

# XCraft: A Dynamic Optimizer for the Materialization of Active XML Documents

Gabriela Ruberg, Marta Mattoso

Department of Computer Science – COPPE/UFRJ, Brazil

{gruberg, marta}@cos.ufrj.br

Technical Report ES-709/07

May, 2007

## Abstract

An active XML (AXML) document contains special tags that represent calls to Web services. Retrieving its contents consists in materializing its data elements by invoking all its embedded service calls in a P2P network. In this process, the results of some service calls are often used as inputs to other calls. Also, usually several peers provide each requested Web service, and peers can collaborate to invoke these services. This implies many equivalent materialization alternatives, with different performance.

Optimizing the AXML materialization process is a hard problem, which often involves searching a huge space of solutions. Current techniques for workflow scheduling and distributed query processing are insufficient for this problem, since in AXML materialization: (i) the set of participating peers is not known in advance; (ii) service calls in the result of other calls forbid a simple “optimize-then-execute” strategy; and (iii) due to the peer volatility in the network, a plan computed by the optimizer may become invalid at the moment of its execution. Moreover, most of the current optimizers are based on centralized coordination.

We propose a dynamic, cost-based optimization strategy to efficiently materialize AXML documents considering the volatility of a P2P scenario. We formalize the problem from a performance-oriented perspective, and present an optimization strategy that incrementally generates and executes materialization plans. This enables the optimizer to reduce the size of the search space, get more up-to-date information on the status of the peers, and deliver partial results earlier. Our strategy can handle arbitrarily complex AXML documents, and exploits decentralization in many levels.

We also present a service-oriented optimization architecture called **XCraft**. We evaluated our approach in an XCraft prototype for the ActiveXML system, an open-source P2P platform. Our results show promising performance gains compared to centralized, static materialization strategies.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Running application</b>	<b>3</b>
<b>3</b>	<b>AXML Basics</b>	<b>5</b>
<b>4</b>	<b>Service Invocation Constraints in AXML</b>	<b>7</b>
4.1	Precedence Constraints . . . . .	7
4.2	Consequence Constraints . . . . .	8
4.3	Dependency Graph . . . . .	8
4.4	Dynamic Graph Updates . . . . .	12
<b>5</b>	<b>Materializing AXML Documents</b>	<b>15</b>
5.1	Equivalent, Replicated and Generic Services	16
5.2	Exploring P2P Collaboration in AXML . . .	17
5.3	Enacting AXML Materialization . . . . .	18
5.4	The AXML Optimization Problem . . . . .	19
5.5	Main Problem Topologies . . . . .	22
<b>6</b>	<b>Optimizing AXML Materialization</b>	<b>23</b>
6.1	Dynamic Optimization Strategy . . . . .	23
6.2	Extracting Materialization Plans . . . . .	24
6.3	Determining Materialization Tasks . . . . .	28
6.4	Dynamic Plan Generation . . . . .	29
6.5	Delegating AXML Optimization . . . . .	35
<b>7</b>	<b>Cost Analysis of AXML Materialization</b>	<b>36</b>
7.1	Representing Heterogenous Scenarios . . .	37
7.2	Heuristic Cost Analysis . . . . .	37
7.3	Costs of Plan Operators . . . . .	38
<b>8</b>	<b>XCraft Architecture</b>	<b>38</b>
<b>9</b>	<b>Experimental Results</b>	<b>40</b>
9.1	Devising the Search Space . . . . .	41
9.2	Plan Delegation Effects . . . . .	42
<b>10</b>	<b>Related Work</b>	<b>42</b>
<b>11</b>	<b>Conclusions</b>	<b>43</b>

# 1 Introduction

Data management techniques for peer-to-peer (P2P) systems have been extensively exploited in the last years. Two technologies have been crucial in this scenario: the XML format, as the universal media for data exchange; and Web services, as the standard for program and data interoperation. Web services are described as independent, self-contained programs whose interfaces can be published, discovered and invoked throughout the Web [60]. They encapsulate heterogeneous business processes, and their related standards [49, 52, 62] are a practical and effective underpinning for reconciling disparate systems. The *combination of XML and Web services* has enabled new models to express powerful distributed computations, raising a new class of *active XML documents* [5]. These documents consist of a highly-adaptive media for distributed information. Hence, solutions to efficiently support them can significantly contribute towards Web computing.

Besides regular XML data, active XML (AXML, for short) documents contain special XML elements which represent calls to Web services. These embedded service calls can be invoked automatically or on-demand; once a service call is invoked, its result is gathered and merged (according to some predefined criteria) into the corresponding XML document. Invoking embedded calls can be thought of as *materializing some intensional data* of the AXML document. Therefore, to retrieve the contents of a document, all of its service calls need to be materialized. Notice this is a quite common scenario in Web applications, where delivering XML documents usually requires materializing their contents first. We are interested in the efficient materialization of AXML documents in a P2P system.

Materializing AXML documents is quite 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. In this case, nested service calls must be invoked first to provide input for their respective outer service calls. These dependencies enforce some precedence constraints on the materialization process, and they often imply some *data flows* between service invocations. Some of these invocation results are required to embody the final materialized document, while others are *intermediate results that do not need to be kept to the end of the materialization*. There may also exist *invocation consequences* in an AXML document, when materializing a service call should automatically trigger another call. Invocation constraints of embedded service calls correspond to some basic control flow patterns, namely sequence, parallel split, and synchronization [54]. 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 seldom meaningful in workflow systems.

Another issue in materializing AXML documents comes from *intensional answers*. Namely, in AXML-enabled systems, *service calls may return other service calls* as the result of their invocation. This means the problem specification (the actual document to materialize) may evolve *at runtime*. This is very different from traditional distributed query optimization settings: a query can be optimized either statically or dynamically [16, 22], but the query specification itself does not change during optimization (except for parameterized queries). With AXML documents, the system must be able to dynamically update materialization plans accordingly. Also, ideally the system should reduce the scope of impact of these changes in the planning process, thus avoiding excessive reoptimizations.

Basically, the intensional nodes of an AXML document point to specific service references, including the service URL and other parameters that are required to invoke a Web service (as defined in the SOAP and WSDL standards [59]). In a more flexible approach, Web services can be addressed by abstract references, *e.g.*, based in some ontology of services, as in OWL-S [41]. This approach is very convenient to describe AXML data, specially because locating the best resources to execute service requests in a P2P system is often burdensome for users. Considering abstract service references, an AXML document can be materialized by many alternative strategies [44]. Regardless of the services invocation order, these strategies may differ in the choice of: (i) the peer that *executes* each service call; and (ii) the peer that *invokes* each service call. Observe that possibly several peers can invoke a service call, even if it refers to a specific Web service endpoint. This means that the master peer can delegate the invocation of a Web service to another peer. Such a decentralized approach adheres to a typical P2P execution model, and it can reduce communication costs since it avoids transferring intermediary results to the master peer.

Efficiently materializing an AXML document concerns both determining which peer invokes and/or executes each service call (by taking into account, *e.g.*, their communication costs), and ordering the invocation of relevant service calls (by exploiting possible parallel executions). This problem is complicated by the *membership fluctuations* of a P2P system, where peers can join or leave the community at any time. Therefore, the optimizer cannot afford to spend much effort to generate plans that may be no longer valid at runtime. In fact, unpredictability is endemic in large-scale systems, and *peers are not required to generate complete materialization plans before starting their evaluation*. Instead, partial plans can be generated and executed (possibly in parallel). This approach has several advantages. First, the

optimizer can access *up-to-date knowledge* about the system, which increases both the quality of the plan statistics and the plan validity. Going further, *much of the decision process can be deferred* until the system is better informed on service statistics and/or service location.

**Contributions.** In this paper, we show that dynamic techniques are vital for efficiently materializing AXML documents. As a consequence, we propose a dynamic optimization strategy that addresses Web service invocations along with the volatility of a heterogeneous P2P environment. We make the following contributions:

- **A canonical model to represent service invocation constraints of AXML documents.** We capture relevant issues related to the invocation dependencies and consequences between service calls into a DAG-based representation, and we define the necessary criteria to check its validity (w.r.t. deadlock and execution termination) and to eliminate redundancy. Furthermore, we propose efficient techniques to update the graph of an AXML document at runtime with intensional answers;
- **A P2P enactment model for AXML materialization with abstract service references.** We characterize the main participants of the AXML materialization process, and their possible interactions in a P2P scenario. We use this enactment model to define the search space of materialization alternatives for AXML documents, and to formalize the corresponding optimization problem;
- **A dynamic algorithm to generate and evaluate AXML materialization alternatives.** We propose an optimization algorithm that can handle arbitrarily complex AXML documents. This algorithm splits the materialization problem into smaller parts, and then interleaves planning and execution, thus enabling the system to yield partial materialization plans and deliver partial results earlier. Plans are encoded with operators of a materialization algebra, which we introduce to properly evaluate embedded service calls;
- **A cost model to evaluate AXML materialization alternatives.** We devise a cost-based strategy which represents typical characteristics of P2P systems, such as replicated Web services, heterogeneous machines and communication links, parallel execution, and the delegation of materialization tasks; and
- **An AXML optimizer architecture and prototype.** We outline a decentralized, service-oriented optimizer architecture, called **XCraft**, to support the proposed techniques. We implemented an XCraft prototype upon the ActiveXML system [13], an open-source P2P

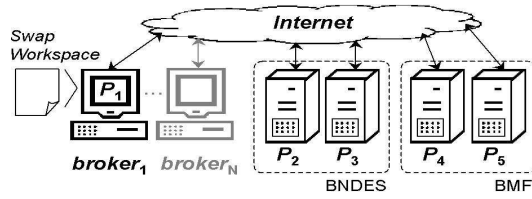
platform to manage AXML documents. We describe an experimental evaluation of its effectiveness.

Although the proposed strategy can benefit AXML materialization in general, it is specially targeted at AXML documents containing calls to *data-intensive Web services*, involving large input or output transfers within a heterogeneous P2P network. Despite the great improvements that we have witnessed in communication speed, data transfers through Web services protocols remain costly due to operations such as packing/unpacking and parsing XML data [44]. Moreover, in such a scenario the main optimization goal is not to find best plans, but mostly to *avoid the worst ones* (which may become unfeasible on critical performance). Also, materialization plans may fail due to many reasons, such as security restrictions and hardware errors. Yet, we believe a dynamic strategy contributes to improve system recovery, since smaller tasks can be better monitored to enable early error detection and fixing.

