Projet ROSES

Programme MDCO – Edition 2007

Livrable no D2.1

Survey on the Web Syndication Current Issues

Identification

Acronyme du projet	ROSES
Numéro d'identification de l'acte attributif	ANR-07-MDCO-011-01
Coordonnateur	Paris 6
Rédacteur (nom, téléphone, email)	Cedric du Mouza et.al. (+33) 01 58 80 85 66, cedric.du_mouza@cnam.fr
No. et titre	RSS-XML freshness measures
Version	
Date de livraison prévue	31 décembre 2008 / t0+12
Date de livraison	3 mars 2009 / t0+15

Résumé

Ce document présente un état de l'art sur les problèmes et les solutions actuels concernant la syndication web. Il remplace l'état de l'art sur les modèles de flux de données prévu initialement dans la liste des livrables ROSES. Ce changement est justifié par le fait que les aspects “flux de données” ne représentent qu'une partie des problèmes rencontrés quand on veut construire un système de syndication web comme ROSES.

Survey on the Web Syndication Current Issues

Camelia Constantin[†], Jordi Creus[†], Cédric du Mouza[‡], Roxana Horincar[†] and Nicolas Travers[‡]

† LIP6, Univ. Pierre et Marie Curie Paris VI, France, firstname.lastname@lip6.fr
‡ CEDRIC, Conservatoire National des Arts et Métiers, firstname.lastname@cnam.fr

Abstract

We aim with that survey at proposing an overview of the principal issues connected to the web syndication. With the continuously growing amount of information avalaible on the Web, it has quickly turned out that the traditional *pull* approach, where a user has to check periodically his sites of interest to retrieve updated information, could not be a long-lasting solution. Web syndication recently emerges as the solution. Nowadays a user can select the set of web sites from which he wants to be notify whenever information is updated/added thanks to RSS or Atom feeds. However, as often for new technologies, Web syndication has come along new challenging research issues, like the way to produce feeds, how to store, cluster or replicate information, how to query, filter index it, and to rank the results,...

1 Introduction about Web syndication

`Web syndication` is a way to be aware about information *via* a subscription. A pull approached is used to bring new information to the user. An application that treats web syndication asks the '*flow*' of information and shows new data to the user depending on updates on the '*flow*'. There is several types of web syndication : application updates (*ie. Microsoft Updates*), web browsers information publisher (*ie. netscape*), feeds (*ie. RSS* and *Atom*)... Here, we will speak more precisely on **feeds**' web syndication, especially `RSS` and `Atom`.

RSS and Atom are two Web Syndication XML formats. Each of them are used in many applications over the Web in order to inform users of new information. Each new piece of information is called an *item*, and the composition of all items that have been into the XML file is called a `Feed`. It contains metadata on the feed and its items. A feed is represented by an XML file which format will be detailed further. Several types of RSS or Atom feeds can be found on different news : daily news, blogs updates, web site modifications, mails, triggers...

A *feed* can be seen has a *flow* since its content changes threw time. Each feed's content can be modified by adding or deleting items into the XML file. RSS/Atom readers see feeds as a view on the changing XML file, showing new items threw time. For example, an RSS/Atom aggregator crawls the feed regularly, each time a new item is registered into the feed, the user is notified that the *flow* contains a new item.

1.1 RSS : Really Simple Syndication

Really Simple Syndication began on 1999 with the 0.90 version (called RDF Site Summary) by Netscape. At this time, RSS is based on RDF [65] that can give rich description of information. From 0.90 to 2.0 versions, RSS changed many times. It has set new elements, modifying attributes, deleting RDF descriptions, changing compatibility. To the end, RSS 2.0 rose in 2002 at Harvard University and had been chosen by the New York Times.

A RSS file is composed of `channel`s that contain information on a feed and `item`s, each one contains new data. A `channel` can contains several tags as *title, description* and *link* which are mandatory, and *lastBuildDate, language, category, language...* which are optional. No precise type is defined for these tags, just recommendations. An `item` contains only optional tags: *title, link, pubDate, description, author, category, source, guid, comments and enclosure*. This format has no official DTD, nor XML-schema, nor namespace in order to make compatible all RSS versions (0.90 to 2.0). Moreover, we can notice that most tags are optional. In matter of fact, RSS creators can do anything on items and it makes difficulties to integrate several types of structured information: no identifier, no date (or incorrect format), XHTML in description (without saying), absolute or relative references (without notification)... Then, RSS readers or aggregator must take into account all versions and all possible errors made by the feed creator.

Below, you can see an example of a RSS file that contains one channel with two different items. We can notice that between to items, we can have many differences: date, guid, link, pubDate.

```
<?xml version='1.0' encoding='UTF-8'?>
<rss version="2.0">
    <channel>
        <title>My feed example</title>
        <link>http://www.example.org</link>
        <language>en</language>
        <pubDate>Fri, 20 Jul 2008 13:36:10 GMT</pubDate>
        <item>
            <title>My first feed item</title>
            <link>http://www.example.org/rss.html?item=1</link>
            <description>This text describes the item, and give more information about the link's content.</description>
            <pubDate>2008-07-20 13:24:10</pubDate>
            <guid isPermaLink="false">http://www.example.org/rss/item1.xml</guid>
        </item>
        <item>
            <title>My second feed item</title>
            <link>./rss.html?item=2</link>
            <description>This text can contain HTML tags.</description>
            <pubDate>Fri, 20 Jul 2008 13:36:10 GMT</pubDate>
        </item>
    </channel>
</rss>
```

1.2 Atom

Atom [64] refers to two `W3C` standards: an XML syndication format and the *Atom Publication Format* (APP, see below). From 2003, the IETF has designed a new syndication format more usable for the database community. By bringing more flexibility with XML-schema and interoperability with APP, Atom became the main competitor of the RSS format.

The Atom XML format is far more precise than RSS 2.0. Needed tags (for the database community) are mandatory, ambiguous information must be notified (references, XHTML, language, date...), description is decomposed into a summary and a content, and a namespace has been designed for this format. A comparison between RSS and Atom can be found in section 1.3.

Atom also refers to a publication protocol called `APP`. It defines the way how to modify Atom

feeds as it was an XML database. It is often used for concurrently modifications and interoperability of systems, mainly in huge news' database. It extends the HTTP protocol with new operations in order to interact with the storing system (consultation, add, remove, update, get). Using XML for data, you can embed all types of data in your messages (RSS items, audio or video for *podcast*, images...). In fact the APP protocol says which type of data you want to store, and then you create an Atom item to link the embedded file.

Below you can see an example of Atom feed, containing same information as the RSS example (saw before):

