Construction and Maintenance of a Set Of Pages of Interest (SPIN) using ActiveXML

Serge Abiteboul — Grégory Cobéna — Benjamin Nguyen — Antonella Poggi

INRIA
Domaine de Voluceau
78153 Le Chesnay CEDEX
FRANCE
Email: Firsname.Lastname@inria.fr

RÉSUMÉ. *Dans cet article, nous nous intéressons à la construction d'entrepôts dynamiques à partir de ressources du Web, en utilisant notamment des services disponibles sur le web. Notre contribution tient essentiellement à la définition d'une nouvelle architecture basée autour de services web (e.g. SOAP) et du language ActiveXML, nous permettant de résoudre notamment les problèmes suivant: (i) l'acquisition de pages du Web, (ii) le contrôle des changements de ces pages et (iii) l'enrichissement des données et méta-données par l'utilisation de services Web. Le système est développé en ActiveXML, un langage et un système permettant l'intégration d'appels à des services Web au sein d'un document XML.*

ABSTRACT. *In this article, we examine the problem of constructing a temporal data warehouse using web services. There are many important aspects in the construction of such a warehouse. Our particular contribution in this article regards the global architecture of a system that can (i) acquire specific pages from the web (ii) control page changes (iii) easily be enhanced using various web services. In order to present a concise architecture, the whole system was designed using ActiveXML, a language and a system based on the embedding of web service calls into XML documents.*

MOTS-CLÉS : *Entrepôt de données, XML, Bases de données semi-structurées, Données temporelles, Services Web*

KEYWORDS: *Data warehousing, XML, Semi-structured Data Bases, Temporal Data, Web Services*

1. Introduction

The number of pages available on the web is increasing, and with it, the quantity of information. A classical problem is that of maintaining a collection of web resources (URLs) of interest for a particular community. Important aspects are (i) the acquisition of pages (from the web or from users), (ii) the control of changes, (iii) the enrichment of data of meta-data, (iv) the storing of the collection and (v) the support of topic specific query mechanisms. In this paper, we are concerned with the first three aspects. More precisely, we present a framework to support the contruction of sets (collection) of pages related to a given interest called SPIN (for *Set of Pages of INterest*).

The SPIN apprach is based on ActiveXML [ABI 02], a language and a system that allows the embedding of calls to web services in XML documents. The ActiveXML language is simple yet provides an extremely powerful form of data-centric distributed computation with XML and XML query languages as cornerstones.

The starting point of the present work is the strong belief that the construction of thematic collections (a very common task today, for instance in thematic portals) should be based on a declarative specification. SPIN designers specify the perimeter of their collections through a formal declaration named the *intension* of the SPIN. Our intensional definition of SPIN (based on ActiveXML) is a first step in that direction. It facilitates the definition, deployment and maintenance of such thematic collections. In the specification, the designer may rely on simple functionalities such as full-text search and navigation. We will see that the specification may support much more complex needs.

Once the intension of a SPIN has been defined, it is the responsibility of the system to construct the *extension*, i.e., the collection of resources that match the specification. (One may want to think of it as a collection of URLs or as a collection of actual pages.) The information around these pages may be enriched (as specified in the intension) with additional meta-information, such as the last date of change, the size of the page, the MD5 signature of the page, its page rank, the number of links pointing to it, etc. The SPIN designer controls when such an extension is computed and may request to have it be computed regularly, e.g., weekly.

An important characteristic of the SPIN approach is that new functionalities may easily be added because of the ActiveXML basis, under the assumption that the new functionalities are provided as web services. These functionalities enable many features such as :

– the integration of "real time" information, e.g., based on subscription services that may push resources of interest to a particular SPIN.

– the enrichment of the content of the collection with some processing, e.g., one may use a classification service to classify all the documents according to a particular hierarchy.

An important aspect of the SPIN system is its management of change. The system can detect changes that occurred both inside pages of the collection, but also in the collection itself (new pages). Furthermore, it can detect changes in meta-data obtained by processing the collection, e.g., detect if the classification of a page changed. The change detection system is base on the XML diff algorithm described in [COB 02], a diff tool that computes changes in a labeled tree. By using this algorithm and the successive versions of a SPIN, we can produce data (reports) that describe the changes between versions.