**Paper outline.** This paper is organized as follows. We describe in Section 2 an application to motivate the need of optimization for AXML materialization, and in Section 3 we present some basic concepts in AXML documents. We define the canonical formalism to represent service invocation constraints in Section 4. Section 5 describes the P2P enactment model for AXML materialization, and states the optimization problem addressed in this paper. Our optimization strategy is proposed in Section 6, including its dynamic algorithm and the algebra to encode materialization plans. In Section 7, we detail the cost model used to analyze alternative materialization plans. We describe the XCraft optimizer architecture in Section 8. Experimental results obtained with an XCraft prototype are analyzed in Section 9. Section 10 discusses related work, and Section 11 closes with some conclusions and perspectives.

## 2 Running application

There are many interesting applications for AXML documents [6], such as RSS news syndication [2] and management of electronic patient records [1]. In the Acware system [10], AXML documents are explored to build *active content warehouses*, which help biologists to continuously integrate and transform information for food risk assessment. Abiteboul *et al.* [9] describe an application based on AXML documents to manage the production and distribution of Open Source software, in the *eDos* European project. In [44], 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*.



(a)

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	$P_1, P_4, P_5$

(b)

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

Basically, currency swap operations rely on exchanging debts made in a specific currency against either another foreign currency or a fixed interest rate. For example, suppose a Brazilian company has a contract for a loan in US dollars. As a security against adverse exchange rate movement, this company can negotiate with a *funding bank* to swap its debt currency against a fixed interest rate. This debt is converted into Brazilian *Reais* according to the exchange rate of the swap operation date. On the contract settlement date, if US dollars have become more expensive, then the company will only have to pay the converted debt plus the interest rate, and the funding bank will provide the remaining difference.

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-realtime. Hence, optimization is a strong requirement of these systems in distributed settings.

***CurrencySwap* setting.** Figure 1(a) shows the *CurrencySwap* 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 BNFES, the major Brazilian funding bank. In turn, BNFES 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

```

<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>
    <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>

```

(a)

```

<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>

```

(b)

**Figure 2. *SwapWorkspace* document at  $P_1$ , (a) before and (b) after the materialization of  $sc_5$ .**

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 [52].

During business negotiations, brokers can follow swap information for relevant contracts in a *SwapWorkspace* document, such as the one in Figure 2(a) (in a simplified AXML notation). Basically, the *SwapWorkspace* document contains the contract number, the company name and its swap status at BNFES, 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 2(b).

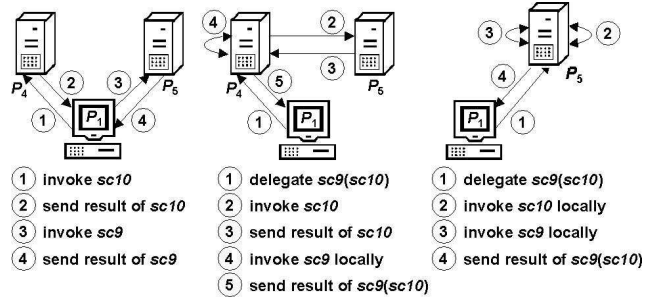
**Materializing AXML data.** The “sc” elements of the *SwapWorkspace* document refer to Web services that are provided by several different peers (see Figure 1(b)). When a service call is invoked at a peer, the system can lookup for its possible service providers, and then choose the best peer to execute 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(a), but only the result of  $sc9$  is required to build *SwapWorkspace*.

Figure 3 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 particularly 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 transferring 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 alternatives grows dramatically even for restricted scenarios. For instance, in our simple *CurrencySwap* example, there are at least  $1.898 \times 10^{16}$  alternative materialization plans<sup>1</sup> for the *SwapWorkspace* document. 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.

<sup>1</sup>Considering only the possible combinations of peers for service executions and invocations.



**Figure 3. Some alternatives for AXML materialization.**

Some heuristics can significantly reduce this search space. In our example, we can apply the *Divide&Conquer* (D&C) heuristic [44], which partitions the document materialization into independent tasks. This would reduce the search space to approximately  $1.265 \times 10^{14}$  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). In Section 5.4, we provide some formula to estimate the number of AXML materialization alternatives.

Clearly, such optimization delays are not acceptable when materializing an AXML document. Exhaustive and optimal search methods are often unfeasible in AXML settings. Therefore, it is imperative to reduce the search space of materialization alternatives to a manageable size, regardless of the size of the problem. Greedy and opportunistic approaches are typical solutions for complex workflow planning. Nonetheless, they are insufficient for AXML materialization since they are based on local decisions, which cannot explore delegation to reduce communication costs. In this paper, we present a cost-based strategy to enumerate and rank AXML materialization plans, based on a hybrid metaheuristic. The core of our strategy is a dynamic algorithm that progressively generates and evaluates these plans, thus avoiding to explore complete search spaces, while still considering relevant performance properties.

### 3 AXML Basics

A typical AXML environment involves a P2P system, whose peers can execute and/or invoke Web services, and an AXML document model that combines XML data and service calls. In this section, we present the system architecture and the document model (borrowed from [3, 5, 39]) defined for the ActiveXML framework [13], an open-source P2P platform that supports AXML documents.

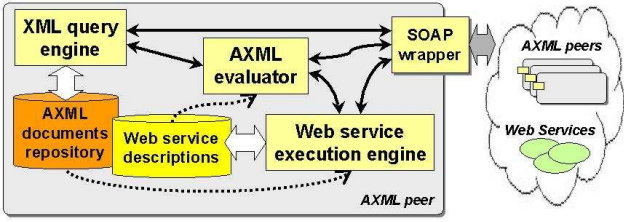


Figure 4. Outline of an ActiveXML peer.

**System architecture.** The architecture of an ActiveXML peer is depicted in Figure 4. Basically, a peer holds a repository to store local AXML documents, and a catalog of Web service descriptions. The XML query engine provides access to the AXML documents repository, and (when necessary) can request the AXML evaluator to materialize some AXML content. In turn, the AXML evaluator triggers embedded service calls, and updates the corresponding AXML documents accordingly. The invocation of triggered service calls is handled by the Web service execution engine. Observe that each ActiveXML peer can act both: as a *server*, by providing Web services; and as a *client*, when invoking service calls that are embedded in local documents. In particular, an ActiveXML peer can provide a distinguished class of Web services, called *declarative services*, which consist of parameterized queries over local documents.

**AXML document model.** An AXML document is 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*. The latter can encode all the information required to access a Web service (URL, operation name, etc.). When a service call is activated, this information is used by the ActiveXML peer to actually invoke the service. Consider the following disjoint infinite sets of labels:  $\mathcal{D}$  of data values,  $\mathcal{E}$  of element names, and  $\mathcal{S}$  of service names. More formally, an AXML document  $d$  is denoted by the expression  $\langle \tau, \lambda, \prec \rangle$ , where  $\tau$  is an ordered tree with a finite set  $N$  of nodes and a distinguished node  $root$ ,  $\lambda : N \rightarrow \mathcal{E} \cup \mathcal{D} \cup \mathcal{S}$  is a labelling function for nodes in  $N$ , such that only leaf nodes are mapped to  $\mathcal{D}$ , and  $\prec$  associates with each node in  $N$  a total order on its children. An important subset of  $N$  is that of all the service call nodes of  $d$ , called  $SC_d$ ; for every node  $v$  in  $N$ , if  $\lambda(v) \rightarrow \mathcal{S}$ , then  $v \in SC_d$ .

The children of a service call node stand as its *input parameters*; in the example of Figure 2(a), they are represented by “param” elements (we omit this element if the service call has only one parameter, such as in sc2). When a service call node is invoked, its respective subtrees are passed to the Web service, and the invocation result replaces the call node in the document. Strictly speaking, service

call nodes are not eliminated from the AXML document, but kept for possible further invocations [4]. Nonetheless, our analysis focuses on a set of service calls to be evaluated *at a given moment in time* in order to materialize an AXML document. Therefore, without loss of generality, we can ignore the fact that service call nodes may remain in the document after their invocation, as long as they do not need to be invoked again during the materialization process. Figure 2 shows the *SwapWorkspace* document: (a) before and (b) after the invocation of the service call sc5. Furthermore, elements resulting from new service invocations have a special *timestamp* attribute to indicate the current snapshot of the document contents. Thereby, users may choose to consider either all the previous results in the document or only the last invocation results for feeding the new service requests (namely, which data elements are going to be used as service inputs).

Notice that both service input parameters and results may be AXML data (i.e. contain service call nodes). Moreover, the input parameters of a service call may be either explicitly provided by nested AXML elements or specified by XPath expressions [65], such as the “company” parameter of sc3 in Figure 2(a). An input parameter defined by an XPath expression represents a query on the elements of the AXML document; such a query is evaluated whenever the service call is invoked and its result is passed in the service request as subtrees of the parameter element.

**Materialization strategies in ActiveXML.** The materialization of some AXML data can be either explicitly requested by the user or implicitly triggered by queries that requires the (materialized) content of a document. The materialization process always starts at the peer that hosts the corresponding document, which we call the *master peer* (e.g., peer  $P_1$  for the *SwapWorkspace* document in Figure 1). Currently, ActiveXML peers consider neither flexible service execution (with abstract service references) nor system performance to reason about possible evaluation strategies for AXML materialization. Instead, peers can adopt only a *type-driven strategy* [39], which is defined by the user and derived from the analysis of the input and output types of the requested Web services.

For example, to materialize the service call sc9 in Figure 2(a), the user may specify an expected schema for its result, stating that  $P_1$  accepts only regular XML data for an answer to sc9. Then, the user has to ask  $P_1$  to invoke sc9 (and its entire subtree) at either  $P_4$  or  $P_5$ , for instance. That is,  $P_4$  (or  $P_5$ ) would have to invoke both sc9 and sc10, thus sending back to  $P_1$  only the result of sc9. Observe, however, that the user has to determine exactly which peers are going to participate in the materialization process, and how they should exchange AXML data. In fact, the work of [39] makes room for a performance-based approach to guide the materialization process.

For query processing with AXML documents, selection predicates can be used to avoid the execution of irrelevant service calls, as proposed in [3]. In this case, the goal is to materialize only the intensional elements that contribute to the query result. Furthermore, when declarative services are used, some selection predicates can be *pushed* to be processed at the service providers.

## 4 Service Invocation Constraints in AXML

The relationships between the service calls of an AXML document encode some *explicit and implicit constraints on the invocation of its service call nodes*. 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. In this Section, we present a canonical formalism to represent the invocation constraints of an AXML document into *dependency graphs*, which are the basis of our optimization strategy.

