OptimAX: Optimizing Distributed ActiveXML Applications

Serge Abiteboul Ioana Manolescu
Spyros Zoupanos
INRIA Saclay–Île-de-France, Gemo group & Université Paris Sud, LRI
4, rue J. Monod, 91893 Orsay Cedex
firstname.lastname@inria.fr

Abstract

The Web has become a platform of choice for the deployment of complex applications involving several business partners. Typically, such applications interoperate by means of Web services, exchanging XML information.

We present OptimAX, an optimization Web service that applies at the static level (prior to enacting an application) in order to rewrite it into one whose execution will be more performant. OptimAX builds on the ActiveXML (AXML) data-centric Web service composition language, and demonstrates how database-style techniques can be efficiently integrated in a loosely-coupled, distributed application based on Web services. OptimAX has been fully implemented and we describe its experimental performance.

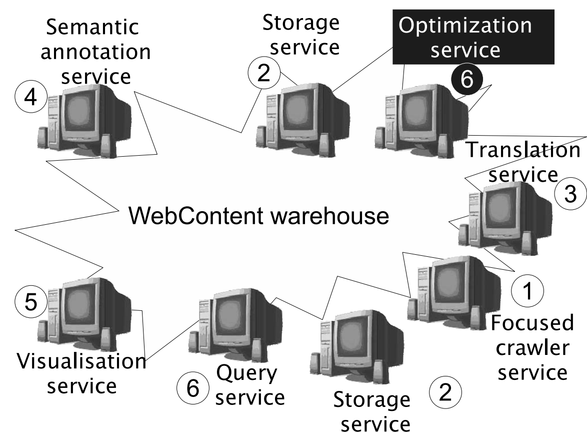

Figure 1. WebContent architecture outline.

1 Introduction

The Web has become the platform of choice for the delivery of business applications. In particular, the popularity of Web service technologies (WSDL [22], BPEL4WS [12] etc.) and their closeness to HTML and XML, the predominant content delivery languages on the Web, has opened the way to the development of complex business applications by integrating Web services provided by different parties. This model has several advantages. From a development viewpoint, it relies on widely accepted standards, and benefits from the plethora of available application building blocks. From a business viewpoint, it allows organizing the activity in cleanly defined modules, each of which is implemented by some Web services. This enables several entities to provide implementations of a given module, and facilitates replacing one entity with another.

We are currently involved in a large R&D project called WebContent [25], whose purpose is to build and exploit large-scale repositories of rich, semantically annotated Web data. The overall setting of the project, outlined in Figure 1, exemplifies the kinds of applications discussed above. A *focused crawler (1)* service returns Web documents related to specific domains, in our case, aircraft sales by Airbus and Boeing (for a continuous, online market survey), respectively, food risk information, for a consortium of food companies seeking to organize and structure information related to different food problems (contaminations, allergens etc.). The crawler service returns XML documents with information-rich headers (crawling date, origin site etc.). A *storage (2)* service can be invoked to make the crawled document persistent in the WebContent warehouse. Observe that multiple *Translation (3)* services are used to translate to and from English, French, Chinese etc. *Semantic annotation (4)* services are invoked to analyze the text of the crawled pages and extract, e.g., specific aircraft brands, names of edible plants or bacteria that taint food etc. The annotations are added as a semantic header to the XML documents, under the form of XML-ized RDF snippets, and the modified documents are put back in the store. *Visualization* (5) and *query* (6) services can be used at this point to exploit the corpus, either via advanced user interfaces (e.g. "fish-

978-0-7695-3261-5/08 $25.00 © 2008 IEEE
DOI 10.1109/ICWE.2008.11

IEEE computer society

eye lens" view on documents) or by querying it, using a subset of XQuery (with full-text search) or SPARQL [19].

The WebContent warehouse is deployed in two settings. First, in a "closed" scenario, in a company Intranet, all services are provided by in-house components and communication takes place via an ESB [13]. Second, in a distributed, decentralized setting, computers are connected via the Internet and communication takes place via Web services exchanged over SOAP [23]. In both cases, there can be several instances of each service, in particular, storage services are provided by multiple machines; and, services can be called from inside or outside the federation of sites implementing the warehouse.

Two problems have to be solved in both settings: identifying services that implement a given interface, and efficiently executing the Web service calls. Efficiency is a particularly important concern in the distributed setting, since data transfers from one site to another may become the bottleneck. However, distribution is a great asset for large-scale warehouses such as the ones envisioned in our target applications, with large (and growing!) data volumes, therefore we focus on the distributed setting. Another source of inefficiency concerns repeated (redundant) execution of identical service calls, which may occur in large computations.

This work considers the problem of *efficient execution of distributed Web services*. Our solution is based on a composition language, namely ActiveXML (or AXML in short) [10], which in our setting can be seen as equivalent to a subset of BPEL. An ActiveXML document is an XML document specifying which services to call, how to build their input messages, and how the calls should be ordered. The contribution of this paper is an *AXML optimizer* called OptimAX, which given an AXML document, applies *equivalence-preserving rewriting* that transforms it into a different document, producing the same results, but possibly very different in shape and in the set of services it invokes. Thus, the execution of the rewritten document is likely to both be faster and consume less CPU resources than that of the original document.

Following the service-oriented architecture illustrated in Figure 1, we have implemented OptimAX as a Web service which, when invoked with an AXML document, returns the rewritten document. This step allows to benefit from the kind of performance-enhancing techniques typically applied in distributed databases [21], but in a new setting: losely coupled (vs. tightly controlled servers), generic (vs. tailored to specific indices and execution techniques), extensible to any service (vs. limited to the "inside" of the database server box). Another important difference is that AXML (and OptimAX) support continuous (streaming) services, such as the crawler service in Figure 1, or more generally any RSS feed. XML streams are at the core of many modern Web applications, e.g. for keeping a portal's content up to date, or for implementing continuous business interactions in a workflow-style setting.

OptimAX was demonstrated at [6].

This paper is organized as follows. Section 2 describes the AXML language and the extensions we bring to it to enable optimization. Section 3 formalizes the AXML optimization problem, and Section 4 describes our optimizer. Section 5 presents experimental results. Based on our problem analysis (Section 3), we classify and compare this work with previous related AXML works and with the state of the art, then conclude.

