A Lightweight Architecture for RSS Polling of Arbitrary Web sources

Sergio Bossa, Giacomo Fiumara and Alessandro Provetti
Dip. di Fisica, Università degli Studi di Messina
Sal. Sperone 31, I-98166 Messina, Italy
sergio.bossa@gmail.com, {fiumara,ale}@unime.it

***Abstract*— We describe a new Web service architecture designed to make it possible to collect data from traditional plain HTML Web sites, aggregate and serve them in more advanced formats, e.g. as RSS feeds. To locate the relevant data in the plain HTML pages, the architecture requires the insertion of some meta tags in the commented text. Hence, the extra mark-up remains totally transparent to users and programs. Such annotated HTML documents are then routinely pulled by our Web service, which then aggregates the data and serves them over several channels, e.g. RSS 1.0 or 2.0. Also, a REST-style Web Service allows users to submit XQuery queries to the feeds database. Finally, we discuss scalability issues w.r.t. polling frequencies.**

I. Introduction

This article describes a new, experimental architecture for automated data collection and RSS delivery of data from traditional HTML Web sites. Our solution requires minimal and totally transparent changes on their HTML pages. The data of interest will be routinely *polled* from the actual sources by standard HTTP querying. Subsequently, the so-created Web service can be queried with REST-style sessions that extract the aggregated data at their wish. As a result, we provide a complete layout for the implementation of RSS Web services that interact with the traditional Web in an almost seamless way.

Even though this research project is only at the beginning, and only a proof-of-concept implementation is available, we believe that there is room for the application of this type of approach to bridging Web services and the traditional Web. Let us discuss why. Today we find on the Web several interesting, popular news sites that consist, essentially, of plain old HTML pages. Even though the content is continuously updated, the site layout and organization is not changing much. Several advanced techniques for news broadcasting and syndication are now available, the main one being RSS feeds, yet it seems that a large set of relevant news sources will carry on *by inertia,* with their existing Web architecture. Our architecture enables extracting the relevant data from plain HTML and makes it available to the contemporary Web service techniques.

Indeed, today Web portals are publishing, along with traditional HTML pages, RSS documents, mostly known as RSS feeds [1], [2].

Inasmuch as HTML is aimed at content visualization for end user experience, RSS is an XML format aimed at capturing channels of data items, thus enabling automated data processing. RSS today is used mainly for content syndication; it organizes the semantics in a *channel* element, containing overall information regarding the resource, and a set of *item* elements, each containing logically related pieces of information. Moreover, every channel or item contains a *title* element, a *link* element and a *description* element.

Even though it has been developed for syndication purposes, RSS can be applied to realize sophisticated forms of content manipulation, like aggregation or advanced querying. Using RSS feeds is indeed simple: Web portals must publish, together with HTML documents, the related feeds. Users can then *consume* these feeds by a particular client, called RSS aggregator, by which they can read, query or *aggregate* feeds.

However, this simple process has some limitations: Web masters have to create their RSS feeds by some RSS generation tool, which are often proprietary and may limit interoperability. Moreover, users may not be able to view older feeds, nor to query feeds *on the fly,* directly on the server.

The architecture described here[1] overcomes these limitations by proposing a *pull-based* Web service to generate, store, aggregate and query contents using RSS standards. With this application it is possible to:

- Automatically and dynamically generate RSS feeds starting from HTML Web pages
- Store them in chronological order
- Query and aggregate them thanks to REST [4], [5] Web services acting as software agents

Clearly, there are scalability issues involved in our architecture, and the pulling policy for each site must be carefully considered. Section VI-B below describes a common structure for pulling policies.

II. Adding meta-tags to existing HTML pages

HTML documents contain a mixture of information to be published, i.e., meaningful to humans, and of directives, in the form of tags, for graphical formatting, i.e. intended for browsers interpretation. Moreover, since the HTML format is designed for visualization purposes only, its tags do not allow sophisticated machine processing of the information contained therein.

[1]The architecture was first outlined in the first author's graduation project [3].

