HAL open science

Monitoring Peer to Peer Systems

Serge Abiteboul, Bogdan Marinoiu

To cite this version:

Serge Abiteboul, Bogdan Marinoiu. Monitoring Peer to Peer Systems. Bases de données avancées, Oct 2007, Marseille, France. inria-00259064

HAL Id: inria-00259064

https://inria.hal.science/inria-00259064

Submitted on 26 Feb 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Monitoring Peer to Peer Systems*

Serge Abiteboul Bogdan Marinoiu
INRIA Futurs† and University Paris Sud
firstname.lastname@inria.fr

July 23, 2007

Abstract

In this paper, we are concerned with the distributed monitoring of P2P systems. We introduce the P2P Monitor system and a new declarative language, namely P2PML, for specifying monitoring tasks. A subscription is compiled into a distributed algebraic plan which is described using an algebra over XML streams. We introduce a filter for streams of XML documents that scales by processing first simple conditions and then, if still needed, evaluating complex queries. We also show how particular tasks can be supported by identifying subtasks already provided by existing streams.

Keywords : distributed data management, stream processing, peer to peer systems, XML, databases, Web services, active documents

1 Introduction

Peer to Peer systems have become popular over the last decade mainly because they provide support for community content sharing and for loosely coupled distributed applications. Their use is still hindered by the difficulty to observe such highly dynamic systems, and to gather information on their functioning. This is the topic of the present paper where we propose a very generic monitoring system for P2P systems. A main contribution is that subscriptions, specified in a high-level declarative language, are compiled into distributed algebraic plans over XML streams. We present a new algorithm for efficient filtering and introduce some novel P2P technology for re-using already deployed monitoring tasks.

We introduce a system, called *P2P Monitor* (P2PM for short) for monitoring P2P systems. P2PM is itself a P2P system. So, we have two P2P systems that coexist, the monitored one (possibly several monitored systems) and the monitoring one (namely P2PM). Clearly, the same machine may participate in both kinds of P2P networks. Each peer in the monitored P2P system can observe some activities (e.g. data changes or communications) happening *locally* and thus become a *primary source* of monitoring information. We represent such information as a *data stream*, and more precisely as an ActiveXML stream (to be defined). Such data may then be transmitted between peers in *channels* (e.g., using point to point or broadcasting). The peers in P2PM perform operations on data streams to produce new streams or publish resulting streams.

A main characteristic of the system is the use of a high level Subscription Language, P2PML, short for *Peer-to-Peer Monitor Language*. A monitoring task is specified to the system in this high level declarative language. This specification is then compiled into an *algebraic mon-*

*This work is partially supported by ANR-06-MDCA-005 grant DocFlow

†Address: Parc Club Orsay Université, ZAC des vignes, 4, rue Jacques Monod - Bâtiment G, 91893 ORSAY Cedex

itoring plan involving both the monitored system and P2PM. Thus, our work is founded on an algebra over data streams (i.e., a library of services) that is based on the ActiveXML algebra framework for distributed data management [4]. In particular, the algebraic operators include:

Alerters These 0-ary operators detect local events of interest and produce data streams.

Stream processors These stream operators are used to filter stream or apply more complex operations such as merge or join of streams.

Publishers The role of these operators is to publish the data of resulting streams.

Alerters are specific to the particular software components that one wants to monitor. We will mention alerters we implemented for various systems.

The most important stream processor is the *Filter*, which can perform efficiently a large number of filtering queries over a stream with possibly intense traffic. An important aspect, from a performance viewpoint, is that it checks separately simple test conditions that are evaluated on the fly and more complex ones that require the use of an XML query processor. Other stream processors support operations such as *Join*, *Merge*, *DuplicateElimination*, or *Aggregate*. Some of them (e.g. *Join*) need auxiliary storage.

A monitoring subscription also details how resulting data streams should be published. We may require that items from a data stream be sent by email, saved in a file or published on the Internet as an XML document or more specifically as an RSS feed. Finally, we may request to have a stream published as a *channel*, i.e., a stream that users or other processors may subscribe to.

A main goal for this work is to be able to scale to a large number of subscriptions and a large number of peers. An important issue in this context is the placement of operators such as filters close to the data they work on when possible, to save on data transfers. As we will see, ActiveXML (besides serving as an algebraic framework) is also used for reducing the amount of data that is transferred by providing information *intensionally* when possible to avoid useless transfers and replication. With respect to replication, P2PM also includes the means to reduce the load on the system by re-using existing data streams. When a new monitoring subscription arrives, the system searches for existing streams that could help support (portions of) the new task. This (monitoring) service discovery is implemented on top of a P2P content management system, namely KadoP [3].

Motivations A wide range of applications can benefit from such a P2P monitoring of P2P systems. To conclude this section, we mention briefly some, insisting on particular ones that motivated the present work.

P2PM can be used to monitor Web services in a Web community. Towards this goal, we implemented an alerter to monitor SOAP messages. The implementation was for Jakarta Tomcat with *Axis* support for Web services. This could serve, for instance, to follow the concurrent execution of large number of workflow instances in telecom services (e.g., BPEL workflows [7]) to detect malfunctions, gather statistics, understand usage patterns, provide better user support, support billing, etc.

This monitoring of Web services was originally developped for monitoring the activities of ActiveXML peers [6]. Furthermore, we also implemented an alerter for monitoring updates of the ActiveXML repository. Such alerters are in fact in the important class of database systems' alerters, or to use the database terminology, *database triggers*. It should be noted that P2PM will replace the actual monitoring system in future releases of ActiveXML peers.

An application we considered for testing is the surveillance of the content published by Web servers (e.g., for a community portal). In

particular, we developed an alerter to monitor changes in RSS feeds. We can also monitor changes in XML and XHTML pages using the XyDiff library [9]. Using P2PM, it then becomes immediate to obtain a push-based surveillance of a very large number of Web pages.

A main motivation of our work is the monitoring of the Edos content sharing network [11]. Edos is a P2P distribution system that is developed within a European project, notably in cooperation with the Mandriva company (originally known as MandrakeSoftware). In Edos, the data consists of the Mandriva Linux distribution, i.e., about 10 000 software packages and the associated metadata. The metadata for one Mandriva distribution is more than 100 megabytes of XML data. Of course, the system has to support simultaneously many (time-based) versions of the distribution. The peers (when the application will be deployed) will be Mandriva Linux developers, so potentially a population in the hundreds of peers. The monitoring is primarily used to gather statistics about the peers (e.g., number, efficiency, reliability) and the usage of the system (e.g., query rate, query response time, publication latency). More generally, Mandriva designed some metrics for evaluating the quality of service of the distribution system and the monitoring system is in charge of providing measures in this metrics.

The paper is organized as follows. Section 2 introduces the subscription language. Section 3 defines the notions of streams and channels and presents the architecture of *P2PM* as well as the *Stream Algebra*. Section 4 focuses on the presentation of a module efficiently implementing stream filtering. Section 5 shows techniques allowing stream reuse. In Section 6, we compare *P2PM* to other systems and in Section 7, we conclude.

```
for $c1 in outCOM(<p>http://a.com</p>
                  <p>http://b.com</p>),
 $c2 in inCOM(<p>http://meteo.com</p>)
let $duration := $c1.responseTimestamp
       - $c1.callTimestamp
where
 $duration > 10 and
 $c1.callMethod = "GetTemperature" and
 $c1.callee = "http://meteo.com" and
 $c1.callId = $c2.callId
return
 <incident type = "slowAnswer">
  <client>{$c1.caller}</client>
  <tstamp>{$c2.callTimestamp}</tstamp>
  <request>
  {$c2/alert/call//temperature/city}
  </request>
  <duration>{$duration}</duration>
 </incident>
by email to monitor-office@meteo.com;
```

Figure 1: A monitoring subscription

2 P2P Monitoring Language

In this section, we briefly describe the subscription language and thereby the functionalities of the system.

