OptimAX: efficient support for data-intensive mash-ups

S. Abiteboul, I. Manolescu, S. Zoupanos

INRIA Futurs, Gemo group & LRI Université Paris Sud,
4, rue J. Monod, 91893 Orsay Cedex
firstname.lastname@inria.fr

I. Setting: mashing up the Web

The Web evolution trend known as the "Web 2.0" entails moving from a few content publishers and huge number of content readers, towards a network of peers, where rich, interactive applications are deployed on the Internet as a platform. Witness to this evolutions are collaborative editing and interaction tools like Wikis, blogs, social networking sites, P2P file-sharing platforms etc. A very recent class of such applications consists of *mash-ups*, which compose information artifacts (such as database queries, calls to Web services, geo-location information etc.) in a single Web-based application [1], [2]. Mash-up platforms can be seen as the ultimate tools for building data integration applications: data feeds are instantly mixed'n'matched, often with the help of GUIs. Mash-up success is also due to the wide availability of public interfaces to valuable data sources, based on XML (SOAP, RSS etc.) and/or Web services.

The ActiveXML language (AXML, in short) [3] has been proposed as a tool for decentralized, data-centric Web service integration. An ActiveXML document is an XML document including calls to Web services. A call can either be a simple request-response exchange, or a long-running subscriptions to some data stream. The language has several interesting features. It is *fully composable*: AXML trees can be sent as parameters to a service call, and a service call can return XML data with more embedded service call. It allows for *resource sharing and composition*: a query over an AXML document can be exported as a Web service, to be used by other peers in their own documents. Optimizations have been proposed [4], [5] to speed-up query evaluation on ActiveXML documents, and a user-friendly interface has been developed. All these aspects make AXML a great technology for mash-up style data integration.

The purpose of this demo is to demonstrate the power of AXML as a support for complex (yet efficient!), easy-to-build and easy-to-use mash-up applications. We propose to demonstrate how users can build an AXML mash-up application (with the help of a GUI). Further, we demonstrate the inner workings of OptimAX, our *AX*ML *optim*izer, which reformulates a user-specified mash-up into one producing the same results, but with better performance.

The technical innovation of our demo is twofold. (i) We show that the AXML language (with a set of minimal extensions) can be used as a *specification language*, *optimization language* (akin to logical plans) and *distributed execution language* (akin to physical plans), for dynamic, distributed Web-based mash-ups, in varied P2P settings. (ii) We demonstrate OptimAX's optimization rules and rewriting engine, also with the help of GUI. The optimizer draws up an initial query plan, and then rewrites it using a combination of heuristics and cost information in order to improve the plan's performance estimates. The process is outlined in Figure 1, which we comment in more detail later on.

By itself, the optimization problem we address is unique, and notably differs from classical distributed query optimization. The reasons are: distribution over a set of sites *potentially unknown at compile time* (see Section III); the *recursive* character of AXML (a service call may return another service call); and finally the *dynamic* aspect of an AXML application, parts of which can be modified (e.g. the user can add new service calls etc.) while other parts are running, going through optimization and evaluation steps.

The rest of this document is organized as follows. Section II briefly presents the (A)XML data model and the main elements of our plan algebra. Section III describes the demonstration scenario. We compare OptiMAX with similar projects and technologies in Section IV and we conclude.

II. AXML and its algebra

In this section, we review the basic concepts framing our OptimAX work: ActiveXML documents and their associated evaluation algebra. To ease reading, we will rely on a simple paranthesized notation for XML trees, e.g. writing $a(b)$ to denote an a node with a b child, and abstract away some details of the AXML notation [3].

A. ActiveXML documents

ActiveXML documents are XML documents including some special elements labeled sc (for *s*ervice *c*all). We now describe the basic features of the language as they have been laid out in [3]. Elements labeled sc describe a given Web service that should be called, and may include XML parameters of the call. When the call is *activated*, a request message (including the parameters) is sent to the Web service, and when the results are received, they are inserted in the AXML document as siblings of the sc node.