2 The AXML language

To introduce the AXML language, we use the following alphabets: a set $\mathcal{P}$ of *peer names*, a set $\mathcal{D}$ of *document names*, a set $\mathcal{S}$ of *service names*, a set $\mathcal{N}$ of *node identifiers* (for the XML tree nodes), and a set $\mathcal{L}$ of *labels* (for the XML tree tags). All peer names are distinct, thus they also serve as peer identifiers (or IDs in short). Document and service names are unique inside each peer, and they also serve as document/service *address*. The triple (peer name, document name, node identifier), suffices to identify a node, therefore we term it *node ID*, or node address. We may omit the document or peer when it is obvious from the context. Elements in the sets $\mathcal{P}, \mathcal{D}, \mathcal{S}, \mathcal{N}, \mathcal{L}$ are respectively denoted p, d, s, n and l, possibly with adornments such as subscripts or primes. Trees are denoted by the letter t, possibly with adornments. For sets, we use capital letters. By convention, we prefix node IDs with $\sharp$.

Intuitively, a peer represents a context of computation; we make no assumption about how the peers are logically connected, i.e. whether the peer network is structured or not.

2.1 Documents and services

We view an XML tree as a pair of $E \subseteq \mathcal{N} \times \mathcal{N}$, and a labelling function λ from the nodes in E to $\mathcal{L}$. Using the standard XML syntax, a sample XML tree t is:

⟨person id="#5"⟩⟨email⟩jdoe@ms.com⟨/email⟩
⟨first⟩john⟨/first⟩⟨last⟩doe⟨/last⟩⟨/person⟩

In this example, the node identifier #5 is depicted as an attribute. An XML document is a pair (t, d) where t is an XML tree and $d \in \mathcal{D}$; we may refer to it by d. A given document d, resp. service s, on a peer p is denoted $d@p$, resp. $s@p$.

We consider *deterministic* services, returning the same answer when invoked with the same parameters. In a Web environment, we may allow "relatively slow" variations in

call results, and consider the answers at time t and $t+\epsilon$, for some small ϵ, to be equally acceptable. In its simplest form, a service can be seen as a function with XML inputs and outputs, in the style of the request-response operation [22].

Continuous services We also consider *continuous services* that work on streams of trees and start processing their input incrementally, *before* it has been fully received. A particular class of continuous service have no inputs and emit a stream of XML trees. This corresponds to an XML subscription, in the style of RSS.

Let s be a service with n inputs. When the service is running, it expects to receive a stream of XML trees for each input. Any stream finishes with a special token denoted *eof*, that no tree may follow. Trees can arrive in all inputs in parallel. When a tree is received in one input, the service may perform an internal computation and/or may output zero or more trees. More specifically, we consider a large class of *distributive* services, such that for each $1 \leq i \leq n$, and for any finite streams $T_1, \ldots, T_i', T_i'', \ldots, T_n$, the following holds:

$s(T_1, \ldots, (T_i' + T_i''), \ldots, T_n) = s(T_1, \ldots, T_i', \ldots, T_n) + s(T_1, \ldots, T_i'', \ldots, T_n)$

where $+$ stands for stream concatenation. This property holds for all services defined as XPath queries, and also for a large class of XQuery queries, namely, for-where-let-return (also known as FLWR) expressions.

Example Consider a query service defined by the following query:

for \$x in \$in1, \$y in \$in2 where \$x/b=\$y/b return ⟨z⟩{x/a}⟨/z⟩

A possible sequence of inputs and outputs for this service is the following (here and in the rest of the paper, we use bold fonts to highlight some nodes for readability):

\$in1	\$in2	result
⟨x⟩⟨a⟩0⟨/a⟩⟨b⟩**1**⟨/b⟩⟨/x⟩	⟨y⟩⟨b⟩**0**⟨/b⟩⟨/y⟩	
	⟨y⟩⟨b⟩**1**⟨/b⟩⟨/y⟩	⟨z⟩⟨a⟩0⟨/a⟩⟨/z⟩
	⟨y⟩⟨b⟩**2**⟨/b⟩⟨/y⟩	
⟨x⟩⟨a⟩3⟨/a⟩⟨b⟩**0**⟨/b⟩⟨/x⟩		⟨z⟩⟨a⟩3⟨/a⟩⟨/z⟩

A non-continuous service may be seen as a particular case of a continuous one, delaying output until it has received an eof token from each of its inputs. At this point, the service outputs its complete results followed by eof.

2.2 Active XML data

An *AXML document* is an XML document where some nodes labeled with the label *sc* (standing for *service call*) are given particular semantics. Specifically, an *sc* node has:

- Two children, labeled *peer* and *service*, specify a peer name $p_1 \in \mathcal{P}$ and a service name $s_1 \in \mathcal{S}$, where $s_1@p_1$ identifies an existing Web service.
- A set of children labeled *param* specify the parameters.

Let $d_0@p_0$ be an AXML document containing a service call to a service $s_1@p_1$ as above. When the call is *activated*, the following sequence of steps takes place: (1) p_0 sends a copy of the *param*-labeled children of the *sc* node, to peer p_1, asking it to evaluate s_1 on these parameters; (2) p_1 evaluates s_1 on this input; (3) a copy of the result is inserted as a sibling of the *sc* node.

When a continuous service call is activated, step 1 above takes place just once, while steps 2 and 3, together, occur repeatedly starting from that moment. The response trees successively sent by p_1 accumulate as siblings of the *sc* node.

Observe that sc nodes may appear as children of other sc nodes. Moreover, the results of an activated service call may contain other service calls.

AXML supports several mechanisms for deciding when to activate a service call. One may *explicitly* request each call activation. For instance, consider a service call $sc_2(sc_1)$, i.e. sc_1 is a parameter of sc_2. The user may choose to activate just sc_2, in this case the sc_1 element *as such* is used as a parameter for sc_2. The call sc_1 may be activated in the future.

Another more frequent case occurs when users want to activate *all the necessary service calls* to bring the document to a certain state. For instance, before sending a document d to a partner that does not understand AXML, we need to activate all the calls in d, the calls which may be received in their results, and so on. Or, it may be necessary to bring d to a given (A)XML type by selectively activating some calls only; algorithms to find these calls are given in [8].

In this work, we define a simple, yet flexible approach for deciding when to activate calls. This approach is based on a set of *default activation order* constraints, which apply by default, and on some *explicit activation order* constraints, which can be manipulated by the user.

The first default activation order rule is dao_1 in Table 1. The reason for this rule is that the majority of the services available today requires plain XML inputs and returns plain XML outputs. Activating the inner call first is more likely to lead to call sc_2 with XML input. Rule dao_1 cannot influence the activation order of two calls when none is an ancestor of the other. To capture such constraints, we enable users to specify that a given service call should be activated only after another call's activation and more precisely, after receiving the first answer of that service. Moreover, for continuous services, we may wish to distinguish between activating service call sc_1 after sc_2 has been *activated* (but has not finished executing), and activating sc_1 after sc_2 has been activated and has finished, i.e. it has sent its eof. Syntactically, such constraints are expressed using two attributes afterActivated and afterTerminated, whose interpretation is provided by the rules eao_1 and eao_2 in Table 1.