An important type of constraint is expressed by *invocation dependencies*, which occur when a service call takes other calls as input parameters. These are *data dependencies*; there may also exist *service dependencies* [44], namely when some information about the Web service pointed by a call is determined by the result of another call. We focus here on data dependencies, which are defined in Section 4.1. AXML documents may also contain *invocation consequences*, introduced in Section 4.2. We use these different types of constraints to define the *dependency graph* of an AXML document in Section 4.3, where we also analyze issues related to validity and redundancy of these constraints. Going further, we define in Section 4.4 efficient mechanisms to update the dependency graph with intensional answers at runtime.

### 4.1 Precedence Constraints

Invocation dependencies represent precedence constraints on the materialization of some AXML data. Namely, one can determine that some service calls must be invoked *before* a given call, because the latter consumes their results. There are two types of such parameters: *concrete* and *non-concrete*, which are defined in the following.

**Concrete parameters.** A first class of AXML invocation dependencies is attained by *nesting a service call in a parameter of another service call* – namely, by specifying a concrete parameter. For example, in Figure 2(a), service calls sc2 and sc3 are nested as input parameters of sc1. Such a node nesting entails that the call to CheckSwapStatus *depends on* both the call to GetCurrentSwaps and GetSwapLimit. Next, we define this relationship.

**DEFINITION 1** Let  $d$  be an AXML document, and  $v_i$  be a node in  $SC_d$ . A node  $v_j$  is a concrete parameter of  $v_i$  iff:

- $v_j \in SC_d$  and  $v_j$  is a descendant node of  $v_i$  in  $d$ ; and
- there is no node  $v_x \in SC_d$  such that  $v_x$  is both a descendant of  $v_i$  and an ancestor of  $v_j$  in  $d$ .

We refer to the set of all the concrete parameters of  $v_i$  as  $\nabla(v_i)$ , where  $\nabla(v_i) \subseteq SC_d$ .

Furthermore, we use the notation  $|A|$  for the cardinality of set  $A$ ; thus, the number of nodes of  $SC_d$  is denoted by  $|SC_d|$ . Checking whether a service call node is a concrete parameter of another call is rather trivial, and it can be done in advance by a static analysis, when the document is loaded and/or updated by an ActiveXML peer.

**Non-concrete parameters.** Users may also specify input parameters that are provided by XPath queries, such as the “foreign\_currency” parameter of sc7 in Figure 2(a). In this case, sc4 is not explicitly nested in sc7; it has actually no ancestor relationship with sc7. Yet, materializing sc7 depends on the invocation of sc4, since sc4 contributes to the input of sc7. This yields another class of invocation dependencies, as follows.

**DEFINITION 2** Let  $d$  be an AXML document,  $v_i$  a node in  $SC_d$ , and  $Q$  an XPath expression within an input parameter of  $v_i$ . A node  $v_j$  is a non-concrete parameter of  $v_i$  iff:

- $v_j$  is in the result of  $\Phi(Q)$ , where  $\Phi(Q)$  is a function that returns the set of all the service calls that contribute to  $Q$  in  $d$ , such that  $\Phi(Q) \subseteq SC_d$ ; and
- $v_j \neq v_i$  and there is no node  $v_x$  in  $\Phi(Q)$  such that  $v_x$  is an ancestor of  $v_j$  in  $d$ .

The term  $\vec{\nabla}(v_i)$  denotes the set of all the non-concrete parameters of  $v_i$ , where  $\vec{\nabla}(v_i) \subseteq \Phi(Q)$ .

Conversely to concrete parameters, a sophisticated analysis may be necessary to compute the set of service calls that contribute to a given XPath expression, such as in [3]. Notice that the XPath expression has to be evaluated first in order to obtain concrete inputs, and then invoke the service. Namely, non-concrete parameters can only be determined *after their respective query is evaluated*. Also, they may imply dependency cycles, possibly leading to an invocation deadlock. To detect this problem, we define in Section 4.3 some validity criteria for the dependency graph of an AXML document. When these cycles are detected, we assume they are either broken prior to our optimization analysis, as in [3], or the materialization process is aborted.

**Transitive dependencies.** In Definition 2, we disregard some redundant dependencies that may occur in the same

attribute. Basically, such a redundancy happens when some service calls in  $\Phi(Q)$  are nested in the AXML document. For example, suppose the “text” and “input\_format” parameters of sc9 in Figure 2(a) are rewritten as:

```
<param name="text"><xpath> // </xpath></param>
<param name="input_format">XML</param>
```

Then, potentially all the other service calls in the document can contribute to the result of the XPath expression in “text” (assuming that self-dependencies and cycles are properly eliminated). In particular, both sc1 and sc2 are in  $\Phi(//)$ . However, according to Definition 2, only sc1 is a non-concrete parameter of sc9, since sc2 is nested in sc1. The rationale behind this simplification is that the redundant parameter no longer exists (in its active form) after its first execution, which is triggered by the dependencies of its closest outer service call. In Section 4.3, we use this principle to reduce redundancy of an entire AXML document.

Redundant dependencies are essentially related to the transitivity of service call parameters, which we define next.

**DEFINITION 3** *Given two service call nodes  $v_i$  and  $v_j$ , we say that  $v_j$  is an intensional parameter of  $v_i$ , denoted by  $v_j \rightarrow v_i$ , iff  $v_j \in \nabla(v_i) \cup \vec{\nabla}(v_i)$ . The term  $\text{fanIn}(v_i)$  represents the number of intensional parameters of  $v_i$ , which corresponds to  $|\nabla(v_i) \cup \vec{\nabla}(v_i)|$ . Similarly, the term  $\text{fanOut}(v_j)$  denotes the number of nodes which have  $v_j$  as an intensional parameter.*

**DEFINITION 4** *Given two service call nodes  $v_i$  and  $v_j$  of an AXML document  $d$ , the node  $v_j$  is a transitive parameter of  $v_i$ , denoted by  $v_j \xrightarrow{*} v_i$ , iff there is a path of the form  $v_j \rightarrow v_{x1} \rightarrow \dots \rightarrow v_{xn} \rightarrow v_i$  in  $d$ , with  $n \geq 1$ .*

The transitive parameter with the largest path length determines the *abstract critical path* of an AXML document, that is the longest path of sequential service invocations. Notice we are considering only the number of service call nodes in the path. When materializing a document, another important path is that with the longest execution time.

## 4.2 Consequence Constraints

Another way to express invocation constraints in AXML documents is specifying a “followed-by” attribute in the service call elements. For example, in Figure 2(a), this clause constrains the invocation of the service call sc1 with respect to sc5. Namely, once sc5 is invoked, an invocation of sc1 must be triggered *immediately after* (as a consequence of) the invocation of sc5. However, invoking sc1 should not disturb sc5.

Notice the semantics of consequence constraints differs significantly from that of invocation dependencies. While these dependencies are intrinsically associated to data flows

to feed Web service inputs, consequence constraints denote an invocation sequencing of service calls that do not necessarily have data dependencies between each other.

**DEFINITION 5** *An AXML document with collateral calls is of the form  $\langle d, \hookrightarrow \rangle$ , where  $d$  is a regular AXML document and  $\hookrightarrow$  is an one-to-one partial mapping of nodes in  $SC_d$  to nodes in  $SC_d$ . Given two service call nodes  $v_i$  and  $v_j$ ,  $v_i \neq v_j$ , if  $v_i \hookrightarrow v_j$ , then each invocation of  $v_i$  automatically triggers an invocation of  $v_j$ , which is referred to as a collateral call of  $v_i$ .*

For simplicity, we assume that a service call may be associated with only one collateral call. Observe that collateral calls may trigger other calls, which in turn may have their own collateral calls, and so on. Hence, the invocation consequences of an AXML document may determine transitive constraints, as defined next.

**DEFINITION 6** *Let  $v_i$  and  $v_j$  be two service call nodes of an AXML document  $\langle d, \hookrightarrow \rangle$ . The node  $v_j$  is a transitive collateral call of  $v_i$ , denoted by  $v_i \xrightarrow{*} v_j$ , iff there is a path of the form  $v_i \hookrightarrow v_{x1} \hookrightarrow \dots \hookrightarrow v_{xn} \hookrightarrow v_j$  in  $d$ , with  $n \geq 1$ .*

It is worth mentioning that one cannot express collateral calls by using only invocation dependencies. Invoking an outer call always implies the previous materialization of its dependencies. On the contrary, executing nodes that are collateral calls should not disturb their respective counterparts. In general, the distinction between process dependencies and consequences can be also found in workflow specifications, such as the **precede** and **enable** constructs of the *ActivityFlow* language [34]. This is also similar to the *enabling flows* of the Vortex system [30].

The invocation constraints of an AXML document correspond to some basic workflow patterns [54]. More precisely, invocation dependencies can be mapped to the *synchronization* control pattern. In particular, shared dependencies can also produce *parallel splits*. On the other hand, collateral calls correspond to the *sequence* control pattern. Additionally, they allow multiple instances of services calls without synchronization, which corresponds to the structural pattern 12 of the classification in [54].

Observe that, when optimizing the materialization of an AXML document, intensional parameters represent essentially some clustering criteria to reduce communication costs. Conversely, collateral calls correspond to the sequential invocation of new service calls possibly without implicit data flows.

## 4.3 Dependency Graph

We present here a formalism based on *directed acyclic graphs* (DAG) to express the invocation constraints of an

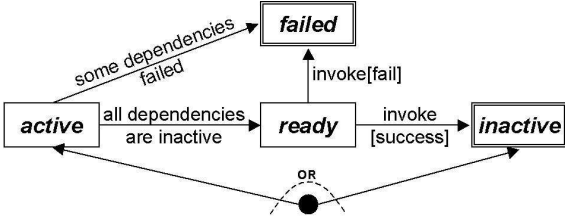


Figure 5. Statechart of service call nodes.

AXML document. We use this abstract representation rather than the AXML tree mainly because non-concrete parameters and collateral calls result into dependencies that can be arbitrarily complex. Such relationships cannot be naturally expressed by tree structures. Furthermore, reasoning about invocation constraints, such as verifying shared dependencies and non-concrete (or collateral) transitivity, is done at this level.

In our analysis of invocation dependencies, we consider that most of the Web services do not understand AXML data, and thus cannot process correctly intensional (concrete and non-concrete) parameters. For regular Web services, these parameters must be invoked before executing the service call. On the other hand, such a invocation might be unnecessary, or even incorrect, for AXML-enabled services. Thus, the user has to specify whether intensional parameters must be invoked *a priori*. In practice, we do not distinguish AXML-enabled Web services, and the execution of intensional parameters is enabled by setting an attribute of their respective “sc” elements. Hence, our analysis is focused on intensional parameters that are always invoked *a priori*.

