Towards cost-based optimization for data-intensive Web service computations*

Nicolaas Ruberg[1,2] Gabriela Ruberg[1,2] Ioana Manolescu[1]

[1]INRIA Futurs – LRI, Gemo group, PCRI, France

[2]UFRJ, Rio de Janeiro, Brazil

firstname.lastname@inria.fr

Abstract

The recent popularity of XML and Web services has lead to a surge in models and platforms for distributed XML data management applications. This work investigates performance issues involved in the deployment of the ActiveXML (AXML) platform for such applications. AXML documents are XML documents, part of which is extensional (present in the document), while part is intensional (specified as calls to Web services). Materializing an AXML document thus requires activating all service calls, and gathering the call results in the document.

In this work, we demonstrate that many distributed materialization strategies exist for a given AXML document; basically, these strategies may differ in the choice of the peer that executes each service call, or of the peer that makes the call. The AXML system has to choose among the strategies in order to generate efficient materialization plans. We formally characterize the optimization search space, and provide some heuristics to improve plan generation. We describe our optimization model and how it fits in the actual AXML P2P architecture. We describe the possible decision models; the parameters of this model, and the required infrastructure to provide them. Finally, we present empirical results that validate the proposed methodology.

1. Introduction

The increased popularity of XML and of Web services have opened up new possibilities for building large-scale, distributed XML data management applications. Several platforms based on XML an Web services have been proposed recently [3, 4, 22]. We consider the efficient deployment of such applications based on the ActiveXML platform [3]; this platform allows to declaratively specify XML data and Web service management applications, to be enacted over a set of distributed peers.

The contribution of this paper is to establish a framework for cost-based optimization of AXML computations. In this section, we introduce our motivating problem, and its AXML model description. Then, to illustrate, we present the query scenario for our problem and its optimization issues.

1.1. Motivating application: The PRONAF program

The Brazilian government runs the PRONAF (http://www.pronaf.org.br) loan program to rural families for farming activities. The credit regulation organism is the Central Bank of Brazil

*N. Ruberg is supported by BNDES, and G. Ruberg by CNPq and Central Bank of Brazil. The paper contents represent the viewpoint of the authors, and do not represent either the position of these instituitions. This work was also partially supported by the French government grant ACI MDP2P.

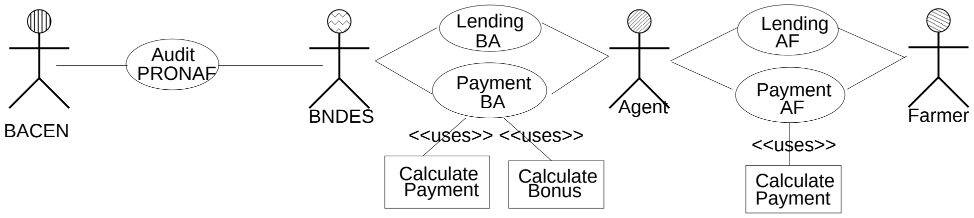

Figure 1: Outline of the PRONAF funding application.

(*BACEN*). The funding organism is the Brazilian Bank for Economic and Social Development (*BNDES*). Several *individual farmers* get loans in the PRONAF program. Since the farmers are spread over the whole contry, loans are provided to farmers through intermediary *credit agents* - typically, smaller regional banks.

Figure 1 summarizes the interactions among actors in PRONAF. The LendingBA activity is the loan made by BNDES to a local credit agent. In turn, each agent is involved in many LendingAF activities - that is, granting individual loans to farmers. Farmers make periodic payments to the local credit agent, including the partial reimbursement, and the interest for that period (PaymentAF activity). In turn, agents make periodical payments back to BNDES (PaymentBA activity). Note that BNDES keeps track only of its loans toward credit agents; it is the agents who manage the details of individual loans to farmers. Payment activities involve some computation to determine the amount to be reimbursed, depending on the loan type, on the BACEN interest rate, the payment date etc (rectangular boxes in Figure 1).

BACEN must be able to audict the whole lending process. However, BACEN is only interacting with BNDES, since only the BNDES knows who the lending agents are; and each agent may have delegated some of its loan program to others. These arrangements stem from normal banking practice.

Application requirements. The PRONAF example illustrates a class of P2P applications, from which we retain the following requirements:

1. Data exchanges, and computations, must be executed:
   (a) in an *efficient* manner
   (b) *over a network of distributed, autonomous peers*, namely: BNDES, a large set of banking agents, and BACEN.
2. *The number and identity of participating peers is a priori unknown*, as new agents join and leave PRONAF. However, the peers are relatively stable.[1]
3. The whole lending and reimbursement process must be *fully documented*. It must be possible to *audit the program by asking queries*, typically unknown in advance.
4. The workload of data manipulations in this application consists of *both read-only and read-write operations*.
5. An important part of the PRONAF activity (e.g. payments, bonus computation etc.) must take place on a *periodical basis* (e.g. every month).
6. As a general requirement, *ease of deployment and ease of use* of the application is an important plus for all actors.

To further clarify item 4 in this list, let us detail the data manipulations issued by each actor. Agents may ask how much of their BNDES-alotted credit is still available for them; they also

[1] By *relatively stable*, we mean that the peer set does not change e.g. over a period of a day.

```
<loanFile><loanNumber>110-334S-H1/2002</loanNumber>
  <rate> <sc>BACEN.getInterestRate(<day>14/5/04</day>)</sc></rate>
</loanFile>
```

```
<loanFile><loanNumber>110-334S-H1/2002</loanNumber>
  <rate> <sc>BACEN.getInterestRate(<day>14/5/04</day>)</sc><value>2.455 %</value>
  </rate>
</loanFile>
```

Figure 2: AXML document evolution through service call activation.

add new farmers, and update farmer credit files when payments are made. BNDES adds new agents, and wants to know how many payments were received on time. Finally, BACEN audits the PRONAF program using ad-hoc queries traversing the competences of all PRONAF peers involved.

