INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Towards Cost-based Optimization for Data-intensive Web Service Computations

Nicolaas Ruberg — Gabriela Ruberg — Ioana Manolescu

N° 5222

June 2004

Thème COM

Rapport technique

ISSN 0249-0803

Towards Cost-based Optimization for Data-intensive Web Service Computations

Nicolaas Ruberg* , Gabriela Ruberg † , Ioana Manolescu

Thème COM — Systèmes communicants
Projet Gemo

Rapport technique n° 5222 — June 2004 — 33 pages

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; these strategies may differ in the choice of the peer that executes each service call, or of the peer that makes the call. Such strategies have different execution costs; thus, a cost-based optimization is necessary to choose the optimal strategy. We present a comprehensive cost model characterizing each strategy; the parameters of this model are automatically inferred, using "calibrating" AXML documents. We formally describe the search space, and provide some heuristics to accelerate the search. Our preliminary performance measures validate the interest and the correctness of our optimization strategy.

Key-words: Databases, cost models, query processing, query optimization, semi-structured databases, XML, Web services

The paper contents represent the viewpoint of the authors, and do not represent either the position of the Central Bank of Brazil or the Brazilan Bank for Economic and Social Development - BNDES.

* N. Ruberg is supported by BNDES
† G. Ruberg by CNPq and Central Bank of Brazil

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 72 92 59 ??

Vers une optimisation à base de coûts pour l'évaluation des services Web orientés sur les données

Résumé : La récente popularité du format XML et des services Web a conduit au développement de plusieurs modèles et plateformes d'application distribuées pour la gestion de données XML. Ce travail étudie les problèmes de performance impliquès dans le déploiement de la plateforme ActiveXML (AXML) pour ce type d'applications. Un document AXML est un document XML, dont une partie est extensionelle (présente dans le document), tandis qu'une autre partie est intensionelle (spécifiée par des appels à des services Web). La *matérialisation* d'un document AXML suppose l'activation de tous les appels de services contenus dans le document, et l'ajout des résultats de ces appels dans le document.

Dans ce travail, nous démontrons que plusieurs stratégies de matérialisation peuvent être employées pour un certain document AXML; ces stratégies peuvent être différentes dans les choix du pair qui exécute chaque appel de service, ou du pair que fait l'appel. Ces stratégies ont différents coûts d'exécution; ainsi, une optimisation à base de coût est nécessaire pour choisir la stratégie optimale. Nous présentons un modèle caractérisant chaque stratégie: des paramètres de ce modelé sont automatiquement inférés, en utilisant un document de calibration. Nous décrivons formellement l'espace des stratégies possibles, et nous fournissons quelques heuristiques pour accélérer la recherche. Nos mesures de performance préliminaires valident l'intérêt et l'exactitude de notre stratégie de optimisation.

Mots-clés : base de données, modèle de coût, traitement de requête, optimisation de requête, base de données semi-struturés, XML, services Web

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, 25]. 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.

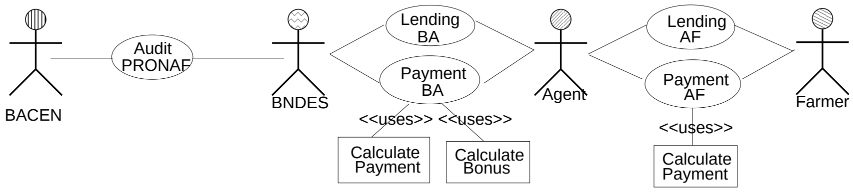

Figure 1: Outline of the PRONAF funding application.

1.1 Sample 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 (*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).

Finally, it must be possible for BACEN 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. From the PRONAF example, 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 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 fulfill some of the above requirements. Requirements 1 and 3 may be provided by mediator systems such as [11, 22, 15]. However, such systems most often do not support intensive update operations (requirement 4), and require all participants to be known (contrast with requirement 2). Existing peer-to-peer systems allow to query an a-priori unknown set of peers [13, 25, 18, 14]; however, cost-based optimizations (requirement 1) are not targeted by these works. Finally, distributed workflow platforms satisfy requirements 3, 4 and 5; however, their installation, maintenance, and optimization is quite involved, thus they provide limited support for requirement 6.

In this paper, we investigate the efficient usage of the ActiveXML (AXML) [16, 2], a P2P platform, for applications such as PRONAF. Section 2 presents the AXML model, and