Among other things, one factor preventing the spread of the Semantic Web is the complexity of extracting, from existing, heterogeneous HTML documents machine-readable information. Although our project addresses only a fraction of the Semantic Web vision, our management of HTML documents needs some technique to locate and extract some valuable and meaningful content.

Therefore, we define a set of annotations in form of meta-tags, which can be inserted inside an HTML document in order give it semantic structure and highlight informational content. In our application, meta-tags are used as annotations, to describe and mark all interesting information, in order to help in the extraction and so-called *XML-ization* phases. The set of meta-tags we defined (and recognizable by our application) is listed in Table I below. The meta-tags are enclosed in HTML comment tags, so they remain transparent to Web browsers and do not alter the original HTML structure of the document.

The conceptual model of the meta-tags described above is rather straightforward and remains orthogonal to the object tags found in the page.

A. Meta tags vs. dynamic XSLT transformations

An obvious alternative to our approach to the treatment of existing HTML structures is that of applying, after the polling phase, some clever XSLT transformation [6] to the HTML file. It should be considered, however, that applying such type of XSLT transformations is possible (or at least greatly facilitated) only when the [X]HTML document is well-formed. This, regrettably, seems rather unrealistic to us, exp. for old documents. Viceversa, our solution relieves the webmasters from any time-consuming translation of her HTML documents into well-formed XHTML ones, which would then make a subsequent XSLT transformation successful.

III. STRUCTURE OF THE XML OUTPUT

Once HTML documents are processed by our application, annotated semantic structures are extracted and organized into a simple XML format which will be stored and used as a starting point for document querying and transformation. This XML format has been simply called *XMLData*. This *neutral* format has also been introduced in order to avoid storing the same information in both RSS 1.0 and 2.0 formats. Indeed, we found more economic for our application to create RSS feeds *on the fly* rather than store them. This approach is also more flexible as the support of new syndication formats (see for example, the Atom format) does not require the re-design of the lower levels of the application (see further). The structure of the XML output resembles the structure of meta-tags previously defined and the RSS XML structure, in order to facilitate transformations from the former to the latter. It is defined by the following XML Schema Definition [7]:

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

<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="unqualified">

<xsd:complexType name="imageType">
  <xsd:all>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:element name="link" type="xsd:anyUri"/>
    <xsd:element name="url" type="xsd:anyUri"/>
  </xsd:all>
</xsd:complexType>

<xsd:complexType name="extensionsType">
  <xsd:sequence>
    <xsd:any namespace="##any"
     processContents="skip"
        minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="channelType">
  <xsd:all>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:element name="link" type="xsd:anyUri"/>
    <xsd:element name="description"
     type="xsd:string"/>
    <xsd:element name="image" type="imageType"
        minOccurs="0" maxOccurs="1"/>
    <xsd:element name="extensions"
     type="extensionsType"
     minOccurs="0" maxOccurs="1"/>
  </xsd:all>
</xsd:complexType>

<xsd:complexType name="itemType">
  <xsd:all>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:element name="link" type="xsd:anyUri"/>
    <xsd:element name="description"
     type="xsd:string"/>
    <xsd:element name="image" type="imageType"
        minOccurs="0" maxOccurs="1"/>
    <xsd:element name="extensions"
     type="extensionsType"
        minOccurs="0" maxOccurs="1"/>
  </xsd:all>
  <xsd:attribute name="index" type="xsd:integer"
   use="required"/>
  <xsd:attribute name="id" type="xsd:string"
   use="required"/>
</xsd:complexType>

<xsd:complexType name="resourceType">
  <xsd:sequence>
    <xsd:element
     name="channel" type="channelType"/>
    <xsd:element
     name="item" type="itemType"
        minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="url" type="xsd:anyUri"
   use="required"/>
  <xsd:attribute name="rssId"
   type="xsd:string" use="required"/>
  <xsd:attribute name="timestamp"
   type="xsd:dateTime" use="required"/>
</xsd:complexType>