Several classes of systems and architectures, such as mediator systems [9, 20, 13], partially fulfill some of the above requirements. However, they do not support intensive update operations, and they require all participants to be known in advance. In this paper, we investigate the efficient usage of the ActiveXML (AXML) [14, 2], a P2P platform, for applications such as PRONAF. Section 2 presents the AXML model, and introduces the need for cost-based optimization. Section 3, presents the space of evaluation strategies when materializing an AXML document, and ways to search this space for an optimal strategy. Section 4 provides a cost model for estimating the cost of an AXML computation; this cost model is an ingredient to the search strategies presented in Section 3.3. Section 5 presents the experimental validation of our cost model and optimization method. We present related work and perspectives in Section 6.

2. Problem statement: optimizing document materialization in AXML

2.1. The AXML model and architecture

The basis of the AXML model [3] consists of *ActiveXML* documents: these are regular XML documents, in which special <sc> elements are used to encode calls to *Web services*. When a service call is *activated*, its result is inserted in the document, as siblings of the <sc> node. For example, Figure 2 shows the transformation of an AXML fragment corresponding to a loan: it includes a call to a Web service provided by BACEN, asking for the interest rate for a given date. Notice the <value> element inserted in the document, as a consequence of the service call activation.

The actual syntax for <sc> elements provides all the information needed in order to make the call, such as service provider URI, port, service and operation names [23] etc; the details can be found at [3].

AXML peer architecture. The AXML model is employed in a distributed setting: a set of *AXML peers* own some (A)XML documents, and/or some Web services. An AXML peer (Figure 3 at right) includes: an XML document repository, an XML query engine, and a Web service messaging infrastructure based on SOAP [3]. The Web service execution engine performs the actuall calls included in the AXML documents. The AXML optimizer (double-lined box) is the new component that our work adds to this architecture: its task is to choose the most efficient strategy for activating the service calls included in the documents. This will be

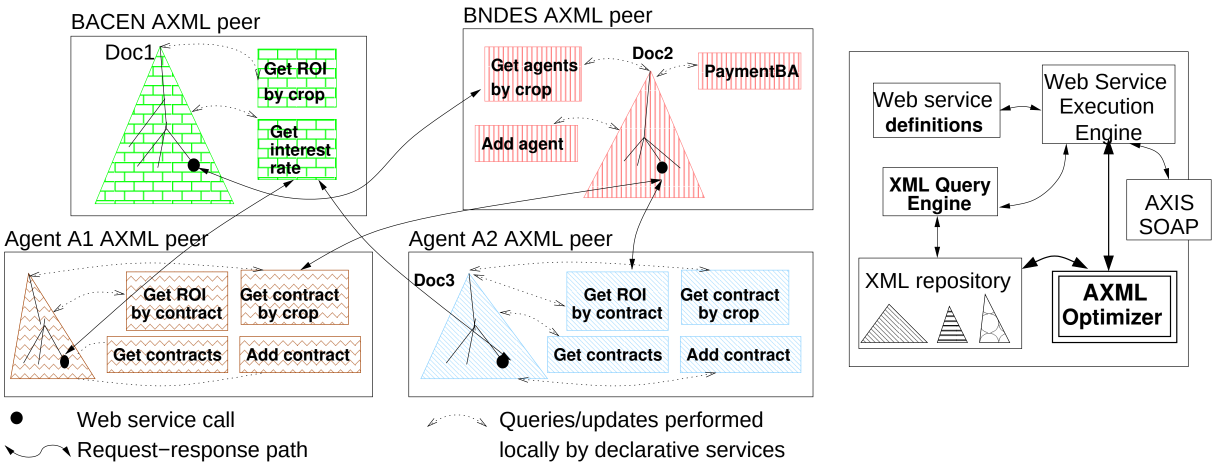

Figure 3: AXML peers for the PRONAF application (left); outline of an AXML peer (right).

described in details in Sections 3 and 4.

AXML Web services. An important AXML feature is the presence of *AXML Web services*: these are defined by *parameterized, declarative query or update statements* over the ActiveXML documents of a peer. Such services are exposed to other peers via the regular WSDL interface. When a service is called, its input message [23] encapsulates XML parameters, which are passed to the query implementing the service. In Figure 2, the element <day> is the parameter. AXML also allows to specify a special parameter value $\$user$, which allows to ask the user at run time to provide the value of a given parameter. Thus, AXML services allow to: *query XML data*, and *modify* it, either directly by adding service call results to the calling document, or indirectly, by activating an update service call.

Intensional data. AXML documents can be thought of as having two parts: an extensional one (plain XML present in the document), and an intensional one (data *potentially* present in the document, to be obtained by activating a service call). An AXML service call may have *intensional parameters*, or *intensional results*, e.g. the upper document in Figure 2 could be an intensional parameter for another service call. Activating a service call transforms some intensional data into extensional data (*materializes* it). Typical application queries (e.g. BACEN audit queries) require fully extensional (plain XML) results. However, *during query processing*, partial results can be exchanged *under their intensional form*. We will extensively build on this feature in the sequel (Section 3).

Generic query services. An AXML peer provides *a generic query Web service*, that is, a service whose parameters include a *declarative query*, and whose effect is to apply the query with the remaining parameters. For example, assume a "sum(/ROI//amount)" query is needed over the return on investment (*ROI*) of all agents. This results may be obtained by either of the following two calls:

BACEN.queryService(<**query**>sum(/ROI//amount)</**query**>,<**input**><ROI>...</ROI></**input**>),
or
BNDES.queryService(<**query**>sum(/ROI//amount)</**query**>,<**input**><ROI>...</ROI></**input**>)

More generally: *any query can be evaluated on any peer, by calling the generic query service provided by the peer*. This only requires that the query inputs be available on the peer.

Call activation parameters. AXML allows specifying various parameters for service call activation, among which: *when* to activate (e.g. at user's click, every hour, every week etc.) [3].