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

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.

```
<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.

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 [27] 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 described in details in Sections 3 and 4.

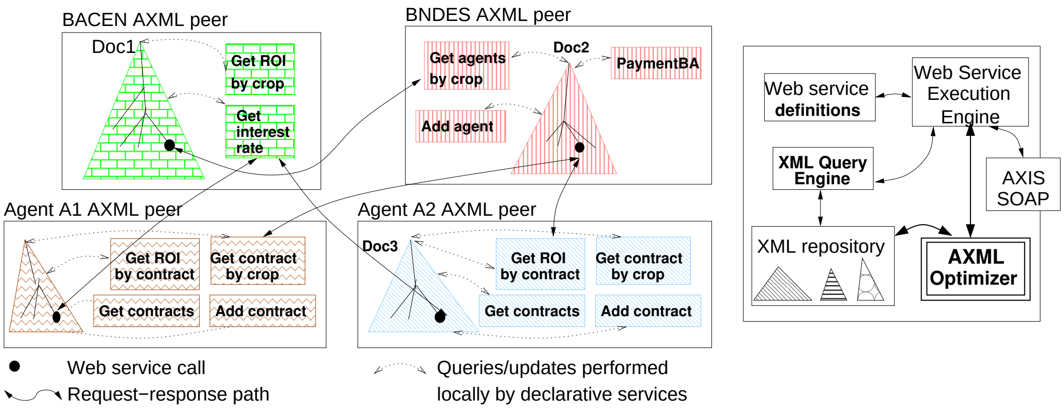

Figure 3: AXML peers for the PRONAF application (left); outline of an AXML peer (right).

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 [27] 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].

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. Service call parameters are specified as XML attributes of the <sc> elements: no programming is needed. At the user's level, AXML documents can be transformed into interactive 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.

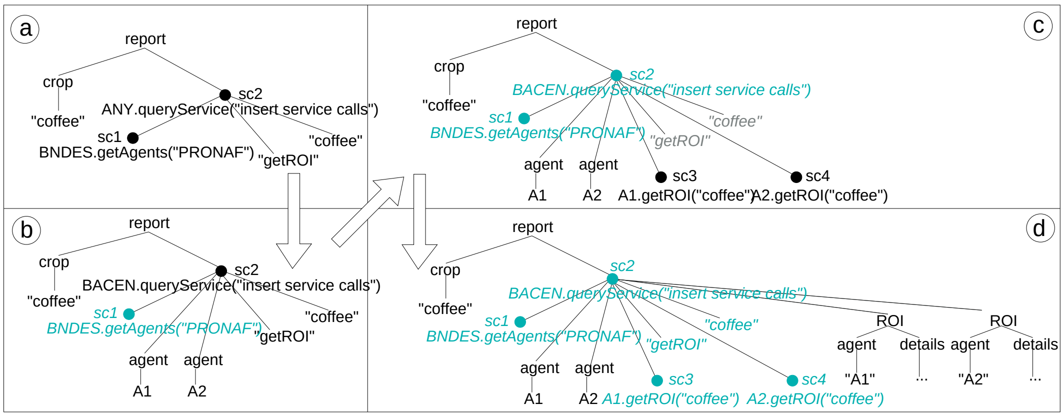

Figure 4: Sample AXML document for a BACEN query: overall return on investment for a specific crop.

Optimization and constraints Besides performance reasons, there are many types of considerations that may impact the choice of a materialization strategy. First, type restrictions on a service's input and output may require that a given peer activates a given service call [16]. Second, security reasons may also pose such restrictions. Third, special calls to *no-impact* services may be used: such services (e.g. compression/decompression, encryption/decryption) do not affect data semantics, but may improve execution performance, or may simply make it possible, in the case of encryption. In such cases, in order for the performance improvement to be attained, natural constraints arise on the execution sites of various service calls: compression must take place at the sender, and decompression must take place at the receiver.

Thus, the work presented in this paper targets the optimal strategy *possible given the existing constraints*. We plan to investigate more deeply the interaction between optimization and constraints, in the future.

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 strategy; the document considered is used by BACEN to obtain the return on investment (ROI) from all loans given for coffee crops. Black dots represent service calls; they become gray (and their details appear in italic) after the service call activation. In Figure 4(a), sc_1 obtains the agents of the PRONAF program, from BNDES; in our example shown in Figure 4(b), the agents A_1 and A_2 are returned. The following service call, 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; two different opportunities for choice exist.

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.

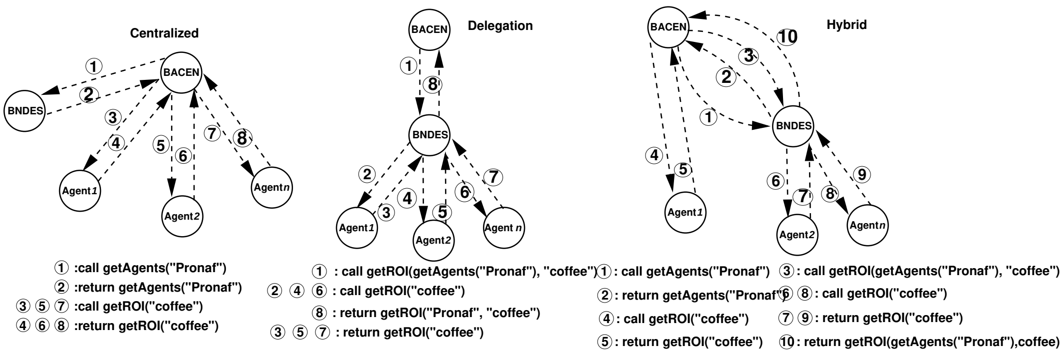

Figure 5: Alternative evaluation strategies for the example in Figure 4(a).

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. Their activation can be written, abusing slightly the notation, as $sc_j(sc_i)$; in the case of a data dependency, $sc_j(sc_i)$ can be delegated to the peer P_j (he has to call sc_i and then apply sc_j). This is not the case for a service dependency, since P_j is unknown (it will be obtained only after calling sc_i).

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.

