XCraft: Boosting the Performance of Active XML Materialization

Gabriela Ruberg
Department of Computer Science/COPPE,
Federal University of Rio de Janeiro
P.O. Box 68.511, Brazil
gruberg@cos.ufrj.br

Marta Mattoso
Department of Computer Science/COPPE,
Federal University of Rio de Janeiro
P.O. Box 68.511, Brazil
marta@cos.ufrj.br

ABSTRACT

An active XML (AXML) document contains tags representing calls to Web services. Therefore, retrieving its contents consists in *materializing* its data elements by invoking the embedded service calls in a P2P network. In this process, the result of some service calls can be used as input of other calls. Also, usually several peers provide each requested Web service, and peers can collaborate to invoke these services. This often implies a huge search space of many equivalent materialization alternatives, each with different performance. In this paper, we model AXML documents from a workflow perspective and propose a dynamic cost-based optimization strategy to efficiently materialize them, considering the volatility of a typical P2P scenario. Our strategy enables the optimizer, called **XCraft**, to get more up-to-date information on the status of the peers, and to deliver partial results earlier. Based on a service-oriented algebra of plan operators, we exploit P2P collaboration to delegate both execution and optimization control. Our tests with an XCraft prototype show important performance gains w.r.t. a centralized approach, whilst the optimizer also achieved to drastically reduce the size of the search space.

1. INTRODUCTION

Data management in P2P systems have been widely explored in the last years, with emphasis on two technologies [24]: XML, as the universal format for data exchange; and Web services, as the standard framework for programs and data interoperation. The combination of these technologies raised powerful models for distributed computations. In particular, we highlight the *active XML (AXML) documents* [2, 15, 1], which contain special elements representing calls to Web services, and consist of a highly-adaptive media for distributed data. To retrieve the contents of an AXML document, the results of its embedded service calls need to be properly materialized. We are interested in producing and evaluating efficient materialization plans for AXML documents in a P2P scenario.

The service calls that are embedded in an AXML document represent *intensional elements*. Namely, they point to other elements which constitute the ultimate contents of the document. Also, they can feed their input parameters from the AXML document. In this paper, we look at the AXML materialization problem from a workflow viewpoint. In fact, *materializing AXML documents is very similar to executing workflows*: embedded service calls are tasks to be performed, which are often related to each other, causing some invocation constraints and data flows. For example, *invocation dependencies* occur when service calls takes the result of other calls as input parameters. Actually, the invocation constraints of an AXML document correspond to some basic workflow patterns (namely sequence, parallel split, and synchronization [21]). However, AXML materialization always involves some data flows towards the peer that is gathering the document contents – called *master peer*. Hence, an AXML document can be incrementally composed and consumed, while partial results are usually just temporary in workflow systems. Another issue particular to AXML materialization comes from *intensional answers*. That is, service calls may return other calls as the result of their invocation. This means the problem specification may evolve at runtime, and the system has to dynamically update materialization plans accordingly. Ideally, the system should limit the impact of these changes on the plan, thus avoiding excessive re-optimizations.

A materialization plan determines the peers that are necessary to execute the Web services requested by an AXML document. Basically, embedded service calls include a URL and other specific attributes that are required to invoke a Web service, as defined in the SOAP and WSDL standards [24]. In a more flexible approach, Web services can be pointed by abstract references, based in some ontology of services, as in OWL-S [16]. Abstract references are very convenient to describe AXML data, specially because locating the best resources in a P2P system is often burdensome for users. This way, regardless of the services invocation order, an AXML document can be materialized by many evaluation alternatives [19], which differ in: (*i*) the peer that *executes* each service call; and (*ii*) the peer that *invokes* each service call. Notice the master peer can delegate the invocation of a Web service to another peer, thus adhering to a typical P2P execution model and thereby improving performance (since it avoids sending intermediary results to the master peer).

Optimizing AXML materialization is further complicated by the *membership fluctuations* of a P2P system, where peers can join or leave the community at any time. The optimizer cannot afford to spend much effort to generate materialization plans that may be no longer valid at runtime. Since unpredictability is endemic in large-scale systems, peers are not required to produce complete plans before starting their evaluation. Instead, partial plans can be generated and executed (possibly in parallel). This approach has several advantages, since the optimizer can access up-to-date information from the system, thus increasing both the quality of the plan statistics and the plan validity. Moreover, it enables to get first results as fast as they are produced.

The main contribution of this paper is a dynamic cost-based optimization strategy that addresses the invocation of Web services along with the delegation of both execution and optimization control. This dynamic approach enables materialization plans to adapt to the volatility of a heterogeneous P2P scenario. The basic idea is to handle arbitrarily-complex AXML documents by properly splitting the materialization problem into smaller parts, and then interleaving planning and execution. Therefore, the system can yield partial plans and deliver partial results earlier. Our optimizer, called **XCraft**, determines independent tasks for a materialization plan, and distribute them in a decentralized manner. Plans are encoded with an algebra tailored for Web services, and contain abstract operators that enable incremental analysis. Moreover, plan operators suport deferred service location, which fits well in dynamic P2P scenarios. To rank alternative materialization plans, we use a cost model that takes into account tasks delegation, parallel execution, and the main task scheduling heuristics. An experimental evaluation with an XCraft prototype shows significant performance improvements compared to centralized and static approaches.

This paper is organized as follows. We discuss related work in Section 2, and motivate the AXML optimization problem with an application in Section 3. Then, we present basic concepts on AXML documents in Section 4. In Section 5, we introduce a workflow-based formalism for AXML documents, and a P2P enactment model for AXML materialization. We devise our optimization strategy in Section 6, and in Section 7 we outline the XCraft service-oriented architecture. We describe experimental tests with an XCraft prototype in Section 8. We conclude in Section 9 with research perspectives.

2. RELATED WORK

We represent AXML invocation constraints in a formalism based on *directed acyclic graphs* (DAG), similarly to models used for business processes orchestration in workflow systems [6, 12, 21]. We use graphs rather than the AXML tree mainly because relationships between service calls result into invocation constraints that can be arbitrarily complex. These constraints are not naturally expressed by a tree structure. In this section, we analyze works related to resource planning in grid and P2P systems, and AXML optimization.

As in scheduling workflow tasks for grid computing [6, 14, 22], we are interested in determining an efficient assignment of tasks (Web service executions) to distributed resources (peers). However, in grid systems tasks usually are assigned to sites, which encapsulates many servers and perform local optimization. These systems aim mainly for load balance and throughput, while optimizing AXML materialization concerns reducing response time by minimizing plan costs. Nonetheless, planning workflows in distributed heterogeneous systems is an NP-complete problem [12], and it remains a research challenge. Likewise, optimizing AXML materialization is a hard problem, with additional complications from the P2P volatility.

Allocating resources and scheduling tasks to efficiently execute workflows is indeed an important issue. However, although decentralization has become a key feature in both P2P and grid computing, current systems do not support a decentralized planner [6, 7, 9, 10, 17, 25]. The few exceptions [5, 8, 22] do not consider the cost of task delegation when generating plans, and randomly assings tasks coordination to peers. Yet, our results highlight important performance gains achieved by a decentralized approach. Also, tasks delegation can significantly benefit process pipelining techniques for memory-constrained systems, such as in [13], since it enables to concentrate related processes in peers connected by fast links. Moreover, most of the current planners are based on static analysis [6, 8, 10, 25]. Even when they are dynamic or adaptive, they do either greedy [17, 22] or opportunistic [14] resources selection. Therefore, since they work with local decisions, they perform a myopic performance analysis.

We consider that AXML documents are similar to decision flows [11] in the sense that their materialization is *attribute-centric*. Namely, it aims at determining the values of certain data elements. Still, conversely to [11], our strategy is dynamic and enables decentralized evaluation.