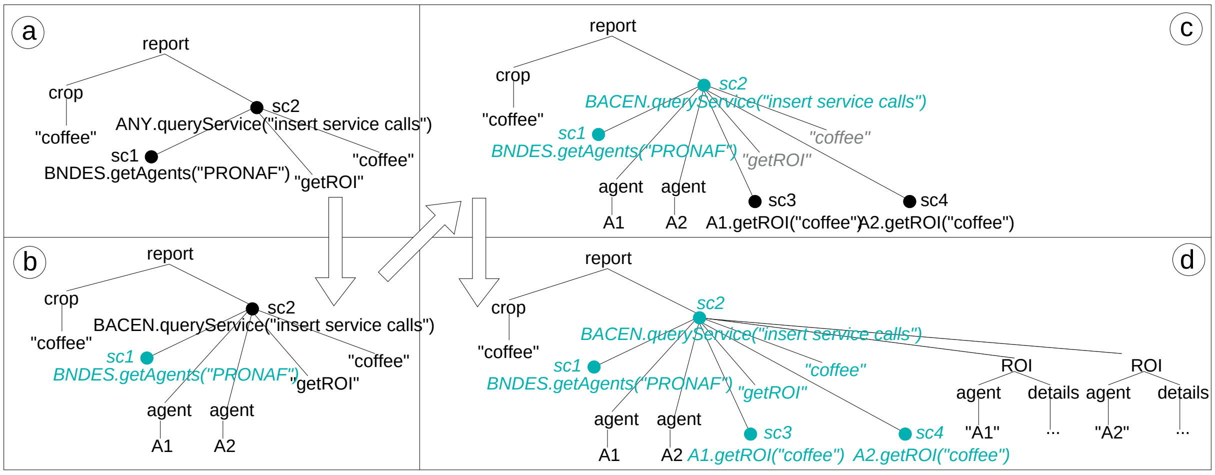

Figure 4: Sample AXML document for a BACEN query: overall return on investment for a specific crop.

2.2. Meeting the PRONAF application qualitative requirements with AXML

Figure 3 shows how the PRONAF application can be enacted with the AXML platform. AXML peers represent: BACEN, BNDES, and two bank agents A_1 and A_2. Triangular shapes denote AXML documents, black dots are <sc> elements. Rectangles denote Web services; service names describes their meaning, e.g. "Get ROI by crop" collects the ROI for loans used to grow a crop (e.g. maize).

Requirement 1b is naturally met by the distribution of AXML peers. Requirement 2 is also met: in order for a new agent to join, once he is running AXML, he has to call the BNDES.addAgent Web service, which will register him in BNDES's database. From that point on, BNDES will know that a query from BACEN, e.g. "Get ROI by crop", has to be transmitted also to the new agent. Requirement 3 is met, since on each AXML peer queries can be asked over local documents. Whenever queries cross <sc> nodes (intensional data), the materialization of this data is triggered; this, in turn, involves activating calls to services provided by other peers. Such services are queries over those peers' documents; recursively, evaluating these queries may trigger more service calls [2]. Since AXML allows reads and updates, requirements 4 is satisfied. The possibility to activate calls periodically fulfills requirement 5.

Finally, the greatest advantage of the AXML platform resides in its high-level, declarative aspect. To customize an AXML peer, one only needs to specify a set of AXML documents, and the queries defining the AXML services. AXML documents can be transformed into interactive user interfaces using stylesheets. These features make AXML a very good candidate for our last requirement.

The need for optimization. As is usual in declarative frameworks, a high-level specification such as the PRONAF AXML application must be compiled into an *execution strategy*, and if several strategies are possible, an *efficient* one should be chosen. We formalize this intuition in the next section.

3. Materialization strategies for AXML documents

Many possible strategies exist for materializing an AXML document, as shown in Section 3.1 (PRONAF example). When a peer has to materialize an AXML document, it is up to its optimizer to choose the best evaluation strategy. Section 3.2 formalizes this by describing the set of possible strategies; methods for searching this set are described in Section 3.3.

3.1. Alternative materialization strategies for the PRONAF example

Figure 4 depicts the successive stages of one possible AXML materialization strategy; the document is used by BACEN to obtain the ROI from all loans given for coffee crops. Black dots represent service calls; they become gray (and shown in italic) after the call activation. sc_1 obtains the agents of the PRONAF program, from BNDES. sc_2 is a call to the generic query service; the query transmitted to this service[2] outputs one service call per agent appearing in its input. The parameters of sc_2 are thus: the agent list; the name of service to be used in the new service calls ("getROI"); and the parameter of the new calls ("coffee").

Notice that sc_2 is a call to the generic query service of ANY peer: the user leaves to the system the choice of the specific peer to be used; in Figure 4, the choice is BACEN. After sc_1 returns agents A_1 and A_2, sc_2 inserts in the document two new calls to services provided by A_1 and A_2. Then, BACEN peer calls A_1 and A_2's services. In the final document, shown in Figure 4(d), all service calls have been activated, and the materialized XML result has been obtained (subtrees rooted in ROI elements).

The stages in Figure 4 correspond to a distributed *evaluation strategy coordinated by BACEN*: the call sc_2 to the generic query service is *executed* by BACEN, and all the remaining calls are *invoked* by BACEN. This strategy is represented in a different notation at left in Figure 5: numbers on the arrows depict the order of communications (sending service call requests with parameters; receiving service call results). But this is not the only possible evaluation strategy; alternatives are as follows.

First, the *execution* peer for calls to the generic query service may be chosen among those participating to the materialization. For example, at the center of Figure 5, the ANY of sc_2 is replaced by BNDES. This means that BNDES will: (i) locally call its getAgents("Pronaf") service; (ii) insert the results in a temporary document; (iii) insert in this document the service calls sc_3, sc_4 etc.; (iv) activate the calls to A_1 and A_2, obtain the results; and (v) ship the results to BACEN.