<xsd:element name="resource" type="resourceType">
  <xsd:key name="itemId">
    <xsd:selector xpath="item"/>
    <xsd:field xpath="@id"/>
  </xsd:key>
  <xsd:key name="itemIndex">
    <xsd:selector xpath="item"/>
    <xsd:field xpath="@index"/>
  </xsd:key>
```

Meta-tag	Description
<channel:title> ... </channel:title>	Channel title
<channel:description > ... < /channel:description>	Channel description
<channel:image url=" link=" title=" />	URL, link and title of an image associated to the channel
<channel:extension uri=" prefix="> ... </channel:extension>	Channel extension (e.g., publication date)
<item:link index="> ... </item:link >	item link
<item:description index="> ... </item:description >	item description
<item:extension uri=" prefix="> ... </item:extension >	item extension (e.g., item publication date)

TABLE I

THE SET OF META-TAGS

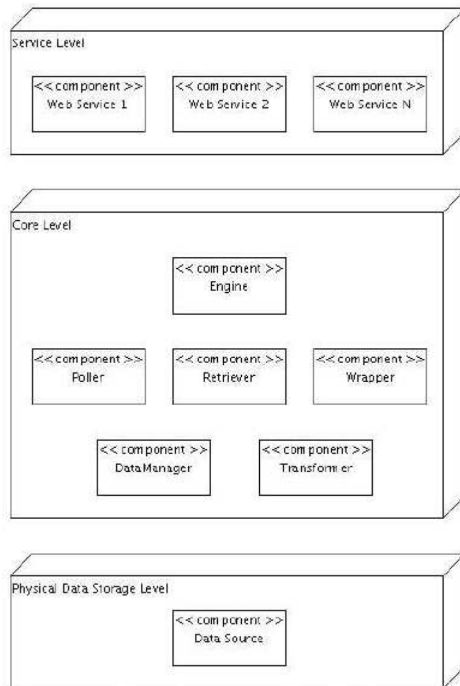

Fig. 1. The general schema