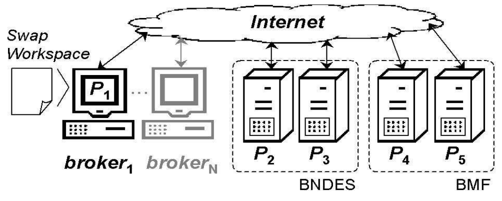
Previous work on AXML optimization mostly addressed typing control [15], XML query processing [1], and data and Web services replication [2]. Mechanisms to generate alternative strategies for AXML materialization, including basic cost formula for performance prediction, was first presented in [19]. XCraft is built upon these ideas, and focuses on the problem of efficiently producing and evaluating materialization plans in dynamic P2P systems. Abiteboul *et. al* [3] proposed a high-level algebraic framework to generate AXML materialization alternatives, with strong emphasis on Web services that can be described by queries. The optimization strategy and algebra of XCraft is complementary to the techniques presented in [3]. Going further, we contribute by handling search complexity, decentralized optimization, resources heterogeneity and P2P membership dynamics. In XCraft, all these issues are addressed by a systematic service-oriented approach.

3. A MOTIVATING AXML APPLICATION

There is a wide range of applications for AXML documents. In [19], AXML documents are used as a practical framework for a financial application, to support a loan program for farming activities. In this paper, we illustrate the main AXML materialization issues with a financial application for *foreign exchange swap*, named *CurrencySwap*. Basically, currency swap operations rely on exchanging debts made in a specific currency against either another foreign currency or a fixed interest rate.

An interesting fact about most of the financial applications is that performance is just as important as other traditionally critical issues, such as security and reliability. For instance, stock trading systems operate in near-real time. Hence, optimization is a strong requirement in this context.

***CurrencySwap* setting.** Figure 1(a) shows the *Curren-*



(a)

Web Service	Providers
CheckSwapStatus	P_2,P_3,P_4
GetCurrentSwaps	P_4,P_5
GetSwapLimit	P_2,P_3,P_4
GetContractPrincipal	P_4,P_5
CalculateDebt	P_2,P_3,P_4,P_5
GetContractSwaps	P_4,P_5
GetExchangeRate	P_2,P_3,P_4,P_5
GetLocalDate	P_1,P_2,P_3,P_4,P_5
GetContractPDF	P_4,P_5
ExtractExcerpt	P_1,P_2,P_5

(b)

Figure 1: *CurrencySwap* application (a) in a P2P setting, and (b) its Web services.

cySwap application in a P2P setting. Companies interact with the system through *brokers*. The central player is the Brazilian Mercantile&Future Exchange (BMF), which manages all the swap operations coming from brokers. Swap contracts are negotiated with BNDES, the major Brazilian funding bank. In turn, BNDES limits the amount of debt subject to swapping for each company, to reduce its financial risk. In Figure 1(a), dotted lines indicate peers in the same intranet (*e.g.*, peers P_3 and P_4 in the BMF intranet). We assume data transfers in an intranet are 50 times faster than through an Internet connection. Information on swap contracts and financial indices are published through Web services. Figure 1(b) lists the main Web services provided by each peer. Peers can gather Web services descriptions either directly from service providers or from catalogs available on the network, such as UDDI servers [24].

During business negotiations, brokers can follow swap information for relevant contracts in a *SwapWorkspace* document, such as the one in Figure 2 (in a simplified AXML notation). Basically, the *SwapWorkspace* document contains the contract number, the company name and its swap status at BNDES, the debt principal in foreign currency, the corresponding converted amount (due to swap operations), the current date, and an excerpt of the contract settlement. The contents of the *SwapWorkspace* document must be gathered from Web services by the invocation of embedded calls, which are represented by the "`sc`" elements. We denote a service call element as `scX`, where `X` is the value of its "id" attribute. In our example, the broker just need to set the contract number and the company name, and then to request (either on-demand or periodically) the system to refresh the workspace contents. A materialized version of the *SwapWorkspace* document is shown in Figure 3.

Materializing AXML data. The "`sc`" elements of the *SwapWorkspace* document refer to Web services that are provided by several different peers (see Figure 1). When a service call is invoked at a peer, the system has to lookup

```
<current_contract><number> 12345 </number>
  <company><name>XTechno Acme Ltd</name>
    <can_swap><sc id="1" service="CheckSwapStatus">
      <param name="swaps">
        <sc id="2" service="GetCurrentSwaps">
          <xpath>//company/name</xpath></sc></param>
      <param name="current_limit">
        <sc id="3" service="GetSwapLimit">
          <param name="company">
            <xpath>//company/name</xpath></param>
          <param name="date">
            <xpath>/current_contract/today</xpath></param>
          </sc></param></sc></can_swap></company>
  <principal><sc id="4" service="GetContractPrincipal">
    <xpath>/current_contract/number</xpath></sc>
  </principal>
  <swap_debt>
    <sc id="5" service="CalculateDebt" followed_by="1">
    <param name="principal">
      <xpath>/current_contract/principal/amount</xpath>
    </param>
    <param name="swaps">
      <sc id="6" service="GetContractSwaps">
        <xpath>/current_contract/number</xpath></sc>
    </param>
    <param name="rate">
      <sc id="7" service="GetExchangeRate">
      <param name="foreign_currency">
        <xpath>/current_contract/principal/currency</xpath>
        </param>
      <param name="date"><xpath>/current_contract/today
        </xpath></param></sc></param>
    </sc></swap_debt>
  <today><sc id="8" service="GetLocalDate"/></today>
  <contract_excerpt><sc id="9" service="ExtractExcerpt">
    <param name="text">
      <sc id="10" service="GetContractPDF"/></param>
    <param name="input_format">PDF</param>
    <param name="output_format">XML</param>
</sc></contract_excerpt></current_contract>
```

Figure 2: AXML document *SwapWorkspace*.

for its possible service providers, and then choose the best peer to execute the call. To materialize an entire AXML document, such a decision is usually influenced by the invocation of other service calls in the document, specially when some input parameters contain other "`sc`" elements. A peer may decide to delegate some related calls to be invoked at another peer, gathering only the results that are necessary to build the AXML document. For example, `sc9` takes as input the result of `sc10` in Figure 2, but only the result of `sc9` is required to build *SwapWorkspace*.

Figure 4 illustrates three alternatives to materialize `sc9` and `sc10`. The left-most alternative represents a centralized strategy (P_1 invokes both service calls), whereas in the two others P_1 delegates service invocations to either P_4 (on the center) or P_5 (on the right). Delegation strategies are par-

```
<current_contract><number> 12345 </number>
  <company><name>XTechno Acme Ltd</name>
    <can_swap>yes</can_swap></company>
  <principal><amount>75000</amount>
    <currency>USD</currency>
    <due>06/20/2006</due></principal>
  <swap_debt>
    <amount>196500</amount><flow>-15720</flow>
    <currency>BRR</currency></swap_debt>
  <today>04/15/2005</today>
  <contract_excerpt> ... </contract_excerpt>
</current_contract>
```

Figure 3: *SwapWorkspace* document after AXML materialization.

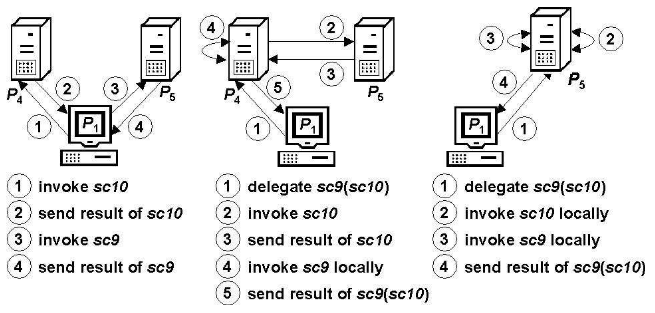


Figure 4: Some AXML materialization alternatives.

ticularly interesting to evaluate nested service calls when the respective executors can communicate through a link faster than the link to the master peer.