Second, the *invocation* peer may be chosen, in the case of calls to non-generic query service. For example, the call to BNDES.getAgents is activated from BACEN (Figure 4, and Figure 5 at left), or from BNDES (Figure 5 at right). As another example, at right in Figure 5(c) we depict a hybrid strategy: the execution of sc_2 is delegated to BNDES, while the invocation of the calls to agents is split among BACEN and BNDES. Such choices entail different execution costs, due to: different *query processing* costs of different execution peers, and different *data transfer* costs, depending on the parameters and results transferred between the execution and the invocation peers. In a nutshell: *different AXML materialization strategies lead to different costs, thus, cost-based optimization is needed to pick the best possible strategy*.

[2] Shown in natural language in Figure 4 for readability.

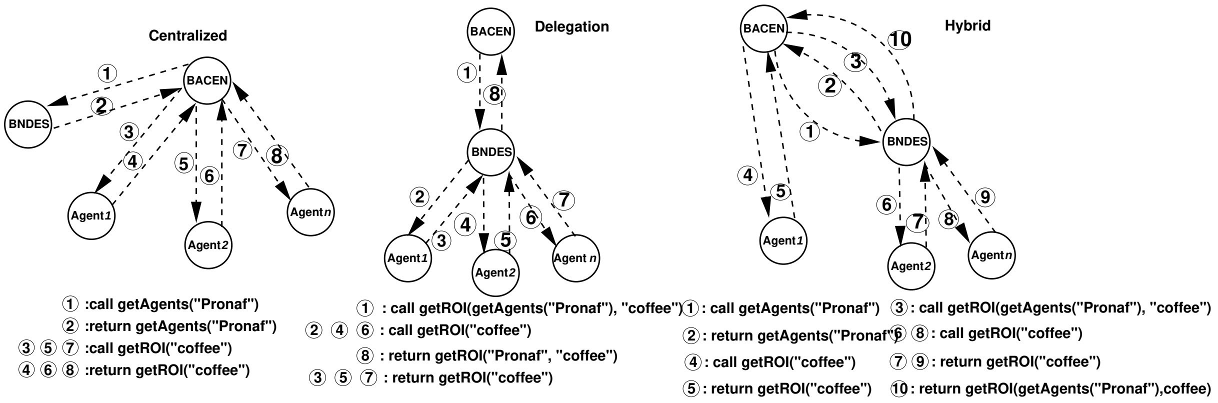

Figure 5: Alternative evaluation strategies for the example in Figure 4(a).

3.2. The space of possible materialization strategies

Notations. Let D be an ActiveXML document, residing at peer P_0, and containing the service calls sc_1, sc_2, ..., sc_n. For every i, let P_i be the peer providing the service called by sc_i. Let $SP(D)$ be the set of distinct peers in $P_1, P_2, \ldots P_n$.

Data and service dependencies. We say a *data dependency* between sc_i and sc_j exists, when the results of sc_i are needed as parameters for sc_j; for example, there is a data dependency from sc_1 to sc_2 in Figure 4(a). We say a *service dependency* between sc_i and sc_j exists, when some information about the service to which sc_j refers (e.g. the peer URI, the service name, the operation name etc.) is returned by sc_i. For example, in Figure 4, there is a service dependency from sc_2 to sc_3, and one from sc_2 to sc_4. A data or service dependency from sc_i to sc_j requires activating sc_i before sc_j.

Dynamic set of materialisation strategies. In the presence of service dependencies, the set of possible materialization strategies cannot be determined in advance, since some service calls (their peers, definition, parameters etc.) are not known. For example, in Figure 4(a), sc_3 and sc_4 are not in the document, so the materialization strategies known at this point are incomplete. Later on in Figure 4, they become known, and the space of strategies includes delegation and/or execution choices for such calls. Data dependencies are "easier", in the sense that execution strategies can be made before calling any service.

Thus, we now consider the space of materialization strategies for D *given a set of service calls, and dependencies among them*; we thus simplify the problem to a "snapshot" instance. Based on the service calls sc_1, sc_2, $\ldots$, sc_n, we define G, the *dependency graph*, as a directed graph with a node for each sc_i, and an edge from sc_i to sc_j when there is a data dependency from sc_i to sc_j. We assume G is acyclic, which corresponds to many useful cases (if G is cyclic, the AXML peer will break the cycles at some arbitrary point). We term the subgraph of G nodes that transitively reach sc_i the *subgraph of* sc_i. A sample dependency graph $\mathcal{G}$ is depicted in Figure 6(right); the subgraph of sc_1 is the tree of sc_1, sc_2 and sc_3.

Let $S(G, P_0)$ be the set of materialization strategies for all sc_is in G, such that the result arrives at P_0. The space of materialization strategies can be described by the enumeration strategy in Figure 6(left). Intuitively, materialization strategies are enumerated by making all possible choices of *execution* peers (lines 1-4 in the algorithm), and *invocation* peers (lines 5-8 in the algorithm). In Figure 4, ANY is replaced by BACEN, then sc_1, sc_2, sc_3 and sc_4 are

	Algorithm **ExhaustEnum**
1	for each node $sc_i \in G$
2	if sc_i calls the generic query service at ANY peer
3	then for each $P \in SP(D)$, instantiate ANY to P
5	for each node $sc_i \in G$
6	for each peer $P \in SP(D)$
7	delegate the subgraph rooted in sc_i to P: P must fully materialize it, and send it back to P_0.
8	replace G by: $G \setminus$ the subtree rooted in sc_i; goto 1

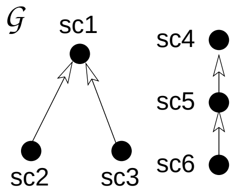

Figure 6: Search space enumeration of materialization strategies (left); dependency graph $\mathcal{G}$ (right).

invoked by BACEN. In the "Delegation" scenario in Figure 5, ANY is replaced by BNDES, then BNDES is the invocation peer for sc_1, sc_2, sc_3, and sc_4.