The monitoring language called here P2PML allows specifying in a declarative manner the events a user is interested in, i.e., where they come from, their kinds, some conditions on them. A statement in this language, called a *(monitoring) subscription*, also specifies how the user should be notified of detected events. Such a subscription may be submitted to any peer in the system. The system is then globally in charge of performing the corresponding *(monitoring) task*, and in particular, of assigning operators that build and process streams of data.

For instance, consider the subscription involving 3 peers in Figure 1. The monitor office of *meteo.com* wants to detect when the *meteo* service it provides to some peers, *a.com* and

b.com, is too slow (takes more than 10s). When such a situation occurs, an email is sent to the monitor office with some information about the call that was too slow.

As one can notice, the syntax is inspired by the XQuery's FLWR [24]. Statements in the language use five types of clauses that are discussed next.

The *FOR* clause specifies the *information sources*. Three sources are used in the example: two alerters of outgoing calls (*outCOM*), one on peer *a*, one on *b*, and an alerter of incoming calls (*inCOM*) at *meteo.com*. Note that the same "call" is an out-call for *a.com* and an in-call for *meteo.com*. The main role of the clause is to define XML variables: $\$c1$ for events detected by the two clients and $\$c2$ for the events detected at the server. In the example, there is a fixed number of peers that are monitored. We will see also how to monitor a possibly large and varying number of peers.

The functions in the FOR clause define the nature of the alerters that are used. In the example, the *outCOM* and *inCOM* alerters monitor the communications in SOAP RPC calls. Such communications consist of a pair of a *Call* and a *Response*. The types of these functions determine the type of the variables they introduce. An alerter produces a stream of XML trees. An element of such stream is called a *stream item*. The nature of the alerter (e.g., here a Web service alerter) determines the type of the stream item. In a stream item, we distinguish two parts:

1. the attributes of the root that typically gather some generic information. In the case of a Web service alerter, these will include information such as the identifier of the call, the identifier of the server that was called or the time of the call. This kind of information is very important for users since a large number of subscriptions will typically impose selection conditions on these attributes.

2. the subelements of the root that possibly have some more complex structure. In the case of the Web service alerter, the entire SOAP message or an error message may be included in the alert.

The main reason for this distinction is that the filter will process selections over these two kinds of data in the alert differently for performance reasons.

The *LET* clause enables the user to define more variables based on the already defined ones. This may be using arithmetic expressions as in the definition of $\$duration$ (the time between the firing of a call and the reception of its answer). New variables may also be defined using XPath expressions. The variables in expressions are required to evaluate to atomic values. When this is not the case, the variable will be assigned the value *NULL*.

The *WHERE* clause imposes some Boolean conditions on the variables. (For the moment, the system supports only conjunctions of conditions.) The conditions are equality or inequality conditions on the atomic variables (integer or strings) or on some atomic values that can be extracted from variables using XPath queries such as $\$c1/alert/call//date$ or Web service call. An example of restriction on the alerts is:

$$\$c1/alert[@callMethod = \text{“}GetTemperature\text{”}]$$

Since such conditions on the root attributes of alerts are very common, we use a dot notation as syntactic sugaring. For instance, the previous condition can be expressed as:

$$\$c1.callMethod = \text{“}GetTemperature\text{”}$$

Conditions on the root attributes will be treated differently for performance reasons. They are called *simple conditions*. The *Where* clause in the example is the conjunction of 4 simple conditions.

The *Return* clause specifies the output of the stream. Each time some values of the variables

over the input streams match the conditions, an output XML tree is obtained in the output stream. This output is defined as some XML data with possibly curly brackets-guarded expressions that are to be evaluated only at runtime. The use of XPath expressions or of *Web service calls* is authorized in these guarded expressions.

Finally, the *BY* clause determines how the user gets notified: by email, through a *channel* or in a file (an XML document which can be an XHTML Web page or a RSS feed). For email, the user can specify the number of stream items that have to be sent simultaneously or the time frequency of the emails. In all cases, the user can specify some bound on the number of stream items. Publication in a channel (the most interesting case) consists in publishing a stream that clients can subscribe to or other subscriptions can refer to. This will be detailed in Section 3.

As previously mentioned, the main concept of the system is the stream. A subscription produces a stream. Furthermore, the syntax declaration of a FOR clause is:

```
<fclause> :- FOR <var> in <stream>
   (, <var in <stream>  )*
```

So, in particular we can nest subscriptions:

```
for $x in ( for $y in ... ) ...
```

Also, functions such as *inCOM* take a stream as input and produce a stream as output. The input stream does not have to be fixed. So for instance, one can define:

```
for $j in
  areRegistered(<p>s.com/dht</p>)
for $c in inCOM($j) ...
```

The peer *s.com* is a member of a DHT. The service *s.com/dht* is an access point to that DHT from Peer *s.com*. Then, the variable *j* is bound by the function *areRegistered* to the set of peers currently in that DHT (a stream). The *inCOM* function maintains the collection of registered peers and monitors in calls for the peers in that collection. Note that *areRegistered* may produce messages of the form:

```
<p-join>a.com</p-join> % a joins
<p-leave>a.com</p-leave> % a leaves
```

In this latter case, *inCOM* removes peers from the collection of monitored peers. In a DHT context, the dynamicity of the network is typically very standard, so such a feature is essential.

One can also request duplicate-free results by preceding the content of the *RETURN* clause by the *distinct* attribute as in:

```
return distinct <a> {$y} </a>
```

As could be observed, the syntax of P2PML has the flavor of that of XQuery. Clearly, it is very different because its role is not to query XML documents but to monitor P2P systems; in particular the BY clause has no analogue in XQuery. The influence of XQuery is motivated by the emphasis on XML streams. Indeed, a related approach is that of StreamGlobe [22, 15] where XQuery is used to query streams of XML data. The streams in our paper form the core of the P2P monitoring.

3 P2P Monitor

We first present the general architecture of P2PM. Then we focus on two key aspects: (i) the underlying ActiveXML algebra that is used within the system and (ii) the generation of monitoring plans.