Figure 5 shows the possible states of a service call node during AXML materialization; the states “inactive” and “failed” are shown with double rectangles because they are terminal (i.e., when nodes become stable). Service call nodes start with either state “active” or “inactive”. A node may be initially set as “inactive” due to many reasons, such as bad AXML specification, user selection, or because some preliminary analysis has considered the invocation of this node unnecessary for the document materialization, as in [3]. On the other hand, an active node becomes “ready” when all of its intensional parameters are “inactive”. If ready, a node can be invoked, and then it reaches either the “inactive” state or the “failed” state, according to the success or fail, respectively, of its invocation. Also, a node moves to the “failed” state if the invocation of some of its dependencies fails.

**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. Service call nodes with persistent results are defined next.

**DEFINITION 7** A node  $v_i$  is a first-level service call of an AXML document  $d$  iff  $v_i \in SC_d$  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  $\xi$  the set of all the first-level service calls of  $d$ , such that  $\xi \subseteq SC_d$ .

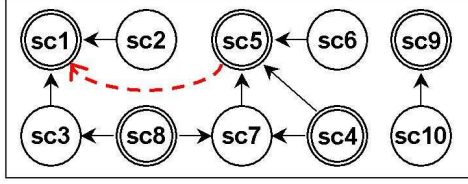
First-level calls can be found statically by a straightforward procedure. In Figure 2(a), the nodes sc1, sc4, sc5, sc8, and sc9 are first-level service calls. Such a distinction is relevant for the optimizer to assess the costs of delegating parts of a document to be materialized by other peers.

**Dependency graphs defined.** We are now able to formally define the *dependency graph* of an AXML document. This graph concisely represents all the synchronization constraints that must be enforced on the service calls within the document. It is a central input to our optimization effort. Although this graph explicitly refers to invocation dependencies, it also encompasses collateral calls. Our emphasis is on intensional parameters because they reflect essential aspects of the AXML document, while collateral calls come from optional annotations on the service call nodes.

Basically, a dependency graph can be obtained by static analysis. Since users may specify AXML documents with cyclic dependencies and infinite execution loops, first we introduce a general definition for the dependency graph. After that, we restrict the valid dependency graphs to acyclic instances. For that purpose, we have to consider a particular definition of cycles in a dependency graph, since invocation dependencies and collateral calls represent distinguished types of edges between nodes, and they can contribute in a tricky way to form cycles.

**DEFINITION 8** The dependency graph  $\Delta$  of an AXML document  $\langle d, \hookrightarrow \rangle$  is denoted by the expression  $\langle \mathcal{G}, \otimes, \overrightarrow{E}, \epsilon \rangle$ , where  $\mathcal{G}$  is a directed graph with a set  $V$  of nodes,  $V = SC_d$ , 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 a intensional parameter of  $v_i$  in  $d$ . The subset  $\overrightarrow{E}$  denotes the collateral edges of  $E$ ; there is an edge  $v_i \hookrightarrow v_j$  in  $\overrightarrow{E}$  iff  $v_i \hookrightarrow v_j$  in  $d$ . The term  $\epsilon$  denotes a state function that maps each node in  $V$  into  $\{active, ready, inactive, failed\}$ .

Notice that a dependency graph encodes a *partial order* on the service calls of the respective AXML document. Figure 6 depicts the graph derived from the *SwapWorkspace* document in Figure 2(a). Nodes are represented by circles labelled with service call IDs, where double-line circles are



**Figure 6.** Dependency graph of the *Swap-Workspace* document.

persistent nodes. Dashed arrows indicate collateral edges. We assume all nodes are in the “active” state.

**Graph validity.** Users may specify AXML documents with arbitrary dependency graphs, possibly containing *invocation deadlocks* and *infinite execution loops*. We consider that such documents are invalid, since they cannot be properly materialized.

Figure 7 shows the basic invalid combinations of invocation constraints between service call nodes (illustrated with “scX”, “scY”, “scW”, and “scZ”). Loosely speaking, the subgraph of simple edges of a valid graph must be acyclic, to avoid dependency cycles such as in Figure 7(a). Also, the subgraph of collateral edges should be acyclic, thus preventing the pattern of Figure 7(b). Moreover, an infinite execution loop occurs if either a node is a collateral call of some of its dependencies (Figure 7(c)) or an alternate sequence of simple and collateral edges converge to a cycle, as shown in Figure 7(d). Notice the forbidden patterns of Figure 7 also apply for transitive constraints; for example, if  $v_x \xrightarrow{*} v_y$ , then we cannot have  $v_y \xrightarrow{*} v_x$  in the graph.

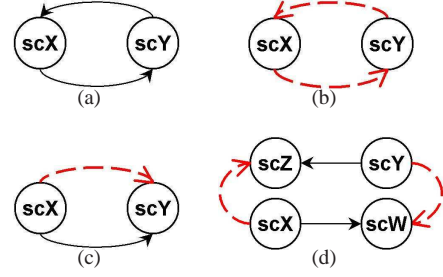
In summary, about the interactions between simple and collateral edges, we have that:

- they do not concur to form deadlock cycles, since collateral calls do not involve data dependencies; but
- their combination may contribute to form infinite execution loops, as shown in Figures 7(c) and 7(d). Hence, to check a dependency graph for execution termination, we have to consider collateral edges as simple edges with opposite direction.

From the above, let the *sequencing component* of a dependency graph be the graph resulting from replacing all the collateral edges by inverted simple edges. Then, we can summarize these validity criteria in the following.

**DEFINITION 9** A dependency graph  $\Delta$  is valid iff:

- the subgraph with all the simple edges of  $\Delta$  is an acyclic digraph; and
- the sequencing component of  $\Delta$  is an acyclic digraph.



**Figure 7.** Basic invalid invocation constraints of a dependency graph.

The first bullet of Definition 9 avoids invocation deadlocks, while the second addresses infinite executions. We clearly distinguish between deadlock and termination detection because we consider the user may need a relaxed notion of graph validity, where termination is guaranteed by some pre-determined fixpoint.

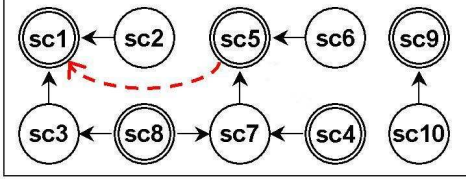
Graph validity can be checked when documents are created or altered by the user. However, such a verification may also be required during AXML materialization, at runtime, in case of intensional answers. Checking for validity can be done in  $O(|V|^3)$ , based on the time complexity of computing the transitive closure of the graph using the well-known Warshall’s algorithm [11]. For AXML optimization, we consider only valid graphs.

**Redundant dependencies.** An AXML document may derive a dependency graph with redundant dependencies. Notice that we can suppress the edge  $sc4 \rightarrow sc5$  in Figure 6, since  $sc4 \xrightarrow{*} sc5$  (from  $sc4 \rightarrow sc7$  and  $sc7 \rightarrow sc5$ ) makes it redundant. The resulting reduced graph is shown in Figure 8. Previously, we eliminated redundancy of non-concrete parameters within the same input parameter of a service call. We extend this idea to the entire document.

Following [50], we say that two graphs are equivalent if they are *bisimilar*. However, [50] considers that only edges are labeled. Here, we also consider node labels, and we focus bisimulation on redundant invocation dependencies, as stated next.

**DEFINITION 10** A dependency graph  $\Delta_a$  subsumes another dependency graph  $\Delta_b$ , denoted by  $\Delta_a \subset \Delta_b$ , iff:

- $V_a = V_b$ ,  $\otimes_a = \otimes_b$ ,  $\overrightarrow{E}_a = \overrightarrow{E}_b$ ,  $\epsilon_a = \epsilon_b$ , and  $E_a \subset E_b$ ; and
- there are some nodes  $v_i$  and  $v_j$  in  $V_b$ , such that if both  $v_i \rightarrow v_j$  and  $v_i \xrightarrow{*} v_j$  are in  $E_b$ , then  $v_i \rightarrow v_j$  is not in  $E_a$ . The edge  $v_i \rightarrow v_j$  is said a redundant dependency of  $\Delta_b$ .



**Figure 8. Reduced version of the dependency graph of Figure 6.**

The minimum reduced graph of  $\Delta_a$  is a graph  $\Delta_a^{MR}$  such that  $\Delta_a^{MR} \subset \Delta_a$  and  $\Delta_a^{MR}$  has no redundant dependency. Moreover, we say that the graphs  $\Delta_a$  and  $\Delta_b$  are bisimilar if  $\Delta_a \subset \Delta_b$  or  $\Delta_b \subset \Delta_a$ .

Several redundant dependencies may occur in an AXML document, and their reduction is important to avoid unnecessary optimization efforts. Notice bisimulation is a formal basis for the elimination of redundant AXML dependencies. Therefore, the minimum reduced graph is a *canonical representation* of the invocation constraints of an AXML document, as follows.

**PROPOSITION 1** *Let  $\Delta$  be a dependency graph with redundant dependencies. There is at least one reduced graph  $\Delta^R$  which is equivalent to  $\Delta$  and at most one minimum reduced graph  $\Delta^{MR}$ .*

*Proof:* By definition of graph bisimulation, if  $\Delta$  has redundant dependencies of the form  $v_i \rightarrow v_j$  and  $v_i \xrightarrow{*} v_j$ , then we can eliminate at least one  $v_j \rightarrow v_i$ , and the reduced graph  $\Delta^R$  remains equivalent to  $\Delta$ . Furthermore, one can always reduce  $\Delta$  by applying a sequence of one-edge eliminations, such that if  $\Delta^R$  cannot be further reduced, then  $\Delta^R = \Delta^{MR}$ . The ultimate set of eliminated edges is the same regardless of the order of the eliminations steps, and therefore there is only one possible  $\Delta^{MR}$ .

A dependency graph with only concrete parameters is reduced by definition, since redundancy is intrinsically related to shared dependencies (which occur due to non-concrete parameters). On the other hand, for non-concrete parameters, graph reduction has time complexity  $O(|V|^3)$ , based on the transitive closure of the graph. However, only nodes with  $fanOut \geq 1$  can be origin of redundant edges, and the tight bound is actually  $O(|\otimes| \times |V|^2)$ . Notice that non-concrete parameters are, in general, quite expensive to be handled in AXML materialization.

Unless stated otherwise, hereafter we will use the terms dependency graph and minimum reduced graph interchangeably.