From the exhaustive strategy space enumeration, denoting the cardinality as $|\cdot|$, we derive:

$$|S(G, P_0)| \quad = \quad O(|G|! \times |SP(D)|^{|G|}) \tag{1}$$

3.3. Searching in the space of materialization strategies

A first naive optimization algorithm could be: enumerate all strategies, estimate their costs, and pick the strategy with the best estimated cost. While complete and optimal, this algorithm explores a potentially large search space. To overcome this problem, we introduce two search space pruning heuristics.

The "Divide and Conquer" heuristic. The dependency graph G may have disjoint components (each node in one component is unreachable from the nodes of the other component). For example, for the graph $\mathcal{G}$ in Figure 6, component $\mathcal{G}_1$ contains sc_1, sc_2, and sc_3, and component $\mathcal{G}_2$ consists of sc_4, sc_5, and sc_6. Each G_i naturally defines a smaller materialization problem. Our heuristic assumes that picking the optimal strategy for each component leads to the optimal strategy for the problem determined by G; thus, the problem is decomposed into several smaller ones. In our example, $|\mathcal{G}| = 6$, $|\mathcal{G}_1| = 3$, $|\mathcal{G}_2| = 3$, and if $|SP| = 4$, this reduces the complexity from $O(6! * 4^6)$ to $O(2 * 3! * 4^3)$, a reduction factor of almost 4.000. The heuristic can be wrong, if the processing of two subproblems interfere (e.g. if they use the same peer at the same time).

The "Context" heuristic. The ExhaustEnum algorithm considers in lines 2 and 6 all peers mentioned in D. The heuristic consists of limiting this choice of peers, as follows. We define the *context* of a service call $sc_i \in G$ to be the set of peers that provide the services called by G nodes:

- reachable from sc_i in G: these peers are potential "consumers" of sc_i's results
- that can reach sc_i in G: these peers are potential "producers" of data consumed by sc_i.

Intuitively, the "Context" heuristic attempts to avoid useless communications. For example, consider the "Hybrid" strategy in Figure 5. When choosing an invocation peer for the calls to Agent_2 and Agent_n, the "Context" heuristic dictates that the peer Agent_1 is not an option, since it is unrelated to the call. In contrast, with the naive exhaustive strategy, the Agent_1 peer would also be considered. To compute the size of the space explored by the "Context" heuristic, the multiplication factor $|G|$ is replaced by the context size for each sc_i; the larger (and shallower) the graph, the bigger reduction to the search space. This heuristic can be suboptimal, if one of the peers that has been avoided is so fast (or so well connected to the others) that this benefit outweighs the cost of shipping data to or from the peer.

The two heuristics presented have different scopes, and thus can apply simultaneously.

4. A cost model for data-intensive Web service computations

We now present our proposed cost model for the scenarios described in Section 3. The metric we consider is *response time*, since this is typically the parameter with the greatest perceived impact on the user [6, 18]. We study the processing costs related to the activation of a service call, in Section 4.1, analyze the cost components and provide basic cost formulas. In Section 4.2, we describe the AXML approach for capturing the values of some peer- and network-dependent cost parameters.

4.1. Costs involved in making one service call

Invoking a service call consumes resources at both service client and provider, thus predicting its performance requires some cooperation between peers on the publication of basic costs information. Consider peer P_i activates the service call sc, which is provided by peer P_j. From the *client viewpoint*, the following steps are required to activate sc:

1. initializing P_i modules that are responsible for activating sc;
2. packing sc's parameters into the Web service input message;
3. sending the call message to P_j;
4. waiting for P_j to process its request and send the result back; and
5. unpacking, parsing, and inserting the result in the document P_i.

Therefore, we identify the respective cost components (in time units) of sc as:

$$\begin{aligned} ccost_{P_i}(sc) \;=\;& init_c + callSize \times pack_u + (callSize + envSize) \times net(1, P_i, P_j) \\ & + scost_{P_j}(sc) + resSize \\ & \times (unpack_u + parse(1, resSize) + merge(resSize, docSize(sc))) \qquad (2) \end{aligned}$$

Factors in Equation 2 are ordered according to activation steps; hence, $init_c$ is the time spent in step 1. Cost ingredients are explained in Figure 7. Notice that we charge communication costs from senders. The $merge$ function depends on the system platform where sc is processed; the *parsing* step is required to identify new service calls in the call result. The performance of activating sc also depends on P_i and P_j workload; to model this, we apply to $ccost_{P_i}(sc)$ an *inflating factor*, depending both on the peer capability (in terms of CPU and memory size), and on its current number of active tasks.

Equation 2 performs a cost analysis at the level of the SOAP messaging protocol (we do not decompose costs underlying the SOAP transfer infrastructure, but rather focus on the interaction between Web services and other costs). The two main components in Equation 2 are: the size of the service call (mainly the size of its parameters), and the size of the result.

The $scost(sc)$ component comprises both the costs of service provider processing, and network transfers to return the result to P_i. On server side, we decompose the costs for handling

$init_c$ and $init_a$	Time to initialize software modules at, respectively, service client and provider.
$callSize$ and $resSize$	Size (in bytes) of, respectively, the service call and its result.
$net(b,P_1,P_2)$	Average time to transfer b bytes from P_i to P_j.
$envSize$	Average size (in bytes) of the SOAP envelop.
$pack_u$ and $unpack_u$	Average time of, respectively, SOAP packing and unpacking of 1 byte of data.
$docSize(sc)$	Size (in bytes) of the document containing service call sc.
$parse(b, size)$	Average time to parse b bytes of an AXML data fragment with $size$ bytes.
$merge(b, size)$	Average time to merge b bytes of data into an AXML document with $size$ bytes.
$scost(sc)$	Time spent between the SOAP message containing sc is received by the service provider, and sc result is received at the client.
$srvExec$	Time of the service call execution on the provider.