3.1 Architecture

The functional architecture of a peer in the P2PM is shown in Figure 2. Between them and with other peers, the modules mostly exchange streams of ActiveXML trees. The notion of ActiveXML stream will be made more

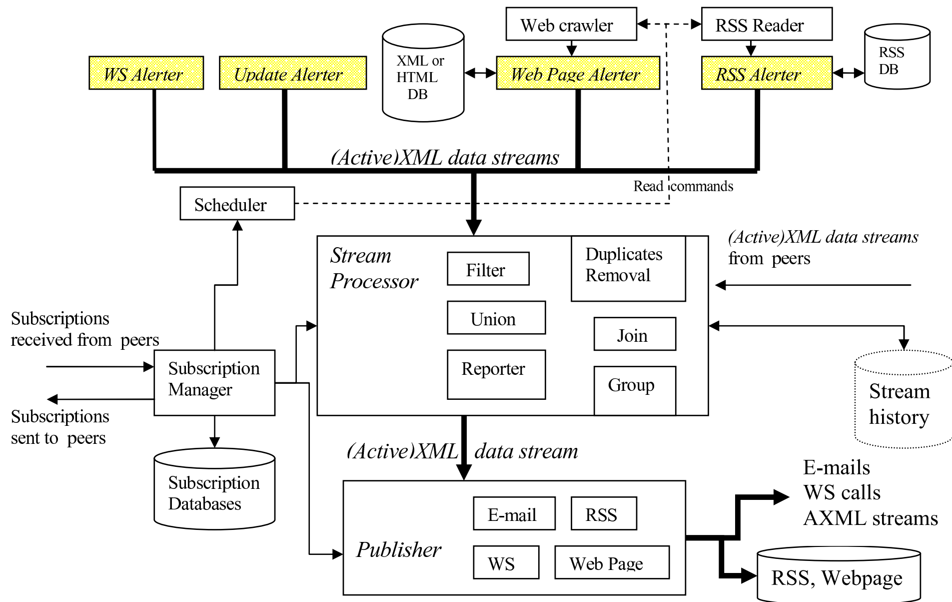

Figure 2: The architecture of a peer in P2PM

precise in Section 3.2 and an algebra over ActiveXML streams will be described in Section 3.3.

The minimum required to be a P2PM peer in a P2PM network is to run a Subscription Manager. Clients not equipped with one can always address their subscription requests to any peer of the system. Alerters may be installed on P2PM peers. They may be also installed on machines that are not P2PM peers but are accepting to be monitored. Note that this will prevent the system from pushing monitoring processing tasks such as filtering close to such sources. A peer may also host some stream processors and some publishers. These are discussed next.

Subscription manager When a user requests a *monitoring task* in P2PML, she forwards the subscription to a peer which becomes *Subscription Manager* for this subscription. A peer keeps the information about all subscriptions under his responsibility in a database named *Subscription Database*.

The *Subscription Manager* is in charge of translating the subscription into a monitoring plan, optimizing this plan, and then deploying the optimized plan (See Section 3.4). The deployment of the plan is done by issuing *local subscriptions*, expressing demands for *monitoring subtasks*, to each peer that is involved in the global plan.

Each of these peers will first store the *local subscription* in a *Local Subscription Database*, then generate a local plan, optimize it locally and install the needed stream processors.

Alerters Each alerter (a 0-ary operator producing a stream) is specialized in detecting particular events in some systems that are external to P2PM. The system currently support four kinds of alerters:

- *Web Application Server Alerters* are continuously surveying the communications of Web application servers. They are implemented as *Axis* handlers.

- *eXist Update Alerters* are continuously surveying updates in eXist XML database system [12].

- *RSS Feed Alerters* are periodically polling specific feeds and are reporting the changes, i.e., added, modified or deleted items.
- *WebPage Alerter* are periodically polling specific XHTML pages and comparing their snapshots.

Alerters that do polling need to be triggered at specific time intervals. The *Scheduler Submodule* is responsible of implementing scheduling policies and of triggering the alerters.

In all cases, the alerting may be tailored, e.g., by using *delta* techniques to represent data changes in Web pages or by including or not the SOAP message for a Web Application Server alert.

Stream processors Some of the stream processors are *stateless*, i.e., their behavior does not depend on the history of their input streams, e.g., *Filter* (σ), *Restructure* (Π), *Merge* ($\cup$). Others are *stateful*, e.g., *Duplicate-removal*, *Join* ($\bowtie$) or *Group-by*. We next briefly discuss the main processors we already support. The *Filter* processor whose performance have deep implications on the usability of the system, will be discussed in Section 5.

Join takes two streams as input and generates an output stream. *Join* can be parameterized by a *join predicate*. For instance, in Section 2, we need to "join" the alerts of the two streams using the equality of the *callId*s. Such a join is typically very used in monitoring systems to follow a task across different peers. For each new tree t in one of the input streams, the *history* of the other stream is searched for a tree t' so that (t, t') matches the *join predicate*. An index over that history is used to speed up the search. The result of *Join* includes information about the matching pair of trees. *Duplicate-removal* detects similar trees based on a *duplicate criteria*. *Merge* takes several streams as inputs and generates a single stream that is their union. One could connect it to *Duplicate-removal* to eliminate duplicates.

Restructure takes as input a single stream. It is defined by a *template* that specifies the restructuring that has to be done at runtime based on the input. In the simplest case, the template specifies a projection of an input tree. It may specify some more complex restructuring and require the use of XQuery, XSLT or Web services. *Restructure* is typically used to perform the construction specified in the *return* statement of a subscription. For the input tree that passed the *Where* test (or for a tuple of trees, e.g., in case of joins), the template is used to construct the output tree.

Publisher *Publisher* is an operator in charge of publishing streams generated by other stream processors. As already mentioned, depending on the subscription, the publication is performed by emails, in XML files (ordinary XML documents, XHTML Web pages or RSS feeds) and most interestingly as *channels*.

As already stated before, the *channel* is the basis of our *Pub/Sub* mechanism. An entity interested in some stream, has to subscribe to a channel publishing it. If a peer P_1 is interested in the output stream of a service evaluating at P_2, it asks P_2 to execute the service and to publish its results on a channel $\#x$, if this is not already the case. P_2 then subscribes to this channel $\#x@P_1$.

Clearly, many peers may subscribe to the same channel. It is an optimization issue whether a channel is implemented using point-to-point communications or using some form of multicasting or even broadcasting. This issue will not be detailed here.

Implementation P2PM is implemented in Java, on top of *ActiveXML Peer*. It is a Web application using *Axis* libraries to handle SOAP Web services and the *Jakarta Tomcat* servlet engine. *JavaCC* libraries are used to build a parser for P2PML. Code from the *YFilter* [10] project was modified and used to implement an *adaptable* automata-based query processor for XML document streams.

3.2 Streams, channels, services

The language ActiveXML [5] has been proposed to support distributed query evaluation and optimization. The Active XML algebra [4] is an algebra over (Active) XML streams. We show here how it can serve as the basis of a P2P monitoring system. Indeed, *P2PM* is based on the exchange of ActiveXML streams and it uses the ActiveXML algebra. Furthermore, it uses the ActiveXML peer system that we redesigned and partly reimplemented to serve the needs of monitoring.

We briefly recall some definitions of ActiveXML and sketch the ActiveXML algebra.

An ActiveXML document is an XML document where some of the elements (*sc* elements), denote calls to Web services. The *evaluation* of such a call results in performing the call and enriching the document with its result (e.g., by appending the result at the place of the call). All the information needed for performing the call (e.g., for accessing the service, deciding when to perform the call or what to do with the result) is provided inside the *sc* element.

An XML (respectively, ActiveXML) stream is a possibly infinite sequence of XML (respectively, ActiveXML) trees. A particular symbol *eos* may be considered to denote the termination of the stream. Typically, a stream is sent from one peer to a set of peers using the auxiliary concept of *channel*:

A *channel* is defined by a tuple

$$(peerID,\ streamID,\ subscribers)$$

where *peerID* is the peer that published this particular stream as a channel and *subscribers* is the set of peers that are subscribing to it.