Optimization opportunities. Many different evaluation strategies can be used to materialize the *SwapWorkspace* document, considering all the invocation possibilities of each embedded service call. The materialization performance may vary a lot for each strategy. For example, if transfering the result of `sc10` through an Internet connection costs $50s$ (*e.g.*, from P_5 to P_1), then it would cost only $1s$ by intranet transfers. For large data transfers and many service calls, such a difference can be much more important. Thus, a naive materialization strategy may lead to an unacceptable execution time.

On the other hand, optimizing the materialization of AXML documents raises a hard problem: the number of evaluation alternatives grows dramatically even for restricted scenarios. For instance, in our simple *CurrencySwap* example, there are at least 1.898×10^{16} alternative evaluation plans for the *SwapWorkspace* document, considering only possible combinations of service executors and delegations. Suppose each plan is generated and analyzed in $0.5ms$ (a quite reasonable measure for modern PCs). An exhaustive search would last more than 305 thousand years (sic!) to find the best plan. Some heuristics can significantly reduce this search space. In our example, we can apply the *Divide&Conquer* (D&C) heuristic [19], which partitions the document materialization into independent tasks. This would reduce the search space to approximately 1.265×10^{14} alternative plans. Still, this means more than 2 thousand years only to choose the best plan, to actually start materializing the document. Despite the great improvement, the time spent in optimization remains critical. Notice that our example has only 5 distinct peers; if this number raises to 10 peers, the search space becomes 4096 times larger (with the D&C heuristic).

4. AXML BASICS

In the AXML universe, the basic elements of a P2P setting are [1, 2, 4, 15]: peers, Web services, and AXML documents. Peers are uniquely-identified agents connected through a network, and Web services are operations that peers can perform. A Web service may require input parameters that are bound at runtime. The invocation of a Web service is an event, namely a compact occurence that enables: 1) the flow of input parameters to the peer that shall execute the service; 2) the service execution; and 3) transfering the result to the peer that requested the service. It has also a termination status, such as "*success*" and "*fail*". By default, a service call node is invoked by the peer owning the corresponding AXML document (called *master peer*), but this invocation can be delegated to another peer. To be invoked, a service call must have all the information necessary to identify the requested Web service (as defined in the SOAP and WSDL standards [24]); in particular, the executor endpoint.

An AXML document is traditionally modeled as a labelled tree with two types of nodes: (i) *data nodes*, or regular XML nodes, which can be labelled with either element names or data values (only for leaf nodes); and (ii) *service call nodes*, which can encode all the information required to invoke a Web service (URL, operation name, etc. [24]). We denote by SC_d the set of all the service calls of a document d. The children of a service call node stand as its *input parameters*. When a service call node is invoked, its respective subtrees are passed to the Web service, and the invocation result is inserted as a sibling of the call node in the document.

Both service input parameters and results may be AXML data. In this case, an input parameter may be either: *concrete*, if it is explicitly provided by nested AXML elements; or *non-concrete*, when it is specified by an XPath [24] expression. The latter represents a query on some AXML elements; it is evaluated whenever the service call is invoked, and its result is passed in the service request as subtrees of the parameter element. Given two service call nodes v_i and v_j, we say that v_j is an *intensional parameter* of v_i if v_j is either a concrete or a non-concrete parameter of v_i. The term $fanIn(v_i)$ denotes the number of intensional parameters of v_i, while $fanOut(v_j)$ denotes the number of nodes for which v_j is an intensional parameter.

Elements resulting from service invocations have a special *timestamp* attribute to indicate the current snapshot of the document contents. Thus, users can choose to consider either all the previous results or only the last invocation results for feeding the new service requests.

5. MODELING AXML DOCUMENTS FOR MATERIZALIZATION

To efficiently materialize an AXML document, the optimizer has to identify its invocation constraints, and determine how embedded calls should be mapped to a P2P system.

We present in Section 5.1 a DAG-based formalism to express these invocation constraints into *dependency graphs*, which are the basis of our optimization strategy. In Section 5.2, we characterize the participants of the materialization process, and define materialization plans.

5.1 AXML Invocation Constraints

The relationships between the service calls of an AXML document encode some explicit and implicit constraints on their invocation. These constraints are mostly derived from producer-consumer relationships between service calls, and they cannot be properly represented in the AXML document tree since they rather form a complex graph.

Invocation dependencies are produced by intensional parameters of service call nodes, and represent precedence constraints on services invocations. Namely, one can determine that some service calls must be invoked *before* another call, because the latter consumes their results. To materialize an AXML document, the optimizer has to detect both the concrete and non-concrete parameters of each service call node.

Checking concrete parameters is rather trivial and can be done in advance, when the document is loaded and/or updated. Conversely, non-concrete parameters may need a sophisticated analysis, such as in [1]. Also, they may imply shared dependencies, possibly causing cycles (*i.e.*, invocation deadlocks). We consider these cycles can be detected and broken prior to our optimization analysis, according to the techniques proposed in [18]. Moreover, we assume that redundant dependencies (incurred from the transitivity of intensional parameters) are properly reduced as in [18].

Collateral calls. Another way to express invocation constraints in AXML documents is specifying a "`followed-by`" attribute in service call nodes. We say that the service call node pointed by this attribute is a *collateral call*. This means that an invocation of the collateral call must be triggered *immediately after* (as a consequence of) the invocation of the node containing the "`followed-by`" attribute. For simplicity, we assume that a service call node may be associated with only one collateral call.

First-level service calls. Some service call nodes play a distinguished role in the AXML materialization process because the results of their invocation constitute the contents of the AXML document. These results must be kept in the document after its materialization finishes. This is opposed to the results obtained from nested calls, which are often temporary and only needed to invoke their dependant calls. A node v_i is a *first-level service call* of an AXML document d if $v_i \in SC_d$ and for any node v_x in d, if v_x is an ancestor of v_i, then $v_x \notin SC_d$. We denote by ξ the set of all the first-level service calls of d, such that $\xi \subseteq SC_d$.

Dependency graph. This graph is a central input to our optimization effort, and it concisely represents all the invocation constraints that must be enforced on the service calls within an AXML document. It can be obtained by static analysis. Since users may specify documents with cyclic dependencies and execution loops, we consider only valid dependency graphs, as stated in [18].

Definition 1. The *dependency graph* Δ of an AXML document d is denoted by $\left\langle \mathcal{G}, \otimes, \overset{\hookrightarrow}{E}, \epsilon \right\rangle$, where $\mathcal{G}$ is a directed graph with a set $V = SC_d$ of nodes, and a set E of edges. The set V has a distinguished subset $\otimes$ of *persistent nodes*, such that a node v_i is in $\otimes$ iff either $v_i \in \xi$ or $fanOut(v_i) > 1$. Edges in E are either *simple* or *collateral*. For any two nodes v_i and v_j in V, there is a simple edge $v_j \rightarrow v_i$ in E iff v_j is an intensional parameter of v_i. The subset $\overset{\hookrightarrow}{E}$ denotes the collateral edges of E, containing an edge $v_i \hookrightarrow v_j$ iff v_j is a collateral call of v_i. The term ϵ denotes a state function that maps each node in V into $\{active, ready, inactive, failed\}$.

Nodes are active if they need to be invoked, and ready if all their dependencies are inactive. Once invoked, they are either inactive or failed (see [18] for a detailed statechart). Figure 5 shows the dependency graph obtained from the *SwapWorkspace* document. Nodes are circles labelled with service call IDs. Double-line circles are persistent nodes. Dashed arrows indicate collateral edges. Notice a dependency graph encodes a *partial order* on its service calls.

Exit points. Nodes that do not have outgoing simple edges (*i.e.*, with $fanOut = 0$) are particularly important: they are said *exit points*, since they represent points where the

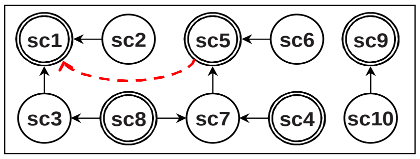


Figure 5: *SwapWorkspace* dependency graph.

materialization process finishes. They can be used to unfold a graph into spanning trees, thus enabling the optimizer to break the materialization problem into smaller parts.