(dao_1)	A call sc_1 which is a parameter of sc_2 is activated before sc_2.
(eao_1)	A call sc_1 having an afterActivated attribute whose value is the ID of another call sc_2 is activated after sc_2 has been activated.
(eao_2)	A call sc_1 having an afterTerminated attribute whose value is the ID of a call sc_2 is activated after sc_2 has terminated its execution.
(dao_2)	A call to the service $send@p$ is activated before all the service calls comprised in its parameters have been activated.
(dao_3)	A call to the service $receive@p$ is activated when the first message from the corresponding $send@p'$ call reaches p.
(noa_1)	Let sc be a call to $send@p$, in some document $d@p$. After activating sc, the calls descendants of sc are never activated (at p).
(dao_4)	A call to $newnode$ is activated before all its descendant calls.
(noa_2)	After a call to $newnode$ has been activated, its descendant calls are never activated at the original peer.

Table 1. Activation order rules.

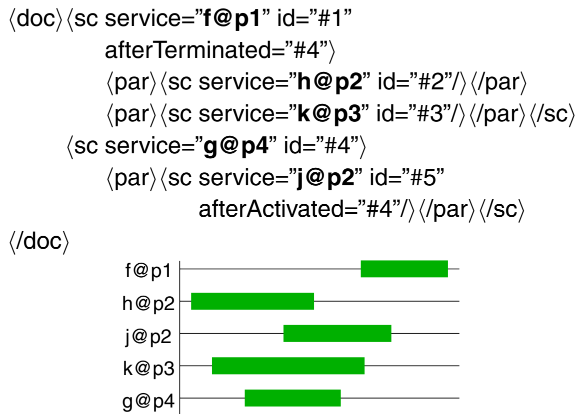

Figure 2. Sample AXML document and timeline for its service call activation.

Example: activation order Figure 2 depicts a simple AXML document at peer p_0, and a possible timeline of the activations of its calls. The period between the activation and termination of each service is shown by a horizontal bar.

doc1@p1	⟨doc⟩⟨sc service="**send@p1**" id="**1**"⟩ ⟨what⟩⟨sc service="f@p3" id="2"/⟩ *⟨fres⟩1⟨/fres⟩⟨fres⟩2⟨/fres⟩* ⟨/what⟩ ⟨where⟩**p2.doc2.#3**⟨/where⟩ ⟨/sc⟩ ⟨/doc⟩
doc2@p2	⟨doc⟩⟨sc service="**receive@p2**" id="**3**"⟩ ⟨from⟩**p1.doc1.#1**⟨/from⟩ ⟨/sc⟩ *⟨fres⟩1⟨/fres⟩* ⟨/doc⟩

Figure 3. Sample activation of calls to $send$ and $receive$.

2.3 Extension: built-in AXML services and replication

To the basic AXML model above, we add a small set of predefined services, which we assume available on all peers.

Send and receive are two services used to send (streams of) XML data from one place to another. The $send$ service has two parameters. The $what$ parameter represents the data to be sent from one site to another. This may be plain XML, some service calls or references to service calls. The $where$ parameter is a node ID. The $receive$ service has one $from$ parameter which is a node ID. The following integrity constraint applies: for each call to a $send$ service, there is exactly one call to a $receive$ service, such that the value of the $where$ child of the $send$ call is the ID of the $receive$ call, and the value of the $from$ child of the $receive$ call is the ID of the $send$ call.

One of the main applications of $send$ and $receive$ concerns the sending of data streams, as Figure 3 illustrates. Consider for now only the XML content shown in upright part of the table. The document $doc_1@p1$ contains a call to the local service $send@p_1$, with a call to $f@p_3$ as parameter. The destination address is the node identified by $(p_2, doc_2, \#3)$, which is the call to $receive$. Once the call to $f@p_3$ is activated, it returns $fres$ elements shown in italic font in $doc_1.xml$ in Figure 3; activating the calls to $send$ and $receive$ transmits these elements into the document $doc_2@p_2$. In the figure, the last element has not yet arrived in $doc_2.xml$.

A call to $send@p$ or $receive@p$ can only be activated when the call is in a document at peer p. This is a syntactic simplification only; we will show that it is possible for a peer p to trigger the sending of some data from another peer p'.

The introduction of the $send$ and $receive$ services requires new activation order rules, namely dao_2, dao_3 and

noa_1 in Table 1. Rule dao_2 specifies that by default, $send$ distributes the computation (not its result). Rule dao_3 shows that $receive$ calls are not activated individually but only as a consequence of receiving a message. Finally, the *no* activation rule noa_1 states that if a service call sc is sent from p to p' by a $send$, sc is not be evaluated at p.

The newnode service installs new AXML trees on a peer. It has a single $what$ parameter, which is an AXML tree. Activating the call to $newnode@p(t)$ creates a new document at peer p, whose associated data tree is t. The service returns the identifier of the new document's root. Observe that $newnode$ is quite powerful, since it enables the distribution of data and computations among peers.

The activation order rules dao_4 and noa_2 apply to calls to $newnode$. Rule dao_4 favors distribution, i.e. it causes AXML code to be sent before being activated. The no-activation rule noa_2 is similar to noa_1.

Activation order: putting it all together Together, rules dao_1, dao_2, dao_3 and dao_4 provide the default evaluation order for service calls appearing in AXML documents. These rules cannot cause cyclic dependencies. Explicit order constraints (eao_1, eao_2) override the default rules, and may introduce cycles. Documents with cyclic constraints are invalid and we do not consider them further.

Activation schedule Let d be a document and $sc_1, sc_2, \ldots, sc_k$ be the service calls from d. An activation schedule (or schedule, in short) for S is a list of pairs $[(sc_1, \tau_1), (sc_2, \tau_2), \ldots, (sc_k, \tau_k)]$ such that for $1 \leq i \leq k$, τ_k is a moment in time, sc_1 is activated at the moment τ_1, sc_2 is activated at the moment τ_2 etc. The schedule is said *valid* iff it respects: (i) all the eao_1 and eao_2 constraints of d; and (ii) as many of the $dao_1 - dao_4$ constraints as possible without violating the (eao_1) and (eao_2) constraints.

Observe that valid schedules largely allow parallel activation of continuous services (the only limitation being the explicit use of afterTerminated). The activation order example of subsection 2.2 illustrates a valid schedule. We focus on valid ones from now on.