The main contributions of the paper are :

1) the use of ActiveXML as the core language of our system. The *intensions* of collections are defined in ActiveXML.

2) a library of web services also called SPIN to support specific features of SPIN management (users management, web crawler,...)

3) an architecture that enables the construction of such collections using ActiveXML servers, standard services available on the web, and the SPIN library.

ActiveXML is a peer-to-peer system. One could imagine the collection being distributed over several web sites that cooperate in building it. This aspect is somewhat orthogonal to the specification of the collection (the topic of this paper). We will ignore it here. It remains nonetheless an essential asset of our approach.

In the following section, we present a motivating example for our system. In Section3, we present a short state of the art, and the tools used in our system. In Section 4, we describe the basic data model, and the core services, using the ActiveXML formalism. In Section 5, we detail some other advanced features that can be added by using the appropriate web services. In Section 6, we detail the temporal aspects of the system, and follow up with the explanation of how this is all integrated. We conclude by mentioning experiments performed with our first prototype.

2. Motivation

In this section we present motivating examples for the use of our system. We start by presenting basic examples. We then describe the running example used throughout the paper.

We first consider three simple scenarios. These are actually directly supported by existing software. We will show in the following sections how all these examples can be generalised in a larger unified framework, SPIN. The advantage of our system is that these examples can all coexist in a much larger context supporting a variety of other scenarios and that they can very easily be specified in a declarative manner.

– **A web-master wants to detect broken links** in the Web site he maintains. For instance consider the set of pages whose URLs begin with *www-rocq.inria.fr/* (INRIA, Rocquencourt). He wants to fetch pages, parse the files to detect links, and fetch the links in a recursive fashion while recording URLs that have been already processed and those that are broken. Such a service can easily be specified, using our declarative language. Furthermore, one may enrich the list of all the pages (as it is constructed) with information such as the type of the page (html, pdf, etc) or the appearance of certain keywords.

– **The person in charge of a portal may want to monitor [NGU 01] a web site** or a set of web pages on a specific topic, and be informed when pages are added, or updated. Such a service can again be specified in a concise way. Let us stress that the use of ActiveXML makes the integration of all the web services called much simpler. The user of the system can easily improve its possibilities, e.g., processing the newly discovered pages with tools that enable their automatic translation in a given language or their classification according to a particular ontology.

– **A user wants to search the web using several search engines (meta search) :** The goal is to integrate results from various search engines. (See, e.g., Kartoo [KAR].) It is also straightforward to specify such an application using SPIN. It is easy to “tailor” this integration and add some specific processing of the resulting set of URLs, e.g., searching for synonyms.

As we can see, the requirements differ greatly from one application to another. We propose some *basic* functionalities, and a simple yet expandable platform that integrates them.

Motivating example

We next introduce a running example that will be used in the entire paper. Suppose we want to maintain the collection of web pages of interest for the inhabitants of a particular city, for instance Sèvres in France. The first web sites that come to mind on this topic would be the city official web site, or the local theater web-site. To find other interesting web-pages, we may use search engines and keyword search. At this point, the interaction between our system and the users is critical : they may add sites of interest, provide new keywords that should (or should not) be investigated, give positive or negative feedback on specific web sites and pages. In our case, a user may, for instance, specify that a page containing the keywords `Sèvres` and `92310` (the zip code of Sèvres) is potentially interesting, whereas a page containing `Deux-Sèvres` (a French department) is probably not.

Once we have the collection of documents, the work is not yet over. Many more tasks are left, such as : classifying the pages according to a given ontology, indexing pages, using the links inside pages of interest to obtain more pages that could also be interesting.

Finally, it is very important to keep the collection up to date, and possibly archive some of it regularly, i.e., perform change control.

3. Context

In this section, we first present ActiveXML, that plays a central role in our work. We then briefly discuss the topic of Web Portal creation, that presents a number of common goals with SPIN support.

The choice of ActiveXML We decided to use ActiveXML. For a complete overview of ActiveXML, we refer to [ABI 02]. We could have based our system on a relational or object database model, or used a semi-structured language such as Ozone [LAH 99] or Microsoft Smart Tags [POW]. Our choice of XML and web services is more than simply following a trend. First of all, XML documents do not suffer the limitations of strict typing. They can be easily enriched ; the flexible typing aspect of XML is crucial here. Secondly, the interaction with the users (via web browsers) and with web functionalities such as search engines or crawlers are much simpler in ActiveXML than with relational or object databases. Finally, ActiveXML fits nicely into the web context, where the system should not stop running if a service call fails, or if another one is very slow in returning its answer. Nonetheless, many ideas presented here could be transposed to an ODMG warehouse with a little work (methods encapsulating web services, etc.) or to a relational context.