```
    </xsd:element>
</xsd:schema>
```

IV. THE OVERALL APPLICATION ARCHITECTURE

Figure 1 shows the overall architecture of the application. Our application is based on a modular structure, for maximizing the flexibility and the extensibility of configuration. Three levels can be distinguished:

- Physical Data Storage Level It is the lowermost level, which stores resources, and provides a means for retrieving and querying them. It can be implemented in various ways, using also established technologies like relational or XML database[8].
- Core Level. It holds the core part of the entire architecture, including the software components which implement the logic of information management and processing; each component can be implemented using different strategies or algorithms, and plugged into the system without affecting other components, i.e., by simply tuning the application configuration files.
- Service Level. It is the highest level, interacting with Web clients by means of REST Web services.

A more detailed explanation follows, starting from the Core level, the foundation over which our application is based.

A. The Core Level

The Core level is composed by several components defining how the application i) retrieves HTML resources, ii) processes to extract information about channeling, iii) manages this new piece of information and finally iv)transforms and prepares it for client consumption:

- **engine:** the code that routinely invokes the Retriever and thus the whole polling process.
- **Poller**: it monitors changes in a set of HTML resources configured in a particular file, using some polling policy (see next section). Moreover, the poller has the important task of coordinating other components in the retrieving, extraction, and storing phases.
- **Retriever**: when invoked by the Poller, it captures the Web resource from its URL and makes it available to other components.
- **Wrapper**: it takes care of extracting the annotated semantic structures from the retrieved HTML resources, wrapping them in a new one, that is, assembling the extracted structures in a fresh, pure XML format, containing the desired informational content: the previously-described *XMLData* format. So, this component must produce a well formed XML document, ready to be stored by the Physical Data Storage level.
- **DataManager**: it acts as a gateway to the Physical Data Storage level, taking care of managing information in the form of the new XML documents previously created, storing them and permitting client components to query their contents.
- **Transformer**: it finally takes care of transforming the stored XML documents into the RSS format requested by clients, using XSLT transformations.

Typical parameters of this level can be changed simply modifying the corresponding parameters which are listed in some configuration files, in XML format. The configuration file of the Engine Component, for example, allows to set the type of polling policy of the Web resources. Currently, the choice is between *flat,* i.e, constant over time, or *smart,* i.e.,

depending on the recent rate of updates. Other parameters are: the type of data manager (currently, the Exist native-XML database together with its connection parameters) and the format of the RSSs sent to Dynamo subscribers (currently RSS1 and RSS2).

B. *The Physical Data Storage Level*

The Physical Data Storage level can be implemented with various technologies: our choice has been to implement it using a native XML database. This choice allows us to store and manage XML documents produced by the Wrapper software component in their native format, and to use the powerful XQuery language for advanced content querying and aggregation. The native XML database is organized as a set of collections of XML resources, where the nesting of collections is allowed. In our application, we store XML resources as provided by the Wrapper software component, one collection for each resource. Each collection holds the various chronological versions of the resource: so, each collection effectively contains the history of the resource, all its informational content and a changelog.

When a new resource is to be stored, a check is done by the DataManager software component, in order to avoid duplicate resources. Two resources are considered to be different if their informational content changes. More precisely, they are different if changes to titles, links or descriptions of the resource channel or items are detected. Once stored, the resource is chronologically archived and ready for later retrieving and querying.

C. *The Service Level*

The Service level lets Web clients access the RSS feeds through the use of REST Web Services [9]. REST, an acronym for *Representational State Transfer,* is an architectural style which conceives everything as a resource identified by a URI. In particular, it imposes a restriction about of the URL defining the page info, that, in the REST view, are considered resources. Each resource on the Web, such as a particular part specification file, must have a unique URL (without GET fields after it), that totally represents it.

With respect to the well-known SOAP architecture[2], in REST we never access a method on a service, but rather a resource on the Web, directly using the standard HTTP protocol and its methods, To put it differently, in REST the hypertext linking controls the application state. This feature of REST allows greater simplicity and maximum interoperability with any Web client, either *thick,* like a desktop application, or *thin,* like a Web browser.

D. *Accessing REST Web Services and resources*

Adhering to the REST architecture and vision, everything is a resource and so any request and any search returns to the client an RSS resource, actually in the format of RSS 1.0 or 2.0, depending on the client choice.

[2]Please refer to [10] for an introduction to SOAP

In our application, these RSS resources are accessed through HTTP requests, using the GET method of HTTP 1.1 protocol; clients can ask for:

- A list of collections of RSS resources, each representing the chronological history of a resource.
- A list of RSS resources contained in a given collection.
- An RSS resource, identified by an index.
- An RSS resource containing only up to a given number of items, starting from the most recent one.
- An RSS resource obtained by querying a collection of resources, searching for keywords in titles, links or descriptions of items.

The GET method, in principle, should not modify the original resource. A detailed description of how REST Web Services and resources are accessed follows.

a) /resources[?type=rssType]: Accesses an RSS resource listing all collections of resources that clients can request and query. The optional *type* parameter identifies the RSS type of the requested resource.

b) /resources/rssId[?type=rssType]: Accesses an RSS resource listing all resources contained in the collection identified by the resource id, the *rssId* URL section. The optional *type* parameter identifies the RSS type of the requested resource.

c) /resources/rssId?index=n & [type=rssType]: Accesses an RSS resource identified by its *rssId* and the *index* parameter, that is the index number into the chronological history: use "1" for the first resource (the most recent one), "2" for the second and so on. The optional *type* parameter identifies the RSS type of the requested resource.

d) /resources/rssId?max=n & [type=rssType]: Accesses an RSS resource identified by its *rssId*, containing only up to *max* items. The optional *type* parameter identifies the RSS type of the requested resource.

e) Complex queries: The following query:

```
/resources/rssId?max=n & [type=rssType]
                       & [(title| link | description | desc)=value]
                       & [op=(and| or)]
                       & [(title| link | description| desc)=value]
                       & ...
```

is intended to query all resources identified by the given *rssId*, requesting only up to *max* items and combining, using logical "and/or" operators, searches for title, link, or description of items. The optional *type* parameter identifies the RSS type of the requested resource.

V. THE APPLICATION AT WORK

To illustrate how our application works we consider a fragment of a HTML document taken from the reference Web site *www.theserverside.com.* After the insertion of the meta-tags, the fragment looks as in Figure 2. Then the fragment is converted in XML format and, if not already present in the database, is stored in the appropriate collection of the database. Upon request from the client, the XML file is extracted and converted into one of the two formats currently supported by our application, that is to say RSS 1.0 or RSS 2.0. For sake of brevity we present here only the RSS2 version of the output (see Figure 3). It should be noted that in order to work properly our application strongly relies upon the insertion of

Collection path	Description
/db/resources	Root collection
/db/resources/headlines.rss	Collection holding XML resources related to the headlines.rss resource, that is, its history
/db/resources/headlines.rss/123	XML resource identified by its time-stamp (123)

TABLE II
COLLECTION EXAMPLES