Figure 7: Cost model ingredients.

sc as follows:

$$\begin{aligned} scost_{P_j}(sc) \quad = \quad & init_a + callSize \times (unpack_u + parse(1, callSize)) + srvExec + \\ & resSize \times pack_u + (resSize + envSize) \times net(1, P_j, P_i) \end{aligned} \tag{3}$$

Cost components are defined similarly as for $ccost_{P_i}(sc)$ in terms of the provider performance. We remark that this formula identifies some relevant information that the service provider must publish in order to allow clients to estimate the costs of calls to its services. Among the above cost components, notice that the coefficients $pack$ and $unpack$ play an important role since they are related to two critical variables in Equation 3, representing the sizes of the service call parameters and result.

4.2. Calibrating the cost model of an AXML peer

In the AXML architecture, a centralized costs catalog is unfeasible, both because of peer autonomy, and for scalibility and performance reasons. Nevertheless, if AXML peers are to collaborate efficiently, each peer must have some information about its own and other peers' costs, in order to choose efficient evaluation strategies. To this purpose, we rely on *on-demand cost propagation* among peers. Namely, in the described AXML scenarios, whenever peer P_i must estimate (by Equations 2 or 3) a cost depending on the behavior of peer P_j, P_i may call *cost information Web services* provided by P_j in order to learn its required cost ingredients. For example, a *processing cost Web service* on P_j takes as argument the name of a service provided by P_j and returns the average server-side execution time of this service. Over time, cost information from P_j can be cached at P_i, thus avoiding subsequent calls to P_j's cost service.

We now explain how a peer learns its own cost parameters. Once known, the parameters are written in a specific *cost sheet AXML document*; then, *cost information Web services* are defined as declarative services (implemented by queries over the cost sheet) to be called by other peers.

Estimating message sizes. Equations 2 and 3 include terms accounting for call information (e.g., service and operation name), parameters and fixed-size envelope for the call, result and envelope for the response. The call information size is usually small, and easy to derive.

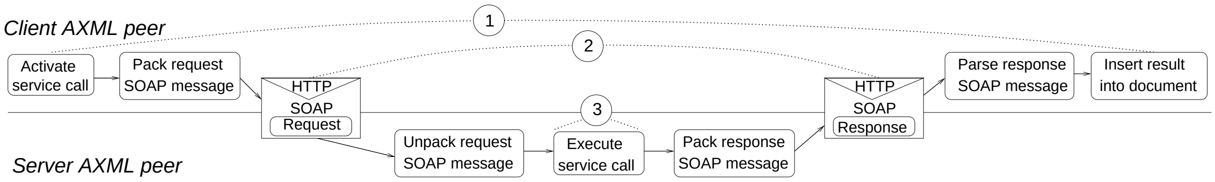

Figure 8: Operation details when activating a service call.

Estimating parameters size is more tricky, and it depends on the way they were specified: (i) parameters can be hard-coded in advance in the document, or known at the time of the estimation, thus their size can be measured; or (ii) parameters may be given as queries over the local document (e.g., in Figure 2 we might have used a path expression as a parameter to getInterestRate), hence their size can be deduced from query selectivity estimates.

The size of a parameter can only be estimated once we decided which service calls (included in the parameter) to activate *before* the parameter is sent, and after these calls have all returned their results. On the other hand, service result sizes are obtained by the service provider either by query selectivity estimation, or by monitoring executed calls, and writing the result into the cost sheet. Last but not least, messaging costs estimates often varies according to network load. To reflect this, the $net()$ function can use a historical log to predict traffic conditions.

Estimating service execution times. Let us first consider the case when services are *declarative XML queries*. If the XML repository provides some response time estimates, they can be gathered by the AXML peer; this requires a tight integration of the peer with the XML repository, but does not favor interoperability, as there is no standardized interface for obtaining such estimates from an XML repository.

Instead, our current AXML peer implementation considers the underlying XML repository as a black-box, and *infers* rough cost formulas describing its performance. This is achieved by including in each AXML peer distribution: (i) a set of *calibrating (A)XML documents* of increasing, pre-defined sizes, all conforming to a known DTD (e.g., XMark [1]); (ii) a set of *calibrating services* defined as queries over these documents, such as a "point" (simple selection) query, and a "join" (value-based join on two different paths) query; and (iii) a *calibrating script AXML document* including calls to the above services, to be evaluated over each calibrating document.

When the peer is first launched, the calibrating documents and script are loaded into the repository, and the calibrating services are registered in the service repository. Next, since the script contains service calls, the AXML peer will trigger them one by one, and measure the query engine's response time. For different document sizes, each calibrating service will give a different response time. From these times, in conjunction with the known document size, the peer *empirically derives* a rough general cost formula, as we will detail in Section 5. Such a rough formula can then be applied to estimate the response time of any query, based on the size of the input and the query syntax.

5. Experimental validation

In this section, we report on a series of experiments in order to establish the elementary cost components (outlined in Figure 8) of the activation of a service call. On the upper side, the operations are performed by the client peer, while on the bottom side, they are performed by

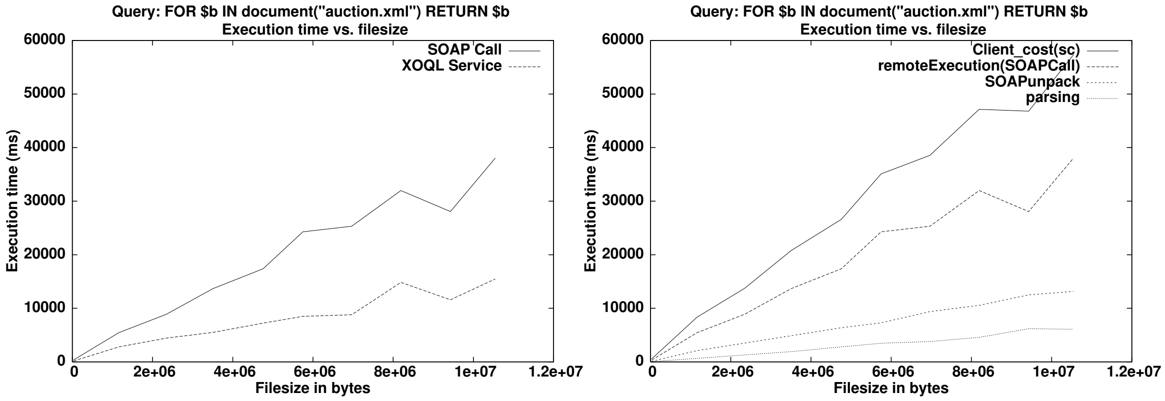

Figure 9: At left, Q_0 performance: query execution only ("XOQL Service"), and SOAP overhead ("SOAP Call"). At right, decomposition of Q_0 costs, on XML documents of increasing size.

the server.