Intensional answers. An AXML-enabled system may allow Web services to return service calls in their results. This artifice can be very useful in many scenarios. For instance, suppose a Web service does not have a certain information that was requested by the user, but it knows which Web services can provide it. In this case, the service may return other calls (to alternative providers), and let the system to pursue the request through other Web services. In this way, the materialization of AXML data can be *dynamically distributed*, thus providing peers with great flexibility for collaboration. We present in [18] mechanisms to efficiently update a dependency graph with intensional answers. Also, our optimization strategy reduces the impact of these answers on materialization plans.

5.2 AXML P2P Enactment

A basic way to write an AXML document consists in hardwiring addresses for the Web services, where the user locates a peer to execute each service call. This approach is not adequate since P2P systems are often complex, highly-dynamic arrangements of peers. Alternatively, one can use abstract service references, based on semantic Web services [16]. An abstract service reference points to a Web service that may be provided by many peers. Users can rely on the system to choose the best provider for each service call. Notice that even if specific service addresses are used, peers can collaborate to materialize a document by delegating some service calls to be invoked by other peers. Therefore, peers need some strategy to distribute the materialization process.

In the sequel, we consider a (possibly infinite) set $\mathcal{P}$ of peers identifiers. We also assume peers can gather information about Web services distribution from the network. Each peer keeps a list of *neighbors* (denoted by $\mathcal{N}$, with $\mathcal{N} \subseteq \mathcal{P}$), namely the peers it can collaborate with.

Execution scope. Given a service call node v, its *execution scope* L_v^E represents the peers that can execute the Web service of v, such that L_v^E = `any` | `unknown` | $\{P_1, \ldots, P_n\}$, where $\{P_1, \ldots, P_n\}$ is a subset of $\mathcal{P}$ which provides v, with $n \geq 0$. We consider service calls can use the symbol "`any`" as the peer providing a Web service, indicating that any provider that the optimizer may find is considered good. The symbol"`unknown`" denotes the optimizer does not have local information on a requested service, but can lookup for it in the P2P system.

Observe that $L_v^E = \{\}$ means the optimizer could not discover any information about peers providing v. Since peers can join or leave the system randomly, the execution scope of a peer is rather a snapshot of the system status. If L_v^E is empty, then the peer can either retry to locate the service afterwards (hoping for some change in the system) or ask other peers to fill in the missing information.

Delegation scope. Distributed computing is inherent in AXML materialization, since service executions usually take place in different peers. Going further, in XCraft other aspects can be distributed. Peers can collaborate to locate service providers, as well as peers can ask other peers to generate parts of a materialization plan for an AXML document. Similarly, a peer can delegate parts of the materialization of a document to other peers. Given a service call node v, the *delegation scope* L_v^C denotes the peers that may invoke v, such that $L_v^C = \texttt{any} \mid \{P_1, \ldots, P_n\}$, where any indicates that all the peers in $\mathcal{N}$ may invoke v, and $\{P_1, \ldots, P_n\}$ is a subset of $\mathcal{P}$, with $n \geq 0$.

In principle, the optimizer may consider any peer in the system to delegate some AXML materialization, accepting that such a peer is allowed to invoke the corresponding service calls. However, this can rapidly make the optimization problem intractable. Hence, we assume a *small-world P2P scenario*, where the optimizer can limit L^C based on L^E (and possibly a given set of peers).

Materialization plans are used to control the AXML materialization process. They are the basis of peers interaction. A plan determines how each service call node is going to be invoked. In XCraft, it can be split and distributed among peers. Moreover, peers can revise some optimization decisions of a plan, and then choose to re-split it among other peers. Thereby, the materialization process can be spread across the system in a decentralized manner.

A materialization plan is obtained from the dependency graph of an AXML document. More precisely, we consider the minimum forest of spanning trees (MFST) [18] of a dependency graph. Plan nodes are labelled based on an algebra $\mathcal{A}$, which consists of a finite set of operators (see details in Section 6.1). Furthermore, given a service call node v held by peer P_v, an *invocation plan* IP_v is an expression $\langle P^E, P^C \rangle$, where P^E and P^C are peers that *can* execute and invoke v, respectively, such that $P^E \in L_v^E$ and $P^C \in (P_v \cup L_v^C)$. The term $\widehat{IP_v}$ denotes the set of all possible invocation plans of v, according to L_v^E and L_v^C. Materialization plans are defined next.

Definition 2. A *materialization plan* $\mathcal{M}$ for a dependency graph Δ is denoted by $\langle \Lambda, \mathcal{O}, \mathcal{L}, \succ, P_m \rangle$, where: Λ is the MFST of Δ; $\mathcal{O}$ associates nodes in Λ with operators in $\mathcal{A}$; $\mathcal{L}$ is a mapping from each node v in Λ to invocation plans in $\widehat{IP_v}$; $\succ$ sets for each node in Λ a total order on its children; and P_m is the *master peer* of $\mathcal{M}$, namely the peer that holds $\mathcal{M}$ and where its persistent results must arrive. We say that $\mathcal{L}$ and $\succ$ are, respectively, the *location scope* and the *invocation schedule* of $\mathcal{M}$. Moreover, $\mathcal{M}$ is *physical* if both $\mathcal{L}$ is total and it maps each node in Λ to exactly one invocation plan; otherwise, $\mathcal{M}$ is *abstract.*

The *makespan* of a materialization plan is the time from the materialization starts until the last service call invocation is completed and the required results are returned to the master peer. Optimizing AXML materialization performance consists in finding a physical plan that minimizes the makespan. This involves two main issues: (*i*) *planning resource selection*, that is determining a caller and an executor for each service call; and (*ii*) *scheduling service call invocations*, to exploit parallelism. Since optimizing AXML materialization has exponential time complexity in the worst case [18], our strategy focuses on finding suboptimal, yet efficient solutions in reasonable time.

```
1   Algorithm DynamicallyOptimize($\Delta$, $k$)
2   {Efficiently materialize $\Delta$ based on dynamic plan
      generation of $k$-depth steps.}
3   begin
4     Generate an initial abstract plan $\mathcal{M}_i$ from $\Delta$
5     Compute the set $T_i$ of materialization tasks
        of $\mathcal{M}_i$
6     Order the tasks of $T_i$ by priority level
7     for each task $t$ in $T_i$ do
8       if $t$ is to be delegated then
10        Pick a new master peer $P'_m$ from $\mathcal{N}$ for $t$
11        Delegate $t$ to $P'_m$ and go to the next task
12      end if
13      Split $t$ into $k$-depth subplans
14      for each subplan $\mathcal{M}_x$ in $t$ in topological order
15        do
16        Locate providers and executors for $\mathcal{M}_x$
17        Generate alternative physical plans of $\mathcal{M}_x$
18        Rank physical plans and pick
            the best $\mathcal{M}_{best}$
19        Execute $\mathcal{M}_{best}$
20      end for
21      Re-evaluate the order of tasks in $T_i$ {optional}
22    end for
23  end
```

Figure 6: Dynamic optimization strategy.

6. DYNAMIC OPTIMIZATION FOR AXML MATERIALIZATION

Efficiently materializing AXML documents involves two major issues: a *huge search space* of evaluation alternatives, and the *unpredictability* of the P2P setting. In a static approach, all the service calls and the interactions among them, their service providers, and communication costs are assumed to be known *a priori.* Clearly, this is not suitable for AXML materialization. Instead, the optimizer should react to changes in the environment, and it should not be based solely on plan re-optimization (which is often expensive in an unstable setting). We propose an optimization strategy that exploits dynamic techniques to reduce search complexity and to adapt to both system performance and membership fluctuations. In our strategy, materialization plans are not produced at once, and re-optimization is triggered only when really necessary. Not surprisingly, our techniques rely on splitting a plan into smaller pieces. Yet, they partition the problem specification such that relevant aspects are preserved.