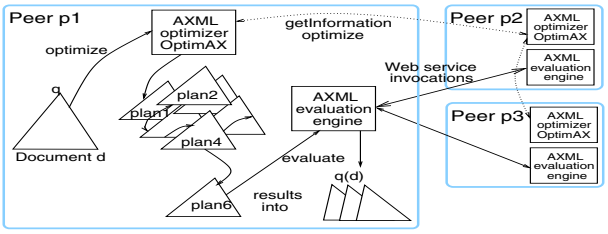

Fig. 1. AXML optimization process.

While the basic mode of interaction with a Web service consists of a single request and a single response message, as part of an ongoing thesis in the Gemo group, *stream services* are being developed. Once a call to a stream service has been activated, a stream of XML answers will be returned asynchronously, and they are all inserted in the caller AXML document as siblings of the sc node. After all answers have been returned, a special token "end-of-stream" (or *eos*) is returned. Observe that a "regular" service (returning just one answer) is a special case of continuous service returning a short stream. Thus, from now on, without loss of generality, we will refer to continuous services only.

We are mainly concerned with *declarative services*, defined by XQuery [6] queries. A few optimization techniques (e.g. factorization, see Sec. II-C) are also possible for non-declarative services, however, these are more limited. We consider a distributed setting, where different *peers* provide services and/or host AXML documents. We consider a set of distinct *peer identifiers* of the form $p_1, p_2, \ldots$, and a set of distinct *document identifiers* inside each peer, of the form $d_1, d_2, \ldots$. Inside a given document, sc nodes are identifiable by their *node IDs* of the form $n_1, n_2, \ldots$ etc.

The *default evaluation strategy* of an AXML document $d@p$ consists of activating all calls in d, receiving the results and inserting them at p, if there are any service calls among these results, activating those too etc. The process is potentially infinite (and the system stops it after a number of rounds), but practical applications are in fact much simpler and have a finite depth of recursive calls. The activation order of the calls in a document obeys the following constraint: whenever a service call sc_1 is a parameter of sc_2, sc_1 had to be activated before sc_2. Beyond this constraint, all calls can be activated in parallel. If an application needed to ensure that sc_x was activated prior to sc_y overriding the default order, the element corresponding to sc_y was annotated with an attribute of the form isAfter sc_x. Observe that the default evaluation strategy leads to all service call results transiting through peer p.

B. AXML extensions for optimization

We outline here a set of extensions brought in [5] and our subsequent work to the basic AXML language [3]. These extensions expose at the level of the language several features implicitly used in the previous language and platform. The advantage of this exposure is to make available to the optimizer flexible building blocks for efficient plans.

First, three new *special* Web services have been added to the language: *newNode*, *send* and *receive*. These services are special in the sense that calls to them are inserted by the optimizer, not by the AXML user.

The *newNode(tree, address)* service provided by peer p copies the XML *tree* as a child of the node whose address is given as a second parameter. If *address* is null, the tree is installed as a new standalone document hosted by p. Once activated, this service returns a single response message with the ID of the tree installed at p.

The *send(data, address)* service provided by peer p continually sends *data* as children of the receive service call node identified by *address*. Different from *newNode*, *send* can transfer a whole stream (if *data* is a stream) as children of the destination nodes, as they arrive. Observe that *send* can transfer all the results obtained from the call to a continuous service, but we restrict it so that it does not transfer activated service calls which have not received eos yet. Intuitively, we do not want to "migrate executing calls"; instead, we are allowed to move calls which have not yet been activated, and calls which have completed execution.

The *receive(data, address)* service provided by peer p is used as a counterpart to the *send*. The *receive* is a place marker indicating where data from a *send* should be inserted. The calls to *send* and *receive* are created together (in pairs); streams of results are transferred between a pair of activated calls to *send* respectively to *receive*. This is in the spirit of communication channels in pi-calculus [7]. The *address* is the identifier of the corresponding *send* that transmits data. The *data* argument of *receive* is not actually used by the service itself, rather, it serves to describe the data which is going to be received there. This description is useful for the optimizer to get a global view of the ongoing computations, as we will illustrate.