Web Portals Constructing web portals is an important and complex task that presents a number of common goals with the construction of SPINs. Many companies, such as IBM, with Websphere Portal Service [Web] or Sybase (Enterprise Portal) [Syb] propose complex systems to build Web Portals. In domains such as e-business, or surveillance systems, other companies also offer specific and complicated solutions. Our proposal features a very simple system, based on ActiveXML, that also simplifies the task of building web portals. As it will be shown, the declarative description of a SPIN takes only a few lines of code. Thanks to its modular approach, our system is highly adaptive and is not limited to a specific type of application.

Web Services It should finally be observed that more and more services will be offered on the web that can participate in SPIN construction. For instance, the SPIN library includes methods to access search engines via the SOAP protocol. Recently, Google started providing such a service. Indeed, it is very likely that the low level services supported by SPIN will soon be available on the web. SPIN should be viewed as the glue needed to combine them. To continue with new services found on the web, Google now offers news search [Goo]. This is in some sense a SPIN service : it is an access to a particular collection of pages, the news headlines. The SPIN system goes further, since it lets users define their own interests and interesting sites, as well as the functions to perform on them.

Why SPIN ? The goal of this work is to present a global and generic framework that enables the creation of many different sorts of applications. While most commercial systems are aimed at a specific task, it is our belief that many simple web services exist on the web, and that by using them within the correct architecture we can have a simple to use yet highly modular and efficient system.

4. Specifying the collection

In this section we give a brief description of the basics of the data model. We first present the intension, then the extension of the SPIN. Finally, we turn to the presentation of web Services

4.1. *Data Model*

A *data-warehouse* consists of a header, and a certain number of *spins*. Each SPIN consists of an *intension*, an *extension* and some service definitions. The warehouse is seen as an ActiveXML document. As mentioned earlier, an ActiveXML document may use external services (via service calls) that may appear anywhere in the document. To simplify, in the SPINs we consider here the service calls will be concentrated in the service definitions. In general, an ActiveXML document may also offer (to the external world) a number of services[1] that are defined using queries (typically in XOQL[AGU], Xquery or XPATH) and updates, of which we will also give some examples. The specification of a SPIN for Sèvres may look like this :

```
<spin:warehouse name="Sèvres">
<spin:head>
  <spin:owner id="Serge" />
  <spin:title>Sèvres Warehouse</spin:title>
  <spin:accessControlList>
   <spin:access group="friends" mode="call"/>
   <spin:access group="all" mode="read"/>
  </spin:accessControlList>
</spin:head>
<spin:spin name="sevres">
  <spin:intension> ... </spin:intension>
  <spin:extension> ... </spin:extension>
  <spin:services> ... </spin:services>
</spin:spin>
<spin:spin name="sevres-sculpture">  ... </spin:spin>
</spin:warehouse>
```

This first part of the SPIN (the header) describes the owner of the SPIN, its general title, and the access control list. In this case, all users are allowed to view the contents of the SPIN, but only the users in the `friends` list can call the services provided by our warehouse. Now consider the intension of a particular SPIN :

```
<spin:spin name="sevres">
  <spin:intension>
   <spin:bound>3000</spin:bound>
   <keywords>
```

1. These services may be restricted by only letting specific access groups to use them.

```
    <keyword>Sèvres</keyword>
    <keyword>92310</keyword>
   </keywords>
   <interestingSites>
     <site>http://www.ville-sevres.fr/</site>
     <site>http://www.vertsdesevres.com/</site>
   </interestingSites>
  </spin:intension> ...
</spin:spin>
```

The `intension` subtree contains all the data (parameters) used by the services to perform the operations that construct the SPIN. The `spin :bound` element indicates the maximum number of URLs we want to have in the extension. The intension also provides a list of keywords `Sèvres` and `92310`, and a list of interesting sites such as *http ://www.ville-sevres.com/*. Note that *keywords* and *interestingSites* are not part of the namespace `spin`. Different SPINs may use different services and thus different kinds of data in the intension. The only constraint is that the intension must contain *all* the parameters needed by the services that will be called.

