988672.988754
64. Oracle XML Developer's Kit for Java. http://otn.oracle.com/tech/xml/
65. The PHP Hypertext Preprocessor. http://www.php.net
66. Powell, J., Maxwell, T.: Integrating Office XP Smart Tags with the Microsoft .NET Platform. http://msdn.microsoft.com (2001)
67. RSS 1.0 Specification. http://purl.org/rss/1.0
68. Sahuguet, A., Hull, R., Lieuwen, D., Xiong, M.: Enter Once, Share Everywhere: User Profile Management in Converged Networks. In: Proc. of CIDR (2003)
69. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP
70. Stonebraker, M., Anderson, E., Hanson, E., Rubenstein, B.: QUEL as a data type. In: Proc. of ACM SIGMOD (1984)
71. T. Özsu and P. Valduriez: Principles of Distributed Database Systems, 2nd Edition. Prentice-Hall (1999)
72. Tatarinov, I., Ives, Z., Levy, A., Weld, D.: Updating XML. In: Proc. of ACM SIGMOD (2001)
73. The Apache Tomcat Servlet Container. http://jakarta.apache.org/tomcat
74. Ullman, J.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)

75. Vieille, L.: Recursive axioms in deductive databases: The Query/Subquery approach. In: L. Kerschberg (ed.) Proc. First Intl. Conf. on Expert Database Systems, pp. 179–193 (1986)
76. Weikum, G. (ed.): Special Issue: Infrastructure for Advanced E-Services, vol. 24, no. 1 (2001)
77. Widom, J., Ceri, S.: Active Database Systems: Triggers and Rules for Advanced Database Processing. Morgan Kaufmann Publishers (1996)
78. Wiederhold, G.: Intelligent Integration of Information. In: Proc. of ACM SIGMOD, pp. 434–437. Washington, DC (1993)
79. Web Services Definition Language (WSDL). `http://www.w3.org/TR/wsdl`
80. Web Services Flow Language (WSFL 1.0). Available from `http://www.ibm.com/`
81. XLANG, Web Services for Business Process Design. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c`
82. The Extensible Markup Language (XML) 1.0 (2nd Edition). `http://www.w3.org/TR/REC-xml`
83. The XML Schema specification. `http://www.w3.org/TR/XML/Schema`
84. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`
85. XSL Transformations (XSLT) Version 1.0. `http://www.w3.org/TR/xslt`
86. Xyleme Home Page. `http://www.xyleme.com`