Another useful extension to the language consists of *generic* resources [5]. *Generic services* are Web services identified by their name and WSDL type description, but whose provider peer is unknown (or, in WSDL terms, for which bindings are unknown). We designate such services by *s@any*, where *any* stands for any peer. Similarly, *generic documents* of the form *d@any* designate any replica of a given document. A generic resource needs to be resolved into a particular concrete resource prior to being used.

The use of *continuous* and *special* services leads to refining activation order constraints. (i) To constrain the activation order of calls to continuous services, we replace isAfter by two attributes. Using isAfterActivated causes the second service's activation just after the first one's activation. Using isAfterTerminated allows activating one call when another one has produced its end-of-stream, e.g., to evaluate an aggregation over the stream. (ii) A call to the $send$ service is activated prior to all the calls it may have in its parameters. (Those calls will only be activated after they have reached their destination.) (iii) A call to the $newNode$ service is activated prior to all the calls it has in the $tree$ parameter. However, it can be activated only after all service calls in its $address$ parameters have terminated. (iv) A call to the $receive$ service

is activated when the first message from the corresponding $send$ reaches its destination.

C. AXML optimization

The process of *optimizing an AXML computation* consists of enumerating equivalent strategies and choosing one assumed to have lower computation costs. Here, *equivalent* means that the same query result is returned at p, however, different peers, document and services may have been used during the computation. As for costs, we are first interested in reducing the *response time*, and second, the *total work*.

The optimization and evaluation process inside a peer P_1 is outlined in Fig. 1. To evaluate query q over document d, OptimAX enumerates several plans, among which one, say $plan_6$, is chosen. To do so, the optimizer may consult with other peers' optimizers, either to gather catalog and cost information, or to delegate part of the optimization. Inter-optimizer communications are shown by dotted lines. The chosen plan is handed to the evaluation engine, which may interact with other peers' evaluation engines through Web service calls (either regular services or $send$, $newNode$ etc.). Thick arrows in Fig. 1 trace actual service call activations. Although not shown in the figure, the process may be repeated as long as answers to $q(d)$ bring more service calls which need to be activated and optimized.

Exhaustive application of the rewriting rules leads to an extremely large search space even for modest problems. The complete exploration of this space may be impractical. To cope with this potential problem, the optimizer accepts (i) a limit on the number of plans developed and (ii) hints on the relative order of rules to apply. This enables controling the cost of optimization itself. A further quite standard approach to limit the cost of optimization consists of *caching* optimized plans to avoid re-doing the same effort.

1) Cost models for OptimAX: We consider cost models focused on communications (messages). Our simplest model $\mathcal{M}_1$ assigns a cost of 1 to each message crossing peer boundaries (i.e. whenever peer p activates a call to $s@p'$). All other computations, in particular local queries, are considered to have a cost of zero. This simple model reflects our experience that communications are by far more expensive than intra-peer computations [4], [8]. The second model we consider, $\mathcal{M}_2$, assigns a cost of bw_{p_1,p_2} to all messages exchanged between peers p_1 and p_2, where bw_{p_1,p_2} is a constant reflecting the transfer latency between these two peers. The model $\mathcal{M}_2$ takes into account the difference between fast transfers in an intranet, and more lengthy ones, e.g. between a peer in France and one in China.

OptimAX optimizes AXML computations by applying *successive rewriting steps* on its AXML plans. The initial plan is a document containing a call to an ad-hoc query service. This service is simply defined by the query which the user asked. The optimizer rewrites this document by applying a set of rules, as the following simple examples show.

2) Instantiation: Consider the plan $a(f@any)$, where a is an XML node and f a generic (query) service, at some peer p. Before evaluating this, any has to be changed into a peer providing f - for instance, but not necessarily, p itself.

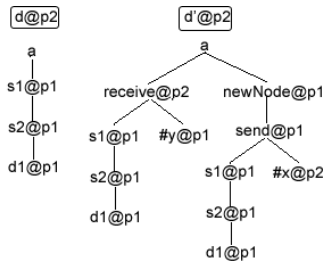

Fig. 2. Delegation example.