Note that strictly speaking the specification of a SPIN is made up of more than simply its intension : We also need to know the web services that use this intension. These will be detailed later.

Consider now one extension :

```
<spin:extension date="31 jul 2001">
  <spin:url id="http://www.mysite.com/mypage.html">
    <content>...</content>
    <link>http://www.yahoo.com/</link>
    <link>http://www-rocq.inria.fr/</link>
    <type>HTML</type>
    <last_update>28 jul 2001</last_update>
    <classification>Group1 (Resume)</classification>
    <site>http://www.inria.fr/</site>
  </spin:url>  ...
</spin:extension>
```

For one SPIN, there may be many extensions, each one computed at a given date. This is captured by the `date` attribute of the `spin :extension` node. For each resource (URL), various information are recorded for each page. Again, exactly *what* information depends on the definition specification of web services. Thus, in the extension also, we are not limited to predefined attributes. The data model is fully extensible, and any kind of (meta) data may be added to the entry defining a page. The only attribute that must be present for a `spin :url` element is the (unique) identifier `id` of the page, which is its URL. All other attributes or child nodes are not part of the basic schema of SPIN. Therefore not all the meta data stored in the extension is part of the `spin` namespace, since it is service dependent. In general, each node will be prefixed with the namespace of the service that provided the information.

4.2. *Web services*

Consider now web services. Calls to web services in ActiveXML are also syntactically in XML as in :

```
  <axml:sc methodName = "myWebService">
    <axml:params>
     <axml:param name="param1">
      XML data or XPATH expression
     </axml:param>
     ...
    </axml:params>
   </axml:sc>
```

ActiveXML service calls return an (Active)XML document. Note that a number of features in ActiveXML allow to control the firing of the call, and the duration of the validity of the data that is obtained. We will ignore this aspect here.

Let us consider how we can obtain URLs of interest using data from the intension. This is done as follows using two services, *askGoogle* and *getSite* :

```
<spin:services>
% Keyword Query
let askGoogle($name) be {
for each $X in
<axml:sc name="http://www.google.com/googleSearch">
  <axml:params>
    <axml:param name="keyword"
                xpath="self//spin:spin[name=$name]
                       /keywords" />
  </axml:params>
</axml:sc>
do insert (self//spin:spin[name=$name]
           /spin:extension/<spin:url id=$X>)
}

% Interesting sites
let crawlInterestingSites($name) be{
for each $X in
<axml:sc name="http://www.myservices.com/getSite">
<axml:params>
  <axml:param name="url"

xpath="self//spin:spin[name=$name]/spin:intension
       /interestingSites/site/" />
  <axml:param name="depth">5</axml:param>
  <axml:param name="bound"
              xpath="self//spin:spin[name=$name]
```

```
                     /spin:intension/spin:bound/"
/>
</axml:params>
</axml:sc>
do insert (self/spin[name=$name]/spin:extension
           /<spin:url id=$X
opinion="yes">)
}
</spin:services>
</spin:warehouse>
```

Explenation :

– The first query calls a (pseudo) Google service `www.google.com/googleSearch/` passing to it lists of keywords coming from the intension (XPATH query). The results (a list of URLs) are then inserted into the extension of the SPIN as nodes named `<spin :url>`. The attribute `id` contains the URL string. Note that this is done by side effect of the call.

– The second query uses a webservice that retrieves all the pages that can be reached starting from a page and following at most 5 links not exiting from the given site. The starting pages are obtained in the `//spin :spin/spin :intension/interestingSites/` subtree. The `bound` parameter indicates that we do not want to retrieve more than a certain number of pages. If we go over that limit we stop crawling. The list of URLs are included into the extension in the same way as before, but each URL discovered this way also has an attribute `opinion="yes"` which indicates a higher degree of confidence in the quality of the URLs discovered this way.

5. Advanced SPIN Features

We may add as many features to the warehouse as we want. The only limit is that these features must be provided by web services. The parameters used by these more complex service calls need to be written in the intension definition, so that they may be accessed with a simple path query.