```
   Algorithm ExhaustEnum
1  for each node sc_i ∈ G
2        if sc_i calls the generic query service at ANY peer
3        then for each P ∈ SP(D)
4                  instantiate ANY to P
5                  for each node sc_i ∈ G
6                        for each peer P ∈ 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 \ the subtree rooted in sc_i
9                        endfor
10                 endfor
11             endfor
```

Figure 6: Search space enumeration of materialization strategies.

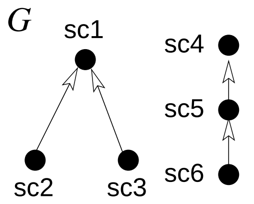

Figure 7: Sample service call dependency graph $\mathcal{G}$.

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

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

$$|S(D, P_0)| = O(|G|! * |SP(D)|^{|G|})$$

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 alleviate 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 7, 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.

Finally, we note that the two heuristics presented, "Divide and Conquer" and "Context" heuristics, 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 [7, 20]. 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:

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

Factors in Equation 1 are ordered according to activation steps; hence, $init_c$ is the time spent in step 1. Cost ingredients are explained in Figure 8. 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 1 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 1 are: the size of the service call (mainly the size of its parameters), and the size of the result.

$init_c$ 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 8: Cost model ingredients.

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 sc as follows:

$$\begin{aligned} scost_{P_j}(sc) \;=\;& init_a + callSize \times (unpack_u + parse(1, callSize)) + srvExec + \\ & resSize \times pack_u + (resSize + envSize) \times net(1, P_j, P_i) \end{aligned} \tag{2}$$

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 2, 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 solve the latter problem, we rely on *on-demand cost propagation* among peers. Namely, in the described AXML scenarios, whenever peer P_i must estimate (by Equations 1 or 2) 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 1 and 2 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. 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

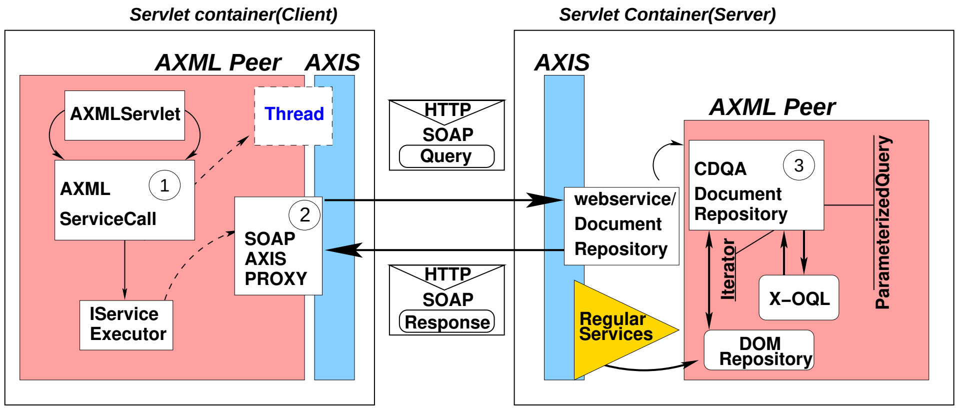

Figure 9: Operation details when activating a service call.

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.

Finally, for a non-declarative services, the execution times are simply monitored, and the result is noted in a cost sheet.

5 Experimental validation

In this section, we report on a series of experiments made with the AXML peer implementation. Our experiments target two aspects. First, we performed a series of measures in order to verify our cost decomposition formulas; we report these measures in Section 5.1. Second, we show how, based on these measures, our optimizer makes correct predictions in a three-peer materialization scenario (Section 5.2). For the later, we present the calibrating document for the participating peers, and the implementation of the Web services that we use to acquire each particular cost. Section 5.3 presents our conclusions.

5.1 Intra-peer cost decomposition

Before presenting our measures, we provide a high-level view of an AXML peer architecture, as sketched in Figure 9. This figure depicts two AXML peers, playing the roles of client, respective server; this distinction is made only with respect to a given service call, as the roles may be swapped at any time. Each peer consists of: an *XML repository*, i.e. persistent store of (A)XML documents; an *XML query engine* that is used for implementing declarative services and evaluating users' queries over the AXML documents; a set of *declarative Web service definitions*; and a *Web service execution engine*, which evaluates the calls included in

Size factor	File size (b)	Size factor	File size (b)
0.00	26703	0.06	6964970
0.01	1161397	0.07	8193163
0.02	2345041	0.08	9430595
0.03	3503730	0.09	10546923
0.04	4756200	1.00	116491402
0.05	5749464		

Table 1: XMark calibrating documents used in the experiments.

the peer's AXML documents, and calls (received from a distant peer) to this peer's services. Inter-peer communication takes place through Web service messaging. An AXML peer includes the popular AXIS [5] SOAP engine to that purpose.

For the purpose of this section's measures, we captured the time spent in the execution between the points shown in circled numbers.

On the server side (right-hand in Figure 9), in (3) we measure query execution time on our in-house XML repository, CDQA (which works on top of a DOM in-memory structure). The actual query processor is the X-OQL module; it sends a parameterized query to the repository, and retrieves an iterator over the results. We measure the time until the iterator completes execution, *scost* in Equation 2.

On the client side (at left in Figure 9), we measure: the overall perception of the time it takes to complete the call. In point (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. Throughout this section, we use *pack* and *unpack* to designate the total cost of packing a SOAP message ($pack_u$, respectively, $unpack_u\times$ the size of the message). Similarly, we use *parse* to denote the time needed to adjust a DOM structure into the client structure.

Experiment Setting We ran our measures on a desktop machine under Linux (named *Mentos*), having 1Gb of memory, a 100 MHz bus speed, and scoring 1202.58 BogoMips 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 the XMark [1] data generator to produce a series of increasing-size documents conforming to the XMark DTD. The size factors and document sizes are described in Table 1.