3) Delegation: Let the plan $d@p_2$ be: $a(s_1@p_1(s_2@p_1(d_1@p_1)))$. Here, the (result of the) call to $s_2@p_1$ is an argument of the call to $s_1@p_1$. The default evaluation strategy (Sec. II-A) would transfer $d_1@p_1$ to p_2, then ship it back to p_1 in order to evaluate the call to s_2, receive these results at p_2, then ship them back to p_1 in order to evaluate the call to s_1 and finally ship the results to p_2. Instead, the optimizer may rewrite $d@p_2$ into $d'@p_2$, shown in Figure 2. This plan represents the delegation of the whole computation to p_1. p_2 would only be responsible to receive the final result from p_1. More specifically the evaluation of d' at p_2 would result to calls to $receive@p_2$ and $newNode@p_1$. The evaluation of the latter would establish the $send@p_1(s_1@p_1(s_2@p_1(d_1@p_1)), \sharp x@p_2)$ at p_1's repository as a new tree. As soon as this tree is installed, its evaluation starts. Before $send@p1$ starts its execution, the subtree $s_1@p_1(s_2@p_1(d_1@p_1))$ has to be materialized (isAfterActivated pointing to the $s_1@p_1$ under the $send@p1$ is omitted in Figure 2 for readability). Note that after $send@p_1$'s activation, results of $s_1@p_1$ will be shipped as soon as they appear. The destination address of $send@p_1$ is the address of $receive@p_2$ (shown as $\#x@p_2$). The address of the send itself is $\#y@p_1$. Note that the execution of $s_1@p_1(s_2@p_1(d_1@p_1))$ at p_1 does not generate any communication cost, because document and services are hosted by the same peer.

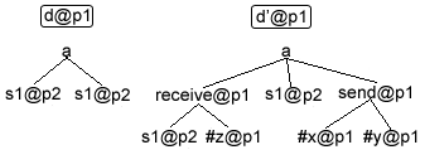

Fig. 3. Factorization example.

4) Factorization: Assume a document $d@p_1$ at peer $p1$ containing two identical service calls, e.g. two subscriptions made by different users (see Figure 3). Calling the service twice leads to an unnecessary overhead. The rewriting, d', is shown at right in (Figure 3). In d', only one call is made to $s_1@p_2$. As soon as the results arrive at p_1, $send$ and $receive$ services are used to copy them to the necessary positions. Observe that the *data* parameter of the *send* is an address,

namely $\sharp x@p_1$), identifying the service call *whose results should be copied as they arrive*.

5) Query pushing: Consider a document of the form $f@p_1(g@p_2(d@p_3))$ where f is defined by a selective query. Then, f can be "pushed" closer to $d@p_3$ without affecting the final result[1], leading to the rewritten plan $g@p_2(f@p_1(d@p_3))$.

How does a peer learns about available peers, documents and services, and cost information? In a "network of friends", peer p learns about other peer's existence gradually, as p calls services provided by these peers; in a DHT-based network, a global distributed index is available to all peers. In this second setting, peers also insert in the index cost (bandwidth) information which all other peers can use.

III. Demo highlights and scenario

First and foremost, our demo aims to illustrate the expressiveness of the AXML language both as a language for specifying mash-up applications, and as an algebra for distributing and executing them. Our second aim is to show OptimAX at work, following it via graphical rendition of the plans (a flavor of which can be found at the URL [9]).