**Exit points.** Nodes that do not have outgoing simple edges (i.e., with  $fanOut = 0$ ) are particularly important for the

materialization of a dependency graph; they are said the *exit points* of the graph. They can be used to unfold a graph into spanning trees, thus enabling the optimizer to break the materialization problem into smaller parts and thereby to reduce the overall complexity. Also, they represent points where the materialization process finishes (i.e., after materializing them and properly triggering their collateral calls, the materialization process should stop). For example, the nodes sc1, sc5, and sc9 are the exit points of the graph in Figure 6.

**LEMMA 1** *Every valid non-null dependency graph  $\Delta$  has at least one exit point.*

*Proof:* The crux here is that, although collateral edges may concur with invocation dependencies to cause cycles, they have no influence on exit points (to determine the  $fanOut$  of a node). Thus, only simple edges must be considered. First, suppose  $\Delta$  is a singleton, namely  $V = \{v\}$ . Then,  $fanOut(v) = 0$  by definition, and  $v$  is an exit point. Consider now that  $\Delta$  has  $|V|$  nodes, with  $|V| > 1$ , such that each node has at least one simple outgoing edge. Pick any node  $v_1$  in  $\Delta$ ; since  $fanOut(v_1) > 0$ , there is a node  $v_2$  such that  $v_1 \rightarrow v_2$ . In turn,  $v_2$  also has an outgoing edge  $v_2 \rightarrow v_3$ , and so forth. The nodes in  $\Delta$  are finite, and this path cannot continue forever. At some point, this path leads to repeated nodes, thus constituting a cycle. Since  $\Delta$  is valid, hence an acyclic digraph w.r.t. simple edges, this is a contradiction. Therefore, there must exist at least one node without outgoing simple edges for  $\Delta$  to be valid.

Lemma 1 is important because it guarantees that, despite the complexity of the shared dependencies and collateral calls, the optimizer has always an exit point to start evaluating a valid dependency graph. Furthermore, assuming service executions always stop after some period of time, valid graphs are termination-safe, as shown in the following.

**PROPOSITION 2** *In a valid dependency graph  $\Delta$ , given any node  $v$ , either  $v$  is an exit point of  $\Delta$  or there is a finite path between  $v$  and some exit point  $v_e$  of  $\Delta$ , such that  $v \xrightarrow{*} v_e$ . Moreover, all the transitive collateral calls that originate directly or indirectly from nodes in  $v \xrightarrow{*} v_e$  (including  $v$  and  $v_e$ ) have a finite path.*

*Proof:* If  $\Delta$  is a singleton, then  $v$  is an exit point by definition. On the other hand, consider  $\Delta$  has  $|V|$  nodes, with  $|V| > 1$ , such that each node has an arbitrary number of both simple and collateral outgoing edges. First, from Lemma 1, we have that  $\Delta$  has at least one exit point. Furthermore, picking any node  $v$  in  $\Delta$ , if  $fanOut(v) = 0$ , then  $v$  is an exit point. Otherwise, there is at least one node  $v_x$  such that  $v \rightarrow v_x$ . In turn, either  $v_x$  is an exit point or it also has an outgoing edge  $v_x \rightarrow v_y$ , and so forth. Since  $\Delta$  is cycle-free w.r.t. simple edges, by induction this path has

to lead to some exit point  $v_e$ . Moreover,  $|V|$  is finite, hence the length of the transitive dependency  $v \xrightarrow{*} v_e$  is also finite. Consider now the sequencing component of  $\Delta$ . Any collateral call that is triggered (directly or not) by some node in  $v \xrightarrow{*} v_e$ , including  $v$  and  $v_e$ , must be in a sequencing path that leads to  $v_e$  (recall collateral edges are inverted in the sequencing component). Notice that such a path does not end with  $v_e$  if this node has a collateral call. However, the sequencing component of  $\Delta$  is an acyclic digraph with a finite number of nodes, and therefore all of its sequencing paths are finite.

Notice Proposition 2 concerns the *snapshot semantics* of a dependency graph. That is, it does not consider the effects of occasional intensional answers, but rather the current graph topology. Next, we discuss how graphs can evolve.

#### 4.4 Dynamic Graph Updates

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 user decide whether 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 [32].

In the *CurrencySwap* example (Figures 1 and 2), suppose the repository of PDF files at  $P_4$  is down, and invoking the service call `sc10` at  $P_4$  returns the following result:

```
<sc id="11" service="Ps2Pdf">
  <sc id="12" service="GetContractPS"/></sc>
```

Then, to materialize the *SwapWorkspace* document, the master peer has now to invoke both `sc11` and `sc12`, to gather input data for the “text” parameter of `sc9`. Moreover, this may require discovering information about the peers that provide the Web services referred by `sc11` and `sc12`.

Intensional answers may significantly change the AXML document – indeed they raise several tough problems for AXML optimization:

- as these answers arrive, the dependency graph has to be updated (at runtime) with the new service calls to allow checking for validity;
- it may be necessary to refresh the service directory with information about the requested Web services. Also, optimizing the newcomers may involve gathering some statistics and costs parameters;
- since the specification of the AXML document is altered, involving new data flows and possibly other ser-

vice providers, previous optimization choices may be contradicted; and

- some Web services might return undesirable or infinite intensional answers.

To guarantee a correct AXML materialization (in terms of data types) and to ensure its termination, the ActiveXML system implements a powerful typing mechanism for service call results [39]. Intensional results are recursively materialized until either the document satisfies a specific type or a fixpoint is reached. In this paper, we rather analyze intensional answers from a performance-oriented perspective. Hence, we consider issues related to dynamically updating dependency graphs with these answers, and their impact on the optimization process.

**Connecting intensional answers.** The idea behind intensional answers is that the AXML document may “evolve” during the materialization process. Consequently, each intensional answer requires an update operation on the dependency graph. Recall that intensional nodes remain either inactive or failed after invoked. A dependency graph update is defined as follows.

**DEFINITION 11** *Let  $d$  be an AXML document,  $\Delta$  its dependency graph and  $v_i$  a node of  $\Delta$ . Suppose  $d_u$  is the AXML data returned by the invocation of  $v_i$ , which is used to update  $d$ . If the graph  $\Delta_u$  obtained from  $d_u$  is not null, then  $\Delta_u$  is said an intensional answer of  $v_i$ .*

**DEFINITION 12** *An update operation  $u$  is a triple  $\langle \Delta, v_i, \Delta_u \rangle$ , where  $\Delta$  is a dependency graph containing the node  $v_i$ , and  $\Delta_u$  is an intensional answer of  $v_i$  to be inserted into  $\Delta$ . The operation  $u$  transforms  $\Delta$  into a graph  $\Delta' = \langle \mathcal{G}', \otimes', E', \epsilon' \rangle$  according to the Update function in Figure 9(a). The node  $v_i$  is said the origin of  $u$ .*

To update the dependency graph, intensional answers must be properly connected to it, such that invocation constraints and their transitive relationships are preserved. In general, propagating these constraints may be very costly. Fortunately, in practice, we find some heuristics that are particularly handy in this context. Based on some properties of the AXML document model, we make the following assumptions:

- (i) *New service call nodes are not affected by the collateral relationships of the AXML document.* The intuition is that collateral calls point to specific service calls references. Also, they represent new instances of service invocations, regardless of previous execution results. Therefore, the newcomers neither inherit the collateral relationships of their origin node nor are referred by pre-existing nodes;

```

1 function Update( $\Delta, v_i, \Delta_u$ ):  $\Delta'$ 
2 {Update  $\Delta$  at node  $v_i$  with  $\Delta_u$ .}
3 begin
4   let  $V' = V \cup V_u \quad \xrightarrow{\quad} \xrightarrow{\quad} \xrightarrow{\quad}$ 
5   let  $E' = E \cup E_u$  and  $E' = E \cup E_u$ 
6   if  $v_i \in \xi$  then {first-level call}
7      $\xi' = \xi \cup \xi_u$ 
8   else  $\xi' = \xi$ 
9   end if
10  if  $\text{fanOut}(v_i) > 1$  then {shared dependency}
11     $\otimes' = \otimes \cup \otimes_u \cup \xi_u$ 
12  else  $\otimes' = \otimes \cup \otimes_u$ 
13  end if
14  ConnectSubGraph( $\Delta_u, v_i, \Delta'$ )
15  Re-evaluate non-concrete parameters of  $\Delta'$ 
16  for each  $v_x$  in  $V'$  do
17    set  $e'(v_x)$  according to  $e(v_x)$  or  $e_u(v_x)$ 
18  end for
19  set  $e'(v_i) = \text{inactive}$ 
20  return  $\Delta'$ 
21 end

21 procedure ConnectSubGraph( $\Delta_u, v_i, \Delta'$ )
22 {Connect the subgraph  $\Delta_u$  to  $\Delta'$  according to
23  $v_i$ 's dependencies.}
24 begin
25   let  $Out$  be the set of outgoing “ $\rightarrow$ ” edges of  $v_i$ ,
26   such that  $|Out| = \text{fanOut}(v_i)$ 
27   for each  $v_x$  in  $\xi_u$  do
28     for each  $v_i \rightarrow v_y$  in  $Out$  do
29       Add  $v_x \rightarrow v_y$  to  $E'$ 
30     end for
31   end for
32 end

```

(a)

```

1 procedure ConnectSubgraphWithPipe( $\Delta_u, v_i, \Delta'$ )
2 {Connect the subgraph  $\Delta_u$  to  $\Delta'$  according to
3  $v_i$ 's dependencies, using a pipe node.}
4 begin
5   Add a new node pipe to  $V'$ 
6   for each  $v_x$  in  $\xi_u$  do
7     Add  $v_x \rightarrow \text{pipe}$  to  $E'$ 
8   end for
9   for each  $v_i \rightarrow v_y$  in  $E$  do
10    Add pipe  $\rightarrow v_y$  to  $E'$ 
11  end for
12  if  $v_i \in \xi$  then
13    Add pipe to  $\xi'$ 
14  end if
15  if  $\text{fanOut}(v_i) > 1$  then
16    Add pipe to  $\otimes'$ 
17  end if
18 end

```

(b)

**Figure 9. Algorithm to (a) update  $\Delta$  and (b) connect  $\Delta_u$  through a *pipe* node.**

- (ii) *Intensional answers do not contain non-concrete parameters referring to nodes from other parts of the AXML document, because Web services are not necessarily aware of the document contents, and these parameters involve context-dependant specification;*
- (iii) *Analogously, the collateral calls of intensional answers refer only to newcomers; and*
- (iv) *New nodes do not depend on the intensional parameters of the origin call, since invocation results either replace the entire subtree of their origin node or are placed as their siblings in the document tree.*

From these points, we can consider that intensional answers are *loosely coupled* with the dependency graph. Therefore, only the outgoing simple edges of the origin service call are used to connect intensional answers, as encoded in the algorithms of Figure 9.

