Distributed Monitoring of Peer to Peer Systems

Serge Abiteboul and Bogdan Marinoiu
INRIA Saclay and University Paris Sud
4, rue Jacques Monod
91893 Orsay CEDEX, France
firstname.lastname@inria.fr

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 P2PML subscription is compiled into a distributed algebraic plan which is described using algebra over XML streams. The operators of this algebra are first *alerters* in charge of detecting specific events and acting as stream sources. Other operators process the streams or publish them.

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 that are already provided by existing streams.

Categories and Subject Descriptors

H.2 [**Database management**]: Distributed databases, Query processing

General Terms

Algorithms, Design, Languages, Performance

Keywords

distributed data management, stream processing, peer to peer systems, 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. Observation turns out to be essential in many contexts, e.g., error management, statistics gathering, workflow control, Web surveillance. This is the topic of the present paper where we propose a generic monitoring system for P2P systems. A main contribution is that subscriptions, specified in a declarative language, are compiled into distributed algebraic plans over XML streams. We present a new algorithm for efficient filtering and introduce a novel P2P technology for re-using already existing streams.

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). 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 stream of XML data. Such data may be transmitted between peers through *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 declarative Subscription Language, P2PML, short for *Peer-to-Peer Monitor Language*. A monitoring task is specified to the system in this language. It is then compiled into an *algebraic monitoring 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) derived from the ActiveXML algebra framework for distributed data management [4]. The algebraic operators include *Alerters* - 0-ary operators, that are situated on the premises of the monitored peers, detect specific local events and produce data streams, *stream processors* - for filtering or applying more complex operations on streams, and *publishers* - for exposing streams to other peers of the system (in *channels*), or to the human users in e-mails, RSS feeds or Web pages.

The most important stream processor is the *Filter*, which can perform efficiently a large number of filtering queries over a stream with intense traffic. An important aspect, from a performance viewpoint, is that it checks separately simple test conditions, evaluated *on the fly*, and more complex ones that require the use of an XML query processor.

An important issue for scaling with many subscriptions and peers 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 is also used for reducing the amount of data that is transferred by providing information *intentionally* 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 ex-

WIDM'07, November 9, 2007, Lisbon, Portugal.

isting 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 monitoring of P2P systems. We will mention briefly some that particularly motivated the present work.

P2PM can be used to observe Web services activity in a Web community. Towards this goal, we implemented an alerter to monitor SOAP messages. This could serve, for instance, to follow the concurrent execution of large number of workflow instances in telecom services (e.g., BPEL workflows [6]) to detect malfunctions, gather statistics, understand usage patterns, support billing, etc.

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.

A main motivation of our work is the monitoring of the Edos content sharing network [9]. Edos is a P2P distribution system that is developed in cooperation with the Mandriva company (originally 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 distribution is more than 100 megabytes of XML data. 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).

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 module implementing efficiently stream filtering. Section 5 shows techniques allowing stream reuse. In Section 6, we compare *P2PM* to other systems and in Section 7, we conclude.

2. P2P MONITORING LANGUAGE

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

The monitoring language P2PML allows specifying in declarative statements, called *(monitoring) subscriptions*, complex events a user is interested in, as well as how the user should be notified of detected events. The system is globally in charge of performing the corresponding *(monitoring) task*, and in particular, of assigning the operators.

For instance, consider the subscription involving three 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).

As one can notice, the syntax is inspired by the XQuery's FLWR [18]. 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 clause defines XML variables: $c1 for events detected by the two clients and $c2 for events detected at the server.

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

```
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>
    </incident>
by publish as channel "alertQoS";
```

Figure 1: A monitoring subscription

calls. Such communications consist of a pair of a *Call* and a *Response*. An alerter produces a stream of XML trees. An element of such stream is called a *stream item*. In a stream item, we distinguish two parts:

1. the attributes of the root that typically gather some generic information. For a Web service alerter, these will include call identifier, server identifier or the time of the call. Selection conditions on these attributes are very common in subscriptions.
2. the sub-elements 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 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, e.g. $*duration*.

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. An example of a XPath query is:

```
$c1/alert[@callMethod = "GetTemperature"]
```

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* = "*GetTemperature*". Conditions on the root attributes are called *simple conditions*. The *Where* clause in the example contains four simple conditions.