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
    <title>My first feed</title>
    <subtitle>An example of Atom web syndication document</subtitle>
    <link href="http://example.org/"/>
    <updated>2008-07-20T13:36:10Z</updated>
    <author>
        <name>Nicolas Travers</name>
        <email>nicolas.travers@cnam.fr</email>
    </author>
    <id>http://example.org/</id>
    <entry>
        <title>My first feed item</title>
        <link href="http://www.example.org/atom.html?item=1"/>
        <id>http://www.example.org/atom/item1.xml</id>
        <updated>2008-07-11T13:24:10Z</updated>
        <summary>A summary of the given item that do not contains HTML tags.</summary>
    </entry>
    <entry>
        <title>My second feed item</title>
        <link href="http://www.example.org/atom.html?item=2"/>
        <id>http://www.example.org/atom/item2.xml</id>
        <updated>2008-07-20T13:36:10Z</updated>
        <summary>A summary of the given item that do not contains HTML tags.</summary>
    </entry>
</feed>
```

We can notice that Atom has an XML-schema that details exactly properties of each tag of the feed. We can't have different ways to analyse a feed's item. Moreover, important tags are mandatory, like identifier, title, date or description. It helps database community to store important information and interacts with data over the web.

1.3 RSS vs Atom

Both RSS and Atom describe its feeds with a structured schema and contain list of items. Each item is composed itself of specific tags, describing information. Main differences can be seen on mandatory elements and ways to structured information for applications. In fact, Atom contains more needed elements for querying feeds into Database Stream Management Systems (DSMS), like identifiers, titles, dates or authors. On the other hand, RSS is more simple and usable developers. We can list several

differences between those two formats.

Some differences are based on type declaration like RSS items can contain text or HTML without notifying, contrary to Atom in which we must declared explicitly the element's content (text, HTML, video, URL...). The description tag in RSS is a catch-all element, thus Atom proposes a summary and a content that differentiate the content. RSS items refer to only one linked document, while Atom entries can link one or more documents (i.e. produced by aggregating items);

RSS lacks of some important types (for applications purposes) like *date* (Atom gives a TimeZone format - RFC 3339), *language* (Atom uses the *xml:lang* type), URI links (Atom specifies the *xml-base* type for absolute or relative URI), unique identifiers (guid) are free (contrary to Atom with **IRI** which, moreover, are mandatory).

Atom is an opened format, free for extensions due to its *XML-schema* and namespace[1], and a *MIME-type* has been registered : *application/atom+xml*. Whereas RSS has a fixed format (copyright Harvard University) but most RSS flows use there own tags to enrich information.

Atom is a more complete concept than RSS since it defines a protocol (*APP*) in order to update collections of entries. Items are inserted into a collection (Atom Entry Document).

On the one hand, RSS is more widespread than Atom over the web due to its simplicity. On the other hand Atom is a standard format of the W3C used by some famous companies (i.e. Google, IBM...). Moreover, *Podcasts* (see below) that publish flows of multimedia files (music, videos) are mostly written in the RSS format. To finish, the acronym *RSS* has been generalised by Medias for *Web Syndication*.

Those reflections highlight differences between web developers and database communities. In fact, one tries to emphasise web facilities with less constraint and no limitation to information. And the other side tends to impose a way to produce information in order to store and find it easier. No format is best, but each community found its interest.

1.4 Architectures and Technologies

`Yahoo! Pipes` [3] is a web application from `Yahoo!` that interacts with feeds by aggregating and manipulating them. It lists best pipes and enables users to create their own pipes. This aggregator helps web developers to manipulate feeds and integrate the resulting one to their own web site. Its graphical interface seduces most users.

A *Pipe* is a sort of tree algebra composed of operations on feeds. Each operation takes and analyses a RSS/Atom file to produce the required result set which is an feed file. The graphical interface lists all operations. To create a pipe, you just drag-and-drop operations into the tree. To link operations, you just have to join two boxes point-to-point. This user-friendly interface helps you to design your own Pipe.

We can list several types of operations: **Sources** (CSV, RSS/Atom, HTML, text, Google, Yahoo search...), **User inputs** (values defined by the user), **Operators** (count, rename, union, web service...), Url (builds an URL from a previous one, with user inputs), **String** (regex, replace, tokenizer, translate...), **Date** (build or format), **Location** (world common addresses), **Number** (Arithmetic operations).

Those operators have filter and transformation properties. They change incoming feeds to produce a new one on which you subscribe *via* an application.

Figure 1 illustrates a Pipe that searches the word *RSS* (parameter box on the right) in the feed *http://example.com/feeds* (URL-builder). The URL-builder box defines which feed to collect and how to fetch it. Then, the feed is fetched from the given URL of the URL-builder. And finally the feed is sorted by the pubDate item and outputted. The output can be seen interactively by selecting the *Pipe output*.

Podcasting is a way to publish media and read new medias. A podcast is composed of a feed (RSS or Atom) and several stored medias. When the podcaster has got a new media file (like audio), he send it

[1] Atom's namespace : http://www.w3.org/2005/Atom

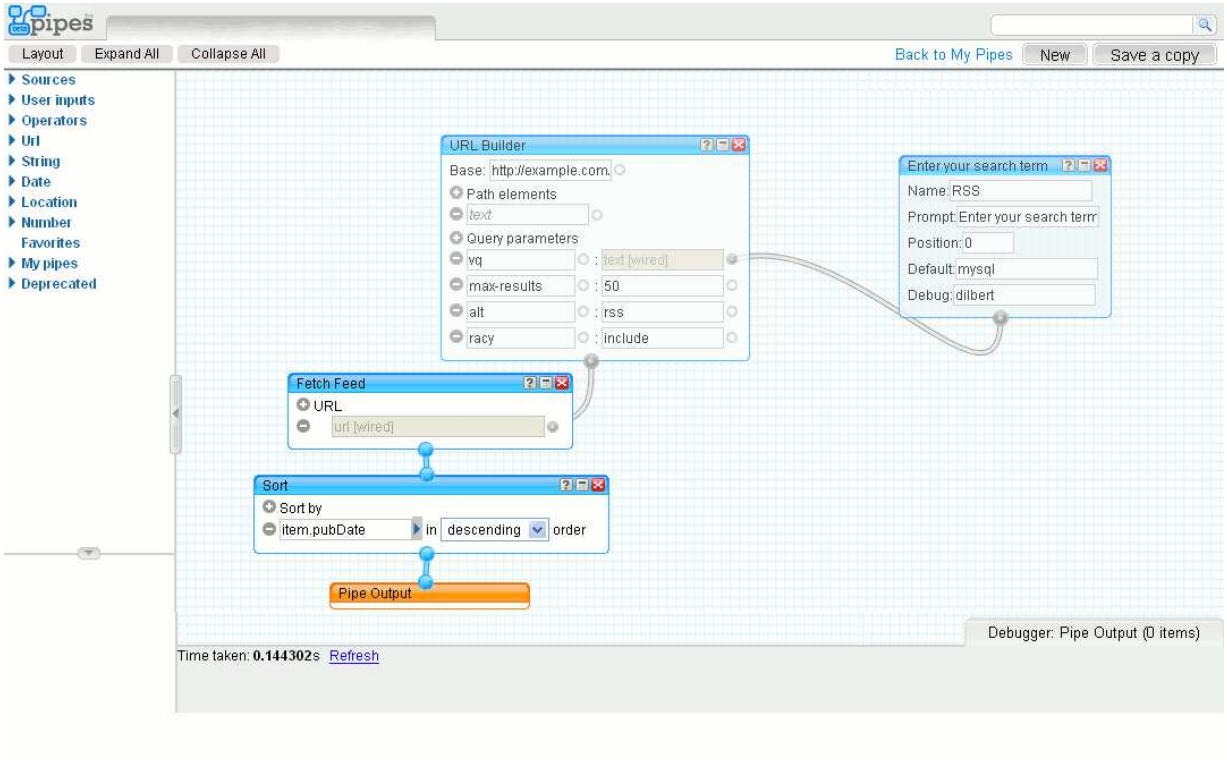

Figure 1: Yahoo! Pipes example

on his server and create a new item in the feed pointing on the media file, with an URI. The reader reads regularly the feed and detects new multimedia files threw new items in the feed. Then it downloads the media file and put it in the reader. So you can be informed of new songs, news, images on your reader.

We can notice that APP describes exactly how you put a new media into the podcast server. In RSS, you do your own way, but you must create a new RSS item linking the media.

Main podcast users read new music items into there mp3 reader over several artistic feeds. Then, you can be informed about best musics of your favourite composer.

Widgets/Mushops are little applications that read a feed in order to change its showing content. Each item is a new event in the application. For example, you can take a widget that shows the weather of your city or country, the feed contains all information on the actual weather in your country. Then, the widget changes with the feed and you can follow the weather in the little application.

Some web sites propose to integrate widgets into your own web site or on your desktop. You can see several types of information on live time: weather, web news, podcast, calendar, images for screensaver...

1.5 Issues of Web Syndication

Web syndication with RSS and Atom brings much more application over the web. Those streams of Web Syndication, so called *feeds*, become necessary to give a dynamic behaviour to web sites. This new technology requires different information treatments. Thus, those *feeds* have small different aspects than traditional data streams. In fact, Pull approach, updates variable frequency, textual information, web links (pages or multimedia), and the specific data structure make stream processing more complicated.

Most data streams management systems works on a push approach, data comes from sources and are treated on the fly. In Web syndication, the system must check updates on sources and cannot loose information since each item of the stream is important. But each source has an unpredictable update

frequency. It depends on coming information, for example a *blog web site* produces information daily and *news web sites* each minute can see more information, contrary to a trigger that can't be predicted. One goal is to approximate stream update frequency to be up-to-date and not overwhelm sources.

Amounts of data increase dramatically with Web Syndication. Contrary to standard Data Stream Management Systems which store aggregates and significant information, all information from Web Syndication is important. From RSS and Atom data structure, each item is necessary. Since the Web produces a huge amount of information, streams processing must store efficiently all this information.

Moreover, RSS and Atom data structure bring new queries issues. In fact, XML query processing with XQuery 1.0 cannot treat unlimited collections of documents. XQuery is set to evaluate finite collections and do not produce result sets for blocking operations on streams like *let* or *count*. Finally, needed treatments for Web Syndication are slightly different since we need to express groups of items (called *window queries*), temporal expressions or events expressions. For this reasons, it is necessary to modify query languages and systems to integrate stream processing facilities.

Below, we will present those aspects brought by Web syndication requirements. Section 2 describes data production, to see how RSS and Atom feeds are generated. Section 5 explains issues and techniques of data storage on Web syndication. Then, section 3 shows datastream processing techniques over relational and XML flows. And finally, we conclude and talk about perspective on Web Syndication.

2 Data production

We present in this section the different solutions for producing RSS/Atom feeds. First we introduce some tools that help for feeds creation and allow to make them avalaible on the Web. Then we describe some recent works dedicated to the aggregation of RSS feeds, that may become, in turn, new RSS feeds. Finally we address the problem of feed generation. While Web syndication has appeared to be a good solution to face with the exponential increase of information avalaible on the Web, there are still few sites that offer RSS or Atom feeds. Consequently, only a small part of the information is actually exploited by syndication. and the subscriber misses partially or totally data that may be in his area of interest.

2.1 RSS feed aggregation