We show an example of connecting intensional answers to a dependency graph in Figure 10. In this example, a service call node “scX” is invoked and returns some arbitrary intensional answer represented by  $\Delta_u$  (Figure 10(a)). Then, for each outgoing edge of “scX”, we connect each first-level service call of  $\Delta_u$  to the respective target node. Namely, to the node depending on “scX”, as illustrated in Figure 10(b).

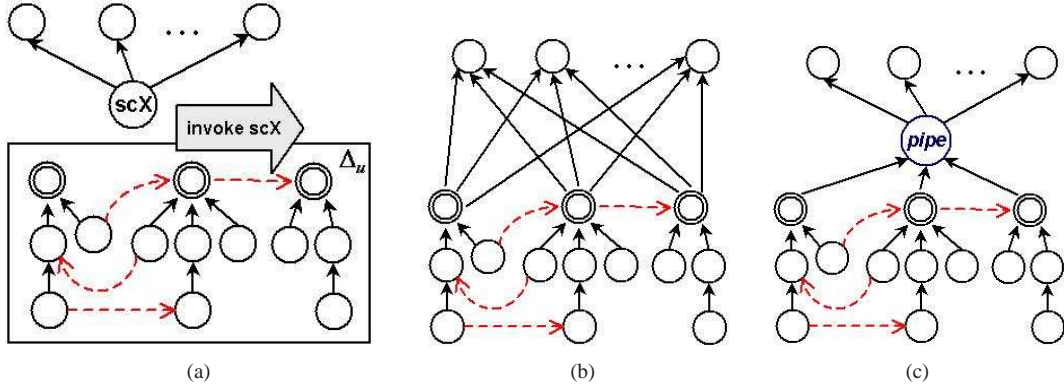
**Pipelined graphs.** Despite these simplifications, updating a dependency graph may involve creating many edges to link the new nodes. The first-level calls of the intensional answer must be joined to the service calls that depend on the origin node (see the double loop in lines 25 and 26 of Figure 9(a)). To improve the performance of this procedure, we introduce a special service call node in the resulting graph. This node refers to a very simple Web service called “*pipe*”, which is a generic service – that is, it can be executed by (potentially) any peer of the system.

The semantics of the *pipe* service is fairly simple: it receives the results from its invocation dependencies, and passes them to its dependant nodes. *Pipe* nodes are transparent with respect to the contents of their inputs; they neither produce new AXML data or eliminate any node. Their goal is to simplify the changes that are necessary to update a dependency graph.

**DEFINITION 13** *Given a dependency graph  $\Delta$  and an update operation  $u$ , the pipelined graph  $\Delta^p$  is the graph resulting from  $u$  such that a *pipe* node is used to connect  $\Delta_u$  into  $\Delta$ , according to the algorithm of Figure 9(b).*

Notice in Figure 10(c) that a *pipe* node concentrates the edges that connect an intensional answer, thus reducing the memory requirements of the resulting graph. If new nodes are connected directly to the dependency graph, then this number is given by:

$$\text{fanOut}(v_i) \times |\xi_u| \quad . \quad (1)$$



**Figure 10.** (a) The invocation of node “scX” returns an intensional answer  $\Delta_u$ ; (b) connecting  $\Delta_u$  to the dependency graph with the *Update* algorithm; and (c) using a *pipe* node.

On the other hand, with *pipe* nodes, the number of edges to link the intensional answer  $\Delta_u$  of a node  $v_i$  is bound to:

$$\text{fanOut}(v_i) + |\xi_u| \quad . \quad (2)$$

Pipelined graphs are particularly useful when the fan-out of the origin call is greater than 1. Also, they are helpful if it is necessary to relax the loosely-coupling assumption. That is, they can be used to “centralize” properties inherited by the new nodes from their origin. For instance, if one considers that intensional answers should inherit the collateral calls of their origin node. Furthermore, it can be proved that the graph  $\Delta^p$  is equivalent (according to Definition 10) to  $\Delta'$  augmented with the *pipe* node and its edges. Pipe nodes are inspired on the use of “*epsilon edges*”, which are introduced in [50].

**Update validity.** Updating a dependency graph requires checking whether the intensional answers lead to deadlocks or infinite collateral loops. Therefore, we state the following criterion.

**DEFINITION 14** An update operation  $u = \langle \Delta, v_i, \Delta_u \rangle$  is correct iff its resulting graph  $\Delta'$  is valid.

Observe that our approach allows to verify the validity of the resulting document before applying the changes to it, by reasoning based on dependency graphs. The heuristics used to update the graph also contribute to restrict such a verification to only the intensional answers. This is possible because we consider that new intensional nodes are not tightly connected to the AXML document.

**PROPOSITION 3** Let  $\Delta$  be a valid dependency graph and  $u = \langle \Delta, v_i, \Delta_u \rangle$  be an update operation. If  $\Delta_u$  is valid, then  $u$  is correct.

*Proof Sketch:* By definition (according to the *Update* algorithm), an update operation connects new nodes such that: they do not depend on existing service calls; and they do not trigger (or are triggered by) collateral calls of the original graph. Therefore, the newcomers can contribute to form neither invocation deadlocks nor collateral cycles with existing nodes. Since the graph is valid before the update, only its new portion needs to be checked. Hence, the validity of  $\Delta'$  is determined by  $\Delta_u$ .

**Lenient updates.** Another performance issue of handling intensional answers can be found at line 15 of the *Update* algorithm, in Figure 9(a). Whenever the AXML document changes, the input queries of non-concrete parameters may need to be re-evaluated. This approach is often quite time consuming. A less expensive alternative to do that consists in performing a *lenient analysis*: the re-evaluation is triggered only after a certain number of service call invocations (or intensional answers), thus allowing the document to evolve freely meanwhile. To proper formalize this idea, we first define the high-level semantics of instantiating an AXML materialization process as follows.

**DEFINITION 15** Let  $d$  be an AXML document, and suppose the dynamic sequence  $I_{\text{timeline}} = [v_1, \dots, v_n]$  is obtained by successively picking a ready node of  $SC_d$  and invoking it, until all nodes in  $SC_d$  are stable, where  $v_n$  is the last node invoked. A materialization phase  $\phi$  of  $d$  during  $I_{\text{timeline}}$  is an expression  $\langle \Delta_0, \wp, IT, U, \beta \rangle$ , where:

- $\Delta_0$  is the dependency graph of  $d$  before  $v_1$  is invoked;
- $\wp$  is the duration criterion of  $\phi$ , which is a boolean predicate defined on some properties of  $d$  and  $\phi$ , such as the maximum number of invoked service calls. The duration criterion is evaluated for each  $v_x$  in  $I_{\text{timeline}}$ ,  $1 \leq x \leq n$ , until it becomes true;

- $IT$  is the invocation trail of  $\phi$ , which consists of a sequence  $[v_1, \dots, v_\ell]$  of service calls invoked until  $\wp$  is evaluated to true, such that  $IT \subseteq I_{\text{timeline}}$ . The number  $\ell$  is said the duration of  $\phi$ , where  $1 \leq \ell \leq n$ ;
- $U$  is a sequence of update operations caused by the service invocations of  $IT$  on  $\Delta_0$ ; and
- $\beta$  denotes the backlog of  $\phi$ , namely a surjection from nodes in  $IT$  to new service call nodes in  $SC_d$ , such that  $\beta(v_x)$  returns the set of all the active nodes that were added to  $SC_d$  by intensional answers from the invocation of  $v_1$  to  $v_x$ , with  $v_x$  included and  $1 \leq x \leq \ell$ .

We denote by  $\Delta_x$  the snapshot graph of  $d$  after the invocation of  $v_x$ ,  $1 \leq x \leq \ell$ . The phase  $\phi$  is incomplete while  $\wp$  is false and  $SC_d$  has active nodes; otherwise,  $\phi$  is complete.

The semantics of a materialization phase is a period, measured in number of service call invocations, while the contents of an AXML document evolve. Notice  $I_{\text{timeline}}$  may be infinite. For example, suppose the AXML document contains a service call that always returns another call to the same service. Also, the duration criterion may be defined on physical properties of the materialization phase, such as the total time spent in the execution of the service calls. Furthermore, only active nodes are accounted in the backlog of a phase, since some new intensional nodes may become inactive (by being invoked) before a phase finishes.

**DEFINITION 16** A materialization phase  $\phi$  of length  $\ell$  is admissible iff the snapshot graph  $\Delta_\ell$  is valid. We say that  $\phi$  is  $x$ -admissible iff  $\Delta_\ell$  is invalid and there is a number  $x$ ,  $1 \leq x < \ell$ , which is the maximum value for that  $\Delta_x$  is valid.

**COROLLARY 1** Given any phase  $\phi$  with a valid  $\Delta_0$ , if all the update operations in  $U$  are correct, then  $\phi$  is admissible.

Based on materialization phases, we can determine checkpoints for the re-evaluation of non-concrete parameters, thereby deferring this analysis to a “reasonable” amount of changes. To perform such an analysis, we consider a *lenient update operation* by excluding line 15 from the *Update* algorithm (Figure 9(a)), and we extend Definition 15 as follows.

**DEFINITION 17** A lenient materialization phase is denoted by  $\phi^{\text{len}} = \langle \Delta_0, \wp, IT, U^{\text{len}}, \beta, \text{len} \rangle$ , where:

- The terms  $\Delta_0$ ,  $\wp$ ,  $IT$ , and  $\beta$  are defined as for  $\phi$ ;
- $U^{\text{len}}$  is a sequence of lenient update operations caused by the service invocations of  $IT$  on  $\Delta_0$ ; and

- $\text{len}$  is the leniency criterion of  $\phi$ , which is a boolean combination of predicates of the form “ $|A| \text{ op } a$ ”, such that  $a$  is an integer, the term  $\text{op}$  is in  $\{=, >, <, \leq, \geq\}$ , and the set  $A$  is in  $\{SC_d, IT, U^{\text{len}}, \beta\}$ . At each  $v_x$  in  $IT$ ,  $1 \leq x < \ell$ , this criterion is checked, and the re-evaluation of the non-concrete parameters of  $\Delta_x$  is triggered if  $\text{len}$  is true.

Additionally, the re-evaluation is always triggered after  $v_\ell$ .

Conversely to  $\wp$ , we restrict the definition of  $\text{len}$ , since a lenient re-evaluation analysis focuses rather on amounts of service calls (invoked or not). An interesting property of  $\phi^{\text{len}}$  is that, at each re-evaluation point, the analysis has to consider only the nodes in the backlog in order to insert new non-concrete dependencies into the graph.