The *RETURN* clause specifies the output stream. When conditions are matched by the values over the input streams, an output XML tree is obtained in the output stream. This output is defined as XML data with possibly curly bracketsguarded expressions that are to be evaluated at runtime.

Finally, the *BY* clause determines how the user gets notified. Publication in a channel (the most interesting case), illustrated by the example, consists in publishing a stream that clients (other peers) can subscribe to or other subscriptions (issued by human users) 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:

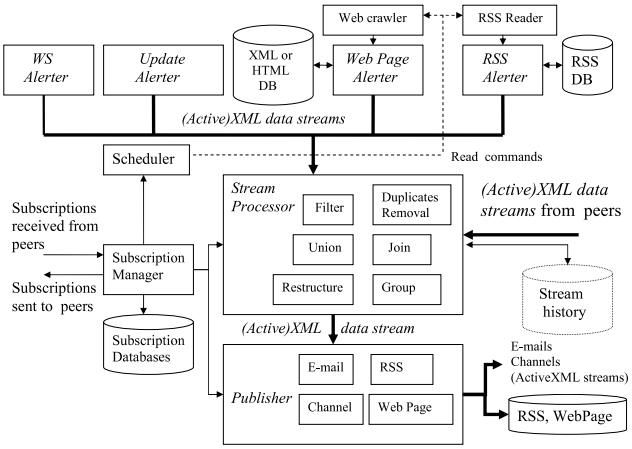

Figure 2: The architecture of a peer in P2PM

```
<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) ...
```

For this example we suppose that our system has the support of a DHT and that the DHT exports a stream of events, corresponding to peers joining or leaving:

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

In the latter case, *inCOM* removes peers from the collection of monitored peers.

One can also request duplicate-free results by preceeding the content of the *RETURN* clause by the *distinct* attribute as in: *return distinct* $<a>\{\$y\}$.

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. XQuery has an influence because the streams contain XML data.

3. P2P MONITOR

We first present the architecture of P2PM. Then we focus on two key aspects: (i) the ActiveXML algebra that is used in 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 minimum required to be a P2PM peer is to run a Subscription Manager. A peer may also host some alerters, 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). A peer can also delegate a part of the monitoring task, by expressing demands in the same P2PML language for *monitoring subtasks* to other peers.

Alerters. Each alerter (a 0-ary operator producing a stream) is specialized in detecting particular events in some systems that are external to P2PM, as described next. An *WS Alerter* intercepts inbound-outbound Web service calls and produces alerts including SOAP envelopes expanded with annotations such as timestamps and the identifiers (DNS/IP) for caller/called entities. They are implemented as *Axis* handlers. An *ActiveXML alerter* detects updates to the ActiveXML peer's repository. A *WebPage Alerter* detects changes in XML/XHTML pages by comparing their snapshots. The alert may provide (if desired) the delta between two pages. (This alerter uses an auxiliary Web crawler for the surveillance of collections of Web pages). *RSS Feed Alerter* detects changes in an RSS feed by comparing snapshots also. With RSS, the alerts have more semantics than with arbitrary XML: e.g., *add*, *remove* and *modify entry*.

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), *Union* ($\cup$). Others are *stateful*, e.g., *Duplicate-removal*, *Join* ($\bowtie$) or *Group*. We next briefly discuss the main processors we support. The *Filter* processor whose performance is critical for the usability of the system, is discussed in Section 4.

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*. *Union* takes several streams as inputs and merges them into a single stream.

Restructure takes as input one stream. A *template* defines 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. 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 stream processors, under different forms: by emails, in XML files (ordinary XML documents, XHTML Web pages or RSS feeds) or as *channels*. 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 publish its results on a channel $\#x$. P_2 then subscribes to this channel $\#x@P_1$.

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* [8] 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. 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 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 interested in 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.

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*.

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, called *generic*, can be offered by many peers. In particular, query services 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 is assumed to be $s@any$.) For deployment, the generic services will be replaced 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 and p' peers, $d@p$ a document at p, $s@p$ a service of arity k, $n@p$ a node in some document at peer p, and the algebraic expressions $e_1, e_2 ... e_k$, the following are also algebraic expressions:

$l\langle e_1, ..., e_k\rangle$, $s@p\,(e_1, ..., e_k)$, $d@p$, $eval@p\,(e_1)$, $send@p\,(n@p', e_1)$, $receive@p\,()$.