Classification

Let us imagine we have a classification service that given a hierarchy of classes (defined with example web pages for instance) and a new web page, classifies the web-page into the hierarchy. We need to first of all define the classes that we are going to encounter in the SPIN. We define them staticaly here ; but these classes could also be obtained by a service call. The classes could for instance be defined in the intension of the SPIN as follows :

```
<spin:intension>
...
 <classification>
```

```
  <art> <music/> <cinema/> <theatre/> ... </art>
   <sports>
    <individual-sports><tennis/>...</individualSports>
    <collectiveSports><football/>...</collectiveSports>
   <sports>
  <history> ... </history>
   ...
 </classification>
...
</spin:intension>
```

We now define a service that uses these parameters to run on a set of pages already in the extension, and add to each page the attribute `class`, which contains the class of the hierarchy that the document belongs to.

```
<spin:services>
let getClassInfo($name) be {
select self//spin:spin[name=$name]/spin:extension/
       <spin:url id=$X ><class>$X</class></spin:url>
from $X in
     <axml:sc name="http://www.class.com/">
        <axml:params>
          <axml:param name="urlList" >
          select $Y
          from $Y in self//spin:spin[name=$name]
               /spin:extension/spin:url/@id</axml:param>
          <axml:param name="classHierarchy"
            xpath="self//spin:spin[name=$name]
                   //classification" />
        </axml:params>
      </axml:sc>
  }
...
</spin:services>
```

User annotations

In this paragraph we give the example of a service that lets users annotate each web page. The code for the service is the following :

```
<spin:services>
...
% Enter a user’s opinion
let addUserOpinion ($ID, $opinion, $text, $class) be {
<spin:url id=$ID>
  <opinion>$opinion</opinion>
  <comment>$text</comment>
  if $class != "" then <class>$class</class>
</spin:url>
}
...
</spin:services>
```

This service takes as input parameters the ID (url) of a page and the opinion (yes/no), comment (a string) and class (a string) that a user thinks the URL should have.

Using other web-services

The previous example shows how to construct the most basic data warehouse. The power of using ActiveXML in the definition of our system lies in the fact that an experimented user will be able to write out his own ActiveXML programs to enhance the semantics of his data warehouse. Up to now, the only temporal aspect of the data warehouse consisted in the dates that appear in the `<spin :extension date=@D>` nodes. In the next section, we show how we manage the data over time in more detail.

6. Temporal Aspects of SPIN

We have defined an efficient architecture for web-services using ActiveXML. It is important to note that ActiveXML is data-based, and that the stored data plays an essential role in the application. We detail in this section the version archiving process and version management of SPIN. Note that the way the system discovers that a page has changed, and the management of loading these pages is beyond the scope of this paper. We refer to [NGU 01] for more details on the topic.

Archive

With ActiveXML, data returned by a web service call is stored in the AXML document itself. Our system can be used for services mediation, proxy/caching and archiving by setting the proper validity and periodicity flags. For instance, a simple web archive is built using a crawling service with a validity time set **forever**. In more complex applications, the archiving process consists in several steps, in particular content filtering, page classification or indexing.

We show how to query the archived data. Then, we will show a simple way to compress the archive by using a *diff*[COB 02] service.

Querying the AXML Document

An AXML document is first of all an XML document and contains data. Thus, it can be queried using an XML query language, such as XQuery [W3C] or XOQL [AGU , AGU 00]. For example, users may query the document to find a page in the archive. Also the SPIN application itself may use queries to retrieve access rights from inside the AXML document, and manage access control using stylesheets in the spirit of [GAB 01]. Queries are also used in the AXML document itself to extract parameters for service calls.

Further ideas

For more elaborate queries on AXML documents, a semantic mediation module is clearly needed. A possible solution is to use an external service, like Xyleme [Xyl] to (i) provide an abstract view of the domain (ii) retrieve mapping rules (iii) translate abstract queries into concrete queries. More work is clearly needed in that direction.

When AXML documents become larger, an index structure is needed to accelerate queries. It is possible, for instance, to store AXML documents in a native XML repository. Another solution consists in creating an simple index inside the AXML document itself (by using an external web service). This is in the spirit of Latex or Microsoft Word that have a function to generate such an index at the end of a document. In this case, the index can also be made available to the user.