Furthermore, recall the XPath expression of an input parameter is always evaluated when the respective service call is invoked. If some intensional nodes contribute to this query, then they are considered non-concrete parameters and the results of their invocation are passed to the outer service call. Hence, although the dependency graph may miss some non-concrete edges in a lenient analysis, these dependencies are always enforced during the materialization. Observe that Proposition 1 also applies to lenient phases. The major drawback of a lenient analysis is that peers cannot attempt to optimize in advance *some* of the data transfers related to service calls in the backlog.

**Final remarks.** Representing service invocation constraints is an important issue in the materialization of AXML documents. In particular, because it enables the system to check relevant properties of a document, and to control its evolution during the materialization process. In spite of that, it is worth mentioning that the canonical model and update techniques that we propose in this Section are not a requirement of the XCraft optimization strategy. They actually provide a formal underpinning that can be explored by any systematic approach to AXML verification and/or optimization. Nonetheless, since our proposal relies on a very popular structure for components dependencies (*i.e.*, graphs), the cornerstone ideas of our optimization strategy can be applied on a general context, regardless of the techniques presented in this Section.

In the next Section, we formalize the problem of determining an efficient configuration of service executors and service callers to materialize an AXML document.

## 5 Materializing AXML Documents

A basic way to write an AXML document consists in hard-wiring a specific address for each embedded service call, including the service URL. Namely, the user has to locate a peer to execute each service call. This approach is

quite cumbersome, specially because P2P systems are often complex, highly-dynamic arrangements of many peers. In a more flexible approach, one can use abstract references to identify Web services, based in some ontology of services, such as in OWL-S [41]. These references correspond to entries in a service directory, as in a UDDI repository [52], where they are mapped to the peers that actually provide the services. Since a Web service may be provided by several peers, ideally users can rely on the system to choose the best provider (in terms of performance, or another set of properties) to execute 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 at other peers. In this case, peers need some strategy to distribute the materialization process. Also, since such a collaboration allows many different materialization alternatives, an automatic strategy requires metrics to reason about delegation.

In this Section, before enumerating these alternatives and pointing appropriate metrics, we describe the main participants of the AXML materialization process, how they can collaborate in a P2P scenario, and their necessary bindings for Web service invocation. Then, in Section 5.4, we characterize the problem of optimizing AXML materialization: we determine some bounds for the search space, introduce the notion of materialization plans for AXML documents, and discuss important issues on generating and ranking these plans. We close this discussion in Section 5.5 with an analysis of the impact of the dependency graph shape on optimization strategies for AXML materialization. In the sequel, we assume an infinite set  $\mathcal{P}$  of peer names in the system. Furthermore, each peer keeps a list of *neighbors* (denoted by  $\mathcal{N}$ , such that  $\mathcal{N} \subseteq \mathcal{P}$ ), namely the peers it can collaborate for AXML materialization.

## 5.1 Equivalent, Replicated and Generic Services

The evolution of the Semantic Web has fostered services orchestration in several distributed scenarios, such as P2P systems [8, 17, 26] and grid computing [14, 21, 29, 57, 66]. An attractive property of semantic-enabled systems is that distributed applications can be defined by *abstract specifications*, and then instantiated on an execution environment based upon *equivalent Web services*. A catalog of services and some matchmaking mechanism are required to enable service selection [19, 23, 36]. This approach has many advantages, and has been widely explored. In P2P systems, *several peers are expected to provide the same Web service*.

To materialize an AXML document with abstract service references, it is important to determine which peers shall be considered to participate in the process, since the Web service requested by each service call node may be provided by several peers. A simple case of service equivalence is

when *Web services are arbitrarily replicated* in the system. Replication of data and/or Web services is typically employed in distributed scenarios to increase throughput and reliability [12, 14]. In the ActiveXML framework, service replication is particularly easy for *declarative Web services*, which are defined by XML queries: *any* peer can evaluate a query (and thus become a service provider), as soon as it has the data on which the query applies. A mechanism for declarative service replication in ActiveXML has been developed in [7]. Another class of services likely to be deployed on several peers consists of *generic services*, such as encryption and data compression services. Usually, these services do not change the contents of the data they operate on, but act on some orthogonal aspects, such as its size and its encryption status.

The AXML optimizer may be free to *dynamically deploy* declarative or generic services to some peers during the materialization process, if this allows to improve the overall materialization performance. However, dynamic service replication may significantly increase the number of AXML materialization alternatives. Hence, the optimizer must consider whether enlarging the search space with these possibilities makes the problem too complex.

We extend the basic AXML document model to allow service call nodes to mention the symbol “any” as the servers providing a given Web service. The semantics is that *the user does not impose any specific provider for this invocation*; any server that the optimizer may find is considered good. For simplicity, we assume that  $\text{any} \in \mathcal{N}$ . Also, we consider the symbol “unknown”, which indicates that the optimizer does not hold information about a requested service, but can lookup for it in the P2P system. We also assume Web services are organized into classes of equivalent services, and peers can gather information about the distribution of these services on the network (as in [8]). Such an information does not necessarily represent the global status of the system, but only the system visibility from a given peer. A basic capability of an AXML-enabled peer is to identify the providers of a service call node, as follows.

**DEFINITION 18** *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  $L_v^E = \text{any} \mid \text{unknown} \mid \{P_1, \dots, P_n\}$ , where *any* indicates that all the peers in  $\mathcal{N}$  can execute  $v$ , the symbol *unknown* denotes that  $L_v^E$  is undefined, and  $\{P_1, \dots, P_n\}$  is a finite set of peers in  $\mathcal{P}$  which provide the service requested by  $v$ , with  $n \geq 0$ .*

Observe that  $L_v^E = \{\}$  means that the optimizer did not discover any information about peers providing the service of  $v$ . Since peers can join or leave the system randomly, the execution scope of a service call is rather a snapshot of the system status. Because of that, if  $L_v^E$  is empty, the peer can either retry to locate the service afterwards (hoping for some

change in the system) or ask other peers to try to fill in the execution scope of the node. The execution scope may also be determined by the user, through explicit peer addressing. It is worth noting that a service provider does not have to be a neighbor. For example, suppose the neighbors of peer  $P_1$  in Figure 1(a) are  $\mathcal{N}_{P_1} = \{P_2, P_4, P_5\}$ . Still, according to Figure 1(b), peer  $P_3$  is among the providers of  $sc_1$  on the *SwapWorkspace* document at  $P_1$ . Next, we discuss ways for peers to collaborate in AXML materialization.

## 5.2 Exploring P2P Collaboration in AXML

Distributed computing is inherent in AXML materialization, since service executions usually take place in different peers. Going further, many other aspects can also be distributed, such as:

- peers can collaborate to *locate service providers*;
- peers can *delegate parts of the materialization* of a document to other peers; and
- similarly, peers can ask other peers to *generate parts of a materialization plan* for an AXML document.

Basically, AXML materialization can be orchestrated in a P2P system by some message exchanges between peers. In this scenario, the basis of peers interaction is the *materialization plan*, which is the fundamental element used to control the AXML materialization process. Such a plan determines how each service call node is going to be materialized; 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. Materialization plans are formally defined in Section 5.4.

Locating service providers is the simplest type of P2P collaboration. If a peer cannot find information about the execution scope of some service call nodes, then it may decide to send a partially-specified plan to another peer, along with a request to properly annotate such an information on the plan. On the other hand, delegating AXML materialization requires more sophisticated control mechanisms. First, recall that the start point of the materialization process is always at the master peer of the AXML document. Then, the master peer decides whether other peers will be invited to collaborate or not. Delegating the invocation of a service call node  $v$  consists in setting another *caller* for  $v$ , which is different from the master peer. In particular, the master peer sends a *delegation plan*, which contains  $v$  (or more nodes), its input parameters (possibly including its invocation dependencies, if  $v$  is not ready), and its collateral call to the caller. In turn, the caller is in charge of triggering the invocation of  $v$  (and of its dependencies and collateral calls,

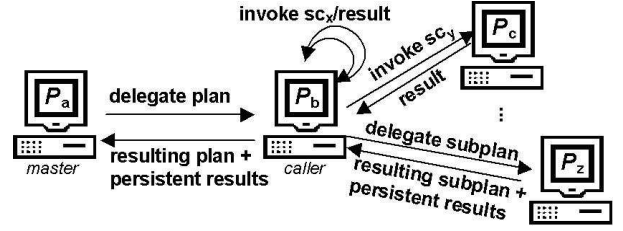


Figure 11. AXML delegation in a P2P system.

if necessary), and gathering its result. The caller must send back the result of  $v$  to the master peer, as well as the result of all the persistent nodes that were sent with  $v$ . The master peer may also choose to evaluate some of the dependencies of a delegated node before requesting its remote materialization. These interactions are illustrated in Figure 11.

Notice that a delegated part of an AXML document does not necessarily correspond to a subtree, but to some partition of its dependency graph. Reasoning about AXML delegation has to consider the performance impact of changing the caller of each node of the dependency graph. Such a decision is mainly oriented by the data flows between service call nodes. For example, delegation may be beneficial to materialize nodes  $sc_9$  and  $sc_{10}$  of the *SwapWorkspace* document, in Figure 3, since it enables to exploit a fast communication link between peers  $P_4$  and  $P_5$ . Nevertheless, there may exist constraints on the invocation of a service call node, such as security and price policies. Thus, users may also choose to specify exactly which peers are allowed to invoke some service calls.

**DEFINITION 19** Given a service call node  $v$ , the delegation scope  $L_v^C$  denotes the peers that can invoke  $v$ , such that  $L_v^C = \text{any} \mid \{P_1, \dots, P_n\}$ , where *any* indicates that all the peers in  $\mathcal{N}$  can invoke  $v$ , and  $\{P_1, \dots, P_n\}$  is a finite set of peers in  $\mathcal{P}$ ,  $n \geq 0$ .

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, as we discuss in Section 5.4. Therefore, we assume a *small-world P2P scenario*, where the optimizer may restrict the delegation scope to the set of service providers that are involved in the document materialization. To further limit this, it may also apply the *Context heuristic* [44], such that only selected executors are used to determine the delegation scope. Also, several P2P systems consider the need of some specialized peers, which are more able than others to perform some tasks, such as query routing and data location [40]. Similarly, some peers may be tailored for some AXML materialization tasks, such as locating the execution scope of nodes or executing delegated

plans (e.g., due to their high connectivity with other peers in the system). Therefore, the optimizer may keep a list of such peers, and include them in the delegation scope even if they are not involved in the execution scope, to improve the materialization performance.

