Distributed monitoring of Peer-to-Peer systems[*]

Serge Abiteboul Bogdan Marinoiu Pierre Bourhis[†]
INRIA Orsay and University Paris Sud
firstname.lastname@inria.fr

Abstract

Observing highly dynamic Peer-to-Peer systems is essential for many applications such as fault management or business processing. We demonstrate P2PMonitor, a P2P system for monitoring such systems. Alerters deployed on the monitored peers are designed to detect particular kinds of local events. They generate streams of XML data that form the primary sources of information for P2PMonitor. The core of the system is composed of processing components implementing the operators of an algebra over data streams.

From a user viewpoint, monitoring a P2P system can be as simple as querying an XML document. The document is an ActiveXML document that aggregates a (possibly very large) number of streams generated by alerters on the monitored peers. Behind the scene, P2PMonitor compiles the monitoring query into a distributed monitoring plan, deploys alerters and stream algebra processors and issues notifications that are sent to users.

The system functionalities are demonstrated by simulating the supply chain of a large company.

1 Introduction

Peer-to-peer systems gained popularity over the last decade by providing support for community content sharing and for loosely coupled distributed applications. A P2P system is a highly dynamic environment with participants acting independently by exchanging information and updating their data. In practice, it is difficult to observe this type of systems and to gather information about their functioning. Observation turns out to be essential in many contexts, e.g., error management, statistics gathering, workflow control, Web surveillance.

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

[†]ENS Cachan.

In this demonstration, we present a distributed system whose purpose is to monitor P2P systems. Our system, called *P2P Monitor* (P2PM for short), is itself a P2P system. So, two P2P systems coexist: the monitored one (possibly several monitored systems) and the monitor (namely P2PM). Clearly, the same machine may participate in both kinds of P2P networks.

We assume here that the monitored systems are willing to cooperate by accepting to run locally software modules, called *alerters*. Alerters are dedicated to the different kinds of events one wants to detect. They produce some information that is represented as XML *data streams*. The processing of these streams is performed by operators of an *algebra over XML streams* derived from the one presented in [4]. The operators of the algebra, called *stream processors*, are distributed over the peers of the monitoring system. Their results are also XML streams. Each stream is physically implemented as a *continuous service* that produces a sequence of asynchronous messages from a sender to a set of receivers. A stream at one peer is available to other peers as a *channel* to which they can subscribe.

The core of monitoring in P2PM is performed by processing *queries over active documents*. An (ActiveXML [6]) *active document* is a tree document with some special nodes representing *functions*. Such a function, once activated, causes the document to subscribe to an XML stream, so to receive a flow of XML data. Each time a new element appears on a stream, one copy of it is received by the active document, which integrates it as a sibling of the function node. An example of an active document is provided in Section 3, Figure 6. (Function nodes are marked with "!"). Now consider a query over an active document. For now, the system supports *monotone queries*, i.e. appending data to the document possibly implicates new data in the output of the query, without dropping old data. So, each new piece of information on an input stream possibly generates a new result for the query. Thus a query also produces an XML stream. Using an incremental algorithm, these queries are processed very efficiently. In particular, a sophisticate form of garbage collection allows removing *no more relevant data* from active documents and unsubscribing to streams whose data could not impact the query output in the future.

By using *active documents*, a user is able to specify in a simple way, e.g. with an XQuery query, some complex monitoring, since an active document possibly aggregates dozens or hundreds of streams coming from many different peers. Moreover, these streams may be specified *intentionally* (e.g., as the result of some query evaluated on a different peer, whose result evolves over time). The system compiles a query into a monitoring plan using subscriptions (in the *P2PML* language). These *P2PML* subscriptions trigger the deployment of alerters and stream processors across different peers. Resulting streams are published by a *publisher* based on the user's demands as continuous services (channels to which other peers may subscribe), RSS feeds, emails, Web pages or stored in XML databases maintained by the monitoring system. Of course, it is also possible for a user to specify a subscription directly in *P2PML*.