We measured the times over the operation spans corresponding to the circled numbers, corresponding to $ccost$ and $scost$ components (Equations 2 and 3). On the client side, we measure: the overall time it takes to complete the call. In (1), we measure the total time it takes to create/activate the AXML internal objects responsible of: composing the outgoing message; and making a SOAP call. In point (2), within the Axis proxy, we measure just the time needed to send the request message, execute it in the server peer, and get the result back. On the server side, in (3) we measure query execution time of our in-house XOQL XML query processor, which is $servExec$ in Equation 3.

Experiment Setting. We ran our measures on a computer under Linux, having 1Gb of memory, a 100 MHz bus speed, and scoring 1202.58 BogoMips (BMs) as CPU power. We ran an ActiveXML peer 0.4.0, using J2SE 1.4.2, and Axis 1.1. As calibrating documents, we used a set of the XMark [1] documents of increasing sizes (from 26 Kb to 116.491 Mb). We used three queries: Q_0 returns all the XMark document; $XMark_1$ and $XMark_8$ are XMark benchmark queries. $XMark_0$ is a "point" query returning the name of a single person; $XMark_8$ joins persons with items they bought.

Influence of SOAP messaging. We first establish the relative importance of the $pack$ and $unpack$ times, since SOAP is typically used in Web service architectures [4]. Figure 9 (left) depicts the execution time of a declarative service defined as Q_0, over various XMark documents. In this experiment, the server and client peers coincide. The lower curve represents the time in (3), and the top curve the time in (2), in Figure 8. The difference between the two is due to the time to pack the calibrating document in the response message (the request message is negligible). The $pack$ time is linear in the document size, and quite important (comparable to the query evaluation time; Q_0 requires traversing all the document !).

XOQL cost estimation. Our XOQL query processor (as well as other in-memory ones) evaluates queries through nested loops, traversing the document as many times as there are *independent* paths in the query, starting from the root. Thus, $XMark_1$'s cost grows linearly, $XMark_8$'s grows quadratically etc. The cost of an XOQL query is estimated as $scost(q) = s^N \times ct_q$, where s is the document size, N is the number of independent paths in q, and ct_q is a constant characterizing the system and query processor.

Client-side cost decomposition. In this experiment, we detail the times measured in points

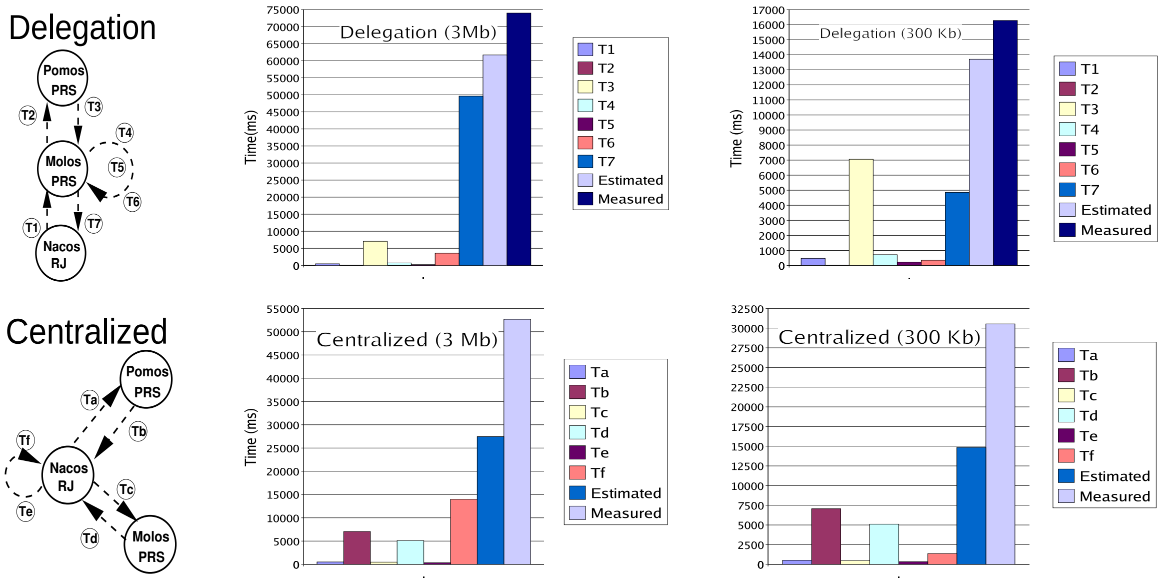

Figure 10: Materialization strategies for 3Mb and 300Kb documents.

(1), (2), and (3) (Figure 8), again when the client and server peers coincide. The results are shown at right in Figure 9; the request unpacking time is negligible, and omitted. The time for $unpack$-ing the query result (the document) is quite important; this time is specific to the Axis SOAP implementation.

Experiments 1, 2, 3 and 4 are performed with the help of the calibrating document (Section 4.2).

Distributed materialization strategies. In this experiment, we validated the usefulness of our cost model for choosing optimal evaluation strategies. Two peers located in Paris (Molos/5570.56 BMs, and Pomos/1202.58 BMs) are connected by a 100Mbs link; a third peer (Nacos/3060.53 BMs) is in Rio de Janeiro, at 236ms of average "*ping*" distance from the others. Nacos must materialize the result of a call to a service joining two XML documents: Users (240Kb, at Pomos), and Auctions (300Kb, at Molos). The options are: delegating to Molos the call to Pomos (getting the Users data), or invoking all services itself. We varied the join selectivity, leading to result sizes of 300Kb, respectively, 3Mb. The two alternatives, and the resulting estimated and measured costs, are shown in Figure 10.