Note that subscribing to a *monitoring task* is different from subscribing to a *channel*. A *monitoring task subscription* is defining a complex distributed interaction between peers. Subscribing to a channel means expressing the will to receive the data published by the channel.

In a P2P setting, typically very dynamic, the peers that are publishing in a channel as well as the set of subscribers may be permanently changing. One could also consider channels where several peers may publish. To simplify, this will be ignored here.

A subscription to a channel can be seen as a call to a service, where the result of the service call is a stream of ActiveXML trees. In ActiveXML terminology, this is a *continuous service*. The trees in the stream are received one after another in an asynchronous manner. Note that a non-continuous service is a particular case: the service returns an ActiveXML tree followed by an *eos*. Furthermore, a standard SOAP call is also a particular case, i.e., it is a non-continuous service returning some XML data that is not Active.

We adopt the following ActiveXML notation. A document d or a service s at peer p, are denoted respectively $d@p, s@p$. Some services, that are called *generic*, have a global name, and can be offered by many peers. This is the case, in particular, for query services that can be offered by any peer with an XML query processor. Such a service is denoted $s@any$, e.g., $\sigma_{//a//b}@any$. (In the following, a service s with no peer location will be implicitly assumed to be $s@any$.) Of course, to turn such a plan into a concrete one that can be deployed, we will have to replace generic services by concrete ones.

3.3 The stream algebra

We next briefly present the stream algebra. Details may be found in [4].

As in [4], we consider the following alphabets: $\mathcal{D}$ of *document names*, $\mathcal{S}$ of *service names*, $\mathcal{P}$ of *peer identifiers*, $\mathcal{N}$ of *node identifiers*, $\mathcal{L}$ of *label identifiers* and $\mathcal{V}$ of variables. Data variables are denoted as $\$x$,$\$y$... and node variables as $\sharp x$, $\sharp y$ .... The set $\mathcal{S}$ contains particular services: *send*, *receive* and *eval*.

ActiveXML expressions are used to model distributed evaluations. For $l \in \mathcal{L}$, p a peer, $d@p$ a document at p, $s@p$ a service of arity k, $n@p$ a node in some document at peer p, $e_1, e_2 ... e_k$ are algebraic expressions, the follow-

ing are also algebraic expressions:

$$\begin{array}{ll} l\langle e_1, ..., e_k\rangle & s@p\,(e_1, ..., e_k) \\ d@p & eval@p\,(e_1) \\ send@p\,(n@p, e_1) & receive@p\,() \end{array}$$

An executing service $s@p$ is noted $\circ s@p$. A service that finished executing is noted $\bullet s@p$. $eval@p\,(s@p'\,(...))$ means p asks for the execution of $s@p'\,(...)$.

The semantics of algebraic expressions is defined using rewriting rules. To illustrate them, we present only the two rules for service invocation:

1. Local service invocation

$$x_0@p : eval@p\,(s@p\,(..., t_i, ...)) \rightarrow x_0@p : \circ s@p\,(..., eval@p\,(t_i)\,, ...)$$

2. External service invocation

$$\sharp x@p\langle eval@p\,(s@p'\,(...))\rangle \rightarrow \sharp x@p\langle \circ receive@p\,()\rangle \;\&\; (new)\,@p' : eval@p'\,(send@p'\,(\sharp x@p, (s@p'\,())))$$

In the second rule, Peer p asks p' to evaluate service $s@p'$ (e.g., a local database call at p') and to send its (stream of) result(s) under the node $\sharp x@p$. Note the fact that p is executing $receive()$ to accept data from p' and to place it at the right spot.

The separator & means that the separated actions are done concurently (in the above expression at peers p and p').

We also present an important rule for query optimization that illustrates the tight interaction with the local optimizer: If $q \equiv q'\,(q_1, ..., q_n)$, then

$$eval@p\,(q@any) \leftrightarrow eval@p\,(q'@any\,(q_1@any, ..., q_n@any))$$

To conclude this section, consider again the example of Section 2. Let us denote by *out@a.com* and *out@b.com* the two alerters over outgoing calls, and *in@meteo.com* the alerter

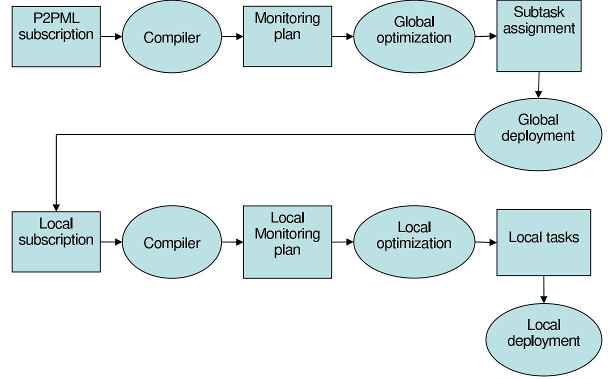

Figure 3: Subscription Processing Chains

on incoming calls. Let p be the peer that processes the subscription. Then the subscription is first compiled into the plan:

$$eval@p(publisher(\Pi_T(\bowtie_P (\cup(\sigma_F(out@a.com), \sigma_F(out@b.com))), \sigma_{F'}(in@meteo.com))))$$

where T is the restructuring template, P the join predicate, F, F' some filtering over the out and in-calls, respectively. Observe that in the above expression, some services are still generic (i.e., non concrete). Observe that the selections were pushed as much as possible to the proximity of the sources to save on communications. Finally, note that operators have not yet been "placed" (they are generic) with the exception of the alerters.

3.4 Monitoring plan generation

As already mentioned, the *subscription manager* is in charge of compiling a *subscription* into a *monitoring plan* that will then be optimized.

The monitoring plans are expressed in the algebra using operators we discussed, and in particular, alerters, stream processors and publishers. Figure 3 presents the steps transforming a subscription into a running *monitoring task*. The processing phases are represented by ovals and the input/output data by rectangles. Observe that at the end of the top processing

chain, the work is distributed between the peers in the system. It may be the case that some of the work is requested locally, which is the purpose of the bottom processing chain.

More precisely, in a first step, the subscription manager computes an optimized plan for the given subscription. The optimization is performed using algebraic rewrite rules and some simple heuristics. In a second step, it searches for resources in the system that cover at least parts of the task plan. This is considered in Section 5. Clearly, one can achieve more optimization by considering these two phases simultaneously. To simplify, this is not considered in the current implementation of the system. Finally, for the new tasks, the subscription manager assigns them to peers, trying to balance the load.