Related Work As far as we know, the processing of continuous queries over active documents (with streams) is a new topic. Previous works over active documents mostly considered non-stream functions: functions that, for each call, return an answer and terminate [6]. Most notably, query processing on active documents (with non-stream functions) is considered in [3]. To some extend, our work may be viewed as the incremental maintenance of the problem they study. The domain of stream processing has recently been very active. For instance, STREAM [12] executes continuous queries over multiple data streams and uses an SQL-like subscription language. Aurora [2], a centralized system and its distributed successor Borealis [1] are also stream processing engines. The StreamGlobe [11] P2P system is specialized in efficiently querying data streams represented in XML using XQuery.

Demonstration We demonstrate our system by monitoring a simulator of a distributed business process described in [10] and summarized in Section 3 of this paper. We show how to provide a distributed *supervisor* capable of monitoring and regulating the business process. We also show the simplicity of use of such a system as well as inherent properties of our distributed monitoring system: efficiency in terms of load balancing on peers, transparency for the user (the system is the one deciding where to place operators), savings on network bandwidth by pushing filters closer to the sources, savings on storage and on CPU due to incremental query evaluation.

```
for $order in channel(gOs@localhost)
where $order//status ="NotAv"
return
  <alert type="stock">{$order//orderId}</alert>
by publish as channel "blockedOrders"
```

Figure 1. A simple P2PML subscription

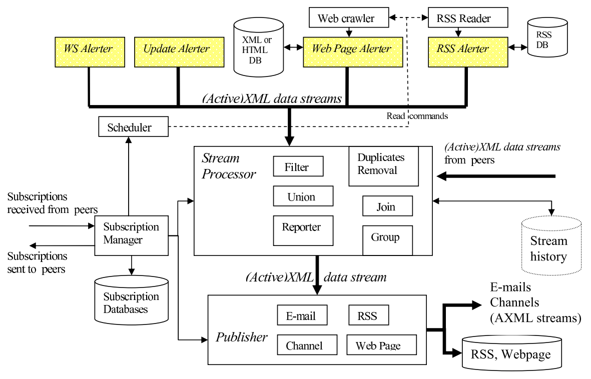


Figure 2. Stream Processing Subsystem

2 The monitoring system

Consider the *P2PML* subscription in Figure 1. Observe that the language is similar to XQuery. Main syntactic differences are the use of the keyword *channel* and of the *publish* clause. The first one designates the data stream sources, whereas the second specifies what to do with the resulting stream.

A simplified model of the stream processing architecture of P2PM is presented in Figure 2. A peer may host one or more alerters, stream processors and publishers. Besides such components that produce and process streams, a peer may host a *subscription manager* that is in charge of managing subscriptions, and in particular of supporting a subscription *catalogue*. Any peer may decide to reuse the stream of results of an existing subscription or to issue a new one. In case of a new subscription, the subscription manager is in charge of detecting (in a "Bits-and-pieces" catalogue) which parts of that subscription are already supported somewhere, and based on that, generates a monitoring algebraic plan for the new subscription and deploys it. In particular, the subscription manager is in charge of optimizing the algebraic plans, e.g. by pushing processing close to the data sources.

P2PM already supports the surveillance of various systems as described next. An *WS Alerter* intercepts inbound-outbound Web service calls and produces alerts including the SOAP envelopes expended

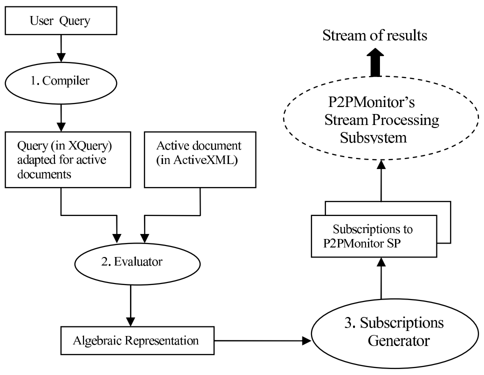
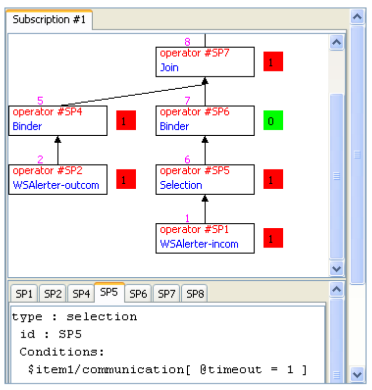