Delta-based compression

We consider here a data warehouse that is updated on a regular basis. It may be an archive, or a caching proxy for instance. The delta-based compression is used on documents for which several versions are stored. The goal is to save storage resources by storing only changes between two consecutive versions of documents instead of storing both documents. In [MAR 01], several storage strategies are presented. For instance, one may want to store only the latest version of the document, and all backward deltas from a version to the previous one. Then, we are able to reconstruct any older version by applying the sequence of deltas to the document.

In some applications (e.g. editing XML documents), the changes applied to a document are known. But in most cases, the set of changes is unknown, and a *diff* algorithm is necessary to detect changes between two consecutive versions [COB 02, DEL]. These services can be integrated into two calls :

```
DetectChanges(XMLdoc V0, XLMdoc V1)
    returns XMLdelta <v0v1> ;
ApplyDelta(XMLdoc V0, XMLdelta d-v0v1)
    returns XMLdoc <v1> ;
```

In our system, these services are provided by XyDiff [COB 02]. Based on XyDiff, we defined a simple aggregation service for SPIN. It receives two XML subtrees representing different versions of a document, and returns an AXML document containing the first subtree, a delta, and a service call to reconstruct the second subtree. The nice thing with AXML is that embedded service calls are transparent to the user. Thus, the result returned by the aggregation service is virtually identical to the subtrees passed in arguments. We show in the `aggregate` function definition how to use these two web services.

Using AXML validity settings, the compression service can be used when a new document is stored, or on a regular timing (e.g. every week). The AXML definition of the aggregation service is as follows :

```
let aggregate($name, $D1, $D2) be {
insert self//spin:spin[name=$name]
       /spin:extension[date=$D1]/
       <delta from=$D1 to=$D2>
```

```
         ... %the delta
        </delta>
        </spin:extension>
        self//spin:spin[name=$name]
        /spin:extension[date=$D2]/
        <axml:sc name="applyDelta">
         <axml:params>
          <axml:param name="from"
                xpath="../spin:extension[date=$D1]"
          />
          <axml:param name="delta-loc"
                xpath="../delta[from=$D1 && tp=$D2]"
          />
         </axml:params>
         <validity>CLONE VALUE</validity>
         <refreshPolicy>ON DEMAND</refreshPolicy>
        </axml:sc>
        </spin:extension>
delete self//spin:spin[name=$name]
        /spin:extension[date=$D2]
}
```

Depending on the XML *diff* tool used, the aggregation service has different features. **XyDiff**, for instance, is very fast to compute and well suited for large AXML documents. Many other XML diff tools would take hours to diff files of a hundred kilobytes.

The change detection service may also be used to monitor changes [NGU 01], for instance, to detect that an important page contains new links.

7. Architecture and Experiments

We present in this section the architecture of the project as well as implementation status and some experiments.

The architecture is described in Figure 1. The SPIN application stores and queries data through the XOQL-based XML repository. The core of SPIN relies on web services integration by the AXML processor. The processor will process the AXML document that describes the data warehouse that we wish to construct, by calling the various services described previously in the paper, and shown in Figure 1. The proposed architecture contains four web services, but more can be added (e.g. classification). Let us note that the XOQL, XyDiff, Xyleme services already existed, and were integrated into the architecture, by making them AXML compliant. Other web services (e.g. Crawler) are built from Unix applications using wrappers. Our ongoing implementation effort is to wrap useful web services to the AXML standard, in order to be able to integrate them afterwards in our framework.

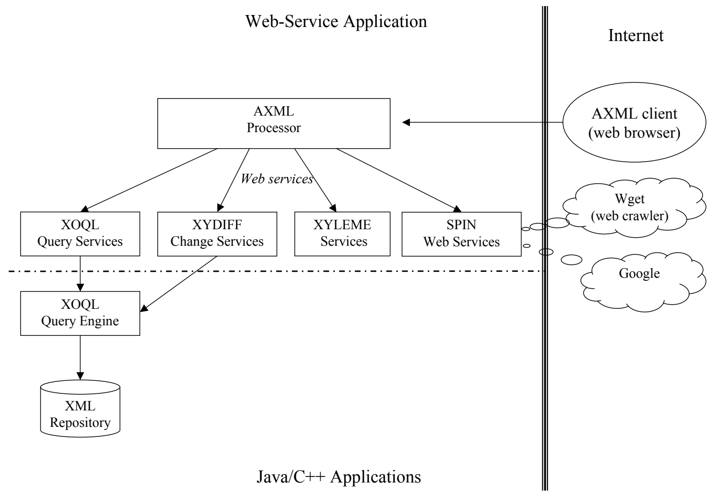

Figure 1. *Architecture*

The user interface of our system consists in the module named “AXML client” in Figure 1. It is based on a browser and uses dynamic HTML pages generated from XML documents using style-sheets (by Cocoon) and the servlet processor Tomcat. The pure-data XML documents are obtained using a light AXML processor that evaluates only the services required for the display.