A scenario we envision is closely inspired from a real application.[2] The scenario concerns the collaborative development and distribution of software packages. Several developers, distributed all over the world, write updates for a set of software packages. Each developer works at his own location, and pushes his updates to one or several servers geographically close to him. Each server hosts some, but not all, of the distribution's packages. The application is highly dynamic. The packages under development and the developer community change over time. The set of packages a given developed contributes to, and the sites where he publishes his updates, may also change, rendering impractical a centralized-catalog solution. In this context, a typical continuous query is: *Whenever there is a new update on the Emacs package, I want to receive the update and the name of its developer.* We intend to demonstrate this, and other distributed scenarios, both on a structured network (DHT) and on an unstructured one. If network connections is available, some peers will run in a cluster in INRIA, the others on the demo site. We will explain the optimizer's functioning via graphical representations of: the OptimAX plans, the evolution of the best plan's cost during optimization, and the relationships between plans in the search space considered. Figure 4 illustrates Optimax's graphical output.

IV. Related works and conclusion

The ideas behind OptimAX' algebra have been presented in [5]; they have since been developed, and implemented in our prototype. A concurrent project [10] has built a divide-and-conquer optimizer exploring two dimensions of the optimization space: using generic Web services, and choosing the peer which activates a given call as proposed in [8].

[1]Peers p_1, p_2, p_3 may or may not coincide.

[2]We encountered this application in the *eDos* EU project, concerning the automatic management of the Mandriva Linux distribution.

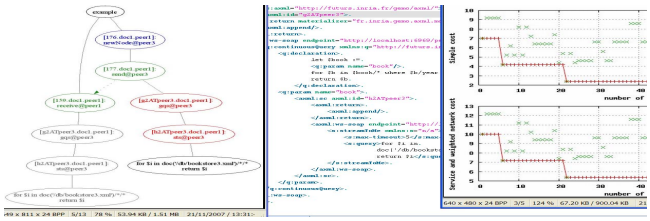

Fig. 4. Sample graphical output of the optimizer.

Our work considers more optimizations, such as lazy query evaluation [4], query pushing, factorization etc.

Many recent works seek to combine Web services, distribution, and XML queries. One area of such works consider Web service orchestration via BPEL4WS [11]. BPEL and AXML share some features, such as the possibility to sequence Web service calls, but have different focus. BPEL is more process-oriented, and in particular it allows handling complex process structure, exceptions etc. In contrast, AXML is data-oriented, sacrificing some expressive power in exchange for a language amenable to optimization. One can conceive an architecture combining BPEL and AXML, where the first one would be used for high-level process description and the latter one for data-targeted optimization of the proposed description.

Relevant related works have considered distributed extensions to XQuery [12], [13] using Web services. Going beyond queries, our approach allows a seamless transition between data, queries and query plans (which all are AXML). This is due to the dual character of AXML, mixing intensional and extensional data. Mutant query plans [14] are similar in spirit, although in a simpler relational setting.

Acknowledgments

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

References

[1] "Programmable web," www.programmableweb.com.
[2] R. Ennals and M. Garofalakis, "Datamator: Mash-up for the masses (demo)," in *SIGMOD*, 2007.
[3] "Active XML Primer," Gemo Report no. 307, 2003.
[4] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda, "Lazy query evaluation for Active XML." in *SIGMOD*, 2004.
[5] S. Abiteboul, I. Manolescu, and E. Taropa, "A framework for distributed XML data management." in *EDBT*, 2006.
[6] W3C, "XQuery: An XML Query Language 1.0."
[7] R. Milner, *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.
[8] N. Ruberg, G. Ruberg, and I. Manolescu, "Towards cost-based optimization for data-intensive web service computations." in *SBBD*, 2004.
[9] S. Zoupanos, www-rocq.inria.fr/~zoupanos/?page=axml.
[10] G. Ruberg and M. Mattoso, "XCraft: Boosting the performance of AXML materialization," Tech. report, UFRJ, 2007.
[11] "Web services business process execution language version 2.0," OASIS Consortium, 2007.
[12] M. Fernandez, T. Jim, K. Morton, U. N. Onose, and J. Simeon, "Highly distributed xquery with DXQ (demo)," in *SIGMOD*, 2007.
[13] Y. Zhang and P. Boncz, "XRPC: Interoperable and efficient distributed XQuery," in *VLDB*, 2007.
[14] V. Papadimos, D. Maier, and K. Tufte, "Distributed query processing and catalogs for p2p systems," in *CIDR*, 2003.