User Query
1. Compiler
Query (in XQuery) adapted for active documents
Active document (in ActiveXML)
2. Evaluator
Algebraic Representation
3. Subscriptions Generator
Subscriptions to P2PMonitor SP
P2PMonitor's Stream Processing Subsystem
Stream of results

Figure 3. Query Processing Subsystem



```
type : selection
 id : SP5
 Conditions:
  $item1/communication[ @timeout = 1 ]
```

Figure 4. Graphical user interface snapshot

with annotations such as timestamps and caller/called entities' identifiers (DNS/IP). An *ActiveXML Alerter* detects updates to the ActiveXML peer repository. A *WebPage Alerter* detects changes in XML or XHTML pages by comparing their snapshots. The alert may provide (if desired) the delta between the two pages. (This alerter uses an auxiliary Web crawler for the surveillance of collections of Web pages.) An *RSS Alerter* detects changes in an RSS feed by comparing snapshots. With RSS, the alerts have more semantics than with arbitrary XML pages: e.g., *add entry*, *remove entry* and *modify entry*.

The streams generated by these alerters are processed by algebraic stream processors that operate on one or more input XML streams (local or not) and produce output XML streams. Web services are used for communications between peers. Some operators are memory-less, e.g., *filter* or *fusion*. Others require information about (a window of) the stream history, e.g., *join*, *aggregation*, *duplicate elimination*. This history information is stored in an eXist [8] XML database. Operators such as *join* may use some application dependent function, e.g. a similarity function.

To be able to support heavy streams, an essential aspect of the work is performance. In particular, we filter streams by combining two kinds of filters. Some simple queries can be performed on-line basically at the speed of receiving this stream using some automata in the spirit of [7]. For more complex queries or when the streams return active documents, we can use the query processor over active documents already described in introduction and illustrated in Figure 3.

P2PM has been implemented in Java as a Web application using support of Axis2 Web services engine [5]. It uses JavaCC [9] for generating P2PML parsers. An applet-based GUI (see a fragment of a snapshot in Figure 4) allows visualizing in real-time the deployed subscriptions on a peer and the state of the deployed stream processors at the level of a peer: how they are interconnected, the received/generated data on streams etc. The GUI also allows specifying new subscriptions in *P2PML* and placing queries on ActiveXML documents.

3 Demonstration

We demonstrate the system with a distributed application simulating the supply chain of a computer manufacturer, namely Dell, as described in [10] and illustrated in Figure 5. The manufacturing system processes continuous flows of orders and has to cope with issues such as distant suppliers. The main modules are as follows. *WWW interface* is the Web site, in charge of processing forms completed by customers and of generating orders to the dispatcher. For a given order, *Dispatch* selects a plant close to the customer to delegate order processing. Each *Plant* processes an order upward, by forwarding orders for different parts to the relevant revolvers. It processes an order downward, by combining the parts that are received into objects (e.g. computers) that are then physically sent to the customers. *Revolver* (or warehouse) is a platform acting as a buffer between suppliers and Dell's plants, performing flow desynchronization; it works in a predictive mode based on statistical information maintained by the *supervisor* (our monitoring system). *Supplier* corresponds to Dell's suppliers. They rarely ship components in large quantities to revolvers, so there is an issue of stock management. Finally, *Bank* is a third party in charge of checking the validity of credit card payments.

The monitoring system is in charge of the surveillance of the entire process. A large number of plants, revolvers as well as banks are involved in the process.