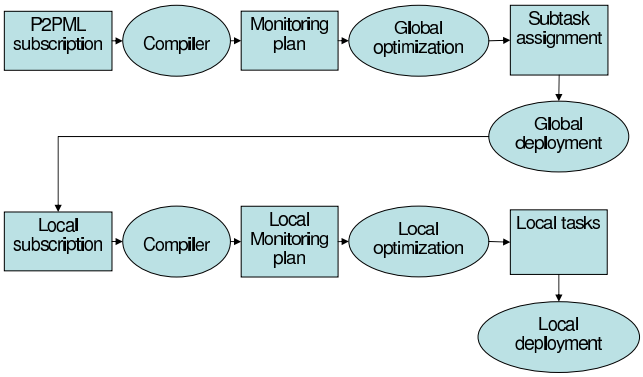

Figure 3: Subscription Processing Chains

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 $s@p'\,(...)$ on peer p'.

The semantics of algebraic expressions are 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

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

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 & between actions means that these are done concurrently (here, 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

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

Let us consider again the example of Section 2 and denote by *out@a.com* and *out@b.com* the two alerters over outgoing calls, and *in@meteo.com* the alerter on incoming calls. Let p be the peer that processes the subscription. Then the subscription is first compiled into the plan:

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

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 and that operators have not yet been "placed" 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 the discussed operators, in particular,

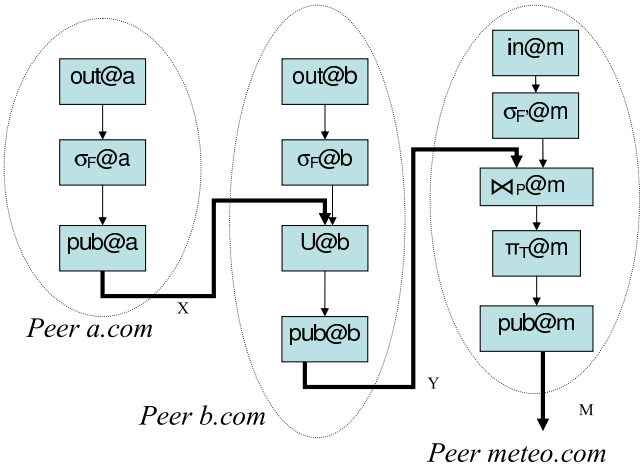

Figure 4: One possible plan for the monitoring task

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. Some of the work may be requested locally (observe the bottom processing chain).

In a first step, the subscription manager computes an optimized plan for the given subscription. The optimization is performed using algebraic rewrite rules and 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. Finally, for the new tasks, the subscription manager assigns them to peers, trying to balance the load.

To illustrate the first step, 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):

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

By rewriting, this yields:

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

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 stream if some other peer is interested in the same filtering. Peer *b.com* filters its own out-calls, merges with the data received

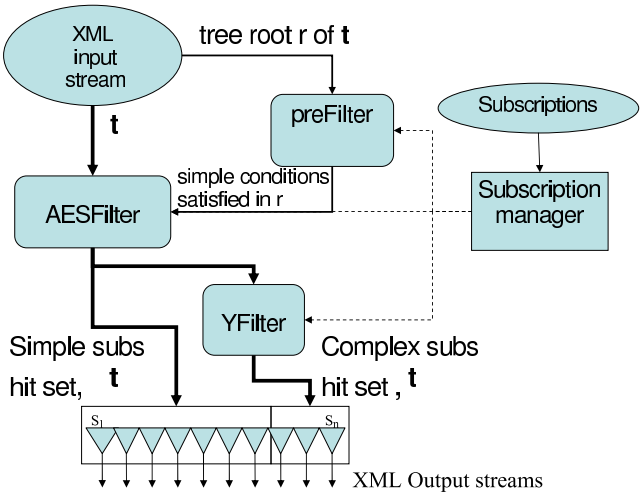

Figure 5: Filter Structure

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 each expression 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 the three peers. Note that local subscriptions can also be expressed in the P2PML language. So, for instance, Peer *a.com* is assigned the task:

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

Observe that the result is published as a channel to which peer *b.com* is automatically subscribed as a first client, and $\#X$ indicates the place where this data is expected at peer *b.com*. Other peers may subscribe to this channel if desired.