Inspired on typical Web protocols, which present results as they arrive (instead of waiting for complete documents), our optimization strategy also allows to minimize the time to obtain the *first results* of the document materialization. We interleave planning and execution, thus the optimizer can decide how to proceed after partial executions, when it may have more fresh information on the system status. The overall algorithm of our strategy is described in Figure 6. Basically, we work on the materialization of an AXML document by using its dependency graph. The optimizer unfolds this graph into trees, and then produces an *initial abstract plan*, whose location scope and invocation schedule are not determined. This plan is partitioned into materialization tasks, which are further split into subplans according to the topological order of the nodes. For each subplan, the optimizer annotates the execution and delegation scopes of the nodes, and then generates alternative physical subplans. These equivalent subplans are ranked based on their makespan, and the "best" (but not necessarily optimal) alternative is picked and evaluated. This process is repeated for all tasks. Optionally, some tasks can be delegated. Next, we detail these steps.

6.1 Generating Materialization Plans

Generating a materialization plan consists of associating the nodes of a dependency graph with adequate evaluation operators, which will actually process each service request. For example, a service call may be invoked by the master peer or be delegated to another peer – each case requires a different operator.

Spanning trees. Since a dependency graph is rather complex, we first encode it using a tree-based structure that is more tailored for distributed evaluation. Namely, we extract the minimum forest of spanning trees of the graph. Several classical algorithms can be used to build spanning trees from an arbitrary graph. For instance, the well-known Prim's algorithm has time complexity $O(|V|^2)$ using an adjacency list as graph structure, and $O(|V|log|V| + |E|)$ for a heap-based graph. This algorithm begins with a node of the graph as the current tree, and then builds its *border*, that is a set of all the nodes that can be reached from it. Nodes in the border are added to the spanning tree and expanded recursively. These steps are repeated until all the spanning trees of the graph are obtained. To generate the MFST, the exit points of the graph are used as roots for spanning. Moreover, given two nodes v_x and v_y, we consider that v_y is *reachable for spanning* from v_x if either $v_y \rightarrow v_x$ or $v_x \hookrightarrow v_y$ is in the dependency graph.

An AXML document usually involves shared dependencies and/or collateral calls, whose subgraphs do not correspond directly to trees. To handle this, we have to build a flat representation of the graph, where each node belongs to exactly one tree. This is done by *node detachment* – namely, by identifying and separating subgraphs that are connected by some service call. It involves two special transformations: (*i*) *node replication*, to replace a node by a set of exact copies of it, which denote the same instance of its service call; and (*ii*) *node cloning*, to add new instances of a service call node to the dependency graph. Node replication separates subgraphs connected by shared dependencies, whereas node cloning expands collateral calls. In [18], we present an algorithm to flatten graphs based on these transformations without loss of information.

For large AXML documents, keeping the entire dependency graph in memory in order to generate its MFST may be prohibitive. Specially after node detachment, since flattening a graph usually inflates its size. Nevertheless, our approach can scale well in limited-memory scenarios, since the optimizer does not necessarily have to generate the MFST at once. In fact, the optimizer can expand and process each spanning tree individually (namely, by processing seed by seed). Even node detachment itself can be performed "*on-the-fly*", as nodes are expanded in the spanning trees.

Algebra of materialization operators. Having computed the MFST of a dependency graph, the optimizer has to turn the resulting trees into a materialization plan. This can be done by labelling tree nodes with operators of an algebra, which has to support Web services invocation and P2P collaboration. Also, it should allow the optimizer to gradually evaluate plans. From these requirements, we propose the algebra described in Figure 7.

We distinguish three operators types:

- *abstract operators* (μ and ρ), which encode the possible executors and callers of service call nodes;
- *physical operators* (**invoke**, **fetch** and δ), which have the information required to actually invoke a Web service; and
- *auxiliary operators* (Θ and **locate**), which are used to decentralize the optimization process.

Notice that both δ and the auxiliary operators do not point directly to Web services that are requested in the AXML document. Instead, they point to some basic Web service for P2P collaboration [18], which in turn can handle one or more service requests of the AXML document.

The optimizer uses these operators to compose materialization plans as follows. First, it generates an initial plan with abstract operators. Then, this plan is successively transformed by replacing, adding and/or consuming operators. Plan transformations may be due to either operators evaluation or traditional rule-based optimization. It is worth mentioning that our algebraic approach is extensible, since one can add other operators to the proposed algebra, as well as new rules to handle these operators. For example, the **invoke** operator could be further specialized into other physical operators, such as an asynchronous operator for continuous Web services.

Generating initial plans. Although there are many plan alternatives for an MFST, abstract operators enable to perform a simple (and fast!) analysis to generate initial plans. We assume each tree node yields either a μ or a ρ operator, the latter for replicated nodes. At this phase, neither $\succ$ nor $\mathcal{L}$ are set for the plan; they are progressively defined during the optimization. Also, collateral calls are denoted by "*cp*" annotations on plan operators. Other supportive information may be annotated, such as node height. Thus, a plan can divided into three areas: the "*main plan*", which has all the spanning trees rooted by exit points, such that replicated nodes are labelled by either ρ or **fetch**; the "*cached plans*" area, which keeps the subplans of replicated nodes; and the "*collateral plans*" area, for the subplans pointed by "*cp*" references.

Figure 8 depicts the initial abstract plan for the dependency graph of Figure 5. Each node represents an algebraic operator, which is specified in the node label.

6.2 Dynamic Plan Generation

In our optimization strategy, instead of processing the entire initial plan at once, the optimizer partitions it: first, into *materialization tasks*, and then into *k-depth subplans.* The idea is to adopt a hybrid search technique to generate physical plans by performing a *workflow-based analysis* on subplans of each task, and evaluating each selected physical subplan.

Operator	Description
$\mu(v)$	The *materialize* operator denotes the set of possible invocation plans of the service call node v, based on L_v^C and L_v^E.
$\rho(v)$	The *retrieve* operator indicates that v is a shared dependency. It works like $\mu(v)$ with a cache lookup.
$\mathbf{invoke}(v, IP_v)$	Represents an invocation of v from P^C to execute the requested Web service at P^E, such that $IP_v = \langle P^E, P^C \rangle$.
$\mathbf{fetch}(v)$	Tells the optimizer to try to use previous invocation results of v from the cache before materializing v.
$\delta(v, IP_v)$	The *delegate* operator asks peer P^C, from $IP_v = \langle P^E, P^C \rangle$, to materialize (possibly optimizing) the subplan rooted at v.
$\Theta(v)$	The *optimize* operator denotes a request for a neighbor to optimize the subplan rooted at v.
$\mathbf{locate}(v)$	Represents a request for a neighbor to determine L^E for all of the operators missing this information in the subtree root by v.

Figure 7: Algebraic operators for dynamic and decentralized AXML materialization.

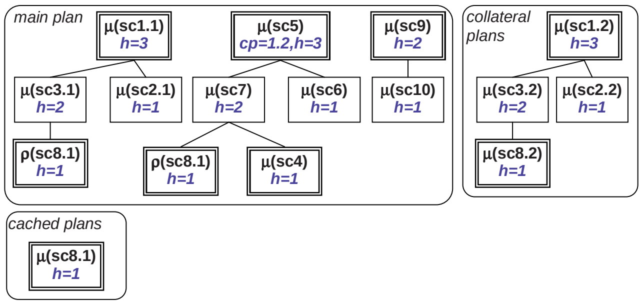


Figure 8: Initial abstract plan for the *Swap-Workspace* dependency graph.