Nowadays, RSS aggregators restrict their scope to crawl user-subscribed feeds and to present them to the user grouped by site source. But as the amount of information increases, it can easily become overwhelming to the user, so a number of approaches have been proposed to improve this drawback. In this section, we describe some of these approaches.

[45] proposes an RSS aggregator which performs a clustering on the feed items set and presents the result to the user in an hierarchical manner. Their system called RCS (for RSS Clusgator System) crawls daily a list of predefined feeds and user-subscribed feeds, then the k-means algorithm is run on this dataset to obtain the items clustering. Finally, a topic extraction is done on the resulting clustering with the aim of to label the nodes on the hierarchical structure of items.

[63] proposes to enhance an RSS aggregator with user-defined ontologies so that users can browse the items from their subscribed feeds according to the concepts defined in their ontologies. They have developed a tool called RSSOwl [59] which allows users to import their own OWL ontology and therefore to seek across the item set the ones related to their interest concepts.

[61] presents DanaÃ¯des, a practical application for continuous query evaluation over geographically tagged RSS feeds compliant with the GeoRSS standard [2]. Their prototype consists of a crawler, a query parser, which translates sql-like queries to execution plans, and of a query processing engine that produces RSS results. Their query language allows to formulate similarity joins, and they make use of some operator optimizations as hash joins.

[55] presents a feed aggregation service in a distributed fashion that delivers personalized RSS feeds to the users. Their system, COBRA (COntent-Based Rss Aggregator), consists of three kinds of nodes: crawlers, filters and reflectors, which provide the personalized feeds to the users. Their main contributions seek to enhance the scalability of such systems, including a topology reorganization algorithm, an optimized polling algorithm and a locality-aware crawler assignment.

2.2 RSS feed generation from (X)HTML pages

Some recent works stress the fact that despite the formidable quantity of information avalaible on the Web, only a small part is prone to syndication. The subscriber that rely only on this information is consequently likely to miss numerous events. To resolve this problem, several propositions were proposed to generate RSS feeds from existing (X)HTML pages that originally do not offer syndication opportunity. These works can be decomposed into two approaches: an automatic one, based on the discovery of patterns, or a semi-automatic one, with a preprocessing work to insert some meta-tags.

Feed generation based on pattern discovery

Note that this approach, applied in several recent works [48, 69, 49, 70, 25], assume a list-oriented information Web pages. When parsing a HTML page, several elements could generally be extracted to constitute the RSS channel. So in the *head* of an HTML document, the title of the page, delimited by the tag `<title>` becomes the title of the RSS channel. Similarly, the content between tags `<description>` and `<keywords>` is the content of the *description* of the channel, `<last-modified` is converted into *pubDate* and `<content-language>` of `<charset>` become *language* of the RSS channel. While this information concerning the channel is quite easy to extract from the HTML document, retrieving information for building the different items of the feed is much more tedious. Indeed, even when assuming that a link, identified in HTML with the `<a>` tag, may correspond to an item whose title could be extracted from the `href` attribute of the `<a>` tag, not all the links inside a page correspond to an item and/or are relevant with respect to the channel topic.

A first feature helpful for discarding information that should not be *itemized* is the time or date, since in news pages, an item is often published with the corresponding release time. Since the formats for dates or times are simple and limited they could be identified inside the Web page to locate potential items. In [69, 70] the authors take inventory around 20 different time patterns that cover all the times and dates within the set of sites they visited. Once time patterns have been detected, a segmentation of the Web page is performed. In [70, 25], the authors rely on a DOM representation of the page and an XPath-based address of each node corresponding to a time-pattern to find sections and subsections inside the page, and the content inside each. Then sections are turned into channels while subsections become items. In [70] they also propose to mine repeated tag patterns, and to proceed similarly, since they assume that all the information is not necessarly associated with a time pattern. In [49] the system collect RSS feeds from the Web to build a language model of title expressions that are used to detect or generated title, or possibly titles, corresponding to an indentified time pattern. Each title along time becomes an RSS feed. [48] proposes to watch for bursts of a word inside blog pages, that is probably a sign of a new of the appearance of a new topic.

Nowadays, exploiting similar approach, several products to generate RSS from HTML have been released like RSS Wizard [60] or FeedFire [1].

Inserting meta-tags for automatic generation

Another approach for producing RSS feeds from HTML documents consists in inserting meta-tags inside Web pages. These meta-tags allow to quickly identify the title, description, link,... of both channels and items. Once tagged, the documents must be checked, either periodically or based on a probability model, to produce easily asociated RSS feeds. In [9] for instance, the author presents the *Dynamo* architecture, where the user manually inserts 7 kind of XML meta-tags to HTML files before storing them in a native-XML database. Thanks to XSLT transformation, *Dynamo* produces RSS feeds from the stored XML data. The RSS resources can access to a list of (collections of) resources, a RSS resources identified by an index, or RSS resources that contain given keywords, in titles, links, description, etc. In [16], the authors extends the previous work to cover dynamic HTML pages, especially in applications where Web pages follow templates, like news forums. The authors propose to insert meta-tags directly inside the templates in order to produce automatically tagged pages. then a periodical polling of the different forums allows to collect information associated to meta-tags and eventually to produce RSS feeds.

3 Querying

Since RSS and Atom are considered as streams, we need to compare different stream processing techniques. For years, data streams management systems processed relational data over events. Those types of events are in push philosophy, but, to query RSS and Atom items threw times, we need to see those techniques and adapt it to asynchronous systems. We present in this section different query language in Data Stream Management Systems (DSMS), and highlight main characteristics that could bring interests in RSS/Atom processing.

3.1 Continuous Query Languages

In literature, many Continuous Query Languages [5][13][14][26][37][44][72] have been design in order to process data flows in different manners. Most of them are based on relational algebra and SQL, and brought specific goals on stream evaluation.

Window Queries
Window queries [26] are specific to continuous queries. When a flow arrives, a window queries is a sort of snapshot of its content at a time. On that snapshot, we can then evaluate typical queries, mostly aggregates or joins that could not be evaluated on streams. We can differentiate three types of window queries :*sliding, tumbling* and *landmark*. All three are defined by the way when windows are created.

- For sliding windows, you can have only one window at a time, it begins with a start event and end with an end event. The next window begins when the previous one finished. This type of window is the most used in DSMS;
- For tumbling windows, you can have several windows, but only one window can begin at a time. Then, when a start event arrives it opens a window and end with an end event. The second window can be opened during the creation of the previous window;
- For landmark windows, there is no constraints for window creation. Several window are created at a time, and may not be closed on same time. This solution needs much more memory than other solutions, and more technical treatments.

Relational Approach
The most famous one is CQL [5] that adds new algebraic operators on flows with queues between them. It present a smart transformation from streams to relational and *vice-versa*. In fact, it is necessary to introduce a notion of infinite relation different from relations for blocking operators (aggregate, joins, etc.). Windows are used in CQL in order to transform a stream into a relation and process sequences.

Parallelism and Adaptability
An interesting DSMS isTelegraphCQ [13] which brings a novel adaptable system. This system is composed of independent operators and a routing module. Operators are not linked together, but tuples that must be treated knows in which operator it has to go. Then, the `Eddy` routing module processes each tuple within operators, dealing with priorities and adaptability of the whole evaluation system. This solution allow the system to adapt itself among all tuples it has to treat.

Negation expressions
SASE [72] is a particular event language that manages RFID events ((*Radio Frequency Identification*). Those expressions can express many type of sequences. It brought a unique expression which can be interesting to manage, the lack of event: *negation*. In fact, when the streams does not respond, it could be an interesting information, and the we can express lack of events into a stream and generates a process. Event expressions are analysed and produce algebraic trees where simple optimisation techniques are done (level up predicates, begin with windows). Evaluating a tree is done by processing a automaton on sequences of items.

LinearRoad benchmark
A stream benchmark has been proposed to emphasises stream management systems. The *LinearRoad* benchmark [6] uses cars and roads events to generate streams. A DSMS must evaluate those streams within five minutes to be relevant. Most of them are not appropriate to web syndication since data are more rich, external data are like to be necessary and composition is mandatory.

3.2 Querying Semi-structured Streams

Since many studies have been done on relational streams, XML streams are more poor in this domain, and are mainly based on relational techniques. We can find some solutions for simple patterns [28] [38], and more complex patterns like XQuery like solutions [10][39][18].