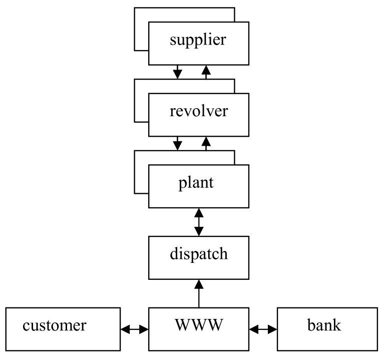


Figure 5. Architecture of the example

Each of these involved participants publishes notifications, e.g., when they receive or submit orders. Consider for instance the supervisor's document in Figure 6. A *!gOs* function (an abbreviation for *getOrders*) is a function denoting the stream of orders issued by the dispatcher, e.g. $!gOs@d$, a plant, e.g. $!gOs@p_2$ or a revolver, e.g. $!gOs@rev_3$. Figure 6 shows the order *129* arriving at the dispatcher, then forwarded to plant p_2. The order's object has not yet been delivered (status *NotDel*) because rev_3 blocks the fabrication process since a component is not available (status *NotAv*). The example has been simplified: there is only one type of products to be ordered and one plant uses a single revolver, e.g. plant p_2 uses revolver rev_3.

As a scenario, we show how different peers are kept up-to-date of the progress of some Web orders. We also show how a customer is informed of the processing of her on-going orders. In another scenario, a plant is warned of revolvers low stocks to avoid ordering some part to a revolver that is soon going to be out of stock.

Let us suppose a supervisor wants to know details about orders blocked at some revolver. The details about all the orders are available on the dispatcher's *gOs* channel. To see the blocked orders, one can directly inspect the streams originating at the revolvers.

A subscription (expressed in XQuery) is initially sent to the user's computer P2PM entity, called here the *supervisor* peer, which could simply wait for data to accumulate in its active document (Figure 6) and repeat the query evaluation on each update. This is non-incremental and centralized query evaluation.

In the demonstration, we will see how our system pushes filters close to the sources, i.e. the revolvers that publish each a stream of alerts for blocked orders as channel *blockedOrders* (a subscription as in Figure 1 is sent to each revolver). The *supervisor* simply subscribes to channels *blockedOrders*, as well as to the dispatcher's *gOs* channel for obtaining orders' details. A *join* of these streams is finally done at the *supervisor*.

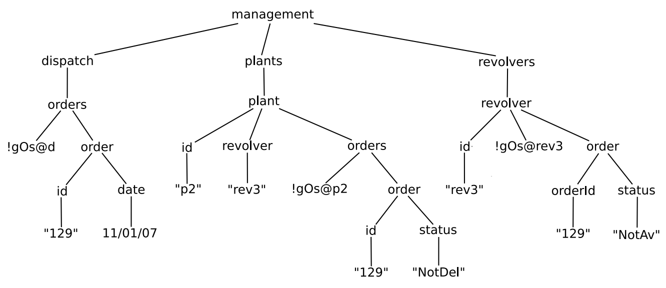


Figure 6. An active document with streams

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, Omar Benjelloun, Bogdan Cautis, Ioana Manolescu, Tova Milo, and Nicoleta Preda. Lazy query evaluation for Active XML. In *SIGMOD Conference*, pages 227–238, 2004.

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

[5] http://ws.apache.org/axis2/.

[6] http://www.activexml.net.

[7] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE*, pages 341–, 2002.

[8] http://exist.sourceforge.net/.

[9] https://javacc.dev.java.net/.

[10] Roman Kapuscinski, Rachel Q. Zhang, paul Carbonneau, Robert Moore, and Bill Reeves. Inventory decisions in Dell's supply chain. *Interfaces*, 34(3):191–205, 2004.

[11] R. Kuntschke, B. Stegmaier, A. Kemper, and A. Reiser. Streamglobe: Processing and sharing Data Streams in Grid-Based P2P infrastructures. In *VLDB*, pages 1259–1262, 2005.

[12] 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.