Replication We assume that some AXML documents may be *replicas* (or copies) of each other, and similarly services may be replicated. A most important example for us is a *query service*. Given the string of the query, any peer equipped with a query processor can provide this query as a service, and all such services are equivalent. Observe that different copies of the same document may evolve independently with time, however, they will eventually reach the same state.

Formally, we consider an abstract peer, called *any*, and use $d@any$ to refer to any of the replicas of d, and similarly for services. We assume that each peer is able to to identify one of the concrete resources corresponding to $d@any$ or $s@any$. For instance, an approach based on semantic service matching is provided in [11].

3 AXML activation and optimization problems

Having introduced AXML, we now chart several interesting problems which arise in this setting, show how they relate to each other, and pinpoint the specific optimization problem addressed in this work.

3.1 AXML activation

Given an AXML document d, we denote by $SC(d)$ the set of service calls in the document. Observe that $SC(d)$ may grow with time, as results (including sc elements) are added to the document. In principle, $SC(d)$ may grow to be infinite, e.g. consider a call to a service $f@p$ that returns exactly one call to $f@p$. We consider the practical setting when the size of $SC(d)$ is bounded.

Cost of an activation Let $d@p_0$ be an AXML document and $sc \in SC(d)$ be a call to $f@p$. The cost of activating sc is defined as:

$$c(sc) = \alpha \times c_f + \beta \times (s_p/bw_{p_0 \to p} + s_f/bw_{p \to p_0})$$

where: α and β are some numerical weights; c_f is the cost associated to the computation of f at the peer p; $bw_{p \to p_0}$ and $bw_{p_0 \to p}$, respectively, are the bandwidths from p to p_0, respectively from p_0 to p; s_p is the size of the parameters of the calls to $f@p$; s_f is the size of the results produced by the calls to $f@p$.

We focus on activations with a finite cost, which requires that c_f, s_p and s_f be finite; the latter implies that f returns a finite number of answers. (A simple extensions to infinite streams would consider the cost per tree in the stream.) If p is any, then $c(sc)$ is set to an upper bound constant max.

We define the *cost of an activation schedule* as the sum of the activation cost of all the calls in the schedule. While a schedule describes very precisely a given AXML computation, we would like to consider activation costs independently of the particular moment when each call is activated. To that effect, we introduce the following definition.

Equivalent schedules Let d be an AXML document and T_1, T_2 be two schedules over two sets of services $S_1, S_2 \subseteq SC(d)$. We say the schedules are equivalent, denoted $T_1 \equiv T_2$, iff applying T_1, resp. T_2 on the document leads to documents that are equal.

Note that S_1 and S_2 may or may not coincide, as shown in the following example.

Empty-result call Let T_1 a schedule over $S_1 \subseteq SC(d)$. Assume that for some $sc \in S_1$, it is known that activating sc does not bring results other than eof. Let $S_2 = S_1 \setminus \{sc\}$ and T_2 the restriction of T_1 to S_2, then T_1 and T_2 are

equivalent (modulo eof, which we ignore by a mild abuse of terminology).

Valid schedule equivalence Let d be an AXML document and $S \subseteq SC(d)$ be a set of service calls from d. All valid schedules for S are equivalent and have the same cost.

Intuitively, valid schedules are equivalent due to the distributive, deterministic services which, called with the same parameters, produce the same results, even if some streams are created at different moments and progress at different rates in different schedules. They have the same cost because our cost model focuses on the total work, which does not change with time.

Set activation Let d be an AXML document and $S \subseteq SC(d)$. We term set activation of S on d the execution of any valid schedule for S. The cost of the activation is the cost of any valid schedule for S.

We term *one-stage activation* of d the set activation of all calls in $SC(d)$. If a call in $SC(d)$ returns another call sc', the latter is not activated in this stage.

We now consider the process of activating all calls in an AXML document until the document reaches a stationary state. Under the assumptions made here (the number of service calls in d is bounded, and services return finite streams), the fixed point state is finite, which entails that after a while, no new calls are returned by running service calls.

Full schedule Let d be an AXML document and SC_0 be the initial set of service calls in d. Let SC_1 be the service calls returned by the set activation of SC_0, and similarly, for $i = 2, \ldots, k$, let SC_{i+1} be the set of service calls returned by the set activation of SC_i. (We chose k so that $SC_k \neq \emptyset$ and $SC_{k+1} = \emptyset$). A full schedule for d is a schedule for all the calls in $\bigcup_{0 \leq i \leq k} SC_i$, such that:

- The restriction of the schedule to SC_i, for any $0 \leq i \leq k$, is valid.
- Whenever a call sc_i returned a call sc_j, sc_i appears before sc_j in the schedule.

Observe that in a full schedule, calls need not appear in the order in which they appeared in d: a call from SC_0 may be activated after a call from SC_5.

As before, all full schedules of d are equivalent and have the same cost. We define the *full activation* of d as the execution of any full schedule of d. If all services return plain XML data, full and one-stage activation coincide.

3.2 AXML optimization

We now consider the problem of *AXML optimization*, focusing first on one-stage.

One-stage optimization Let d be an AXML document and $S \subseteq SC(d)$ a subset of the calls in d. The process of one-stage optimization for d consist of finding a document d' such that:

- one-stage activation of d and d' produce identical documents (up to terminated service calls);
- the cost of the set activation of d' is smaller than, or equal to that of d.

Observe that optimization is a static process, which does not involve call activations. Optimization is *exhaustive* if it produces a document d with the minimum cost among all documents equivalent to d.

Let us now consider the integration of optimization in a full evaluation process, where we have to activate the calls in d, then the possible calls in their results etc. The choice of when and how often to invoke the optimizer impacts the rewritings it may find, thus the full activation cost. The main reason is that the optimizer decides to rewrite the document based on the service calls it contains *at optimization time*, and the latter change as activation proceeds. To characterize the goal of optimization, we define:

Document equivalence Let $d@p$, $d'@p$ be two documents at the same peer. We say d and d' are equivalent, denoted $d \equiv d'$, if the result of full activation on d and d' coincide (up to some terminated service calls).

This notion of equivalence characterizes documents that are *eventually* equal after their full activation. The documents may go through different states during the process, may call services from different peers etc. From the perspective of the user requiring the full activation result of $d@p$, the result of $d'@p$ is the same. From the system perspective, given a document d and a set $\mathcal{R}$ of rewriting rules, optimization can be seen as repeatedly applying $\mathcal{R}$ rules to obtain new documents equivalent to d, and keeping the one with the lowest evaluation cost.