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) [15] for matching conjunctions of simple conditions and the YFilter Algorithm [8] for matching tree-patterns. The goal is to support very high volume input streams. This is of first importance for filtering potentially heavy *source streams* coming from alerters: e.g., telecom services, produce huge volumes of notifications to be filtered.

In Section 2, we mentioned that the attributes of the root of a stream item often contain information 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 simple conditions. The second one treating complex conditions only sees a stream of items that is typically much smaller than the stream being filtered.

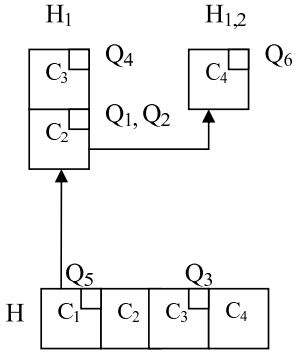

Figure 6: Atomic Event Set

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 matches 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, this aspect will be ignored here. For this section, 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 $\$c.callee = "http : //meteo.com"$. 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: *preFilter*, *AESFilter* and $YFilter_\sigma$.

preFilter. The preFilter module is an automaton that, for each document t, reads the first tag of t (so, in particular, the root's attributes). It tests the simple conditions which are organized in a hash-table with the attribute name as key and the condition as value.

AESFilter. AESFilter is a modified version of the hash tree technique of [15]. Figure 6 represents the hash tree for the subscriptions. The AES algorithm assumes that the set of simple conditions is ordered. So we assume this is the case. 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 with simple conditions satisfied.

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

The 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,$H_{1,2}$ contains C_4 that follows after the sequence C_1, C_2 in subscription Q_6.

The structure we use to implement our AESFilter 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 for Q_4. Its marking is Q_4. Details on the *AES* may be found in [15].

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 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 [15], this organization scales with the number of subscriptions.

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. If we suppose t satisfies C_1, C_3 in the example, AESFilter will detect Q_5 as a matching simple subscription and Q_4, Q_3 as active complex subscriptions. Then YFilter$_\sigma$ is adapted to check only for the complex queries Q'_4 and Q'_3. Suppose that Q'_4 only is verified. The matchings 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. As shown in [8], 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 when the subscription database changes. 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 which provides, on demand, a part of the document that was considered too big to be passed into the stream. Let us consider the following XML document coming on a stream:

```
<root attr1="x" attr2="y">
 <sc service="storage" address="site">
    <parameters>...</parameters>
 </sc>
</root>
```

The active part is rooted at the element *sc*. Suppose that the call to service *storage@site* would evaluate to:

```
<c><d>...</d></c>
```

This data would replace the subtree rooted at *sc*. Let us also assume that a subscription filters this stream with the query:

```
$item.attr1="x" and $item.attr2="z" and $item//c/d
```

The Filter first evaluates the simple conditions, i.e. the conditions on the root attributes. Since the checking for the second condition fails, it will not pursue the checking of the XPath expression. Observe that by replacing the *service call* subtree with the result of the Web service call, the XPath expression would evaluate to *true*. Our strategy avoids the unnecessary call to service *storage@site*.

5. STREAM REUSE

This section presents P2PM's support for stream reuse.

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, 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. Since monitoring subscriptions are tasks that execute over long periods of time, it is therefore important to pay once for optimizing the monitoring plan.

To support stream reuse, the system provides a *Stream Definition Database* that contains the description of all available streams. This database is implemented using the KadoP [3] system, a P2P XML index and repository over a DHT system. The motivation is that a centralized database would potentially be a bottleneck. All the peers in the KadoP network can participate in the storage and indexing of the Stream Definition Database. One can efficiently discover streams of interest even when millions of streams have been declared by tens of thousands of peers.

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. The information about some stream is defined with XML data:

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

The pair (*StreamId,PeerId*) fully identifies the stream.