Throughout the measures, we used the three queries depicted in Table 2. Q_0 returns an entire document. Q_1 and Q_8 are part of the XMark benchmark query set; the former is a "point" query returning just one element, the later is a join query matching buyers with the items they bought.

Q_0	for $x in document("auction.xml") return $x;
Q_1	for $p in document("auction.xml")//person where $p/@id="person0" return $p/name;

Q_8	for $p in document("auction.xml")//person
	return <item person=$p/name/text()>
	COUNT(for $t in document("auction.xml")//closed_auction
	where $t/buyer/@person = $p/@id return $t)
	</item>

Table 2: Queries used in the experiments.

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](for what concerns the SOAP $envSize$, it is $316b$).

Figure 10 (top) depicts the execution time of a declarative service implementing Q_0, when asked to execute over various-sized XMark documents. In this experiment, the server and client peers coincide. The time of the lower curve is measured in point (3) in Figure 9; the other one corresponds to point (2) in Figure 9. The difference between the two is due to the time it takes to pack the calibrating document in the response message (the request message has negligible size). Both curves are quite linear, and can be approximated as $0.0015 \times s$ (where s is the file size), respectively, $0.0037 \times s$, thus, the *pack* time is about $0.0018 \times s$. This time, *pack*, when comparared to the query time is non-negligible; remember that Q_0 requires a full traversal of the document. Thus, for messages of important size, SOAP message packing costs are a significant cost component.

Hot versus cold runs We turned to services implemented by more complex queries, and analyzed the difference between hot and cold runs. Figure 10 (middle and bottom) depicts the query cost measured in (3), for Q_1 and Q_8. We notice a constant overhead for the first (cold) run, of about 600 ms. This overhead is the $init_a$ value in Equation 2. This value is relatively important for Q_1, which has overall low cost; however, it is insignificant for a more costly query as Q_8.

XOQL inferred cost formula From the curves in Figure 10, we can infer a rough cost formula for the queries. Q_1's cost grows linearly, while for Q_8 the curve is quadratic. The reason is that XOQL (as well as other query processors implemented on top of DOM, e.g., Kweelt [24]) typically needs to traverse the document as many times as there are *independent* path expressions in the query (starting from the document root). In Q_1 there is just one such path, but in Q_8 there are two, corresponding to `$t` and `$p`.

Thus, we infer the formula: $srvExec_{Mentos} = s^N \times ct_N$, where s is the document size, N is the number of independent paths in q, and ct_N are system constants.

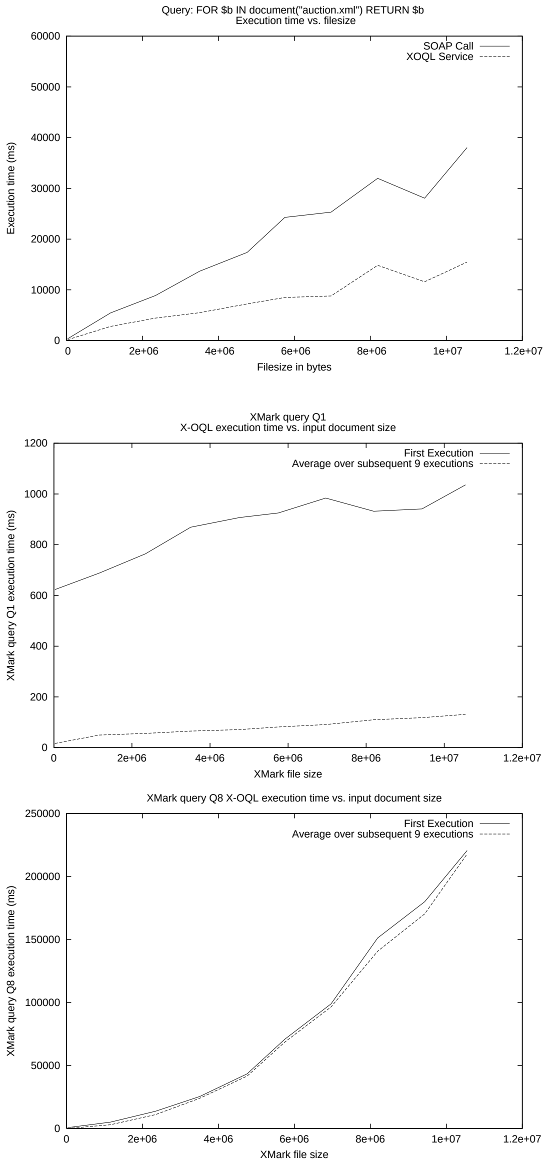

Figure 10: Top: Q_0 execution times: query execution only ("XOQL Service"), and SOAP overhead ("SOAP Call"). Middle: hot and cold runs of XOQL on Q_1. Bottom: hot and cold runs of Q_8.

The above experiments and reasoning take place during cost model calibration (Section 4.2). If the query processor had a more elaborate implementation, Q_1 costs would have been almost constant (due to the presence of an index, for example). Similarly, a more elaborate query processor could have lead to a linear (not quadratic) increase of Q_8 costs. We expect that for the actual AXML peer implementation, the analyze of three measures of each benchmark query (Q_1 and Q_8), and the corresponding document sizes, is enough to observe the exponential law verified above, and consequently derive the constant ct for a peer.

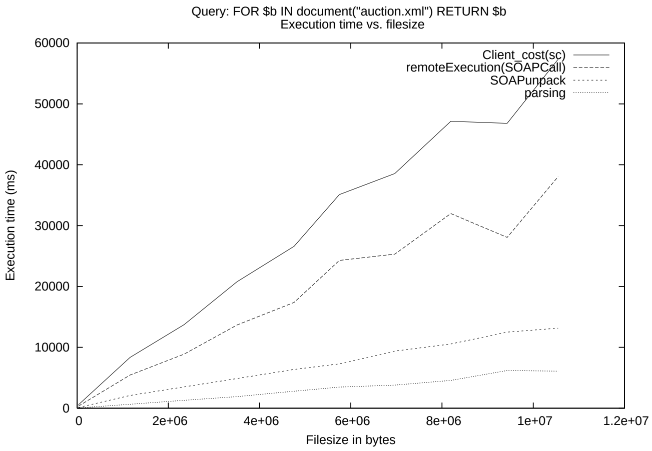

Figure 11: Decomposition of Q_0 costs, on various XML documents from Table 1.

Client-side cost decomposition In this experiment, we detail the times involved in the service call execution, picturing query execution times with several XMark documents. Again, the client and server peer coincide. The results of the measures made in points (1), (2), and (3) (in terms of Figure 9, for the client and server roles) are presented in Figure 11.

We omitted from Figure 11 the unpacking time: since queries are relatively small compared to their results, the time it takes to pack/unpack the query message is negligible. From Figure 11, analyzing the points on each curve, formulas for $unpack$, $parse$ and $scost(sc)$ can be easily inferred, linear in $resSize$. Though, we reasonably assume that time taken by any SOAP implementation, or XML parser, will grow linearly, so the slope (and an eventual offset) provide sufficient information for a good reasoning of the peer's execution behaviour.

Again, the performance curves of an XML query processor more ellaborate than ours may differ. In this case, the processor will have to provide a special service for its own cost estimation; all that is required is to wrap the cost estimator as a Web service, to be invoked when cost estimations are needed.

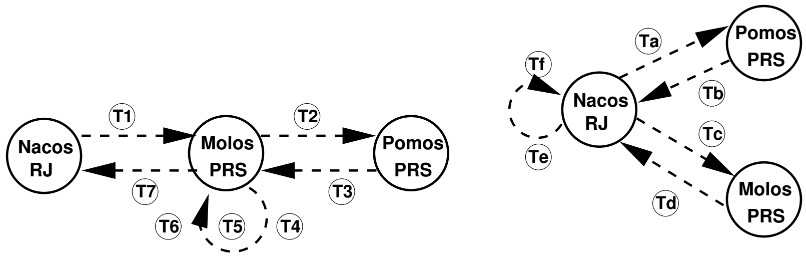

Figure 12: Optimization strategies: delegation (left) and centralized(right).

Name	CPU benchmark [3]	CPU Hardware	CPU Clock	Memory
Nacos	3060.53	AMD Athlon	1.8 GHz	512Mb
Molos	5570.56	Pentium IV	2.8 GHz	512Mb
Pomos	1202.58	Pentium III	0.6 GHz	1024Mb

Table 3: Characteristics of the hardware used in the experiments.

The time it takes to *unpack* the query result (in this case, the document) is quite important, but it is inherent to Axis, which manipulates XML documents based on DOM. However, notice that Axis has one of the fastest DOM parsers, as shown in [9].

5.2 Validation of the distributed cost model