XPath approach
In order to treats XML events (a small XML document with event information), XPath does not prevent stream evaluation, few solutions on the fly were proposed. The most common one is [28] that proposes two algorithms LQ and EQ based on a Lazy Filtering and an Eager Querying Algorithms. The path is converted into a hash table where each bit is set when the appropriate predicate is verified. If all the table is set to true, then given item verifies the query and is selected by the query.

XQuery approach
The *ForSeq* [10] proposed W3C to extend XQuery with window queries. A new clause **ForSeq** is used to declared windows for infinite sequences, which is a new type for the model (i.e. int**). It takes into account sliding, tumbling and landmark windows declaration. This language has been implemented into the *MXQuery* system which now validates the *LinearRoad* benchmark. It also proposes uses cases for XML streams.
Although it is a good proposition for XML streams, in this model time is seen as data. So we can't be

sure that time based queries can be valid. In fact, items that could be late can't be processed. Moreover, since windows produce result set when the end predicate is verified, then, we can't have a snapshot of the window during its evaluation.

SAX approach
FluX [39][38] is a purely event-based queries. It rewrites XQueries into the event-based FluX language. This algorithm uses order constraints from a DTD to schedule event handlers and to thus minimize the amount of buffering required for evaluating a query. It takes a DTD to format SAX events and pre-process evaluation of XML streams. Some optimization rules can group items in order to project them into a result set at the same time. The SAX parser has been extended into *XSAX* that takes into account DTDs that would create automatons for XML files. This proposition is very useful for RSS and Atom that has a fixed schema. This language lacks of expressivity for complex standard queries and continuous queries.

NFA approach
YFilter [18] is an alternative approach which combines multiple queries into a single *Nondeterministic Finite Automaton* (NFA). The use of a combined NFA reduces the number of states needed for sets of user queries and greatly improves filtering performance by parallelizing evaluation. YFilter also extends the NFA model to efficiently handle predicates within path expressions. Each query is decomposed into states of operations, and a graph is composed (redundancy is prevented by reusing states). To process an item, it must pass through the graph into a proper resulting state.

4 Achieving scalability

Most of the syndication applications support large numbers of subscriptions and sources. Consequently considering each subscription on the server's side, independently of the others for both query evaluation and result dissemination would generate to a crushing processing activity and message number. The application then fails to achieve scalability. Moreover, on the client's side, the client may also be flooded by thousands of items, even with a low number of subscriptions if the number of item sources is high. In this section we consider solutions to make applications scalable. We mainly identify four famlies of solutions: enhanced dissemination processing based on subscription indexing to decrease procesing and communication costs for notifications, grouping subscriptions to improve evaluation costs, filtering mechanism to prune and evaluate only a small subset of subscriptions and finally results ranking to allow the client to consult only a subset of items that is the most pertinent with regards to the quality of the sources, user's opinion, etc.

For all these techniques we can consider centralized, partially-centralized or fully-distributed deployement. The rationale for selecting one solution rather than another could be various, like for instance:

- simplicity: centralized solutions are more simple to deploy, to manage, to secure, etc;
- scalability: when the number of data and/or subscriptions becomes too large, centralized systems failed to store, to index and to quickly answer to queries;
- bottleneck: for centralized solutions, both subscriptions and items must be routed to a single server;
- message latency/loss: distributed solutions generally required more messages than centralized ones what increases the answer time, the risk of loosing messages, the bandwith occupation, etc.

The aspects are well known and discussed in litterature. We choose to simply focus on the different optimization techniques that allow to achieve scalability without considering the pros and cons of the distribution.

4.1 Subscriptions indexing

In web-based publish/subscribe applications, servers must quickly face to a large number of subscriptions. The naïve strategy that consists in evaluating each query independently from the other should be consequently discarded for scalable applications. Here the context is slightly different from traditional databases applications, where the number of data is much larger than the number of queries. Oppositely, in pub/sub applications both number of data and subscriptions are large.

To guarantee an efficient evaluation process, the traditional solution consists in indexing subscriptions and to use the index for each incoming item to determine the notifications to produce. However the context of web syndication is highly dynamic:

$i)$ new subscriptions may join or quit the system,

$ii)$ new items are continuously produced by sources.

Consequently the index structure must be also dynamic to support numerous insertions or deletions and to efficiently retrieve the subscriptions concerned.

Inverted lists approaches

A first family of subscriptions indexes consists in inverted lists. Two models are commonly adopted for subscriptions and items with inverted lists: the boolean model and the vector space model. For the latter, if there exists n distinct words possible, both subscriptions and items are represented as a n-dimensional vector $V_s = (w_1, w_2, \ldots, w_n)$ where each w_i is the weight assigned to the $i-th$ word (generally this weight is computed based on a *tf/idf* measure). For the former model data are simply represented as a bit vector $V_b = (b_1, b_2, \ldots, b_n)$ where $b_i = 1$ if the $i-th$ word is present in the subscription/item.

In their brute-force variant, inverted lists consist in maintaining a directory D with all the words extracted from subscriptions placed in a different cell $D[i]$. Each word w_i at $D[i]$ is a key that is associated to an inverted list L_i that contains all the subscriptions s_{ij} that present this key word. When a new item is received, we first extract the different words $q_k, 0 \leq k < N$ from the item. For each q_k we retrieve the corresponding inverted list L_k, and the subscriptions that should be notified are subscriptions in $\cup_{0 \leq k < N} L_k$ The whole technique is depicted in Figure 2.

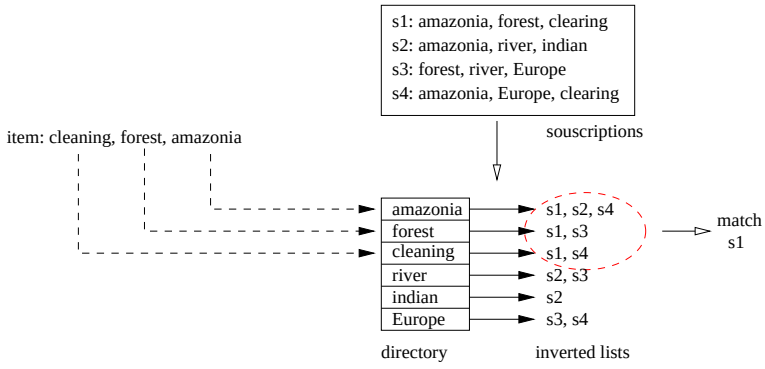

Figure 2: Basic use of inverted lists in pub/sub

The main problem with the brute-force technique is that we must test each word of every incoming item, what can quickly turns to be a drastic time-consummant task. Several variants have been proposed to improve this basic solution. For instance in [73], for the boolean model, the authors propose to manage two additional arrays, one to count the number of words of each subscription,and another to keep count of the occurrences of the subscriptions retrieved with the inverted lists for the set of disctinct words of the item. They also present the key methods where each subscription appears only once in the set of inverted lists. The subscription is allocated to an inverted list by picking up either at random one of its word, or based on the word frequency, choosing the word with the lowest frequency. The idea is to limit the number of subscriptions associated with the more frequent words and thus the number of subscriptions to examine for each item. Here we record for each subscription in an inverted list its words that were not the directory entry. Then when we use an inverted list for any given word of the item, we check one after the other all the corresponding subscriptions associated, *i.e.* we verify that their other words also belong to the item. These techniques were adapted for vector space model in [74], the idea is to rely on threshold values for each word. The approach proposed in [22] use tree index for the different predicates in the subscriptions, and inverted lists for some of these predicates, clustering subscriptions in each list based on their number of predicates. When an item arrives, a bit vectors with a 1-value for each of its predicates is created. Then the inverted lists determine the set of clusters possibly concerned by the item. A matrix of elementary bit vectors representing each subscription it contains is associated to each cluster and allow to efficiently detect the subscriptions within the cluster that match the item.

Tree-based approaches

A second family of indexes for subscriptions are tree-based approach. Various tree structures have already been proposed but they all basically consits in building a tree based on a partial order that relies on containment of subscriptions.

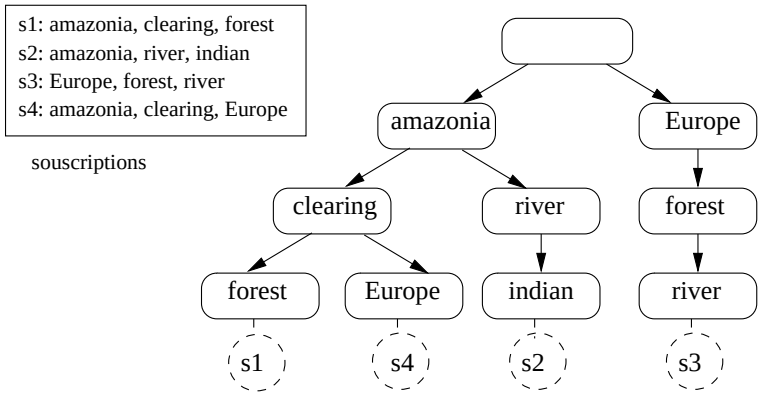

Figure 3: Basic use of prefix tree in pub/sub

Figure 3 illustrates an example of a prefix tree built on a set of subscriptions. Note that these techniques impose that:

$i)$ the subscriptions have a fix structure like set of predicates expressed as triples $(attribute, operator, value)$, $(attribute, value)$ ($AWPS$ model [42]), $(keyword, weight)$ (vector space model) or only $keywords$ (boolean model),. . ., and items have a corresponding structure for their atomic events;

$ii)$ the attributes/keywords are ordered for subscriptions and items;

The root of the tree corresponds to the empty prefix. Each node n is the root of a subtree that indexes all the subscriptions that start with the sequence corresponding to the tree traversal until n.

When a new item is received, a tree traversal permits to determine the set of subscriptions that match the item. We start with the root and read one atomic event after another. Then for each node n reached, the item is forward to its children if the input event match the node predicate. At each node, if there are some subscriptions that correspond, a matching notification is produced.