Experiments and Implementation Status Some modules like XOQL or XyDiff are fully functional, though not yet completely integrated into web services. Both XOQL and XyDiff have been developed as part of an earlier project and have been intensively tested/used since. The ActiveXML processor is also implemented, we are now working on extending its features, in particular the security aspects. We also designed a prototype application with SPIN services, including a lightweight web crawler and a wrapper to Google[2]. These modules have been connected to both XyDiff and XOQL to compute deltas, compress the archive, and send notification reports to users.

For instance, we constructed the SPIN of *www.inria.fr* web-sites that contains over 13.000 web pages, and maintained it during several weeks to see new pages and updated pages. The corresponding XML document weights several megabytes.

To conclude this section, it is important to observe that the extension of the SPIN library becomes each day easier. First, more and more web services become available, that the library may use. Also, tools like GLUE [Min] can now be used to quickly create web services, e.g., from java programs.

2. This was before Google SOAP interface was made available.

8. Conclusion and perspectives

In this article, we presented a simple and expandable system, based on web services, that constructs and maintains a data warehouse over time. The architecture is modular, thus enhancing the warehouse with application specific modules is very simple. We believe this system can make up the core of many data warehousing applications, and provide the basic services that are needed in order to maintain such a warehouse. The use of ActiveXML provides an easy means of adding new functionalities to the warehouse, provided these services are encapsulated by a SOAP protocol.

Acknowledgments We would like to thank Vincent Aguilera, Omar Benjelloun, Laurent Mignet, Tova Milo, Michalis Vazirgianis, Beiting Zhu for discussions on the topic.

9. Bibliographie

[ABI 02] ABITEBOUL S., BENJELLOUN O., MILO T., MANOLESCU I., WEBER R., « Active XML : A Data-Centric Perspective on Web Services », *Bases de Données Avancées*, 2002.

[AGU] AGUILERA V., « X-OQL Query Language for XML », www-rocq.inria.fr/˜ aguilera/xoql/.

[AGU 00] AGUILÉRA V., CLUET S., VELTRI P., VODISLAV D., WATTEZ F., « Querying XML Documents in Xyleme », *Proceedings of the ACM-SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, july 2000.

[COB 02] COBENA G., ABITEBOUL S., MARIAN A., « Detecting Changes in XML Documents », *ICDE*, 2002.

[DEL] DELTAXML, « DeltaXML : Change Control for XML in XML », www.deltaxml.com.

[GAB 01] GABILLON A., BRUNO E., « Contrôles d'accès pour documents XML », *Bases de Données Avancées*, 2001.

[Goo] « http ://www.google.com/news/ ».

[KAR] KARTOO, « www.kartoo.com/ ».

[LAH 99] LAHIRI T., ABITEBOUL S., WIDOM J., « Integrating Structured and Semi-Structured Data », *Int. Worshop on Database Programming Languages*, 1999.

[MAR 01] MARIAN A., ABITEBOUL S., COBENA G., MIGNET L., « Change-centric Management of Versions in an XML Warehouse », *VLDB*, , 2001.

[Min] MIND ELECTRIC, « GLUE », www.themindelectric.com/glue/index.html.

[NGU 01] NGUYEN B., ABITEBOUL S., COBENA G., PREDA M., « Monitoring XML data on the Web », *Proceedings of the ACM-SIGMOD*, , 2001.

[POW] POWELL J., MAXWELL T., « Integrating Office XP smart tags with the Microsoft .NET Platform », http ://msdn.microsoft.com/.

[Syb] « Sybase Enterprise Portal », http ://www.sybase.com/products/ep/.

[W3C] W3C, « XQuery », www.w3.org/TR/xquery.

[Web] « IBM Web Sphere », http ://www.ibm.com/websphere/.

[Xyl] « Xyleme S.A. », http ://www.xyleme.com/.