In order to validate our cost formulas, we built a synthetic example that ilustrates the delegation and centralized document evaluation. Figure 12 presents our experiment scenario. Two peers, Molos and Pomos, are located in Paris; they are connected with each other by a 100Mbs switch. Nacos, the third peer, is located in Rio de Janeiro. The connection speed from Molos to Nacos is characterized by an average *ping* rount-trip time of 236 ms. Table 3 presents the hardware used in our experiments.

We considered the scenario in which Nacos wants to materialize a document obtained as a join over two other XML documents. The first join ingredient contains XMark user data, and is located at Pomos; its size is 240 Kb. The second join ingredient contains auction data, and is located at Molos; its size is 300 Kb. The decision that Nacos has to take is: if Nacos does the query processing (that is, retrieves the two join ingredients and performs the join), or if he delegates the query processing to Molos.

Cost ingredients The first step towards the estimation of each strategy overall performance, it is to acquire the basic costs ingredients for every peer and to publish it in its statistics document. In order to build this document, two basic services are needed: *timeService* and *bounceService*. The *timeService* returns the execution time of any service call. The *bounceService*, which is run on the server side, *echoes* back whatever it receives as input. These two simple tools are very useful for acquiring the cost ingredients of our cost formuas, as we will show.

Our first ingredients are the curves describing our peer (server-side) query processor performance. As mention in Section 5.1, *ct* is the machine dependent factor. The first factor to be obtained is ct_1: we issue Q_1 over two different size XMark documents. ct_2 is determined by issuing Q_8 over three different XMark documents. We are concerned about the time spent in processing those queries, so the queries are wrapped in our *timeService* service, and the results published in the peers statistics document.

As a peer publishes some services, it is important for a client peer to acquire its perspective of the service execution time. With the *timeService* and the service call, we are able to measure the client perspective time of a service call execution, in other words, we capture the *scost* concept presented in Equation 2. When the service provides AXML declarative services, Equation 2 can be used to estimate the service execution time.

The next estimation that our statistics document has to provide is: how long it takes to send a query to the client peer. From Equation 1, this corresponds to the term $(callSize + envSize) \times net(1, P_i, P_j)$, where the *callSize* is small (the length of the query). This estimation can be roughly obtained by measuring the time of *bounceService* from peer P_i to P_j, *bounceService* with a 0 byte parameter.

Last but not least, we need to determine the curve describing the network time. We measure the time to execute a *bounceService* with diferent size parameters, for simplicity in our experiment, we expect a linear behaviour from the network, w.r.t. the expected services result sizes (between 100Kb and 4Mb).

Table 4 presents the statistics summary of Molos and Nacos documents, which were obtained by the methods described above. Table 4 also relates the cost to the strategy decomposition presented in Figure 12.

Cost formulas Table 5 presents the cost formulas for the delegation and centralized evaluation strategies. The delegation strategy can be estimated by the sum of each T component presented in Figure 12.a.

- T1 express the time to send the request to Molos, in the delegation equation in Table 5, this time is obtained by estimating $Molos$: *bounceService*.
- T2 is the request to Pomos for the Users document, in the delegation equation in Table 5. This time is obtained by estimating $Pomos$:*bounceService*.
- T3 is the response from Pomos with the Users document. This time is obtained by estimating $Pomos$:$getUsers$.

- T4 is the time Molos spent retrieving from its database the Auctions document. This time is obtained by estimating $Molos\!:\!getAuctions$.

- T5 is the time spent in joining the data. This time is estimated as $Molos(ct_2 \times resultSize^2)$.

- T6 is the time spent in packing the result. This time is estimated as $Molos(ct_1 \times resultSize)$.

- T7 is the time spent in sending the result to Nacos. This time is estimated as $Pomos\!: net \times resultSize$.

The cost of the centralized strategy can be estimated similarly to the delegation one, by summing up the T components presented in Figure 12b:

- Ta expresses the time to send the request to Pomos, in the centralized equation in Table 5, this time is estimated using the time of $Pomos : bounceService$.

- Tb is the response from Pomos with the Users document, this time is estimated as the execution time of $Pomos\!:\!getUsers$.

- Tc expresses the time to send the request to Molos, in the delegation equation in Table 5. This time is estimated using the execution time of $Molos\!:\!bounceService$.

- Td is the response from Molos with the Auctions document. This time is estimated using the execution time of $Molos\!:\!Auctions$.

- Te is the time to join the data. This time is estimated by the expression: $Nacos(ct_2 \times resultSize^2)$.

- Tf is the time spent in parsing the result. This time is estimated by the expression: $Nacos(ct_1 \times resultSize)$.

Table 6 presents the results of our formulas for the delegation and centralized strategy.

Experiments To illustrate the interest of each possible strategy, and the need for cost-based optimization, we varied the selectivity of the join involved in the materialization problem considered. In the first experiment, the join selectivity was $5\times$, meaning the expected result of the materialized document has size 3Mb. In the second experiment, the join selectivity is $0.5\times$, meaning the expected result is 300Kb. Table 6 presents, for the two experiments, the estimated costs obtained as above, and the actual measured performance.