This approach is adopted for instance in [73, 50]. It allows to save place when the number of subscriptions is large and to reduce the number of tests for each new item. Indeed the worst case corresponds to an attempt with the longuest path, *i.e.* the longuest subscription, while the worst case w.r.t. space is when there exist no common prefix for the subscriptions. what requires as much space as a basic storage of all suscriptions.

[47] describes the *Bounding-based XML Filtering* or *BoXFilter*. This technique consists in indexing the *Prüfer encoding* of the different queries. The Prüfer encoding of a query is generated by iteratively removing the left-most leaf of the tree and adding its label to the sequence. The basic idea for the BoXFilter is somehow similar to the R-tree. It is structured as a balanced tree, where each node has a kind of bounding box determines by an upper and lower bounds for the set of Prüfer sequences this node index. These bounds are built by keeping respectively the highest and lowest labels (alphabetical order) at each position of the sequences. A subscription is inserted in the leaf that will have least enlargement. The split difficulty consists in finding the best partition of Prüfer sequences to minimize overlapping.

Other approaches are based on multi-dimensional indexing. they consider the subscriptions as multi-dimensional points that are indexed, and the events are considered as multi-dimensional points or range queries depending on the structure proposed. For instance in [12] the authors propose a structure to efficiently index range subscriptions like $x \leq attribute \leq y$ using a B-tree like structure in a P2P context, with decentralized indexing/querying algorithms. The basic idea is to relax the stateful subscriptions (*i.e.*, those whose result can not be proceeded only with the incoming event, like query with aggregation operators for instance) into stateless ones, and to map then each range subscription as a 2D point (val_{min}, val_{max}). Then each peer contacted is interested by the new event and/or limit the space that contains peers to contact. An extension named B^A-tree is presented in [11] to efficiently retrieve notifications for a set of subscriptions when the value differ from more than a given radius of a previous notified value. A somehow similar solution is proposed in [66] to index subscriptions viewed as a 2D points thanks to a UB-tree (basically a grid with Z-order) while events are considered as range queries. [62] proposes to distribute subscriptions on a set of peers using Chord hashing strategy applied on a single predicate of the subscription. There is a second level of indexing on each peer with hash table, prefix tree or inverted list, depending on the predicate nature, on each subscription predicate. Other approaches are based on R-tree like [8]. Here subscriptions are considered as multi-dimensional points in the space of the words, and these points are indexed with a R-tree.

4.2 Subscriptions factorization

Several approaches to reduce processing costs by evaluating *simultaneously* several queries may be found in the litterature. They essentially differ one from another by the nature of the queries considered and by the way these queries are grouped for evaluation. We distinguish mainly two families of subscriptions factorizing: proposals that aims at grouping subscription based on its whole definition and those that try to factorize common parts of the subscriptions.

[46] is an example of the first family. They define the notion of topic-cluster that are virtual topic which group topics with similar set of subscribers. Consequently a topic may belong to several clusters. One can subscribe to topics or to topic-clusters and when an event about a given topic is published, all the topic-cluster concerned received the event. Potentially local filters allow to fit exactly to the user's interest. Since the context is highly dynamic, the set of subscriptions is constantly changing an so is the topic-clusters architecture. The authors propose consequently a dynamic clustering strategy based on a cost function that considers both maintenance and dissimination overheads. They place a topic into an

existing cluster if the global cost decreases, noting that the higher publishing frequency of a topic is, the higher number of subscribers to a cluster must be to benefit for from an insertion into a given cluster. They also present conditions for merging two existing clusters. Basically we reduce the global cost by merging two clusters that share many subscriptions.

Another kind of subscriptions factorizing is proposed in [31] for XML like documents where subscriptions involve many documents and consequently joins. Oppositely to previous technique, only parts of subscriptions are factorized in order to evaluate simultaneously several subscriptions in an efficient way. Their rationale is First they tranform subscriptions into query tree patterns with join conditions. These subscriptions are then grouped according to their join graph, *i.e.* the skeleton of the sub-tree from the query that is involved in the join. For each join graph, called template in the paper, a table is created where tuples are each subscription that presents this template. For each template is created a conjonctive query that is evaluate on each incoming document. When a document satisfies this query, it is then propagated to each tuple (query) of the template.

4.3 Filtering

Another commonly adopted way to achieve scalability for evaluation when the number of subscriptions is very large is the filtering mechanism. While it is close in its principles to the indexing mechanism, its goal is quite different. Filters are essentially used either to locally discard events that could not match any subscription of a peer, or to decide if a given event should be forwarded to some other peers. So they are mainly very important in a dissemination process based on broadcasting. Note that in some pub/sub applications the role of queries and data is reversed since they consider that data are submitted and we attempt to match data stored on peers. But both kind of applications are conceptually similar so we discuss indifferently both of them. Another important requirement for filtering, what makes them also conceptually different from indexes, is that the subscription set is supposed static. Filters may differ by the way they consider incoming events (essentially XML documents) and especially how they manage subscriptions: with Bloom filters, automata or graphes.

Bloom filters

Bloom filters are largely used in peer-to-peer context to decide whether a peer should test an event with the subscriptions that it registered. A Bloom filter is essentially a bit-vector v of length m, associated to n independant hash functions, $h_1, \ldots, h_n$ that have a range between 1 and m. Then considering in turn the element e_i of a set of elements $\mathcal{E} = \{e_1, \ldots, e_r\}$, we set to 1 each bit at position $h_k(e_i), 1 \leq k \leq n$. Then to verify whether an element e' belongs to $\mathcal{E}$ we check if all the bits at position $h_k(e'), 1 \leq k \leq n$ are set to 1 in v. If one of the bits is set to 0, we are sure that the event do not belong to $\mathcal{E}$ without testing each e_i. However Passing through the filter do not guarantee the belonging since we may have a false positive.

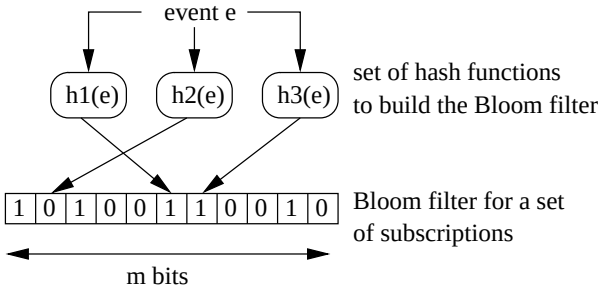

Figure 4: Example of Bloom filter with 3 hash functions

Figure 4 illustrates the principle of a Bloom filter, defined with three hash functions. The bit vector is produced thanks to hash functions applied on elements (atomic condition of the subscriptions). Then for an incoming event we check if all the positions in the bit vectors are 1, what is not the case here, so the event is discarded, no subscription my be interested by it.

[34] propose to store the different subscriptions in two data structures: $bfposet$ to store the atomic predicates of the subscriptions and $bftree$ a tree structure to store a disjunction of conjunctions of subscription predicates, both based on Bloom filters encoding. When an event (item) is received, each atomic event (attribute,operator,value) is matched with the $bfposet$, and for those atomic events that match allow to encode a corresponding Bloom filter for the whole event. This Bloom filter is passed to the $bftree$ that check if this Bloom filter satisfies a disjunction of conjunction of events corresponding to a subscription. In [27] the authors consider XML path queries (subscriptions) as query strings, and all the query strings of a user are represented by a Bloom filter. Different user's Bloom filters are recorded into a routing table on a peer. When a XML packet (small depth XML document) is received the set of pathes it is composed of is mapped to a bit-vector that is compared with Bloom filters in the routing table to determine the potential set of users the packet has to be forward to. [33] describes a distributed Bloom filter technique for P2P where the Bloom filter that summerizes the XML data of a peer is split into segments of similar length which are distributed on a DHT-network according to their segment number. [40, 41] propose multi-level Bloom filters for a content-based routing strategy for path query in a P2P system. The authors define *Breadth Bloom Filters* (BBF) for a XML tree T of j level as a set of simple Bloom filters BBF_i for each level $i < j$ of T, and *Depth Bloom Filters* (DBF) as a set of simple Bloom filters, each DBF_k summerizing the set of pathes with length k in T. Taking account the structure of the documents results in efficient processing of path queries. They investigate hierarchical organization where peers are organized like a tree, only tree roots are connected to the main channel and each parent level has a merged Bloom filter for its children, and peer clustering based on the similarity of their content with a Bloom filter that is a merge of all Bloom filters within the cluster. A new query is received by a root (resp. cluster representant) node and a matching attempt is performed, first with BBF_0 (resp. DBF_1) till BBF_j (resp DBF_j). Note that each root (resp. representant) node also stores the merged Bloom filter for all the other roots (resp. representant). If the matching with the local merged Bloom filter is successful, the query is forwarded to the children (resp. cluster members) while successful matches with some *outer* Bloom filters leads to a query forwarding to these nodes.

Automaton-based filtering

XFilter [4] was proposed to deal with XPath queries on XML documents. each path expression is considered as a Finite state Machine (FSM) similarly to example depicted in Figure 5.