```
<!-- <channel:image
    url="http://www.theserverside.com/skin/images/feed-logo.jpg"
title='The Enterprise Java Community.
    Your Enterprise Java Community'
link="http://www.theserverside.com />"-->
<!-- <channel:extension uri="http://purl.org/dc/elements/1.1/"
prefix="dc" localName="language">
en-us</channel:extension> -->
<!-- <channel:title> -->The Enterprise Java Community.
    Your Enterprise Java Community
<!-- <channel:link> -->http://www.theserverside.com
<!-- </channel:link> -->
<!-- </channel:title> -->
<!-- <channel:description>-->Enterprise Java Community is a
    developer community, containing up-to-date news,
    discussions, patterns, resources and media
<!-- </channel:description>-->
<!-- <channel:extension uri="http://purl.org/dc/elements/1.1/"
prefix="dc" localName="date"> -->
<!-- </channel:extension> -->
<td colspan="2">
<hl><!-- <item:title index="1"> -->wingS 2.0 web framework released
    <!-- </item:title> -->
</hl>
<div class="iteminfo">
Posted by:
<!-- <item:link index="1"> -->
<a href="/user/userthreads.tss?user_id=194346"
    title="view Joseph's recent threads ...">
<!-- </item:link> -->Joseph Ottinger</a>on
<!-- <item:extension index="1" uri="http://purl.org/dc/elements/1.1/"
prefix="dc" localName="date"> -->December 08, 2005 @ 08:25 AM
<!-- </item:extension> --></div>
<p>
<!-- <item:description index="1"> -->
The <a href="http://www.j-wings.org/" target="_blank">wingS project</a>
has just released version 2.0 of its framework with lots of major
improvements.<br><br>wingS is a component based web framework resembling
the Java Swing API with its MVC paradigm and event oriented design
principles. It utilizes the models, events, and event listeners of
Swing and organizes the components as a hierarchy of containers with
layout managers.
<!-- </item:description index="1"> -->
```

Fig. 2. An HTML fragment after the insertion of meta-tags