Figure 13 presents the graphics of each component of the measured timed, as well as the calculated time and the real time.

Molos	Operation	Time / Factor
	Pomos:getUsers T3	4781.65 ms
	Pomos:bounceService T2	21.40 ms
	Molos:getAuctions T4	718.60 ms
	ct_1 - Linear Factor	0.001079
	ct_2 - Join Factor	75×10^{-11}
Nacos	**Operation**	**Time / Factor**
	Pomos:getUsers Tb	7057 ms
	Molos:getAuctions	5103 ms Td
	Molos:bounceService Tc T1	473.9 ms
	Pomos:bounceService Ta	512 ms
	ct_1	$0.004225 - LinearFactor$
	ct_2 - Join Factor	113×10^{-11}
	Pomos:net[4]	0.015

Table 4: Cost Ingredients.

Strategy	Equation
Delegation	$EvalTime = Molos\!:\!bounceService + Pomos\!:\!bounceService+$ $Pomos\!:\!getUsers+$ $Molos\!:\!getAuctions + timeToJoinData+$ $timeToPackResult + networkConstantToSendToNacos$
Centralized	$EvalTime = Pomos\!:\!bounceService + Pomos\!:\!getUsers+$ $Molos\!:\!bounceService+$ $Molos\!:\!getAuctions + timeToJoinData + timeToPackResult$

Table 5: Cost Formulas.

Strategy	Estimated (ms)		Measured (ms)	
	3Mb	300Kb	3Mb	300Kb
Delegation	61687.99	13703.31	73972	16278
Centralized	27459.21	14850.51	52669	30538

Table 6: Estimated and measured cost ingredients results.

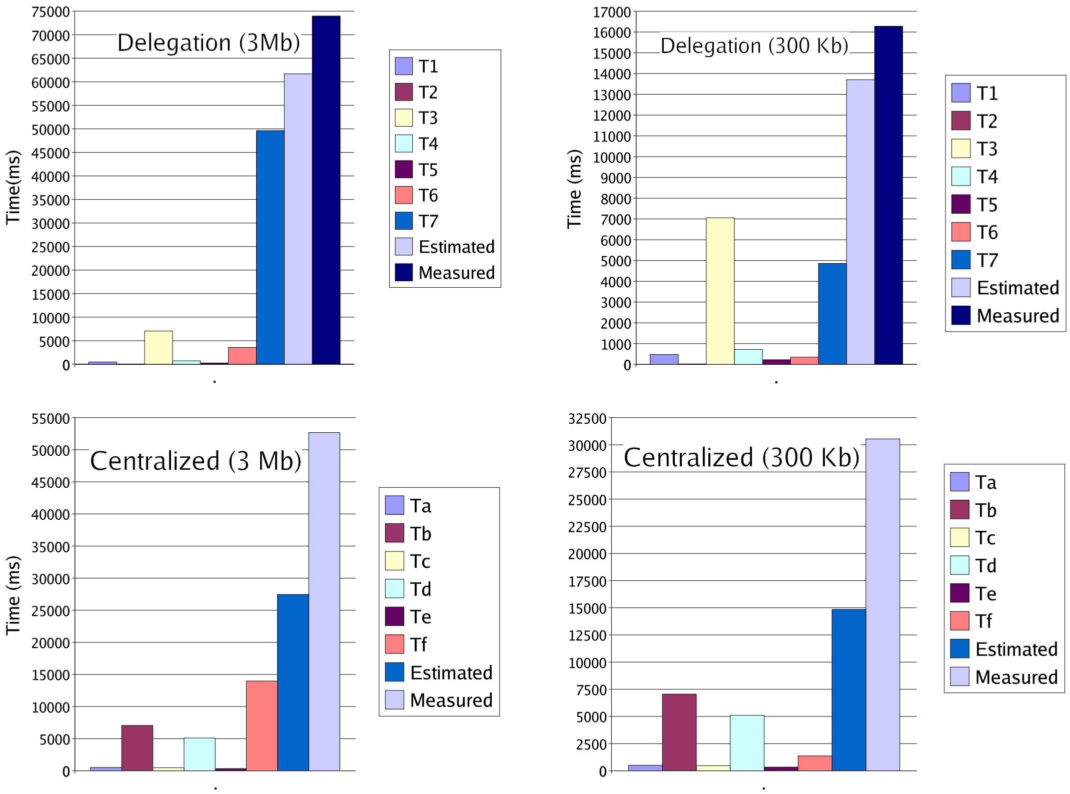

Figure 13: Experiments measurements with a 3Mb result size, and a 300Kb result size.

5.3 Conclusions

From these experiments, we draw the following conclusions. First, our cost formulas were able to predict the general performance of each materialization strategy. Second, for few-bytes documents, and/or a high speed connection among all peers, the dominant costs are due to the query processing, packing and unpacking. Third, for relatively large files (of the order of Mb) the packing and unpacking process is not negligible, even in a scenario with slow networks. This aspect is important, as the AXML platform relies currently on SOAP messaging.

Overall, we conclude that the various costs impacting the performance of document materialization have different impacts in different materialization scenarios, and these impacts are not always intuitive; communication times are important, but so can be query processing times, and, more surprisingly, message packing and unpacking times. We thus establish the need for a comprehensive cost-based model, that accounts for all these various aspects. In the particular case of the AXML system, we have provided such a framework, identified all elementary cost components, and moreover, we have shown how it is possible to gather all these components based on the AXML model itself.

6 Related work and final considerations

Statistics and cost information have long been crucial for the efficiency of distributed query processing [22, 21]. In most cost-based optimization techniques for distributed data management systems (such as multidatabases [28], federated systems [22] and mediated query systems [11]), 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 [15], with the additional complexity of the many-peers decentralized setting. Furthermore, conversely to multidatabase systems, P2P systems [13, 25, 18, 14] cannot rely on a centralized cost catalog. In previous work [23], 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 [12], 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 [8], where only cost information is obtained at runtime [6]. 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 [10, 19], and network coefficients [7, 20], have been proposed. An interesting category of cost ingredients in P2P systems consists of qualitative performance criteria (for example, *site reliability*) [17].

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.

Acknowledgement The authors thank Serge Abiteboul for his comments on an earlier version of this work.

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. In *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] The AXIS SOAP engine. Available at www.apache.org.

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

[7] 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.

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

[9] D. Davis and M. Parashar. Latency performance of SOAP implementation. IEEE Cluster Computing and the Grid, 2002.

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

[11] 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.

[12] 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.

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

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

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

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

[17] 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.

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

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

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

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

[22] 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.

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

[24] A. Sahuguet. The Kweelt XML query processor. Available at kweelt.sourceforge.net.

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

[26] Dorst W. The quintessential Linux benchmark: All about the BogoMips number displayed when Linux boots. *Linux Journal*, 21, 1996.

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

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

Appendix: the AXML documents for the PRONAF actors

In this section, we present in details the AXML documents that can be used to implement our target application.

The AXML document hosted by BACEN is presented in Figure 14. For this role, BACEN is an every day interest rate provider. This particular document does not contain any intensional data.