Full optimization Let d be a document and $\mathcal{R}$ a set of rules. The full optimization problem for d and $\mathcal{R}$ consists of finding a sequence of steps chosen among:

- pick a service call currently in d which can be activated according to the ordering constraints of d, and activate it
- apply a rewriting rule from $\mathcal{R}$, rewriting d into d'

until all services calls in d have been activated, such that the total activation cost (including the past and possibly future service call activations) plus the total cost of optimization is the smallest among all possible such sequences of steps.

In the above, we also took into account the cost of optimization, which can be approximated by some constant c_o. Observe that the problem is more complex than just inserting optimization steps in some places into a given full schedule, because optimization may remove or add service calls, leading to re-scheduling.

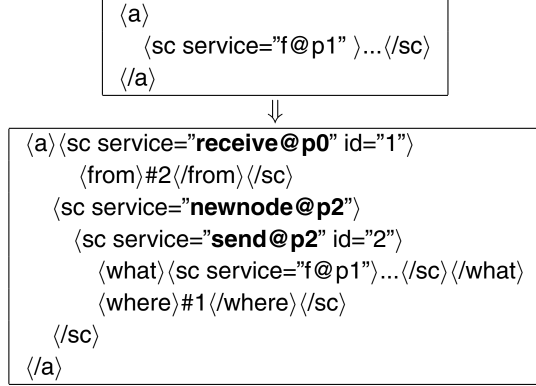

Figure 4. Delegation rule.

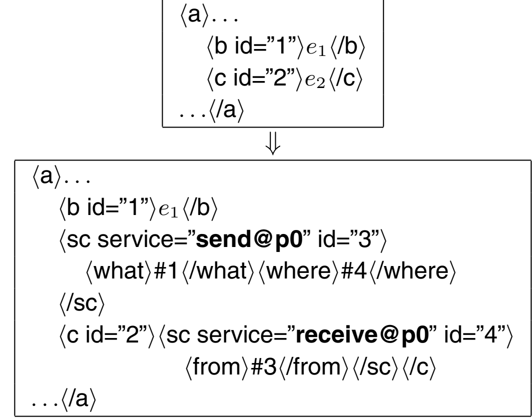

Figure 5. Factorization rule.

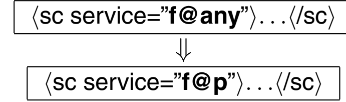

Figure 6. Instantiation rule.

If all service calls return plain XML results, then invoking the optimizer only once, prior to any activation, is a solution to the full optimization problem.

If service calls are allowed to return any AXML trees with service calls, the problem is undecidable. The intuition for this is the following. Optimizing too rarely may lead to poor activation decisions, which could have been avoided if we had chosen to invoke the optimizer more often. Optimizing too often, on the other hand, may also be suboptimal. For instance, the optimizer may rewrite a subtree t of d into t', instead of waiting for some more activations which may have produced, say, a subtree t'' of d, such that considering t and t'' together enables a big cost-saving rewriting, which cannot be applied based on t' and t''. Thus, one can exhibit a document when optimizing *before each call activation*, even assuming $c_o = 0$, is suboptimal. Moreover, in reality, $c_o \neq 0$, thus very frequent optimization is impractical.

Based on this analysis, we have built an optimizer for *one-stage optimization*, whose details are described in the remainder of the paper. In the full evaluation setting, we recommend invoking the optimizer once on SC_0, then on SC_1, then on SC_2 and so forth, a heuristic which we find a reasonable compromise.

4 Optimax

In this section, we describe the actual optimizer we implemented for AXML.

4.1 Search space: optimization rules

The complete search space is the set of distinct documents obtained by repeatedly applying a set of rewriting rules which we describe next. They all preserve AXML equivalence as defined in Section 3.2.

The *delegation* rule (Figure 4) concerns distributing computations. The rule depicts a document before (upper) and after (lower) applying a delegation rule. We assume the document resides at peer p_0. The rule introduces three new service calls to $send$, $receive$ and $newnode$. The effect is to install at p_2 an AXML document (via $newnode$), such that the call to $f@p_1$ will be performed from that document, i.e. from p_2, not from p_0. As soon as f results start accumulating at p_2, the $send$ call will transmit them to the $receive$ call at the original peer p_0, bringing thus the results in the original document. The delegation rule may reduce costs by cutting down data transfers. For instance, assume that $p_1 = p_2$ and the call to f had as parameter a call to $g@p_1$. In this case, activating the upper document would transit the results of $g@p_1$ from p_1 to p_0 and then back from p_0 to p_1. The lower plan eliminates these needless transfers.

The *factorization* rule (Figure 5) eliminates redundant computations. In this rule, e_1 and e_2 are two sets of AXML trees, such that each tree in e_1 is equivalent (as defined in Section 3.2) to some tree in e_2 and viceversa. The rule replaces e_2 with a pair of calls to $send$ and $receive$, which copy e_1 as children of c, effectively in replacement of e_2. If e_2 contained service calls, the rewritten document reduces the actual activated calls and (if the services were on remote peers) also reduces inter-peer transfers. The rewritten document requires a local copy of data from the b to the c element, but such transfers are likely less costly (and our cost formula ignores them).

The *instantiation* rule (Figure 6) turns an abstract service call to $f@any$ into a concrete call to $f@p$, where p is one of the peers providing f. Given that activations of

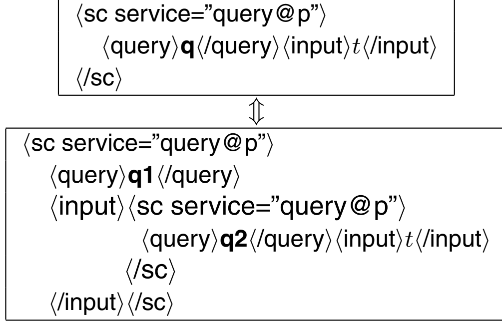

Figure 7. Query (de)composition rule.

calls at any have maximum cost (Section 3.1), this rule always reduces cost. Moreover, when several peers provide f, different cost reductions can be obtained. Documents with calls to services $@any$ give more options to the optimizer. When services are queries (Section 2.1), plans produced by instantiation resemble distributed strategies in mediator systems [21].

The *query composition/decomposition* rule (Figure 7) applies to calls to services defined by queries. In this rule, q, q_1 and q_2 are XML queries such that $q \equiv q_1(q_2)$, i.e. for any XML input t, $q(t) = q_1(q_2(t))$. The upper document calls the query q, while the lower document nests the call to q_2 as a parameter of the call to q_1. In this rule, p can also be any. Query composition (going from the lower part to the upper part in Figure 7) reduces costs by cutting the overhead of one activation. Query decomposition (going from the upper part to the lower part) may reduce costs if the sub-queries q_1, q_2 can be handled very efficiently by some processor unable to handle the full query q. For example, q_1 may be an XPath query which may be answered using an index [4], and q_2 is an XML construction query applying on the results of q_1. This rule applies more generally for any queries $q, q_1, \ldots, q_n$ such that $q \equiv q_1(q_2, \ldots, q_k)$. The rule *de facto* integrates XML query optimization into the larger problem of AXML optimization.