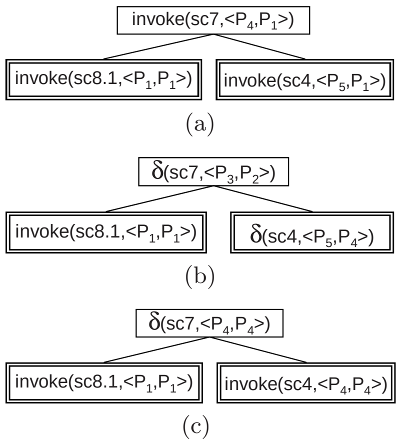


Figure 10: Some physical alternatives for the subplan rooted by μ(sc7).

Materialization tasks. To determine the materialization tasks of a plan, we focus on its exit points: each tree of the MFST yields a task. These trees are not necessarily independent, due to shared nodes. They can be grouped into overlay clusters, such that the trees of each cluster share some service result. Still, once a shared node is evaluated, its related trees may become independent. Since this enables parallel execution and fosters decentralization, we try to find a tasks schedule that would gradually increase the *parallelism potential* of the plan, which is given by the number of clusters of the plan. To rank materialization tasks, the optimizer assigns priorities for them, based on (possibly a combination of) some properties, such as the number of service calls, related clusters and replicated nodes [18]. These last two criteria reflect the degree of, respectively, external and internal data coupling of a task.

In a regular peer (*i.e.*, with only one processor), materialization tasks are usually evaluated in a *blocking mode*. Namely, they are handled one-by-one, and each task blocks the evaluation of the others. In this case, using parallelism potential to compute priority can help the optimizer to explore parallel execution through tasks delegation. This also works for parallel peers, since each task blocks the evaluation only within the scope of its clusters.

Physical subplans. Although materialization tasks are natural candidates to a plan splitting unit, usually they are not small enough to be efficiently optimized. Hence, the optimizer splits materialization tasks into subplans of depth k, where k is a small integer, as described in [18]. For each abstract subplan, the optimizer generates its alternative physical subplans. First, it tries to annotate abstract operators with their respective execution and delegation scopes, from the the peer catalog. If some operators miss this information, then **locate** operators can be used to discover the missing scopes.

Notice P2P systems are highly dynamic, and the location scope of a plan should be preferably provided by *late binding*. By restricting it to subplans, we can defer retrieving services addresses until they are really necessary.

An abstract subplan works as a template for physical subplans. The optimizer uses it to yield different combinations of execution and delegation scope. For example, Figure 10 shows some physical alternatives for the subplan rooted by μ(sc7) in Figure 8, according to the services distribution of Figure 3. Notice that Figure 10(a) represents a centralized materialization approach, such that the master peer (*i.e.*, P_1) has to invoke every service call of the subplan. On the other hand, the subplan alternatives in Figure 10(b) and Figure 10(c) indicate that the master peer will delegate some service calls to other peers, in a decentralized approach.

Several algorithms can be used to produce this search space, such as an exhaustive method or eager enumeration [18]. Yet, our work rather puts emphasis on partitioning a plan to reduce its search space and exploit P2P collaboration. Yet, it is worth noting that most of these algorithms can perform efficiently with our strategy, since it enables them to handle smaller problems.

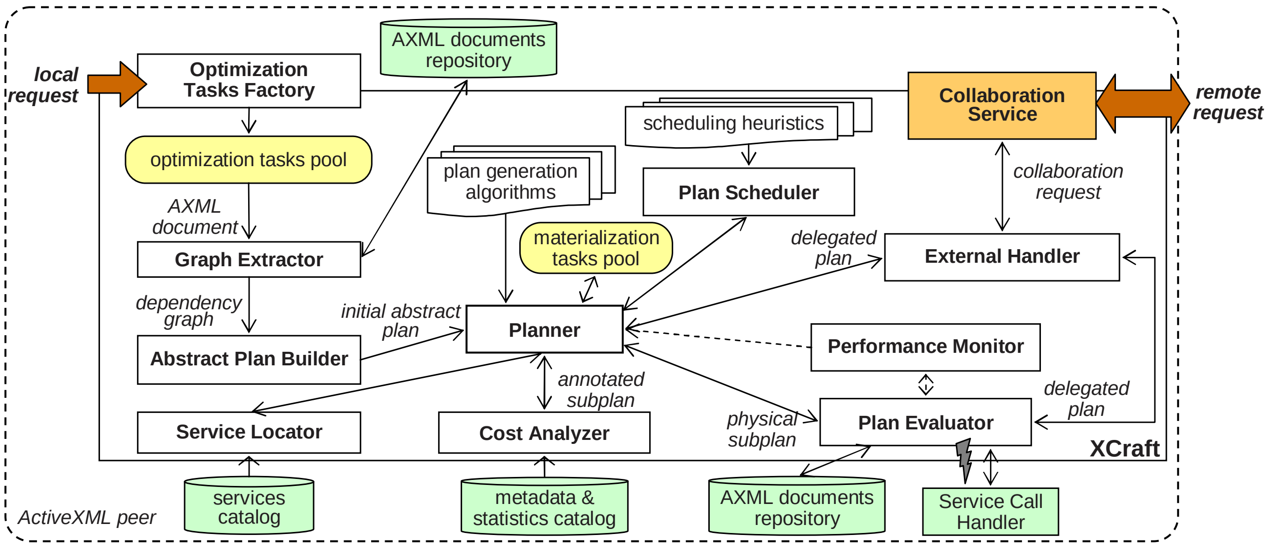


Figure 9: System architecture of the XCraft optimizer.

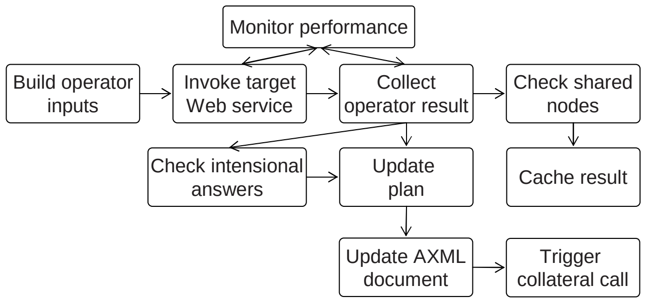


Figure 11: Steps of evaluating a physical operator.

Subplans evaluation. The optimizer evaluates a physical subplan by inspecting operators following the invocation schedule ($\succ$). Since we consider a decentralized environment, where parts of a plan may be delegated to other peers, the optimizer actually evaluates only: *local operators*, namely those whose caller is P_m; and *delegation points*, which are mostly δ and Θ operators. Also, we assume all delegation points return their result to P_m (*i.e.*, transitive delegation cannot be enforced).

To evaluate an operator of a physical subplan, the optimizer builds the required inputs, invokes the service call, gathers the result, and updates both the plan and the AXML document accordingly. Also, it monitors the service invocation and result transfer, and may trigger re-optimization if the performance significantly surpasses expected costs. If some error arises, the optimizer may resubmit the operator, considering the user allows it. Moreover, the optimizer has to check on the result for intensional answers, to properly update the plan. This affects only the current subplan, and eventual re-optimizations do not ripple to the rest of the plan. Finally, the optimizer has also to load shared results into the cache and trigger collateral calls accordingly. The materialization process finishes when all tasks are properly evaluated. These steps are shown in Figure 11.

7. XCRAFT ARCHITECTURE

Based on ActiveXML [4], an open-source P2P platform to manage AXML documents, we present a service-oriented optimizer architecture called XCraft. XCraft works as a facade component of the ActiveXML peer; it interacts with the AXML documents repository, internal catalogs, and the Service Call Handler.

Figure 9 shows the main modules of XCraft. The optimizer conducts AXML materialization as follows. When the contents of an AXML document are requested, its master peer starts a new optimization task at XCraft. The *Graph Extractor* analyzes the service calls embedded into the document and produces its dependency graph (or retrieves it, if it is already available in the cache). This graph is used by the *Abstract Plan Builder* to extract the corresponding MFST and to yield an initial abstract plan. The *Planner*, a central XCraft module, takes the initial plan, breaks it into materialization tasks and calculates their priority.