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 *Operator* argument specifies the operator that is used to produce this stream. When the set *Operands* is empty, this means that the stream is a monitoring source, in other words, it is produced by an alerter. 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 also defined in XML:

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

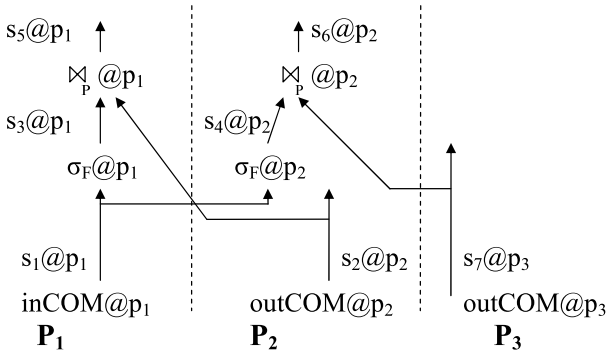

Figure 7: Stream Replication and Equivalence

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). To do so, it also has to provide an Id for the replica, s'. 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.

Algorithm for discovering useful streams. The algorithm searches for existing streams that can be used for serving a newly declared subscription. 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 its output stream 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 (see Figure 7).

Since the operators have been published over the original streams, we are concerned only with searching operators on original versions of 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 have to select either this original stream or one of its replicas. This selection is guided by the optimizer. Typically, we select as provider a peer 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 7 an example of two streams that are equivalent (because of the equivalence of algebraic expressions) without being replicas. The *Reuse* algorithm works on a monitoring plan, trying to find sub-plans 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 streams produced by alerters on p_1, assuming the variable $\$p_1$ holds the peer ID:

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

Suppose now that we found that s_1 contains alerts on incoming calls and that we want some particular filter over s_1. The following query returns all possible candidates that filter s_1 assuming $(\$p_1, \$s_1)$ holds the peer and stream ID:

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

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 7 shows existing streams in the system that may be used. Suppose that we already found that the filtering of incoming calls 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 to substitute for that node. The nodes that have not been matched correspond to new streams that have to be produced.

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 of statistics for file sharing systems, e.g. [16], in order to answer queries such as: which is the most shared video file in this P2P system? The other is network monitoring typically for improving QoS, e.g. [14].

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 as [7], [12] and [15] do centralized monitoring for changes in documents on the Web. NiagaraCQ[7] becomes scalable by regrouping similar structures of different continuous queries expressed in an XML-QL language. PeerCQ[10] 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.

STREAM[13] 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 intend to couple this approach with an efficient garbage collection mechanism which detects and eliminates unnecessary trees from the storage. Borealis[1] and Aurora[2] 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[11, 17], a P2P system for efficiently querying data streams represented in XML, that uses an XQuery-like language and proposes stream sharing to achieve scaling. StreamGlobe performs 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 re-used.

7. CONCLUSION

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

We are currently testing our system by monitoring RSS feeds. We also plan to test our system for monitoring P2P systems running distributed applications like Edos[9], already mentioned. 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.

8. 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*, 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), 2003.

[3] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and querying peer-to-peer warehouses of XML resources. In *ICDE*, 2005.

[4] S. Abiteboul, I. Manolescu, and E. Taropa. A framework for distributed XML data management. In *EDBT*, 2006.

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

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

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet Databases. In *SIGMOD Conference*, 2000.

[8] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE*, 2002.

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

[10] B. Gedik and L. Liu. PeerCQ: A decentralized and self-configuring Peer-to-Peer information monitoring system. In *ICDCS*, 2003.

[11] R. Kuntschke, B. Stegmaier, A. Kemper, and A. Reiser. Streamglobe: Processing and sharing Data Streams in Grid-Based P2P infrastructures. In *VLDB*, 2005.

[12] L. Liu, C. Pu, and W. Tang. WebCQ: Detecting and delivering information changes on the Web. In *CIKM*, 2000.

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

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

[15] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the Web. In *SIGMOD*, 2001.

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

[17] B. Stegmaier, R. Kuntscke, and A. Kemper. Streamglobe: adaptive query processing and optimization in streaming P2P environments. In *ACM International Conference Proceeding Series; Vol. 72*, 2004.

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