The *useless call* rule eliminates calls with anticipated empty results (recall the example for the empty result call in Section 3.1).

Search space size Let d be a document such that all calls in $SC(d)$ refer to specific peers (not any) and assume delegation is the only rule. Let P be a set of peers known to the optimizer. Each call can be delegated to each $p \in P$, leading to a search space of size $|SC(d)|^{|P|}$. On the contrary, assume now that all calls in $SC(d)$ refer to any and enable instantiation: the size grows to $|SC(d)|^{2|P|}$. The impact of the other rules described above is more difficult to quantify since it depends heavily on the document. In any case, exhaustive search may be quite costly, and we are interested in efficient, non-exhaustive optimization.

4.2 Search strategies and heuristics

The applications we consider for AXML have very varied profiles. One class of applications focuses on subscriptions [7], where factorization is crucial (to avoid duplicate data transfers) and query composition may also apply (to efficiently filter out subscriptions). Other applications consider distributed data management workflows [18], where instantiation and delegation are central. As another example, our current WebContent project [25] focuses on XQuery processing in a structured P2P network, based on a distributed XML index; the main rule is query decomposition isolating the largest subquery the index may handle. The total time budget given to the optimizer also varies with the application.

To accommodate such a variety of settings, we have devised a simple XML dialect for specifying the optimizer's search strategy. Each strategy is a sequence of *steps*. Each step applies a given *search algorithm* (which can be: depth-first or breadth-first, and possibly attempt to rewrite the cheapest plan first), using a given *rule set*, and with an *upper bound* on the number of plans developed. For instance, the first step may develop 100 plans in a breadth-first, cost-driven manner, then the second step may apply depth-first, cost-driven search producing 20 more plans, then the best plan found so far is chosen. The default strategy runs a single step, using the depth-first, cost-driven strategy, and develops 100 plans using all rules. In our experience, this simple strategy lead to useful rewritings.

4.3 Implementation issues

OptimAX is implemented in Java and integrated with the recent v.2 of the AXML engine [10]. This version relies on an XQuery-compliant database (eXist [24]) to store and update AXML documents, enabling it to scale up; the previous AXML engine was limited by its in-memory, DOM-based document management. Axis2 is used for Web service messaging. The activation order constraints described in Section 2.2, and continuous query services as described in Section 2.1 were specified and implemented in AXML v2 as part of the optimizer integration effort. We discuss here some notable engineering issues.

For speed, the optimizer must run in memory, however, handling many rewritings of large XML documents may lead to memory problems. Thus, the optimizer creates, from an AXML document, an *active plan*, i.e. a copy omitting XML nodes without sc descendant (which our rules do not need). An optimized plan is assembled back with its plain XML subtrees in the eXist repository by the AXML engine. To preserve the correctness of plan equality tests, omitted data trees are replaced by compact hash codes in the plans.

Applying the factorization rule (Figure 5) requires effi-

cient equivalence checking, which we implement by testing the equality of plans modulo terminated calls (this check is exact for one-stage optimization). To avoid comparing all node pairs from a plan, a hash function $h_{\equiv}$ is efficiently computed on plans, and we only compare pairs of nodes with the same $h_{\equiv}$ results.

Implementing the rewriting rules required special caution to preserve AXML plan coherence. Consider the factorization rule replacing expression e_2 with some $send$-$receive$ calls copying the same data from another node (Figure 5). Assume that a descendant of e_2 is a call to $send$, and the corresponding $receive$ is outside e_2: erasing e_2 would leave the $receive$ without a $send$, thus factorization is not possible. On the contrary, assume that e_2 contains a call to $receive$: then the $send$ call corresponding to this call to $receive$ must be deleted, too. This analysis may follow recursively connections between $send$ and $receive$ calls. The performance remains acceptable, because the plans are quite compact.

5 Experimental analysis

In this section, we illustrate OptimAX usage on two examples derived from the WebContent project. We then study its efficiency, on a set of synthetic documents.

5.1 Case studies

A first simple scenario of OptimAX in WebContent concerns a document processing flow of the form:
$annotate@any(translate@any(store@any(crawl@p)))$
where the services are those described in Section 1, and the nesting of calls encodes their data dependencies, ensured by the AXML default activation order. Moreover, OptimAX is able to instantiate the peers providing annotation, translation, resp. storage services for the documents returned by the crawling service at peer p, based on a load catalog describing the available peers and services. The translation and annotation services update their input documents directly in the warehouse. OptimAX may use delegation to ship the computations somewhere close to p, as to reduce data transfers.

In a more complex query scenario, OptimAX is invoked on documents including a single call to the pre-defined *abstract* service *WebContentQuery*. As parameter of the call, a single XQuery query is given. The *WebContentQuery* service is abstract, i.e., it is implemented only by a combination of other services available in the WebContent distributed warehouse.

A WebContent-specific OptimAX rule rewrites calls to *WebContentQuery*(q) into expressions of the form:

$$gqs@local(join, kadop@any(tpq_1), kadop@any(tpq_2), \ldots kadop@any(tpq_n))$$

In the above, $tpq_1, tpq_2, \ldots, tpq_n$ are conjunctive tree pattern queries (think of XPath queries allowed to return several nodes), and $join$ is an XQuery query, such that:

$$q \equiv join(tpq_1, tpq_2, \ldots, tpq_n)$$

In other words, $tpq_1, tpq_2, \ldots, tpq_n$ are conjunctive tree pattern queries extracted from q, and $join$ is a *recomposition* query specifying how to put together the results of q based on the results of the tree pattern queries. The decomposition is provided by the TGV XQuery algebraic compiler [20], which is embedded (for the WebContent application) into OptimAX. The decomposition is useful, because in WebContent, tree pattern queries can be efficiently executed over a distributed set of XML storage providers, once they agree to maintain, together, a *distributed full-text XML index*. The sub-system responsible of building and exploiting this index, built in a previous project, is called KadoP [4]. In our setting, each KadoP site provides a service $kadop$ whose parameter is a tree pattern query. Upon invocation, the site coordinates execution, retrieves the results, and returns them in streaming fashion (the $kadop$ service is continuous).

Without delving into details, we mention that the execution of each tree pattern query can be assimilated to a distributed join, combining posting lists from the distributed KadoP index. Each join entails some data transfers, which tend to become the bottleneck in query evaluation. OptimAX uses the freedom degrees provided by $@any$ to place the joins (i.e. to chose the sites providing the $kadop$ services) in such a way as to minimize the data transfers.