```
- <rss version="2.0">
  - <channel>
    - <title>
        Enterprise Java Community: Your Enterprise Java Community
      </title>
      <link>http://dynamo.dynalias.org/tss.jsp</link>
    - <description>
        Enterprise Java Community is a developer community, containing up-to-date news,
        discussions, patterns, resources, and media
      </description>
    - <image>
        <title>TheServerSide.com</title>
        <link>http://www.theserverside.com</link>
      - <url>
          http://www.theserverside.com/skin/images/feed-logo.jpg
        </url>
      </image>
      <dc:language>en-us</dc:language>
    - <item>
        <title>wingS 2.0 web framework released</title>
      - <link>
          http://feeds.feedburner.com/techtarget/tsscom/home?m=315
        </link>
      - <description>
          The wingS project has just released version 2.0 of its framework with lots of
          major improvements. wingS is a component based web framework resembling
          the Java Swing API with its MVC paradigm and event oriented design principles.
          It utilizes the models, events, and event listeners of Swing and organizes the
          components as a hierarchy of containers with layout managers.
        </description>
        <pubDate>Thu, 08 Dec 2005 08:28:04 EST</pubDate>
      </item>
    </channel>
  </rss>
```

Fig. 3. The fragment in RSS 2.0 format

meta-tags, which can be accomplished with a very little effort and/or change in currently available content management and publishing systems. It is beyond the scope of our application to be able to discover the appropriate patterns inside the HTML documents and *automatically* insert the meta-tags, which can be successfully done by our application only if the HTML document never changes in its internal structure.

Let us now see how a user interacts with the application. First of all, a user can verify the available RSS resources through his Web browser. She obtains a list of the available resources which can be formatted in one of the two currently supported formats, namely RSS 1.0 or 2.0. Following the link, the user gets the archive of the resource, chronologically ordered from the newest to the oldest. Our application allows also to aggregate RSS items and to query them. It is then possible to keep up-to-date by requesting a fixed number of the newest items. It is also possible to request the newest items containing a certain keyword in the title or in the description.

VI. APPLICATION EXPERIENCE

A. The proof-of-concept: dynamo.dynalias.org

We made a working prototype of our architecture, that we called Dynamo, available at http://dynamo.dynalias.org. By now Dynamo publishes the news feeds, in both RSS1 and RSS2 formats, taken from the Web portals www.serverside.com and www.java.net, each of which produces about 4-5 news (in plain HTML format) every day. In order to avoid any interaction with the portals we resorted to download the HTML pages containing the news, insert the meta-tags we defined and submit them to the entire procedure of extraction, storing and publishing.

B. Scalability issues

A typical problem in the design of an architecture like ours consists in the forecast of all possible critical elements that can raise as work loads become bigger and bigger. First of all it must be considered that an instance of Dynamo can be installed for each Web server. In those cases in which we have a very frequent production of news coming from different sources of information, it is possible to install a ”copy” of Dynamo for each source so to distribute and even balance the load. Another, even more severe, possible limitation to the performance of the proposed architecture is represented from the bandwidth required to forward the requests for updates, because in those cases of non regular updates a lot of requests would be useless thus resulting in wasting bandwidth. This is the reason of an improvement we are studying, that is a polling policy able to fit the frequency of the updates of the news from the Web servers: this policy, we called smart polling policy, adjusts the frequency of the requests for updates to the frequency with which Web portals generate new information.

Another factor that may affect the overall performances of DynamoNews is the host database management system, which is in our implementation is *eXist,* an Open Source native XML database whose performance seems not up to those of the DBMSs normally adopted to service Web portals. To avoid long response times even for simple queries, we have implemented a cache engine where the most frequently requested queries are stored.

We introduced two different polling policies, which can be chosen and plugged in our application independently from each other. The first is called "flat" polling policy, as it does not depend from update frequency, while the second is called "smart", as it tries to fit the update frequency of each Web portal. It is possible to reconfigure at run-time the Poller component of the application (see further), in order to switch policy at runtime. It must however be considered that the smart polling converges asymptotically to the flat one.

With the flat polling policy, Web resources are queried for updates at regular time intervals which can be modified. It is the simplest strategy and it well applies to regularly updated information. The first improvement one can make over flat polling is to compute the frequency of the requests of updated Web documents as an estimate of the frequency. Then such estimate is compared to the *real* frequency with which Web documents are updated or newly generated. Both the estimate and the *real* times are used to compute a new estimate. That is:

$$\tau_{n+1} = \alpha\tau_n + (1-\alpha)t_n \quad (1)$$