```
<interest>
...
  <TJLP date="29/02/2004">
    <rate> 10.00</rate>
  </TJLP>
  <DOLAR date="29/02/2004">
    <rate> 3.00</rate>
  </DOLAR>
...
</interest>
```

Figure 14: BACEN AXML document.

The AXML document hosted by BNDES keeps track of every agent contract and every payment per contract. The document includes information on payments already done, the next payment, the remaining principal amount, and some other information. Intensional data appears in this document, concerning the following aspects: the next payment (and other open payments); the parcel amount; the interest; and the contract percentage calculation for the bonus grant. The other intensional data is the effective amount payed at the cashier. It is important to notice that the interest and the bonus are provided respectively by BACEN, and the agents.

Figure 15 and Figure 16 present an example of a BNDES AXML document, with the BNDES database of loan schedules for the agents having requested the PRONAF credit.

The agent's AXML document keeps track of every farmer and/or cooperative contract. This document is similar to the BNDES AXML document. It contains the payments already done, the next payment, the remaining principal amount, and some other information. However, it is semantically different and has some minor differences from the BNDES document,

```
<contracts>
...
<contract num="89000789511">
  <agent> Bank of Brazil - BB </agent>
  <beginDate> "18/07/2002" </beginDate>
  <endDate> "18/07/2008" </endDate>
  <periodicity>6 − semestral</periodicity>
  <initialAmount>100 000.00</initialAmount>
  <remainingAmount>80 000.00</remainingAmount>
  <program>PRONAF-BONUS</program>
  <payment date="18/05/2003">
   <value> 10 500.00 </value>
   <interest name=TJLP>10%</interest>
   <bonus> 500.00 </bonus>
   <amount> 10 000.00 </amount>
  </payment>
  <payment date="18/11/2003">
   <value> 10 500.00 </value>
   <interest name=TJLP>10%</interest>
   <bonus> 500.00 </bonus>
   <amount> 10 000.00 </amount>
  </payment>
  <payment date="18/05/2004">
   <sc> ANY://getParcel(../remainingAmount, ./interest, ../periodicity, ./bonus) </sc>
   <sc> BACEN://getInterest(TJLP, ./payment@date)</sc>
   <sc> BB://getBonus(../contract@num)</sc>
   <sc> ANY://getCashier(../contract@num) <sc>
  </payment>
</contract>
```

Figure 15: One contract in the BNDES AXML document.