To illustrate, let us consider again the example. Imagine that no existing stream was found that could serve (part of) the subscription. Suppose that the query optimizer selects the plan (using the ActiveXML syntax):

$$\begin{array}{l} eval@p(publisher@p(\Pi_T@meteo.com(\\ \quad \bowtie_P @meteo.com(\\ \quad\quad \cup@b.com(\sigma_F@a.com(out@a.com), \\ \quad\quad\quad\quad \sigma_F@b.com(out@b.com))), \\ \quad\quad \sigma_{F'}@meteo.com(in@meteo.com)))) \end{array}$$

By rewriting, this yields:

$$\begin{array}{l} \quad \%\ \text{at p} \\ \circ publisher@p(\sharp M@p : \circ receive()) \\ \&\ \%\ \text{at meteo.com} \\ \circ send@meteo.com(\sharp M@p, \\ \quad \circ\Pi_T@meteo.com(\\ \quad\quad \circ \bowtie_P @meteo.com(\\ \quad\quad \sharp Y@meteo.com : \circ receive(), \\ \quad\quad \circ\sigma_{F'}@meteo.com(in@meteo.com)))) \\ \&\ \%\ \text{at b.com} \\ \circ send@b.com(\sharp Y@meteo.com, \\ \quad \circ \cup @b.com(\\ \quad\quad \sharp X@b.com : \circ receive(), \\ \quad\quad \circ\sigma_F@b.com(out@b.com)) \\ \&\ \%\ \text{at a.com} \\ \circ send@a.com(\sharp X@b.com, \\ \quad\quad\quad\quad \circ\sigma_F@a.com(out@a.com)) \end{array}$$

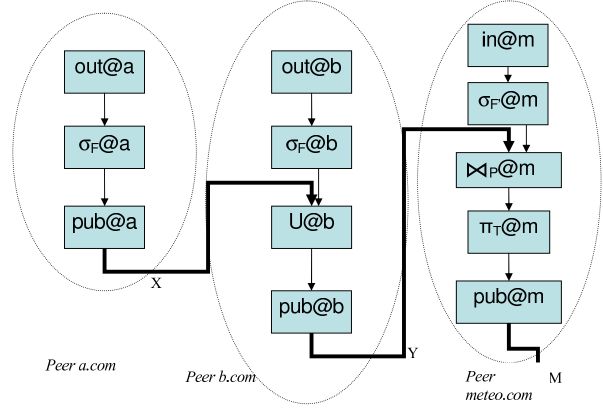

Figure 4: One possible plan for the monitoring task

Peer *a.com* filters its out-calls and sends its result to *b.com*. Observe the use of identifiers in $\sharp X@b.com$ to denote the destination of the message, i.e., the place where the result of the filtering at *a.com* is expected. This is in fact supported by a publication in a channel, namely the X channel published by a. This will allow the reuse of this streams if some other peer is interested in the same filtering. Peer *b.com* filters its own out-calls, merges with the data received from *a.com*. The result of the merge is sent to *meteo.com*, again via a channel, this time Y. This last peer joins what it receives with the result of the filtering of its in-calls. Finally, *meteo.com* also does a transformation (Π_T) to produce the results and sends it to p, via a last channel, namely M.Observe that peer p publishes the results.

Observe that now each of these expressions involves only services executed at one of the peers. So, each peer can start performing its part of the global task. The same plan is represented graphically in Figure 4 for peers *a.com*, *b.com* and *meteo.com*. Note that local subscriptions can also be expressed in the P2PML language. So, for instance, Peer *b.com* is assigned the task:

```
for $x in inCOM(<p>local</p>)
 where
  $x.callMethod = "receive" and
```

```
  $x.place = #X
 return $x
union
for $x in outCOM(<p>local</p>)
 let $duration := $x.responseTimestamp
             - $x.callTimestamp
 where
  $duration > 10 and
  $x.callMethod = "GetTemperature" and
  $x.caller = "http://meteo.com"
  return $x
by channel Y
  and subscribe(meteo.com, #Y, Y)
```

Note the *union* operator that takes the union of the two streams and is very much in the style of *union* in SQL. Observe also that the result is published as a channel.

Then peer *meteo.com* is automatically subscribed to this channel as a first client, and $\#Y$ indicates the place where this data is expected at peer *meteo.com*. Other peers may subscribe to this channel if desired.

We did not discuss one important aspect, namely how we detect that part of subscription can be supported using already executing tasks. This is addressed in Section 5.

4 Filter

In this section, we describe a most important stream processor, namely *Filter*. As we will see, *Filter* is based on two basic mechanisms: the Atomic Event Set Algorithm (AES for short) [19] for matching conjunctions of simple conditions and the YFilter Algorithm [10] for matching tree-patterns. The goal is to support very high volume input streams. This is of first importance for filtering *source streams* coming from alerters. The alerters are typically very simple pieces of software that do not attempt to perform complex selections, so they may produce lots of alerts that one has to filter. Also, some applications, e.g., in telecommunication services, produce huge volumes of notifications to be filtered.

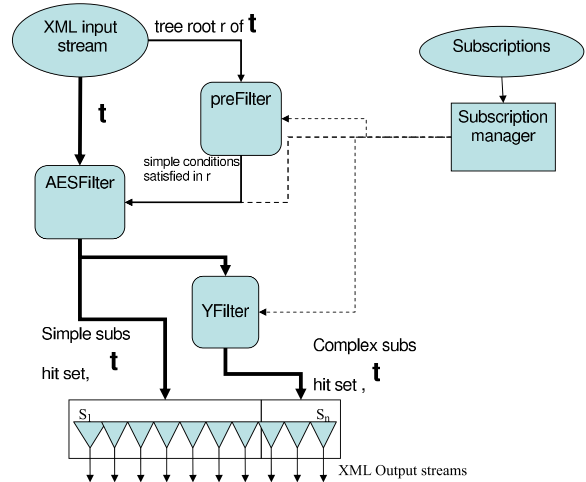

Figure 5: Filter Structure

In Section 2, we mentioned that the attributes of the root of a stream item often contain information that is important for filtering. At the same time, this information is easy to access without requiring complex computation and without the need to read the entire stream item. A system can support the filtering of a very high rate of stream items *on the fly* if only such simple conditions are checked. This is what we do next by separating the filtering in two stages. The first step consists in checking the simple conditions. The second one treating complex conditions only sees a stream of items that is typically much smaller than the stream that is being filtered.

More precisely, suppose we have to apply a large set $\{S_i\}$ of subscriptions over a stream of XML documents. A subscription S_i is specified in Filter as a pair (Q_i, T_i) where Q_i is a conjunctive query and T_i a report template. For each tree t in the stream, *Filter* must find the Q_i that match t. When a matching is found, it also has to apply the template T_i but since the main performance issue is to detect the matchings within a large set of subscriptions, this aspect will be ignored here. For this section of the article, we will refer to Q_i as *subscription*.

The subscriptions supported by Filter may include *simple conditions*, i.e., equalities or

inequalities ($\neq$, $\leq$) between the attributes of the root node of t and constants. An example of simple condition is $\$c.callee = $ "*http : //meteo.com*". Simple conditions are treated differently than the remaining portions of queries. As already mentioned, this is for performance reasons because they do not require any complex computation over t. For each Q_i, $Q_i = \wedge_j C_{ij}(\wedge Q'_i)$ where the C_{ij} are simple conditions and Q'_i performs the remaining complex filtering if needed. We consider here that Q'_i is a general tree-pattern query. If there is no Q'_i, the subscription is said to be *simple*; otherwise, it is *complex*. Filtering is performed by three modules. For each t,

preFilter computes the simple conditions satisfied by t.

AESFilter computes the simple subscriptions matching t and the complex ones such that all their simple conditions are satisfied by t.

YFilter$_\sigma$ computes the complex subscriptions matching t (if some remain to be checked).

We consider them in turn.

preFilter For each document t, the preFilter module is an automaton that reads the first tag of t (so, in particular, the attributes of the root node of t). It tests the simple conditions. Simple conditions are organized in a hash-table with the attribute name as key and the condition as value. The conditions having as object a particular attribute are organized as a balanced tree structure. A test for a $\$Y$ variable may be for instance, $\$Y < 10.0$ or $\$Y < \$X$ (for $\$X$ a variable less than $\$Y$ lexicographically). The preFilter test can be performed in linear time in the number of attributes of the root of t (also, linear time in the number of conditions that are satisfied and in the depth of the balanced tree structure).