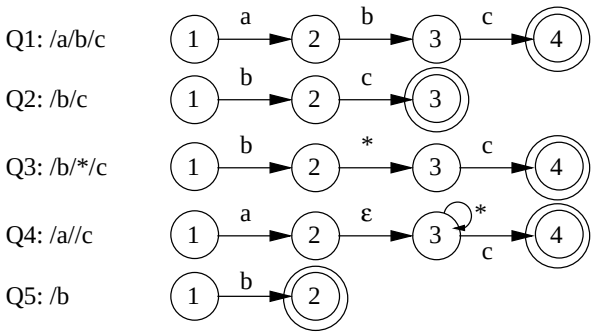

Figure 5: XFilter FSMs for a set of XPath queries

To face with the high number of queries to evaluate, the authors propose to reduce the number of FSMs to be examined thanks to an index structure based on inverted lists. An entry in the query index

is a name element, and for each entry, two lists are associated: the *CandidateList* and the*WaitList* that contains query path nodes The example Figure 6 illustrates the query index for the FSM of Figure 5 after the element b was read. Basically a state transition in the FSM corresponds to moving a query path node from the WaitList to the CandidateList for all the queries that were in the CandidateList of the corresponding element at the previous step.

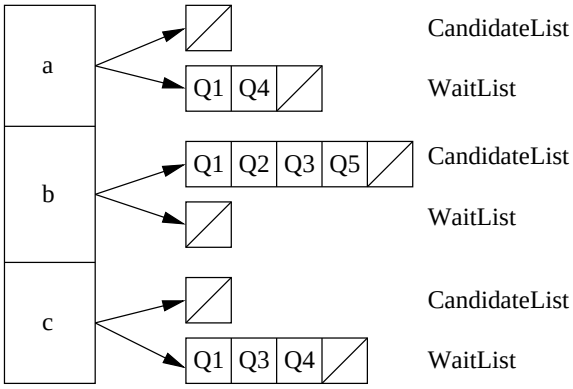

Figure 6: XFilter query index for queries of Figure 5

Partially based on this work, another approach for achieving high-performance in XML filtering is *YFilter* [18, 19, 17]. Authors merge all the XFilter FSM for the different queries into a single Non-Deterministic Finite Automata (NFA like) the one depicted in Figure 7.

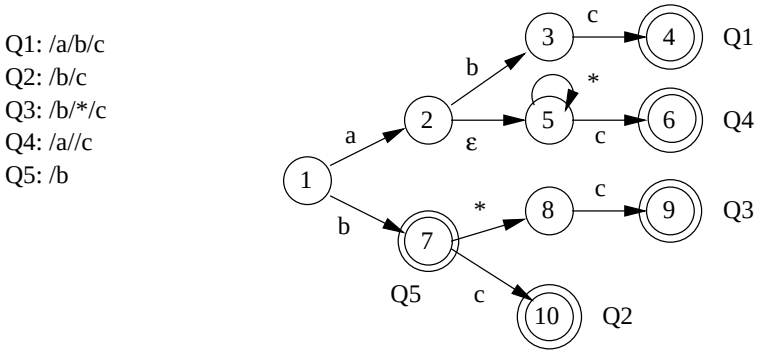

Figure 7: YFilter NFA for a set of XPath queries

When a new XML document is received it is first parsed. Then for each element a transition is triggered. Since each query also presents some value-based predicates on its path, these predicates must be checked during the processing. The authors propose two strategies: (i) *inline processing* where the predicates of the different queries concern by the state reached are immediately checked, what implies to store at each state the corresponding predicates and (ii) the *selection postpone* where the predicates of a given query are tested only when its accepting state is reached. YFilters are efficiently deployed in several implementations [20, 21] to achieve internet-scale dissemination of XML messages.

Another approach consists in representing XPath queries as Deterministic Finite Automata (DFA) [30, 29]. Here again a single automaton is built for the set of queries. Since the number of states grows exponentially with thz number of queries, the authors propose a lazy DFA building, where the states are added when needed during the processing. However to achieve this goal they need to store the NFA of the different queries in a table. Moreover they speed up the XML documents processing by adding some small binary extra-data in each document, called the *Stream Index (SIX)*, that allows to access XML data more quickly.

Graph-based filtering

Another technique used for filtering events is based on graph matching. The idea is that both documents and subscriptions are represented by graphes and we apply techniques dedicated to pattern tree in our context. The main difference is the assumption of a high number of subscriptions what implies to present an efficient matching.

For instance in *G-ToPSS* [53, 52], each document is a set of triples (subject,property,object) that could be represented as a directed labelled graph using RDF semantics. Similarly subscriptions are represented as a pattern of directed labelled graph, with potentially variables for subject and/or object node that can match any node and predicates like $(?x, op, val)$ to mean that "?x op val". Semantics is also considered during matching thanks to the use of a taxonomy. All subscriptions are represented by a single graph. The goal is to retrieve all subscriptions within this graph that match the document graph. To achieve this, the authors build a hash table on couples of nodes, each entry refers another hash table with all the edges between these two nodes (the key is the edge label). The linear matching algorithm proceeds in two steps: (1) for each edge for the document we retrieve the edges of the subscriptions that can match, then (2) we verify the bindings for the different variables and evaluate the constantes.

In [68], the authors introduce semantic Web technologies into pub/sub systems, namely *OPS*, and events must match subscriptions both semantically and syntactically. Events are converted into RDF graphs and subscriptions are graph patterns. The authors propose two general subgraph isomorphism algorithms for matching over overlapping graphs.

4.4 Ranking results

The feeds subscribed by the user might produce an important number of articles, whereas only a small part of them might be relevant to the user. One aim of the aggergators is to present the user a ranked list of articles, depending on their estimated relevance, that takes into consideration the user profile.

One simple solution for buiding a user prfile can be to describe it as a set of kewords he is interested in (similarly to keywords in search engines) and to rank incoming articles by the frequency of those keywords. In the Shoechicken system the user preferences are automatically learned from his behavior (articles that are read, deleted or bookmarked) [32]. Each articles has an associated abstract representation as a list of representative keywords deduced from its title, summary or the body text. Similarly to articles, the learned user model is represented by a vector, whose components corresponds to the TF*IDF scores of terms. For an article, the score of a vector component is obtained as a combination of the TF*IDF scores of that term in the body and the summary of an article. The relevance score of an article is obtained by computing the cosine similarity between the article vector and the user model vector. Weights used in different relavance score computations are adjusted by interpreting the user behavior (positive: the user reads the highly rated articles or negative: user deletes highly rated articles without reading them).

In the NectaRSS [56, 57] system, the user preferences are also automatically acquired from the content of the news he has already read. User profile is also represented as weights of a set of terms, which are computed at each user session as the frequency of these terms in the headlines and the summaires of feeds read during that session. This session profile is further used to update the existing user profile. Articles are ranked by computing comparing the characteristic vectors of their headlines with the one of the profile.

Other criteria, such as their novelty, can be used for ranking news articles. For example, the Newsjunkie system [23] detects the novelty of the infomation for each new story, based on the information that the user has already reviewed. The novelty-analysis algorithms models documents as smoothed probability distributions over words and named entities (people, organization and geographical locations), or as vectors of tf*idf weights on the same space. Documents are organized into groups

(documents on the same topic or produced by the same source). Articles are ranked by novelty-assessing algorithms that start from a seed story of articles the user has already read. The amount of novel information carried by some new article is predicted by comparing it with the seed story, by using several metrics. Such document similarity metrics are based on probabilistic distribution of words (such as the Kullback-Leiber or the Jensen-Shannon divergence) or compute the number of entities contained in the new article that are not in the seed story. Breaking news are reported by comparing each earticle with previous articles published in some given sliding window. Bursts of novelty correspond to bursts in the graph that plots the distance between each article and the preceding window. At the beginning of the burst, additional articles add new details causing the graph to rise.

Statistics on word occurences and on the waiting times in blogs and RSS feeds might help modelling the spread of ideas and opinions, to analyze the developement of trends and to follow dynamic opinion changes [43]. Empyrical analysis shows that the differences in word frequencies are very high ($1 - 10^6$), since their frequency is correlated to their "popularity". The word popularity is due to the popularity of the context/topic to which they are associated (e.g words on topic "music" are more frequent than words associated with e.g "tuberculosis"). The paper analysse whether the word frenquency and the waiting times are similar to the Poisson distributions that model independent events. Starting from the idea that the statistical analysis might be affected by the heterogeneity of word frequencies, the first step of the analysis is to build frequently-equivalent classes of words depending on the number of their occurrences during the considered period. The analysis shows that the distributions deviate from the exponential and that their shapes and deviations are the same for all words (words share the same statistical properties), independently on their frequency classes. The dynamics of the words in RSS feeds are dominated by bursts of activities followed by long periods in which the words do not appear, in the same way as the usage of words in Internet traffic, email or web browsing. Rare events (words that occur on average less that one time a day) have waiting times that are not similar to the Poisson process, and the distribution of waiting times do not depend on the class of words. Words occuring many times a day also have deviations from the Poisson distribution, showing a high probability to observe extreme events. This suggests that word usage is dominated by bursts of activities (maybe caused by a response to an external triggering factors or due to active discutions between bloggers) followed by long periods of rest.

Blogs and RSS feeds can be used for public opinion gathering and marketing purposes, therefore it is interesting to dispose of tools for automatically extracting significant topics. Several approaches have been proposed to identify significant topics, that apply time series analysis and frequency statistics to word/noun/phrase occurences. Topic Detection and Tracking identify significant terms (features) across documents. [54] compares the efficiency of three existing methods of feature selection, to identifying significant terms over a given period in an evolving RSS corpus and for computing the significance of terms at a given date. The significant terms help to identify ignificant topics in RSS feeds. Feature selection is generally used for identifying significant words in each document of a text collection, and help to determine document categories based on their significant terms. χ^2 seems to be the best to determine term significance for RSS feeds.