Materialization tasks are processed such that each task is split into subplans, and each subplan is optimized and completely evaluated before optimization is resumed. First, the Planner asks the *Service Locator* to identify the location scope of the current subplan. According to the plan generation strategy, the Planner roves the search space of alternative physical plans; it uses the *Plan Scheduler* to determine $\succ$ by applying some scheduling heuristic (*e.g.*, min-min, max-min, etc.). For each physical plan, it asks the *Cost Analyzer* to estimate the makespan, and registers the subplan with the best makespan during the search. Then, it sends the overall best subplan to the *Plan Evaluator*, which evaluates the corresponding operators by triggering their service calls according to $\succ$.

As the subplan evaluation proceeds, service call results are gathered and merged into the AXML document. The *Performance Monitor* watches over operators evaluation, and may demand subplan re-optimization if necessary. Both the Planner and the Plan Evaluator may also get plans coming from the *External Handler*, which routes requests from other peers. These requests consist of materialization plans serialized in the XML format. Plans may keep information on their delegation trace to control propagation in the P2P system. Remote requests are received by the *Collaboration Service*, which implements the interface of the basic Web services for P2P collaboration. Finally, evaluation results are sent back to the origin peer through the Collaboration Service reply.

For simplicity, we omitted in Figure 9 two XCraft modules: the *Plan Cache*, where the optimizer keeps shared plans and their results; and the *Optimizer Profile Loader*, which is used to configure a set of properties that control the behavior of XCraft internal modules.

8. EXPERIMENTAL RESULTS

We have implemented and tested the proposed optimization strategy in the ActiveXML system [4]. We extended the ActiveXML peer (version 4-Beta) with the XCraft optimizer. We used the Java language and open-source software, such as Apache Tomcat 4.1.29, JDK 1.4.2, and Axis 1.1. To compute spanning trees, we used a Java implementation of the Prim-Jarnik algorithm from the JDSL library.

In our tests, we deployed three ActiveXML peers extended with the XCraft optimizer and some basic collaboration Web services. At each peer, we also deployed two declarative Web services, which perform respectively a union and a join operation on documents derived from the ACM SIGMOD Record articles database [20]. These services take a single input parameter, and combines it (either by a union or a join operation) with a locally stored file.

We used three heterogeneous machines under different workloads, as described in Figure 12. Processing power is represented by BogoMips [23]. $Master$ indicates the master peer, which is connected through a 512Kbps Internet link to the other two machines. $Laptop1$ and $Laptop2$ are located in a 36Mbps local network. Figure 13 shows these peers connections. Also, we generated sets of AXML documents with different configurations of service call nodes by varying the height and width of the document trees. Recall the height is determined by invocation dependencies and the width corresponds to the number of trees in the MFST of the AXML document. Basically, in the experiments we used AXML documents that contain sequences (*i.e.*, batch-pipelined tasks) and parallel splits patterns from grid workflows [21].

We performed two basic analysis. First, we identified aspects that have relevant impact on the materialization complexity (*i.e.*, the number of alternative plans). We observe the time spent in optimization with different plan generation approaches, and compare these results with the XCraft dynamic strategy. In the second battery of tests, we evaluate the gains achieved by subplan delegation. We focused on delegation of service invocations, since both optimization and service location operators are more related to contingency planning and do not directly reflect performance improvement. These operators rather benefit the adaptivity capabilities of the optimizer.

It is worth noting that most P2P and grid systems neglect the communication costs from transfering data between two operators that are delegated to the same peer. Nonetheless, this simplification is not true for Web services, since they always involve heavy operations for XML handling, as observed in [19].

Peer	O.S.	BogoMips	RAM
$Master$	Debian GNU/Linux	2957.31	512Mb
$Laptop1$	MS Windows^{TM}XP	1718.18	512Mb
$Laptop2$	Linux SuSe^{TM}	1198, 77	512Mb

Figure 12: Hardware of deployed ActiveXML peers.

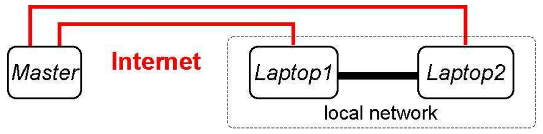


Figure 13: P2P network used in experiments.

8.1 Devising the Search Space

The time spent on the optimization process is a crucial point when efficiently materializing AXML documents. Since this represents a combinatorial problem, exhaustive search is usually prohibitive, thus making heuristics mandatory. Moreover, in P2P systems, the optimization process itself cannot be time-consuming due to the dynamic behavior of peers. Therefore, an important goal of the optimizer is to analyze the size of the search space of a given materialization plan.

The great improvements of hardware performance have made possible to tackle several complex optimization problems. Nonetheless, we observed they are still insufficient to solve the issues posed by AXML materialization, which usually involves very large search spaces. To have a more clear idea of the size of the search space of an AXML document, we used the complexity formulas presented in [18] to identify its relevant dimensions and estimate their impact.

In Figure 14, we varied the $fanOut$ of service call nodes for an AXML document with height $h = 2$ and four first-level service calls (*i.e.*, width is $|\Lambda| = 4$). We consider a system of three peers. Notice the axis of number of plans is in a logarithmic scale. Results represents the complexity of partial plans with only executors and to the complete search space, assuming an exhaustive method. Analogously, Figure 15 shows the size of the search space by varying both the $fanOut$ and the document height. These results clearly indicate the search space grows exponentially with respect to both $fanOut$ and h. In fact, even for small documents, its size is significantly large. In further analysis, we found this exponential behavior stands the same for the number of peers involved in the materialization process.

XCraft uses a dynamic optimization strategy to reduce the number of inspected plans, yet taking into account relevant properties such as communication costs. Basically, XCraft breaks materialization tasks according to a given parameter k. We can observe in Figure 16 the impact of our strategy on the number of inspected alternative plans. We used an AXML document with four first-level service calls of height fixed at $h = 8$. We assumed a very simple case, where the $fanOut$ of each service call node is 1 (namely, the document contains only 32 service call nodes). We estimated the number of inspected plans with our dynamic strategy for different values of the k parameter (*i.e.*, the height of each subplan to be analyzed by the optimizer), considering both the analysis of subplan delegation and choosing only service executors. We also compare these results with the exhaustive

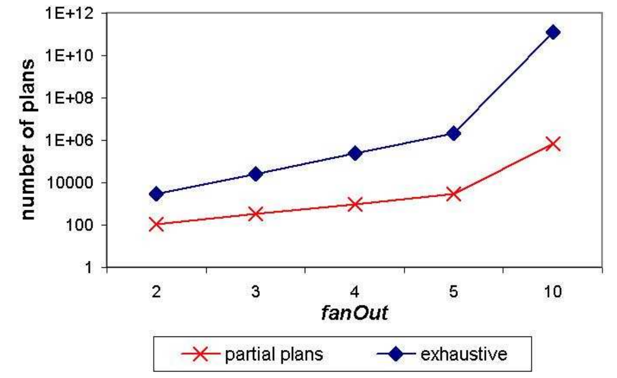


Figure 14: Impact on the size of the search space by varying the $fanOut$ of service calls.

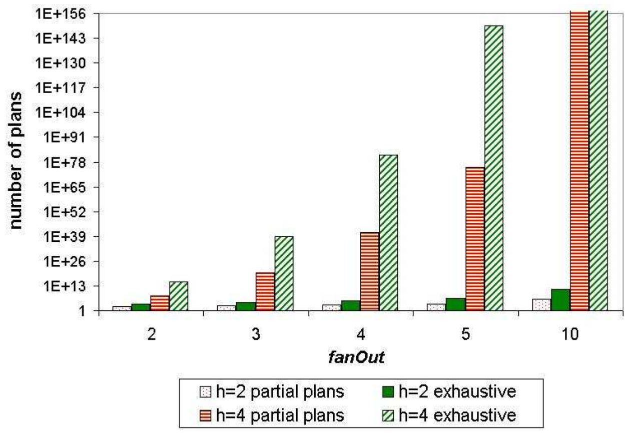


Figure 15: Search space analysis from varying both the document height and the $fanOut$.

search strategy, and results of using the Divide&Conquer heuristic to identify independent materialization tasks. Notice that even in this simple case, the size of the search space prevents adopting the exhaustive strategy. Our dynamic approach provides XCraft with flexibility to deal with complex AXML scenarios, by allowing it to scan search spaces with manageable sizes.