where τ_{n+1} is the estimate at the $(n+1)$-th iteration, τ_n the estimate at the n-th iteration, t_n the *real* frequency at the n-th iteration. The parameter α, whose value stands in the interval between 0 and 1, represents the relative weight of the previous estimate w.r.t. the *real* frequency. As τ_{n+1} takes into account the previous iterations, α represents the importance given to previous iterations.

Some considerations about the parameter α. Its value, comprised between 0 and 1, influences the velocity with which the frequency of polling equals the frequency with which Web portals publish new information. We found, on the other side, that its value does not influence the *convergence* of the frequency of polling to the frequency of publication, but only its velocity. Analogous results can be found in literature, even if in rather different situations. See, for example, the algorithm of processes scheduling known as *shortest job first* [11], as well as the weighed mean frequently used in iterative calculations typical of the Self-Consistent Integral Theories in Statistical Many-Body Thermodynamics. Please refer to the survey in

VII. Relationship with literature

The amount of information currently available in HTML format is really huge, but the main limitation in its fruition consists in its poor machine-readability, that is, in its lack of *structure.* Such problem can be solved, vis-a-vis the size of the Web and of individual Web portals, by making the extraction and annotation phase automated at least to some extent. To the best of our knowledge, the most advanced example of this approach is the LiXto [12] suite. LiXto supports the semi-automated creation of extraction programs, called filters, which, thanks to some clever logic-based representation of the HTML/XML structure [13], [14] of the document, is tolerant to some degree of elaboration of the source. We are planning to experiment with LiXto to make our extraction function capable of re-arranging the meta-tags annotation to adapt to changes in the HTML source.

VIII. Final considerations

We have described a Web application that generates and manages the RSS feeds extracted from HTML Web documents. The proposed architecture is intended to be applicable to arbitrary Web sites, provided that the Web administrator decides to start the service by adding the proposed meta-tags to the commented part of each page.

In order to collect the information relevant for the generation of a RSS feed we have defined a set of XML-like annotations which have to be inserted inside the HTML documents that contain the information we want to convert. The information is then extracted and organized into an XML format for storing. Typical actions which can be made include aggregation, query and conversion to RSS formats for syndication.

The most contemporary Web Content Management Systems (CMS) can handle news publishing and channeling by dynamic procedures which, upon user's request, retrieve data from the DBMS, the insertion of Dynamo meta-tags is accomplish just by some slight modification of those procedures (usually coded in PHP, JSP or ASP). Although content management systems of the last generation allow the publication of news in RSS format, Dynamo has the advantage of preparing and storing XML news and querying the database in a more *semantics-driven* way than with the relational databases which normally underlie CMSs.

We believe that there is room in the current landscape of the Web for this solution as it allows upgrading existing Web portals with minimal effort. As an instance, our recent work [15] describes how to bring a legacy system for the managing of community Web pages up to RSS news channeling. By hosting hundreds of discussion lists, accessed daily by thousands of users, the considered application is a good, and successful testbed for Dynamo.

References

[1] WC, "Rdf site summary (rss) 1.0." [Online]. Available: http://web.resource.org/rss/1.0/spec

[2] "Rss 2.0 specification." [Online]. Available: http://blogs.law.harvard.edu/tech/rss

[3] S. Bossa, *Towards the Semantic Web: a platform for dynamic generation, query and archival of RSS contents (In Italian). http://informatica.unime.it/*: Graduation Project in Computer Science, Univ. of Messina, 2005.

[4] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures.* Ph.D. Dissertation, 2000.

[5] R. L. Costello, "Building web services the rest way." [Online]. Available: http://www.xfront.com/REST-Web-Services.html

[6] W3C, "Xsl transformations (xslt) version 1.0," 11 1999.

[7] ——, "Xml schema part 0: Primer version 2.0," 10 2004. [Online]. Available: http://www.w3.org/TR/xmlschema-0

[8] R. Bourret, "Xml and databases." [Online]. Available: http://www.rpbourret.com/xml/XMLAndDatabases.htm

[9] J. M. Snell, "Resource-oriented vs. activity-oriented web services." [Online]. Available: ftp://www6.software.ibm.com/software/developer/library/ws-restvsoap.pdf

[10] W3C, "Soap version 1.2 part 0: Primer," 06 2003. [Online]. Available: http://www.w3.org/TR/2003/REC-soap12-part0-20030624/

[11] G. G. Abraham Silberschatz, Peter Galvin, *Operating System Concepts VI Edition.* John Wiley & Sons, 2002.

[12] G. Gottlob, R. Baumgartner, and S. Flesca, "Visual web information extraction with lixto," *Proc. of VLDB Conference*, 2001.

[13] G. Gottlob and C. Koch, "Monadic datalog and the expressive power of languages for web information extraction." *Journal of the ACM*, vol. 51, 2004.

[14] G. Gottlob and et Al., "The lixto data extraction project – back and forth between theory and practice." *Proc. of PODS, Principles of Database Systems*, 2004.

[15] F. DeCindio, G. Fiumara, M. Marchi, A. Provetti, L. Ripamonti, and L. Sonnante, "Aggregating information and enforcing awareness across communities: the dynamo rss feeds creation engine," in *Proc. of COM-INF06 Workshop.* Springer LNCS, 2006.