[67] propose an automatic online algorithm that ranks topics extracted from news, and shows to the user with high priority the topics that are both timely and important. The importance of a topic is determined by the frequency and the recency of the reports on this topic by the media, but also the attention of the users on this topic The basis for the topic ranking is the inconsistency between the media focus and the user attention (gap between what the media provide and what the user views).

Information in blogs and its dynamics differ from traditional web content, sice some blog posts generate additional posts leading to discussions [7]. Therefore, the search and the ranking methods in this context must be consequently adapted. Blogscope is a system for the online analysis of temporally ordered blogs, whose methods can be more generally applied to any temporally ordered streaming text sources (news, mailing lists, forums). For some given query keywords, the system returns blogs ranked on their autority. The intuition of the blog authority is that important blogs are the one that are the first to

report on news, and that will be therefore read by a large number of readers. In order to find these blogs, the system first computes the query synopsis, by finding the most correlated words to the query in some given time interval. These words "explain" the popularity of the query for those periods. The popularity of the words is computed by analysing how often they were mentioned in the blog posts during some interval. From the popularity curves, bursts in the popularity of keywords during the considered period are detected. Blogs that are related to the bursts in the query synopsis are rabked based on their authority. Authoritative blogs are temporally close to the occurence of the bursts (they gave rise of the burst in the synopsis query set) and are the one that are most linked in the blogosphere.

5 Data storage

5.1 Freshness/cache

Although the continually increasing information level available on the internet is welcome for the evolution of our society, such fast advancements also need an efficient way of exchanging information.

The bandwidth needed in a classical RSS distribution architecture in which all clients poll periodically a central server grows linearly with the number of subscribers. This centralized architecture can not sustain the RSS growth process and a distributed approach is necessary.

By using a collaborative RSS distribution architecture, the use of clients' network bandwidth can be optimized and the load placed on publishing servers can be dramatically reduced.

Distributed RSS systems

An example of a news feed exchange system is FeedEx [35]. It forms a distribution overlay network and its nodes not only fetch feed documents from the servers but also exchange them with neighbors. This way a node having the role of a client also has the role of a cache. This exchange allows nodes to reduce the frequency of fetching documents from the servers and decrease the server load.

The system FeedTree [58] introduces the usage of a P2P overlay network to distribute RSS feed data to subscribers. By using P2P event notification to distribute the feed items, peers share the bandwidth costs, the load placed on the provider is dramatically reduced and updated content is delivered to clients as soon as it is available.

An extension of the Atom technology that offers reliable data distribution and consistent data replication is presented in [36], where a symmetric feed replication architecture is proposed. The client acts like a mirror of the original feed source. The communication is done both ways, from publisher to the consumers and from the consumers to the publisher, so that the publisher knows that the consumers have received the data intended for them and is also able to measure the freshness of this data.

Freshness

The problem of RSS stream diffusion in a distributed architecture brings along the problem of synchronization between publishers and subscribers. An example of a study made on refresh strategies is introduced in [15]. The application presents the case of the web crawlers that use different policies in order to maintain the local data fresh when the data sources are updated autonomously and independently.

There are different refresh mechanisms to achieve this goal that depend on several dimensions of the synchronization process. Thus, a subscriber may choose to synchronize at different frequencies. If it has several subscriptions to manage, it can decide to refresh them using different orders and it can use different refresh frequencies for each of them, depending on the update frequencies of each publisher.

5.2 Eliminating irrelevant and redundant information

With the constant increase in the number of sources, the user may be quickly overwhelmed by new feeds. However not all this information may be interesting for the user, either because some feeds do not supply information relevant enough, or because some feeds present redundant information. We present in this section the general two-steps approach adopt to filter out irrelevant and redundant information. It consists first in defining a similarity measure for feeds, based on fuzzy-set IR models, and then to use this measure to retrive feeds that should be discarded.

Determining correlation factors

Similarity measures between words/sentences/concepts within a set of documents have been largely used in the information retrieval area. Most of the proposals [71, 24, 51] consider the fuzzy-set IR model where an important requirement for defining a similarity measure is the specification of a correlation factor between words. To determine how two given words are correlated, several approaches rely on statistical analysis of the articles from Wikipedia. With more than 10 millions of articles (more than 2.3 millions in English) written by more than 75,000 authors on various topics, Wikipedia provides a representative set of documents for stimating correlation between words. Each article is preprocessed in order to remove stopwords and to stem remaining words. Stopwords are very frequent words like articles, conjunctions, prepositions, numbers, non-alphabetic characteres, etc. Since they are generally meaningless, they should not be taken into account when analyzing the informational content of a document. The removal is performed by checking the presence of a given word into a commonly used stopwords hash table. Then the article is stemmed, *i.e.* we remove grammatical and conjugation endings to keep *normalized* words. Based on these remaining words, a correlation matrix is built where each cell is the correlation factor that represents how a given word is linked to another. Tipically this factor is estimated by the frequency the two words are found together in a document, the number of occurrences they appear inside a document, and the distance (number of words) between them.

Finding similar feeds

After this preliminary preprocessed work, that do not require any knowledge about the feeds that will be handled, we are now able to detect similar feeds. To compare two articles, the idea issued from fuzzy-logic domain consists generally in comparing each word w_i^1 of the first article of the first feed f_1, with all the words from the second feed f_2 and retrieving the associated correlation factors. We deduce then a correlation between w_i^1 and f_2 for instance through an weighted average of the obtained factors. Then inttegrating all the set correlation values of words of f_1 with f_2 we deduce a correlation between f_1 and f_2. One may argue that such an approach to define similarity between feeds is costly, but as shown in [51], the small size of an RSS document (25 words on average) makes this approach suitable for this case. Note that some works, *e.g.* [51], prefer to rely on a average of the correlation values between each n-gram (sequence of 2-5 words, with better results for digrams or trigrams) of f_1 and all the n-grams of f_2, rather than to compare single words.

Filtering out/finding similar feeds

Once the similarity between feeds has been defined, it can be used to retrieve feeds that present information related to a given feed [71, 51]. It allows to *automatically* detect documents that may be possibly complementary to the ones a user is interested by, what avoids it to intensively mine among thousands of feeds. Another application for the similarity measure is the junk feeds detection since once a junk feed has been identified by the user, all the similar feeds can be discarded, limiting the spam for the

user. Or a user may consider that when he has consulted an article, he won't be interested by feeds that provide similar articles since he consider the first feed provides enough information. Similarity can also be used for classification purpose: clusters of feeds with high similarity values may be constituted what permits the user to easily consult a set of feeds dedicated to a peculiar topic (inside the same cluster). Clusterisation is also useful for filtering out redundant feeds. For instance in [24], a ranking is performed inside each cluster of feeds, according to the similarity value of each document computed *inside* the cluster itself. The lower similarity value, the less informative feed *w.r.t.* the cluster. Consequently, the feeds with lower similarity values are discarded since they provide no (or few) additional relevant information compare to other feeds of the cluster.

6 Conclusion

References

[1] FeedFire. http://www.feedfire.com.

[2] Georss: Geographically encoded objects for rss feeds. http://georss.org.

[3] The yahoo! pipes feed aggregator. http://pipes.yahoo.com.

[4] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 53–64, 2000.

[5] A. Arasu, S. Babu, and J. Widom. The CQL continuous Query Language : Semantic Foundations and Query Execution. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 121–142, 2006.

[6] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 480–491, 2004.

[7] N. Bansal and N. Koudas. Searching the Blogosphere. In *Proc. Intl. Workshop on the Web and Databases (WebDB)*, 2007.

[8] S. Bianchi, P. Felber, and M. Gradinariu. Content-Based Publish/Subscribe Using Distributed R-Trees. In *Proc. Europ. Conf. on Parallel Processing (Euro-Par)*, pages 537–548, 2007.

[9] S. Bossa, G. Fiumara, and A. Provetti. A Lightweight Architecture for RSS Polling of Arbitrary Web sources. In *Proc. Intl. Workshop From Objects to Agents (WOA)*, 2006.

[10] I. Botan, P. M.Fischer, D. Florescu, D. Kossman, T. Kraska, and R. Tamosevicius. Extending XQuery with Window Functions. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 75–86, 2007.