The size of the search space has a major impact on the optimization time, which can be estimated from the results in Figure 16. In our experiments, for the AXML document used in Figure 16, we observed that the optimizer spends an average of 0.5 millisecond to generate and analyze an alternative plan. Although large documents tend to require more time to be generated and analyzed, this average value sets a good performance reference. For example, scanning the search space of 1000 equivalent plans would take about 0.5 second.

8.2 Plan Delegation Effects

Although our dynamic strategy produces suboptimal solutions, it enables the optimizer to exploit subplans delegation, which usually results in significant performance gains. This can be noted in Figure 17. We evaluate the performance achieved by delegating materialization subplans containing service call nodes with $fanOut = 1$, and invocation results with 100Kbytes. This corresponds to AXML documents

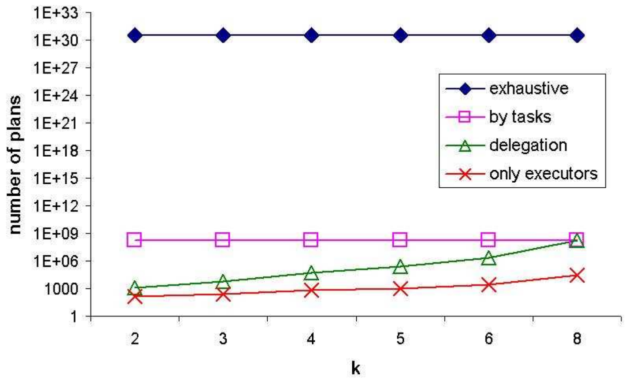


Figure 16: Search space reduction in XCraft.

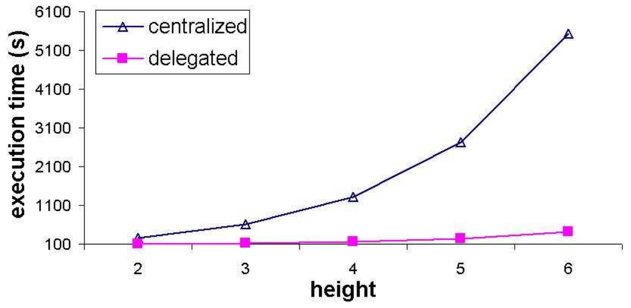


Figure 17: Performance gains obtained by subplans delegation.

that contain batch-pipelined tasks (*i.e.*, with at most one dependency). We vary the height of tasks in the documents from 2 to 6, and use the D&C heuristic to partition materialization plans into independent tasks. The k parameter is set with the task height.

In a centralized evaluation strategy, the optimizer invokes of each service call, and gathers their results from an Internet link. With delegation, the master peer sends physical subplans (*i.e.*, materialization tasks) to be evaluated remotely, and receives only persistent service results to compose the final document contents. Only the root element of each materialization task is a persistent result. It is worth noting that, for higher values of $fanOut$ and of the size of service results, the performance gains of plan delegation tend to be even more expressive.

9. CONCLUSIONS

Materializing AXML documents correspond to a general case of finding efficient assignments of inter-related tasks to heterogeneous machines, which is a hard optimization problem. In this paper, we present an optimization strategy that widely explores dynamic techniques, thus scaling well for decentralized and ad-hoc P2P systems. This strategy also improves system recovery since smaller tasks can be better monitored for early error detection and fixing. The proposed optimization strategy exploits algebraic operators to handle plan complexity and encode collaboration decisions. We also present a service-oriented optimizer architecture to support collaborative AXML materialization.

We believe this work goes beyond the AXML context, and contributes to the efficient execution of workflows, specially in highly-dynamic and heterogeneous settings. Also, with a descentralized architecture for collaborative optimization, we addressed an important issue that has been neglected in most of the current systems.

We are currently evaluating methods based on stochastic local search to incrementally refine physical plans. These methods have been implemented in a simulation tool, to facilitate the analysis of complex P2P settings. There are many other interesting paths to pursue our work, such as investigating contingency planning for service call failures, and extending XCraft with pipeline techniques, such as those proposed by [13].

Acknowledgements. This work is partially funded by CNPq and INRIA. Gabriela Ruberg was also supported by the Central Bank of Brazil. The contents of this paper do not represent the opinion of these instituitions. The authors would like to thank Serge Abiteboul and Ioana Manolescu for the fruitful discussions on AXML optimization, and the Gemo team for the ActiveXML prototype.

10. REFERENCES

[1] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for Active XML. In *ACM SIGMOD*, pages 227–238, 2004.

[2] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *ACM SIGMOD*, 2003.

[3] S. Abiteboul, I. Manolescu, and E. Taropa. A framework for distributed XML data management. In *EDBT*, pages 1049–1058, 2006.

[4] ActiveXML home page. At `www.activexml.net`.

[5] B. Benatallah, M. Dumas, and Q. Z. Sheng. Facilitating the rapid development and scalable orchestration of composite web services. *Distributed and Parallel Databases*, 17(1):5–37, 2005.

[6] J. Blythe, S. Jain, E. Deelman, A. Mandal, and K. Kennedy. Task Scheduling Strategies for Workflow-based Applications in Grids. In *CCGrid*, 2005.

[7] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. A. Hensgen, and R. F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.*, 61(6):810–837, 2001.

[8] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. GridFlow: Workflow management for grid computing. In *CCGRID*, pages 198–205, 2003.

[9] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping Abstract Complex Workflows onto Grid Environments. *J. Grid Comput.*, 1(1):25–39, 2003.

[10] A. Gounaris, R. Sakellariou, N. W. Paton, and A. A. A. Fernandes. Resource scheduling for parallel query processing on computational Grids. In *GRID*, pages 396–401, 2004.

[11] R. Hull, F. Llirbat, B. Kumar, G. Zhou, G. Dong, and J. Su. Optimization techniques for data-intensive decision flows. In *ICDE*, pages 281–292, 2000.

[12] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[13] M. Lemos and M. A. Casanova. On the complexity of process pipeline scheduling. In *SBBD*, pages 57–71, 2006.

[14] L. A. Meyer, D. Sheftner, J. Voeckler, M. Mattoso, M. Wilde, and I. Foster. An opportunistic algorithm for scheduling workflows on grids. In *VECPAR*, 2006.

[15] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. D. Ngoc. Exchanging intensional XML data. In *ACM SIGMOD*, pages 289–300, 2003.

[16] OWL-S: Semantic Markup for Web Services. At *http://www.w3.org/Submission/OWL-S/*.

[17] Pegasus home page. At *http://pegasus.isi.edu*.

[18] G. Ruberg and M. Mattoso. XCraft: A dynamic optimizer for the materialization of active XML documents. COPPE/UFRJ Tech. Report ES-709/07, http://www.cos.ufrj.br/~gruberg/xcraft_rt.pdf, 2007.

[19] N. Ruberg, G. Ruberg, and I. Manolescu. Towards cost-based optimization for data-intensive Web service computations. In *SBBD*, pages 283–297, 2004.

[20] ACM SIGMOD Record articles database. Available at http://acm.org/sigmod/record/xml/.

[21] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[22] P. K. Vargas, I. de Castro Dutra, V. Nascimento, L. Santos, L. Silva, C. Geyer, and B. Schulze. Hierarchical submission in a grid environment. In *MGC*, pages 1–6, 2005.

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

[24] The W3 Consortium. At *http://www.w3.org/*.

[25] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the ASKALON grid environment. *SIGMOD Record*, 34(3):56–62, 2005.