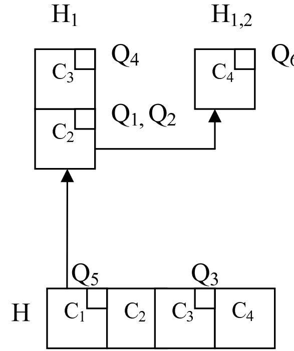

Figure 6: Atomic Event Set

AESFilter AESFilter is a modified version of the hash tree technique of [19]. Figure 6 represents the hash tree for the subscriptions:

$$
\begin{aligned}
Q_1 &= C_1, C_2, Q'_1 \\
Q_2 &= C_1, C_2, Q'_2 \\
Q_3 &= C_3, Q'_3 \\
Q_4 &= C_1, C_3, Q'_4 \\
Q_5 &= C_1 \\
Q_6 &= C_1, C_2, C_4, Q'_6
\end{aligned}
$$

The AES algorithm assumes that the set of simple conditions is totally ordered. So we assume this is the case and furthermore that preFilter produces an ordered sequence of the simple conditions satisfied by t. AESFilter feeds that sequence in the hash tree to obtain the simple subscriptions that are satisfied by t and the *active* complex subscriptions, i.e., those such that their simple conditions are satisfied.

The structure used by the *Atomic Event Sets* algorithm is a *hash-tree*. The root hash-table, named H, has for entries simple conditions specified by the subscriptions in the system. (To simplify, we ignore subscriptions with no simple conditions.) An entry, say the entry for C_{i_1} possibly contains a pointer to another hash table, named H_{i_1} which contains entries for the conditions that follow condition i_1 in some subscriptions. A hash-table in this structure corresponds to a prefix in some subscription. $H_{i_1,i_2,...,i_k}$ exists if at least one subscription has as prefix the sequence: $C_{i_1}, C_{i_2}, ..., C_{i_k}$.

In the example, H_1 contains as entries the conditions C_2 and C_3 that follow condition C_1 in the subscriptions Q_1, Q_2, Q_4 and Q_6. $H_{1,2}$ contains C_4 that follows after the sequence C_1, C_2 in subscription Q_6.

The structure we use to implement our Filter corresponds only to the simple conditions of the subscriptions. The markings correspond to the subscriptions that are still active after the processing of the simple conditions, meaning that their complex queries have to be evaluated by the $YFilter_\sigma$. The marked cells are the last simple conditions in at least one subscription. For instance, the condition C_3 in the hash-table H_1 is the last simple condition in the subscription Q_4. Its marking is Q_4. Details on the *AES* structure may be found in [19].

AESFilter is called with as input, the ordered list of conditions detected as valid by *preFilter* in the XML tree. It returns (i) the list of simple subscriptions that are satisfied by the tree, and (ii) the complex queries that have to be executed by the $YFilter_\sigma$, i.e., such that all the corresponding simple conditions are satisfied by the document.

As shown in [19], this is a most efficient organization that scales with the number of subscriptions. We inherit the performance of *AESFilter* from the performance of the AES-algorithm.

YFilter$_\sigma$ Lastly, YFilter$_\sigma$ uses the YFilter algorithm to test on t the query Q'_i for each active subscription Q_i. Observe that we run a different filter YFilter$_\sigma$, depending on the complex subscriptions that passed the AESFilter test. For instance, suppose t satisfies C_1, C_3 in the example. Then AESFilter detects Q_5 as a matching simple subscription and Q_4, Q_3 as active complex subscriptions. Then YFilter$_\sigma$ is adapted to check only the corresponding complex queries (Q'_4 and Q'_3). Suppose that Q'_4 only is verified. Then, the match are Q_4 and Q_5.

$YFilter_\sigma$ is a modified version of the YFilter automaton. Given the set $\{Q'_i\}$ of queries corresponding to the complex subscriptions, we construct a *YFilter* automaton. Now, given a tree t, only certain subscriptions are active so the automaton is *virtually* pruned to adapt to the specific filtering task for t. Details are omitted. As shown in [10], this is a most efficient organization that scales with the number of subscriptions because it groups path queries based on their common linear prefixes.

Figure 5 describes the Filter's architecture consisting mainly of the three modules previously described. Dotted arrows represent the flow of data corresponding to the offline adjustment of the filter under the effect of the changes in the subscription database. Plain arrows correspond to the data flowing in the Filter during the processing of an XML document coming on the input stream. Input data for the Filter is figured in ovals.

Web service calls To conclude this section, we consider a particular aspect of the filter, namely the use of external services. This is a place where the fact that the trees we monitor may be active (i.e., in ActiveXML) is particularly relevant. Such a tree may include calls to a Web service. For instance, consider the document in Figure 7 corresponding to the detection of a message from *NoSecret.com* to *P.com*. This message is a response for a Web service call of *P.com* previously addressed to *NoSecret.com*. The actual content of the message (the report *P.com* asked for) is provided *intensionally* to reduce the network bandwidth consumed by the monitoring system. If needed, it can be obtained by calling the service *getMessage*. Note the use of rectangles for *service call (function)* nodes in the tree, as opposed to the use of ovals for the representation of normal nodes.

Now consider the subscriptions:

- $type = \text{"}responseWS\text{"} \wedge caller = \text{"}P.com\text{"}$ that monitors the answers *P.com* receives, and
- $type = \text{"}responseWS\text{"} \wedge callee = \text{"}Secret.com\text{"} \wedge method =$

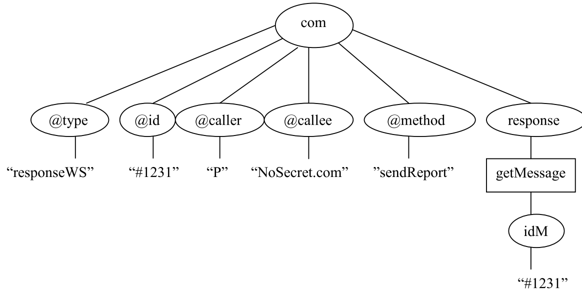

Figure 7: Document with intensional data

”*sendReport*” $\wedge$ *call//SomeCountry*, monitoring the reports *P.com* receives from *Secret.com* containing secret information about *SomeCountry*.

For evaluating the first subscription, the message is not needed. Now, for evaluating the second one, the filter would need to materialize the *intensional* part of this document by calling the Web service *getMessage* to determine whether the current document matches or not. Because of the prefiltering of simple conditions we use, the Web service call is not invoked for this subscription either (attribute *callee* of the considered stream item does not match). As seen in this example, Web service calls may come from intensional data in the stream documents. They may also come from explicit function evaluation demanded by the user (in the *Where* and *Return* clauses).

5 Stream reuse

In this section, we present the support for stream reuse of P2PM.