Basic cost ingredients were collected by a calibrating document, and by applying cost formulas of Equations 2 and 3. Depending on factors such as query cardinalities, performance varies significantly; hence, the need for some mechanism to guide the choice of an execution strategy. Our proposed cost model correctly guides this choice (Figure 10). Furthermore, even when network communication is expensive, for relatively large files (over 1Mb), packing and unpacking costs are significant.

6. Related work and final considerations

Statistics and cost information have long been crucial for the efficiency of distributed query processing [20, 19]. In most cost-based optimization techniques for distributed data management systems (such as multidatabases [24], federated systems [20] and mediated query systems [9]), query execution costs are mainly represented by data transfers, but the impact of the communication protocol is usually neglected. On the other hand, for data-intensive Web services, the standard SOAP protocol represents a significant performance overhead. Furthermore, XML data exchange often involves some data parsing steps that may become very expensive. We introduce basic cost functions that model these characteristics, and we analyze the components of these functions in several experimental scenarios.

Although data-intensive Web services are closely related to distributed data management systems, there are some relevant differences on their optimization problems. First, query optimizers for distributed systems usually produce execution plans based on a set of algebraic operators, and they can profit from some algebraic optimization properties. In our setting, *service calls* (which can be seen as a sort of pre-defined queries) with heterogeneous performance replace such operators; optimizing them is similar to ordering expensive predicates or user-defined functions [13], with the additional complexity of the many-peers decentralized setting. Furthermore, conversely to multidatabase systems, P2P systems [11, 22, 16, 12] cannot rely on a centralized cost catalog. In previous work [21], we presented a distributed architecture to gather and publish costs and statistics from diverse data sources. In this paper, we propose a calibrating mechanism to enable peers to collect and exchange performance coefficients required by cost functions, considering that peers cannot rely on special software modules for that purpose. Cost model calibration was introduced in [10], which required the DBA to manually run calibrating queries. We exploit the AXML model to do it through a calibration script, without requiring user intervention.

A crucial issue in our setting is that execution plans cannot be statically built since their service calls are not completely known in advance, due to service dependencies. Thus, some operators may be added to the execution plan at runtime. This is really different from traditional optimization problems [7], where only cost information is obtained at runtime [5]. Our optimization approach consists of: (i) detecting dependencies between service calls, and exploiting them dynamically during optimization; and (ii) using a "consumer/producer" model to determine the context of service calls in each optimization phase. We formalize the problem for AXML documents, and we present heuristics to tackle the search space size.

Finally, many techniques to estimate our cost ingredients, such as query selectivity [8, 17], and network coefficients [6, 18], have been proposed. An interesting category of cost ingredients in P2P systems consists of qualitative performance criteria (for example, *site reliability*) [15].

We plan to pursue this work by considering more advanced query processing techniques such as query composition [2]; furthermore, we are currently extending AXML with a dynamic hash table lower layer, allowing to locate documents in the network of peers.

References

[1] The XMark benchmark. Available at http://monetdb.cwi.nl/xml/.

[2] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Query evaluation over ActiveXML documents with lazy service calls. ACM SIGMOD, 2004.

[3] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-peer data and Web services integration. In *VLDB*, 2002.

[4] S. Amer-Yahia and Y. Kotidis. A Web-service architecture for efficient XML data exchange. In *ICDE*, 2004.

[5] L. Bouganim, F. Fabret, C. Mohan, and P. Valduriez. Dynamic query scheduling in data integration systems. In *ICDE*, 2000.

[6] L. Cherkasova, Y. Fu, W. Tang, and A. Vahdat. Measuring and characterizing end-to-end internet service performance. In *ACM Transactions on Internet Technology*, volume 3, 2003.

[7] R. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *ACM SIGMOD*, 1994.

[8] J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: Making XML count. In *ACM SIGMOD*, 2002.

[9] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8:117–132, 1997.

[10] G. Gardarin, F. Sha, and Z. Tang. Calibrating the query optimizer cost model of IRO-DB, an object-oriented federated database system. In *VLDB*, 1996.

[11] F. Goasdoue and M-C. Rousset. Answering queries using views: a KRDB perspective for the Semantic Web. *ACM TOIT*, 2003.

[12] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *VLDB*, 2003.

[13] T. Mayr and P. Seshadri. Client-site query extensions. In *ACM SIGMOD*, 1999.

[14] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. Dang Ngoc. Exchanging intensional XML data. In *ACM SIGMOD*, 2003.

[15] W. Ng, Y. Shu, and B. Ling. Fuzzy cost modeling for peer-to-peer systems. In *2nd Workshop on Agents and P2P Computing (AP2PC)*, 1994.

[16] W. Siong Ng, B. Chin Ooi, K-L. Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *ICDE*, 2003.

[17] N. Polyzotis and M. Garofalakis. Structure and value synopses for XML data graphs. In *VLDB*, 2002.

[18] R. Rajamony and M. Elnozahy. Measuring client-perceived response times on the WWW. In *"USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[19] F. Reiss and T. Kanungo. A characterization of the sensitivity of query optimization to storage access cost parameters. In *ACM SIGMOD*, 2003.

[20] M. Roth, F. Ozcan, and L. Haas. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *VLDB*, 1999.

[21] N. Ruberg, G. Ruberg, and M. Mattoso. Digging database statistics and costs parameters for distributed query processing. In *CoopIS/DOA/ODBASE*, 2003.

[22] I. Tatarinov and A. Halevy. Efficient query reformulation in peer-data management systems. In *ACM SIGMOD*, 2004.

[23] Web Services Definition Language (WSDL). `http://www.w3.org/TR/wsdl`.

[24] Q. Zhu and P. Larson. Global query processing and optimization in CORDS multidatabase system. In *PDCS*, 1996.