```
<contract num="89000789512">
  <agent> Rio Grande do Sul State Bank - BANRISUL </agent>
  <beginDate> "21/01/2004" </beginDate>
  <endDate> "21/11/2004" </endDate>
  <periodicity>1 − monthly</periodicity>
  <initialAmount>200 000.00</initialAmount>
  <remainingAmount>160 000.00</remainingAmount>
  <program>PRONAF-BONUS</program>
  <payment date="21/02/2004">
   <value> 20 166.00 </value>
   <interest name=TJLP>10%</interest>
   <bonus> 166.00 </bonus>
   <amount> 20 000.00 </amount>
  </payment>
  ...
  <payment date="18/11/2003">
   <sc> ANY://getParcel(../remainingAmount, ./interest, ../periodicity, ./bonus) </sc>
   <sc> BACEN://getInterest(TJLP, ./payment@date)</sc>
   <sc> BANRISUL://getBonus(../contract@num)</sc>
   <sc> ANY://getCashier(../contract@num) <sc>
  </payment>
</contract>
...
</contracts>
```

Figure 16: Another contract in the BNDES AXML document.

```
<contractGroup num="89000789511">
<sc> BNDES://getInitialAmount(./contractGroup@num) </sc>
<sc> BNDES://getRemainingAmount(./contractGroup@num) </sc>
...
<contract num="1">
  <beginDate> "18/07/2002" </beginDate>
  <endDate> "18/07/2008" </endDate>
  <periodicity>6 − semestral</periodicity>
  <type> 1 − coffee</type>
  <initialAmount>1 000.00</initialAmount>
  <remainingAmount> 800.00</remainingAmount>
  <bonusRight> Yes </bonusRight>
  <payment date="18/05/2003">
   <value> 100.50 <value>
   <interest name=TJLP>10%</interest>
   <bonus> 0.50 </bonus>
   <amount> 100.00 </amount>
  </payment>
  <payment date="18/11/2003">
   <value> 100.50 <value>
   <interest name=TJLP>10%</interest>
   <bonus> 0.50 </bonus>
   <amount> 100.00 </amount>
  </payment>
  <payment date="18/05/2004">
   <sc> ANY://getParcel(../remainingAmount, ./interest, ../periodicity, ./bonus) </sc>
   <sc> BACEN://getInterest(TJLP, ./payment@date)</sc>
   <sc> ANY://calculateBonus(../type, ../bonusRight)</sc>
   <sc> ANY://getCashier(../contract@num) <sc>
  </payment>
</contract>
...
</contractGroup>
```

Figure 17: Agent AXML document.

since it maintains each individual farmer loan schedule. Intensional data in this document consists of the next payment to be done, the open payments, and also the the contract group level, e.g. an initial amount (on the contract group level). Figure 17 presents the AXML document expressing the agent's contract database.

The documents presented above expect a numeric value as the materialization result of the intensional data. These intensional data materializations are encoded in tags as follows:

- **getInitialAmount** returns <contractInitialAmount>;
- **getRemainingAmount** returns <contractRemainingAmount>;

- **getBonus** returns <bonus>;
- **getParcel** returns <value>;
- **getInterest** returns <interest>, and the interest name as an attribute, e.g. TJLP (long-term interest rate), dolar, etc.;
- **calculateBonus** returns <bonus>; and
- **getCashier** returns <amount>.

We do not present the queries implementing the services referred to by the service calls; these queries can be easily inferred from the service names, the input and output data.

Contents

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803