P2PM is a P2P platform providing monitoring services. Services such as *Selection*, *Join*, *Restructure* or *Publisher* are provided by the peers but each peer does not have to support all services. For instance, a *PDA* may refuse to support an expensive service such as *Join* whereas, an enterprise server may typically be willing to do it. When a new monitoring subscription is submitted to a *Subscription Manager* at a particular peer, that module is in charge of selecting a monitoring plan for this subscription. An essential aspect of its work is to determine which already existing streams may be reused for that task to save CPU consumption and network traffic. Clearly, monitoring subscriptions are typically expensive tasks that execute over long periods of time. It is therefore important to pay once for optimizing them (by carefully optimizing the monitoring plan) to obtain savings during their entire life time. Observe also that the system can gather statistics about the monitoring tasks and based on that, modify the monitoring plan (some form of self tuning). This important aspect is left for future work.

To support stream reuse, the system provides a *Stream Definition Database* that contains the description of all available streams. We will describe our support of *Stream Definition Database* over a P2P content management system, namely KadoP [3], at the end of the section. We next consider the representation of streams in our system then present our algorithm for discovering streams that can be used to serve a particular subscription.

Stream representation The description of streams is maintained in a database. The system provides services for publishing information about existing streams, and for querying this information in particular for stream reuse. We briefly describe at the end of the section an implementation of this service in a P2P XML-based content management system, namely KadoP. The information about a particular stream is defined with some XML data of the following form:

```
<Stream PeerId="..." StreamId="..."
   isAChannel="...">
   <Operator> ... </Operator>
   <Operands> ... </Operands>
   <Stats> ... </Stats>
</Stream>
```

where the pair (*StreamId*,*PeerId*) fully identifies the stream in the P2P system. *Operands* provides the list of pairs $(OPeerId, OStreamId)$ of operands, and *Stats* provides statistical information maintained for the stream such as the average volume of data in the stream for some period of time. The other two arguments are explained next.

The *Operator* argument specifies the operator that is used to produce this stream. When the set *Operand* is empty, this means that the stream is a monitoring source, in other words, it is produced by an alerter. The label of the son of *Operator* specifies the type of the alerter, e.g. WS communications, database updates etc.

The boolean attribute *isAChannel* specifies if the stream is *published* under the form of a channel or not. Recall that a channel is a stream that has been published. A channel is typically multicasted to several peers, so *PeerId* is not the single peer that can provide this data. The information about a particular channel is defined with some XML data of the following form:

```
<InChannel PeerId="..." StreamId="..."
 ReplicaPeerId="..."
 ReplicaStreamId="..."
/>
```

Suppose peer p published a stream s in a channel and that peer p' subscribes to that channel. Then p' may choose to publish this information to let it be known that he can also provide (p, s). In doing so, it also has to provide an Id for the replica, s'. In such case, the attributes would be in order, p, s, p' and s'.

When we publish the specification of a stream, we always do it with respect to the original streams and not to the replica. For instance, suppose that a peer provides some filtering of $s'@p'$, say $s''@p''$. When declaring this new stream, it will use $s@p$ as operand. So, even if "physically" $s''@p''$ is obtained by filtering $s'@p'$, it is viewed *semantically* as a filtering of $s@p$. This greatly facilitates the re-use of streams.

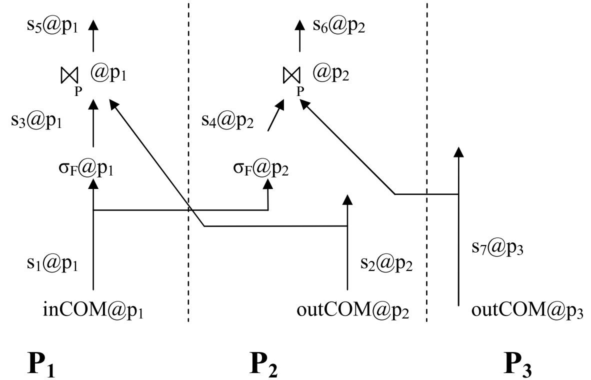

Figure 8: Stream replication and Stream Equivalence

Algorithm for discovering useful streams The algorithm searches for existing streams that can be used for serving a newly declared subscription. To illustrate, suppose the subscription is a selection over the Web communications at peer p_1. One first queries the database to see if a communication alerter for p_1 exists. Say it does and it is $s_1@p_1$. Then one queries the database to see whether there is a filtering of $s_1@p_1$ that performs the desired task. This is illustrated in Figure 8.

Since in the database, we have published all the operators over the original streams, we are concerned only with searching operators on original versions on the streams. The issue of replicas comes only in a second stage, namely when we have discovered a stream we are interested in and when we select either this original stream or one of its replicas. This selection is guided by the optimizer. Typically, we select a peer that can provide the information that is preferably "close" (networkwise) and not overloaded. Clearly, the notion of replica is not a full answer to *stream equivalence*. Indeed, one can find in Figure 8 an example of two streams that are equivalent (because of the equivalence of algebraic expressions) although they have not been defined as replicas. Detecting such equivalences is an interesting nontrivial problem that we intend to study in the future.

The *Reuse* algorithm works on a monitoring

plan, trying to find subplans that are already supported by existing streams. *Reuse* starts its search from the sources of the monitoring stream. For instance, if a source stream in the subscription involves incoming communications at Peer p_1, we can use the following XPath query to find alerters on p_1 assuming the variable $\$p1$ holds the peer ID:

```
/Stream[@PeerId = $p1][Operator/inCom]
```

Suppose now that we found that s_1 is an alerter on incoming calls. Now suppose that we want some particular filter over s_1, the following query returns all possible candidates that filters s_1:

```
/Stream[Operator/Filter]
  [Operands/Operand[@OPeerId=$p1]
    [@OStreamId=$s1]
  ]
```

assuming $(\$p_1, \$s_1)$ holds the peer and service ID.

In the example, we searched for any filtering over $s_1@p_1$. In fact, one can use a more precise query to actually search for a precise filter. Starting from the data sources, the search proceeds in the monitoring plan trying to obtain the largest subtasks possible. For now, we search only for exact matches. One can try to find "sufficient" sources. For instance, a bibliographic source that would provide the sports feeds in a press agency, would suffice to provide the football feeds from that agency, using some extra filtering. This aspect will be ignored here.

Now consider the more complex monitoring plan corresponding to the evaluation of the following expression:

$$\bowtie_P \left(\sigma_F\left(inCOM@p_1\right), outCOM@p_2\right)$$

Figure 8 illustrates existing streams in the system that may be used. Suppose that we already found that the filtering of incalls can be provided by $s_3@p_1$ and $s_4@p_2$; and that the outcalls can be provided by $s_2@p_2$. To search for the join, we can ask the query:

```
/Stream[Operator/Join]
  [Operands/Operand[@OPeerId=$p1]
     [@OStreamId=$s3]
  ]
  [Operands/Operand[@OPeerId=$p2]
    [@OStreamId=$s2]
  ]
```

More generally, the algorithm proceeds from the "leaves" of the monitoring plan, attempting to map nodes in the plan to existing streams. Operators that have all their operands matched generate queries to the database. The result of the queries determines whether this operator will be mapped to an existing stream. For a node that is matched, the algorithm searches for possible replicas of the streams that can substitute for that node. The nodes in the task's plan that have not been matched correspond to new streams that have to be produced.

Implementation The Stream Definition database is implemented using the KadoP[3] system, a P2P XML repository over a DHT system. The motivation is that a centralized database would potentially be a bottleneck.

A P2P repository is more in the spirit of P2P systems we monitor. All the peers in the KadoP network can participate in the storage and indexing of the Stream Definition Database.