To conclude this section, we observe that OptimAX' modular rule set enables it to blend context-specific rewritings (e.g. decomposing calls to *WebContentQuery*) with generic rewritings (e.g. finding the sites which evaluate calls to $kadop@any$). Moreover, given that OptimAX is an "external" optimizer, i.e. it works outside a database core, it can be extended easily to include specialized rule engines, such as the XQuery-specific engine TGV.

In both of the above cases, the optimizer is very fast (tens of a second). We have deployed the WebContent platform (including OptimAX) on four machines in our lab intranet, where the actual impact of optimization was not very visible. Large-scale deployment of the distributed WebContent platform is scheduled for the summer of 2008.

5.2 Efficiency

We now study the *efficiency of plan finding* in OptimAX. The AXML activation cost reductions due to delegation and instantiation have been shown to reach several orders of

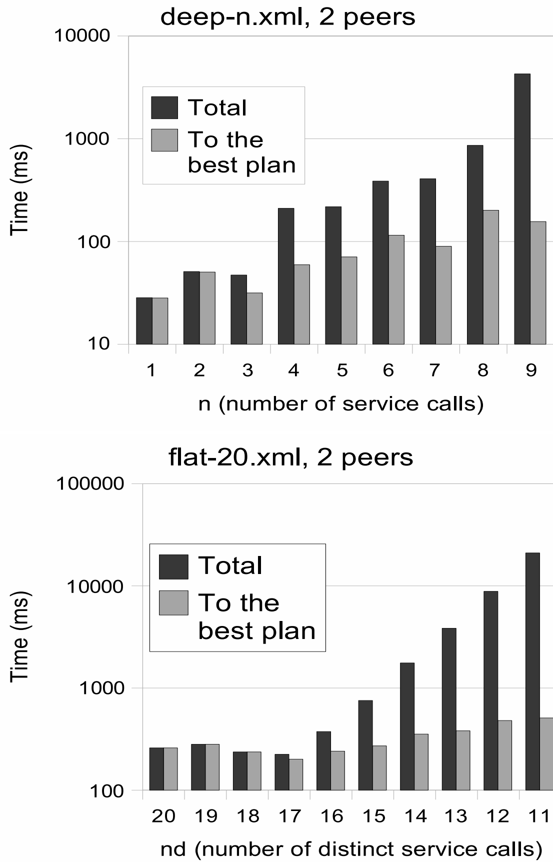

Figure 8. Optimization time for *deep* and *flat* documents.

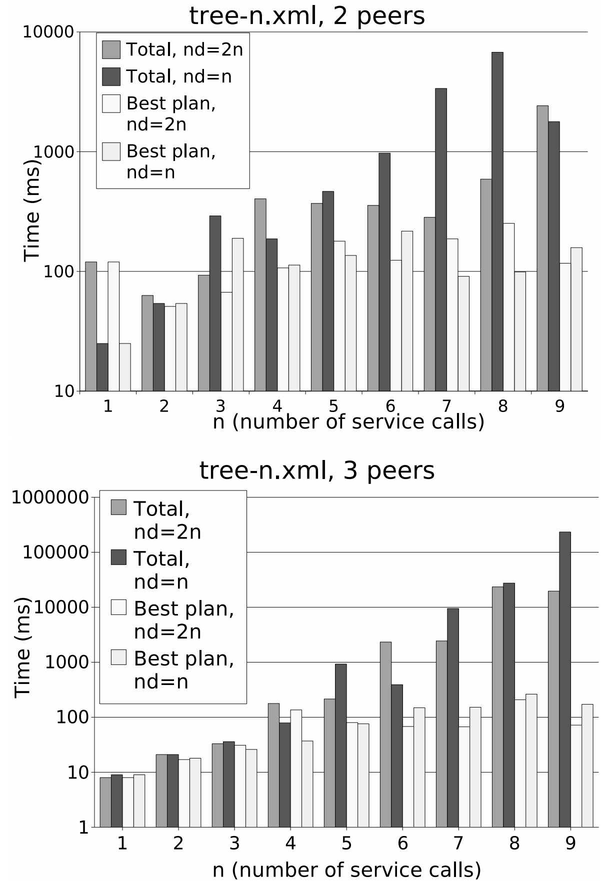

Figure 9. Optimization time for *tree* documents.

magnitude [17, 18]. Clearly, other rules can improve that factor. Experiments ran at a computer with Intel Xeon CPU 5120 @ 1.86GHz, 3GB of Ram and Mandriva Linux 10.

We consider three sets of synthetic parametric documents, consisting exclusively of service calls (except the root). Document *deep-n.xml* consists of n service calls organized in a linear tree of fanout 1. Document *flat-n.xml* consists of a root node and n service call children. Finally, document *tree-n.xml* is an arbitrary tree of n service call nodes where the maximum fanout is $f_{max} = 6$. Each document is characterized by n_d, the number of *distinct* services called. Services assigned randomly (uniform distribution over a set of n_d) to each tree node. Each optimization problem is further characterized by n_p, the number of peers which to which computations may be delegated. In these experiments we consider delegation and factorization.

The graphs in Figure 8 and 9 depict the time for exhaustive optimization, resp. to obtain a plan with the best cost (as determined by of the exhaustive process).

In the top graph of Figure 8, we study the documents *deep-n.xml*, maximizing opportunities for delegation due to the deep nesting of calls. Total optimization time grows exponentially with n, which agrees with the lower bound for the search space size (Section 4.1). The bottom graph considers the document *flat-20.xml* while varying n_d; here, factorization applies more and more as n_d lowers.

In Figure 9, we use the documents *tree-n.xml*, varying n, with $n_p = 2$ (upper) and $n_p = 3$ (lower). We set n_d to n, resp. $2n$. Since the services are chosen independently for each node, for both values of n_d, some calls are likely to refer to the same service, and some factorization occurs (more for $n_d = n$). The graphs show that the total optimization time grows exponentially with n, and generally grows as n_d decreases, since this often means more factorization opportunities. Moreover, the total time grows from a few seconds, to a few hundred seconds as n_p moves from 2 to 3, also as predicted by the lower bound on the search space size from Section 4.1. For applications running for a long time, it may make sense to spend some minutes optimizing, but for others, exhaustive optimization is prohibitive.