[11] B. Chandramouli, J. Phillips, and J. Yang. Value-Based Notification Conditions in Large-Scale Publish/Subscribe Systems. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 878–889, 2007.

[12] B. Chandramouli, J. Xie, and J. Yang. On the Database/Network Interface in Large-Scale Publish/Subscribe Systems. In *Proc. Intl. ACM Symp. on the Management of Data (SIGMOD)*, pages 587–598, 2006.

[13] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, V. R. Sam Madden, F. Reiss, and M. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 11–18, 2003.

[14] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. Intl. ACM Symp. on the Management of Data (SIGMOD)*, pages 379–390, 2000.

[15] J. Cho and H. Garcia-Molina. Effective Page Refresh Policies for Web Crawlers. *ACM Trans. on Database Systems (TODS)*, 28(4):390–426, 2003.

[16] F. de Cindio, G. Fiumara, M. Marchi, A. Provetti, L. A. Ripamonti, and L. Sonnante. Aggregating Information and Enforcing Awareness Across Communities with the Dynamo RSS Feeds Creation Engine: Preliminary Report. In *Proc. Intl. OTM Workshops*, pages 227–236, 2006.

[17] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Trans. on Database Systems (TODS)*, 28(4):467–516, 2003.

[18] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 341–344, 2002.

[19] Y. Diao and M. J. Franklin. High-Performance XML Filtering: An Overview of YFilter. *IEEE Data Eng. Bull.*, 26(1):41–48, 2003.

[20] Y. Diao and M. J. Franklin. Query Processing for High-Volume XML Message Brokering. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 261–272, 2003.

[21] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 612–623, 2004.

[22] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In *Proc. Intl. ACM Symp. on the Management of Data (SIGMOD)*, pages 115–126, 2001.

[23] E. Gabrilovich, S. T. Dumais, and E. Horvitz. Newsjunkie: Providing Personalized Newsfeeds via Analysis of Information Novelty. In *Proc. Intl. Conf. on World Wide Web (WWW)*, pages 482–490, 2004.

[24] I. Garcia and Y.-K. Ng. Eliminating Redundant and Less-Informative RSS News Articles Based on Word Similarity and a Fuzzy Equivalence Relation. In *Proc. Intl. Conf. on Tools with Artificial Intelligence (ICTAI)*, pages 465–473, 2006.

[25] H. Geng, Q. Gao, and J. Pan. Extracting Content for News Web Pages based on DOM. *Intl. Jour. of Computer Science and Network Security (IJCSNS)*, 7(2):124–129, 2007.

[26] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5–14, 2003.

[27] X. Gong, Y. Yan, W. Qian, and A. Zhou. Bloom Filter-based XML Packets Filtering for Millions of Path Queries. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 890–901, 2005.

[28] G. Gou and R. Chirkova. Efficient algorithms for evaluating XPath over streams. In *Proc. Intl. ACM Symp. on the Management of Data (SIGMOD)*, pages 269–280, 2007.

[29] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Trans. on Database Systems (TODS)*, 29(4):752–788, 2004.

[30] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *Proc. Intl. Conf. on Database Theory (ICDT)*, pages 173–189, 2003.

[31] M. Hong, A. J. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively Multi-Query Join Processing in Publish/Subscribe Systems. In *Proc. Intl. ACM Symp. on the Management of Data (SIGMOD)*, pages 761–772, 2007.

[32] J. Horsley, M. Wooten, and E. El-Sheikh. Shoechicken: An Intelligent System for Recommending RSS/Atom Content. In *Proc. Intl. Conf. on on Computers and Their Applications (CATA)*, pages 206–210, 2006.

[33] C. Jamard, G. Gardarin, and L. Yeh. Indexing Textual XML in P2P Networks Using Distributed Bloom Filters. In *Proc. Intl. Conf. on Database Systems for Advances Applications (DASFAA)*, 2007.

[34] Z. Jerzak and C. Fetzer. Bloom Filter Based Routing for Content-Based Publish/Subscribe. In *Proc. Intl. Conf. on Distributed Event-Based Systems (DEBS)*, pages 71–81, 2008.

[35] S. Jun and M. Ahamad. FeedEx: Collaborative Exchange of News Feeds. In *Proc. Intl. Conf. on World Wide Web (WWW)*, pages 113–122, 2006.

[36] F. Kart, L. E. Moser, and P. M. Melliar-Smith. Reliable Data Distribution and Consistent Data Replication Using the Atom Syndication Technology. In *Proc. Intl. Conf. on Internet Computing (ICOMP)*, pages 124–132, 2007.

[37] M. L. Kersten, E. Liarou, and R. Goncalves. A Query Language for a Data Refinery Cell. In *Proc. Intl. Workshop on Event-driven Architecture, Processing and Systems (EDA-PS)*, 2007.

[38] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 1309–1312, 2004.

[39] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 228–239, 2004.

[40] G. Koloniari, Y. Petrakis, and E. Pitoura. Content-Based Overlay Networks for XML Peers Based on Multi-level Bloom Filters. In *Proc. Intl. Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P)*, pages 232–247, 2003.

[41] G. Koloniari and E. Pitoura. Content-Based Routing of Path Queries in Peer-to-Peer Systems. In *Proc. Intl. Conf. on Advances in Database Technology (EDBT)*, pages 29–47, 2004.

[42] M. Koubarakis, T. Koutris, C. Tryfonopoulos, and P. Raftopoulou. Information Alert in Distributed Digital Libraries: The Models, Languages, and Architecture of DIAS. In *Proc. Europ. Conf. on Research and Advanced Technology for Digital Libraries (ECDL)*, pages 527–542, 2002.

[43] R. Lambiotte, M. Ausloos, and M. Thelwall. Word Statistics in Blogs and RSS feeds: Towards Empirical Universal Evidence. *The Computing Research Repository (CoRR)*, 2007.

[44] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proc. Intl. ACM Symp. on the Management of Data (SIGMOD)*, pages 311–322, 2005.

[45] X. Li, J. Yan, Z.-H. Deng, L. Ji, W. Fan, B. Zhang, and Z. Chen. A Novel Clustering-Based RSS Aggregator. In *Proc. Intl. Conf. on World Wide Web (WWW)*, pages 1309–1310, 2007.

[46] T. Milo, T. Zur, and E. Verbin. Boosting Topic-Based Publish-Subscribe Systems with Dynamic Clustering. In *Proc. Intl. ACM Symp. on the Management of Data (SIGMOD)*, pages 749–760, 2007.

[47] M. M. Moro, P. Bakalov, and V. J. Tsotras. Early Profile Pruning on XML-aware Publish/Subscribe Systems. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 866–877, 2007.

[48] T. Nanno, T. Fujiki, Y. Suzuki, and M. Okumura. Automatically Collecting, Monitoring, and Mining Japanese Weblogs. In *Proc. Intl. Conf. on World Wide Web (WWW)*, pages 320–321, 2004.

[49] T. Nanno and M. Okumura. HTML2RSS: Automatic Generation of RSS Feed Based on Structure Analysis of HTML Document. In *Proc. Intl. Conf. on World Wide Web (WWW)*, pages 1075–1076, 2006.

[50] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML Data on the Web. In *Proc. Intl. ACM Symp. on the Management of Data (SIGMOD)*, pages 437–448, 2001.

[51] M. S. Pera and Y.-K. Ng. Finding Similar RSS News Articles Using Correlation-Based Phrase Matching. In *Proc. Intl. Conf. on Knowledge Science, Engineering and Management (KSEM)*, pages 336–348, 2007.

[52] M. Petrovic, H. Liu, and H.-A. Jacobsen. CMS-ToPSS: Efficient Dissemination of RSS Documents. In *Proc. Intl. Conf. on Very Large Database (VLDB)*, pages 1279–1282, 2005.

[53] M. Petrovic, H. Liu, and H.-A. Jacobsen. G-ToPSS: Fast Filtering of Graph-Based Metadata. In *Proc. Intl. Conf. on World Wide Web (WWW)*, pages 539–547, 2005.

[54] R. Prabowo and M. Thelwall. A comparison of feature selection methods for an evolving rss feed corpus. *Inf. Process. Manage.*, 42(6):1491–1512, 2006.

[55] I. Rose, R. Murty, P. R. Pietzuch, J. Ledlie, M. Roussopoulos, and M. Welsh. Cobra: Content-based filtering and aggregation of blogs and rss feeds. In *Proc. Intl.Symp. on Networked Systems Design and Implementation (NSDI)*, 2007.

[56] J. J. Samper, P. A. Castillo, L. Araujo, and J. J. M. Guervós. NectaRSS, an RSS Feed Ranking System that Implicitly Learns User Preferences. *The Computing Research Repository (CoRR)*, 2006.

[57] J. J. Samper, P. A. Castillo, L. Araujo, J. J. M. Guervós, O. Cordón, and F. Tricas. NectaRSS, an Intelligent RSS Feed Reader. *Journal of Network and Computer Applications (JNCA)*, 31(4):793–806, 2008.

[58] D. Sandler, A. Mislove, A. Post, and P. Druschel. FeedTree: Sharing Web Micronews with Peer-to-Peer Event Notification. In *Proc. Intl. Workshop on Peer-to-Peer Systems (IPTPS)*, pages 141–151, 2005.

[59] R. O. Software. http://www.rssowl.org.

[60] R. W. E. Software. http://www.extralabs.net/rss-wizard.htm.

[61] W. Tan, F. Rao, Y. Fan, and J. Zhu. Compatibility Analysis and Mediation-Aided Composition for BPEL Services. In *Proc. Intl. Conf. on Database Systems for Advances Applications (DASFAA)*, pages 1062–1065, 2007.

[62] C. Tryfonopoulos, S. Idreos, and M. Koubarakis. Publish/Subscribe Functionality in IR Environments Using Structured Overlay Networks. In *Proc. Intl. ACM Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 322–329, 2005.

[63] F. M. Villoria, O. Díaz, and S. F. Anzuola. Powering RSS Aggregators with Ontologies - A Case for the RSSOWL Aggregator. In *Proc. Intl. Conf. on Enterprise Information Systems (ICEIS)*, pages 197–200, 2006.

[64] W3C. Atomenabled. http://www.atomenabled.org/.

[65] W3C. Resource description framework. http://www.w3.org/RDF/.

[66] B. Wang, W. Zhang, and M. Kitsuregawa. UB-Tree Based Efficient Predicate Index with Dimension Transform for Pub/Sub System. In *Proc. Intl. Conf. on Database Systems for Advances Applications (DASFAA)*, pages 63–74, 2004.

[67] C. Wang, M. Zhang, L. Ru, and S. Ma. Automatic Online News Topic Ranking Using Media Focus and User Attention Based on Aging Theory. In *Proc. Intl. Conf. on Information and Knowledge Managementon (CIKM)*, pages 1033–1042, 2008.

[68] J. Wang, B. Jin, and J. Li. An Ontology-Based Publish/Subscribe System. In *Proc. Intl. ACM/IFIP/USENIX Middleware Conf. (MIDDLEWARE)*, pages 232–253, 2004.

[69] J. Wang and K. Uchino. Efficient RSS Feed Generation from html Pages. In *Proc. Intl. Conf. on Web Information Systems and Technologies (WEBIST)*, pages 311–318, 2005.

[70] J. Wang, K. Uchino, T. Takahashi, and S. Okamoto. RSS Feed Generation from Legacy HTML Pages. In *Proc. Asia-Pacific Web Conference (APWeb)*, pages 1071–1082, 2006.

[71] K. Wegrzyn-Wolska and P. S. Szczepaniak. Classification of RSS-Formatted Documents Using Full Text Similarity Measures. In *Proc. Intl. Conf. on Web Engineering (ICWE)*, pages 400–405, 2005.

[72] E. Wu, Y. Diao, and S. Rizvi. High-Performance Complex Event Processing over Streams. In *Proc. Intl. ACM Symp. on the Management of Data (SIGMOD)*, pages 407–418, 2006.

[73] T. W. Yan and H. Garcia-Molina. Index Structures for Selective Dissemination of Information Under the Boolean Model. *ACM Trans. on Database Systems (TODS)*, 19(2):332–364, 1994.

[74] T. W. Yan and H. Garcia-Molina. The SIFT Information Dissemination System. *ACM Trans. on Database Systems (TODS)*, 24(4):529–565, 1999.