Given a tree pattern of interest (that describes the stream we are interested in), we submit a query to the network. This results in contacting a small number of peers that index components used in the pattern and then retrieving information from the peers that published streams that are potentially useful. One can thus efficiently discover streams of interest even when millions of streams have been declared by tens of thousands of peers .

6 Related Work

Most of the works in the field of monitoring peer-to-peer systems have addressed two aspects. The first is the gathering statistics for file sharing systems, e.g. PeerMind[20], to answer queries such as: which is the most shared video file in this P2P system? The other is network monitoring for gathering statistics and detecting malfunctions, typically to improve QoS, e.g. Netscout [18] and Sandvine P2P Monitor tool [21].

An interesting system is Astrolabe [23], an information management service based on a peer-to-peer protocol that monitors the dynamically changing state of a collection of distributed resources by computing *summaries* of the data in the system using on-the-fly aggregation controlled by SQL queries. Astrolabe has a strategy of organizing resources in hierarchical domains, called *zones*, very similar to the one of the DNS (Domain Name System).

Our work differs greatly of these, since we are primarily interested in monitoring *events* regarding document updates (database, RSS feeds, Web pages) and distributed applications running in P2P systems. For these reasons, this topic is at the confluence of two research areas : Web-scale monitoring systems and stream processing.

Systems such Xyleme[19] and WebCQ [16] do centralized monitoring for changes in documents on the Web. NiagaraCQ[8], also a centralized system for Web monitoring, becomes scalable by regrouping similar structures of different continuous queries expressed in an XML-QL language.

PeerCQ[13] is a P2P system that performs Web-scale information monitoring using continuous queries and implementing efficient algorithms for allocating the queries on peers. All the processing for a continuous query is done on a peer and not distributed in network like in our approach.

STREAM[17], using an SQL-like subscription language, processes data streams by transforming them into relations. The query results are transformed back into streams. This system uses time-based windows for bounding the necessary storage for the evaluation of joins over streams, for instance. We are considering coupling this approach with an efficient garbage collection mechanism which detects and eliminates unnecessary trees from the storage.

Aurora[2], a centralized system and its distributed succesor Borealis[1] are also stream processing engines. All these systems are based on the relational model, processing streams of data tuples.

A work close in spirit to ours is StreamGlobe[22, 15], a P2P system for efficiently querying data streams represented in XML. It uses an XQuery like language and proposed stream sharing as a way to achieve scaling. Our system and StreamGlobe differ in the approach for stream reusal: StreamGlobe's sharing algorithm is performing in-network search for useful streams while we are using a service (provided by a DHT) for maintaining and querying the stream definitions. Also StreamGlobe shares streams derived from data sources by applying only unary operators, e.g. selections, projections and window-aggregation while the system we present allows sharing for all streams. In particular, the stream resulting from the join of two streams can be shared, detected as useful and reused.

Our system uses YFilter [10], a XML stream filtering engine based on a non-deterministic finite automata (NFA) which achieves scalability due to path sharing between queries. An alternative approach would have been using a deterministic finite automata (DFA) based filtering engine in the style of [14]. NFA-based automata have the advantage of being space efficient over DFA-based automata which have the number of states increasing exponentially with the number of queries, but are time efficient. In our case, NFA automata are convenient also for another reason: they are easily adaptable when the number of queries changes.

7 Conclusion

In this paper we have presented P2PM, a versatile peer-to-peer tool for monitoring generic P2P systems. Alerters have to be developped for each type of P2P system one wishes to monitor. However, the part of the architecture dedicated to processing as well as the stream publishers remain the same, regardless of the monitored application. We have also shown an efficient filtering technique and an algorithm for the detecting useful streams for covering (parts of) a new monitoring task.

We are currently testing our system by monitoring RSS feeds. Scalability is a property our system has because the processing for each *monitoring task* is distributed. We hope the tests we are performing will demonstrate it in practice.

We also plan to test our system for monitoring P2P systems running distributed applications. A good candidate for our tests seems to be the *EDOS* Distribution System[11], a P2P architecture for open-source content dissemination developed jointly by INRIA and Mandriva, a Linux distribution provider. In this system, peers have three roles: *publisher* of content in the distribution network, i.e. the main distribution server, *mirrors*, i.e. secondary trusted servers, replicators of the main server that provide additional downloading sources in the system, and clients which download content. In such a context, our system will be useful for detecting the presence/absence of *mirrors* in the network, the charge of the downloading activity at each and whether the content they replicate is synchronized or not with the one published by the main server.

Certainly, we have met very interesting problems and we plan to explore them in the near future. One is defining and implementing an efficient garbage collection mechanism for reducing the storage needed for our stateful stream processors. A second is finding solutions for the issue of stream equivalence. We are also interested in detecting and reusing streams that hold *sufficient* data. Another problem is adapting the monitoring plans to situations when peers are coming and leaving. Monitoring plans can also evolve so that they consume resources as little as possible (self-tuning). And these are just few examples.

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[3] Serge Abiteboul, Ioana Manolescu, and Nicoleta Preda. Constructing and querying peer-to-peer warehouses of XML resources. In *ICDE*, pages 1122–1123, 2005.

[4] Serge Abiteboul, Ioana Manolescu, and Emanuel Taropa. A framework for distributed XML data management. In *EDBT*, pages 1049–1058, 2006.

[5] Active XML Survey, ftp://ftp.inria.fr/inria/projects/ gemo/gemo/gemoreport-331.pdf.

[6] Active XML, http://activexml.net.

[7] Business Process Execution Language for Web Services, http://www.ibm.com/developerworks /library/ws-bpel/.

[8] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for Internet Databases.

In *SIGMOD Conference*, pages 379–390, 2000.

[9] Gregory Cobena, Serge Abiteboul, and Amélie Marian. Detecting changes in XML documents. In *ICDE*, pages 41–52, 2002.

[10] Yanlei Diao, Peter M. Fischer, Michael J. Franklin, and Raymond To. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE*, pages 341–, 2002.

[11] Edos, http://www.edos-project.org.

[12] eXist, http://exist.sourceforge.net/.

[13] Bugra Gedik and Ling Liu. Peercq: A decentralized and self-configuring Peer-to-Peer information monitoring system. In *ICDCS*, pages 490–499, 2003.

[14] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata. In *ICDT*, pages 173–189, 2003.

[15] Richard Kuntschke, Bernhard Stegmaier, Alfons Kemper, and Angelika Reiser. Streamglobe: Processing and sharing data streams in grid-based P2P infrastructures. In *VLDB*, pages 1259–1262, 2005.

[16] Ling Liu, Calton Pu, and Wei Tang. Webcq: Detecting and delivering information changes on the Web. In *CIKM*, pages 512–519, 2000.

[17] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Singh Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[18] Netscout, http://www.netscout.com/.

[19] Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mihai Preda. Monitoring XML data on the Web. In *SIGMOD Conference*, pages 437–448, 2001.

[20] Peermind, http://www.peermind.com/.

[21] Sandvine, http://www.sandvine.com/solutions.

[22] Bernhard Stegmaier, Richard Kuntscke, and Alfons Kemper. StreamGlobe: adaptive query processing and optimization in streaming P2P environments. In *ACM International Conference Proceeding Series; Vol. 72*, pages 88 – 97, 2004.

[23] Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[24] XQuery, http://www.w3.org/xml/query/.