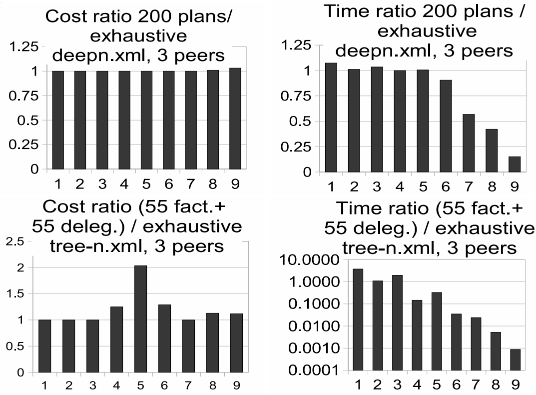

Figure 10. Time and cost trade-offs for *deep-n.xml* **with non-exhaustive strategies.**

The good news shared by all graphs in Figure 8 and 9 is that *the time to the best solution is very moderate*, of the order of 0.1-0.5 seconds. This is due to our depth-first then cost-based search strategy we apply (to rewrite a plan, we consider the most rewritten ones, and among them, we pick the cheapest one).

Our last experiment demonstrates the interest of non-exhaustive strategies. For the two graphs in Figure 10 at the top, our strategy ran a depth-first, greedy search limited at 200 delegations and/or factorizations. For small n values, this strategy is complete, and the running time (a few ms) is almost identical to that of the full exploration. For larger n values, limiting the search to 200 rewritings marginally increases the cost of the best solution, but decreases the search time by an order of magnitude! At the bottom of Figure 10, we used the same plans as at the bottom graph of Figure 9, with the following strategy: explore 55 factorizations, *then* 55 delegations. This strategy finds plans within a factor of 2 of the optimum, but may decrease running time by 3 orders of magnitude!

In conclusion, that while the AXML search space is huge, efficient exhaustive strategies typically find an optimal plan fast. For larger problems, non-exhaustive “smart” strategies critically cut optimization time, while producing near-optimal plans. This demonstrates the practical applicability of our optimizer.

6 Related works and conclusion

The starting point of this work is the AXML language [2], which we extended with the $send$, $receive$ and $newnode$ services and with a flexible yet simple way to control call activation order (Section 2.2). This brings important benefits to users, which may combine continuous and non-continuous services at will. It is also crucial for the efficiency of optimized plans. Consider e.g. the plan $send@p_1(f@p_x, p_2.d_1.\#1)$. Depending on whether p_x is p_1, p_2 or another peer, we may wish to activate $send$ first (thus, push the computation to p_x) or f first (thus, call f from p_1). Our previous algebraic proposal [5] was unable to express both - a shortcoming we detected while implementing OptimAX. We defined valid schedules and formalized the optimization problem accordingly.

A language for AXML replication and some XPath execution strategies were introduced in [3], which does not address optimization. The full optimization problem is solved in [1] in the particular case when the only rule is useless call elimination (Section 4). Delegation and instantiation have first been proposed in [18] in isolation and in an ad-hoc way which could not be generalized. In [5] we proposed an abstract algebra (with the shortcomings mentioned above) but did not discuss how it can be mapped into an implementable language. Finding which calls to activate to bring a document to a given type is shown to be sometimes undecidable in [8, 16] which do not consider optimization. Continuous services are used in [7] to specify monitoring programs, but algebraic optimization is not considered. XCraft [17] is an optimizer for non-continuous AXML. It uses a workflow model for AXML documents, which are split in pieces of fixed size optimized independently. In contrast, OptimAX, based on tree rewritings, is intimately connected with the AXML model, which allows it to include many more rules, e.g. factorization, query composition/decomposition etc., giving it more generality. Moreover, we explore many strategies, and we show that a greedy-based depth-first can quickly identify efficient plans, more reliably than a fixed-size divide and conquer approach.

Web service orchestration in workflow style, e.g. via BPEL4WS, is a very active area [9]. While AXML with ordering constraints has some workflow flavor, it trades many BPEL aspects (complex processes, exception handling etc.) for a data-centric character that enables specific data-oriented efficient optimizations. More generally, one can use either AXML or BPEL4WS to specify relatively simple workflows; we have experimented with a simple translation tool from one to the other. OptimAX functions in the realm of AXML documents, which we found more convenient to handle via rewriting than process specifications.

The work presented here follows previous works on distributed query processing [15, 21], and in particular in the context of mediator systems [14].

Our ongoing and future work in this area concerns the integration of Optimax with an algebraic XQuery optimizer, within the WebContent project [25].

Acknowledgments

This work has been partly funded by the French Government grant RNTL WebContent.

References

[1] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for Active XML. In *SIGMOD*, 2004.
[2] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-peer data and web services integration. In *VLDB (demo)*, 2002.
[3] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD*, 2003.
[4] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE*, 2008.
[5] S. Abiteboul, I. Manolescu, and E. Taropa. A framework for distributed XML data management. In *EDBT*, 2006.
[6] S. Abiteboul, I. Manolescu, and S. Zoupanos. OptimAX: Efficient Support for Data-Intensive Mash-Ups (demo). In *ICDE*, 2008.
[7] S. Abiteboul and B. Marinoiu. Distributed monitoring of peer-to-peer systems. In *WIDM*, 2007.
[8] S. Abiteboul, T. Milo, and O. Benjelloun. Regular rewriting of active XML and unambiguity. In *PODS*, 2005.
[9] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, 2004.
[10] ActiveXML home page. Available at http://www.activexml.net.
[11] S. Benbernou, X. He, and M. Said-Hacid. Implicit service calls in ActiveXML through OWL-S. In *ICSOC*, 2005.
[12] Business process execution language for web services. www.ibm.com/developerworks/library/ws-bpel.
[13] D. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.
[14] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB*, 1997.
[15] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4), 2000.
[16] A. Muscholl, T. Schwentick, and L. Segoufin. Active context-free games. In *STACS*, 2004.
[17] G. Ruberg and M. Mattoso. XCraft: Boosting the performance of Active XML materialization. In *EDBT*, 2008.
[18] N. Ruberg, G. Ruberg, and I. Manolescu. Towards cost-based optimizations for data-intensive web service computations. In *SBBD*, 2004.
[19] SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/.
[20] N. Travers, T. Dang-Ngoc, and T. Liu. TGV: A tree graph view for modeling untyped XQuery. In *DASFAA*, pages 1001–1006, 2007.
[21] P. Valduriez and T. Ozsu. *Principles of Distributed Database Systems*. Prentice Hall, 1999.
[22] W3C. WSDL: Web Services Definition Language 1.1.
[23] W3C. Soap version 1.2 part 1: Messaging framework (second edition), 2007.
[24] Open source native XML database. exist.sourceforge.net.
[25] WebContent, the Semantic Web platform (rntl project). www.webcontent.fr.