A major motivation of AXML delegation is similar to the idea of the *edge-zeroing* algorithms for parallel scheduling [33]. In a dependency graph, these algorithms analyze the edges with high communication cost, and attempt to define clusters of tasks to be run at the same processor. Initially, each node of the graph represents a cluster. Then, at each step of the algorithm, the edge with the largest communication cost is found, and the two clusters incident by this edge are merged if such a merging does not increase the overall performance. However, the constraints on both the execution and delegation scope of each service call node of an AXML document do not allow to arbitrarily assign service executions to peers. Moreover, communication costs are not zeroed if two connected service executions occur in the same peer (though they are really reduced), since SOAP messaging usually involves some additional time-consuming operations, such as parsing and packing the transferred data [44]. Hence, deciding about delegation mostly relies on comparing materialization alternatives.

We assume peers are autonomous, thus the master peer cannot enforce transitive delegation to other peers. Namely, the result of each delegated plan is always sent to the master peer. Considering otherwise would increase significantly an already huge space of AXML materialization alternatives. However, since peers have different perspectives of the system, a peer may decide to reoptimize a delegated plan, possibly by delegating parts of it. Moreover, peers may return a delegated plan intact or partially materialized. In this case, the master peer either reoptimizes the plan, trying to find another peer to delegate the plan, or evaluates it itself. Nevertheless, if a peer decides to reoptimize a plan, then it cannot include the original master peer in the delegation scope of the respective service call nodes. Also, to avoid endless delegations, the original optimizer can determine some limits for plan reoptimizations. These limits and the plan provenance are encoded in the materialization plan.

Distributing the optimization of AXML documents is analogous to delegating materialization tasks, and it can help peers to handle requests overload and insufficient support information. If a materialization problem is too large, the master peer may split the dependency graph without making any considerations about either execution scope or performance, and then send parts of the problem to be solved by other peers. To support P2P collaborations in ActiveXML, we assume each peer has to provide the basic Web services shown in Table 1.

**Table 1. Services for P2P collaboration.**

Web service	Description
<i>locate</i>	Accepts requests to discover the execution scope for some materialization plan. It returns a (possibly partially-)annotated plan.
<i>optimize</i>	Accepts requests to find an efficient materialization plan for some dependency graph. It returns a (possibly partially-specified) materialization plan.
<i>submit</i>	Receives requests to execute some materialization plan, by possibly re-optimizing it. It returns the plan and its persistent results.

### 5.3 Enacting AXML Materialization

In the AXML universe, the basic elements of a P2P setting are peers, Web services and AXML documents. Essentially, *peers* are uniquely-identified agents connected through a network, and *services* are operations that peers can perform. A service may require input parameters that are instantiated at runtime. It also has a termination status, such as “*success*” and “*fail*”. The *invocation* of a Web service is an event, namely a compact occurrence that enables: 1) the flow of input parameters to the peer that is going to execute the service; 2) the service execution; and 3) the transfer of the results to the peer that requested the service. By default, a service call node is invoked by the master peer of its respective document, but this invocation can be delegated to another peer. To be invoked, a service call must have all the necessary information to identify the requested Web service (as defined in the SOAP and WSDL standards [59]). In particular, it must have the address of the peer that is going to execute the service, as stated next.

**DEFINITION 20** *Given a service call node  $v$  held by peer  $P_v$ , an invocation  $\mathcal{I}$  of  $v$  is denoted by the expression  $\langle v, P^E, P^C, status \rangle$ , where:*

- $P^E$  and  $P^C$  are, respectively, the executor and the caller of  $v$ , such that  $P^E \in L_v^E$  and  $P^C \in (P_v \cup L_v^C)$ ;
- neither  $L_v^E$  is empty nor it contains the symbol “*unknown*”; and
- the status of  $\mathcal{I}$  is determined by the termination status of the service requested by  $v$ , such that status in  $\{success, fail\}$ .

For example, in Figure 3 (center), we have  $\langle sc10, P_5, P_4, success \rangle$  and  $\langle sc9, P_4, P_4, success \rangle$ , assuming both invocations are successfully executed.

For simplicity, when considering materialization phases, we omitted in Definition 15 the information about callers and executors of each service call invocation. Now, we can

ground an invocation sequence  $I_{\text{timeline}}$  to a specific location context, as follows.

**DEFINITION 21** *Given an AXML document  $d$ , a grounded invocation sequence  $I_{\text{timeline}} = [\mathcal{I}_1, \dots, \mathcal{I}_n]$  is a dynamic sequence obtained by successively picking a ready node  $v_x$  in  $SC_d$ , setting an  $\mathcal{I}_x = \langle v_x, P_x^E, P_x^C, \text{status}_x \rangle$  and invoking it, until either  $SC_d = \{\}$  or  $SC_d$  has no ready node, such that  $1 \leq x \leq n$ . Moreover, if  $v_i \hookrightarrow v_j$  in  $d$ , then  $\mathcal{I}_j$  is a successor of  $\mathcal{I}_i$  in  $I_{\text{timeline}}$ , and the only invocations between  $\mathcal{I}_j$  and  $\mathcal{I}_i$  are the dependencies of  $v_i$  and their collateral calls. A grounded invocation track  $IT$  of  $I_{\text{timeline}}$  is of the form  $IT = [\mathcal{I}_1, \dots, \mathcal{I}_\ell]$ , with  $\ell \leq n$ .*

Observe that nodes are invoked in proper order in  $I_{\text{timeline}}$  (ready nodes first with subsequent collateral calls), and invocation constraints are enforced accordingly. Hereafter, we assume both  $I_{\text{timeline}}$  and  $IT$  to be grounded, unless stated otherwise.

Materializing an AXML document consists in invoking all of its embedded service calls, as well as the occasional intensional answers. This process yields a new version of the document, as defined next.

**DEFINITION 22** *Let  $d$  be an AXML document and  $\phi$  be a (grounded) materialization phase of  $d$  with length  $\ell$ . A materialized version of  $d$  is another AXML document  $d'$  obtained by the service invocations of  $\phi$ . We say that  $d'$  is complete iff all the nodes in  $\Delta_\ell$  are inactive; otherwise,  $d'$  is partial. Moreover, the materialization of  $d$  into  $d'$  is successful iff it is complete and  $\text{status}_i = \text{"success"}$  for each invocation  $\mathcal{I}_i$  in  $IT$  of  $\phi$ ,  $1 \leq i \leq \ell$ .*

If some invocations fail, their dependant calls are not invoked (since they cannot become ready) and the respective failures are reported to the user. Still, we consider that all the remaining service calls are invoked, if possible. Also, we assume that the user is not interested in *undoing* service calls when some of them fail in the materialization of an AXML document. That is, the document does not represent a transaction unit.

The master peer receives document requests and starts their materialization, possibly by delegating parts of the service invocations. After peers finish their materialization tasks, they must send all the persistent service results back to the master peer. In our AXML settings, we assume communication links between peers may have different bandwidth.

## 5.4 The AXML Optimization Problem

The diversity of Web services providers, P2P collaboration opportunities, peers capabilities, and invocation dependencies allows the materialization of an AXML document to be performed through many different alternatives,

with different performance. The *makespan* of a materialization alternative 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. We adopt this performance metric because it is based on *response time*, which has typically a significant perceived impact on the user [20, 43].

Optimizing the materialization performance of an AXML document consists in minimizing its makespan. This involves two main issues: (i) *planning resource selection*, that is determining a caller and an executor for each service call, such that both service execution and communication costs are minimized; and (ii) *scheduling service call invocations*, to exploit parallelism and thereby minimize the makespan. Clearly, these issues are inter-related; efficiently assigning service executions to peers depends on balancing their load, and vice-versa. We use the term *AXML planning* in a general sense, to indicate both tasks (i) and (ii).

AXML planning is essentially characterized by scheduling a complex DAG to heterogeneous machines, which is an NP-complete problem [15, 33]. Because of such a complexity bound, we focus our optimization analysis on *finding suboptimal solutions in reasonable time*. Furthermore, as discussed in Section 4.4, intensional answers changes the specification of a materialization problem at runtime. Since usually peers cannot foresee these results, we assume AXML planning is initially restricted to the dependency graph obtained before the materialization starts, and occasional intensional answers trigger some re-optimization procedures. However, in our approach, such a re-optimization is localized to specific subgraphs whenever this is possible. We address this problem in Section 6.

**Search space of materialization alternatives.** Thus far we have considered materialization phases as totally-ordered sequences of service call invocations. Nonetheless, the dependency graph of an AXML document does not impose a total order on these invocations. Hence, many different invocation sequences can be used to materialize a document. For example, in the dependency graph of Figure 8, we may have invocation sequences starting with *sc2*, *sc4*, *sc6*, *sc8* or *sc10*. Also, some service calls may be invoked in parallel. In our example, for instance, nodes *sc4* and *sc6* can be invoked independently.

There are several techniques to allocate resources from a pool of available machines for job scheduling [15, 18, 33, 55]. However, planning resource selection for AXML materialization is a different problem due to several reasons. In particular, in an AXML document each service call node has its own execution and delegation scopes, and a good resource configuration has to conciliate the diverse execution and data flows possibilities in a highly-dynamic scenario. Based on the combinations of callers and executors of each service call, the number of possible configurations of a de-

pendency graph  $\Delta$  is given by:

$$\#locationConfigs(\Delta) = \prod_{x=1..|V^*|} |L_x^E| \times (|L_x^C| + 1) \quad , \quad (3)$$

where  $V^*$  is the set of all nodes in  $V$  plus the new node instances triggered by collateral calls. Now, if we consider also the possible invocation sequences, the number of AXML materialization alternatives is bounded to:

$$\#plans(\Delta) = |V^*|! \times \#locationConfigs(\Delta) \quad . \quad (4)$$

In the worst case, when any peer is both a provider and a caller candidate for every service call node, this number becomes  $|V^*|! \times n^{2|V^*|}$ , where  $n$  is the number of distinct peers. Even for simple scenarios,  $\#plans(\Delta)$  tends to be large due to the exponential nature of the problem.

To reason about these possibilities, we introduce a central element of AXML planning: a *materialization plan*, which is derived from the dependency graph of an AXML document. However, instead of a complex graph, we use trees to represent a plan. More precisely, we consider the *minimum forest of spanning trees* of a dependency graph; namely, the minimum set of trees containing all the nodes and edges of the graph. Also, in a materialization plan, these tree nodes are labelled by operators of an algebra, which represent adequate materialization alternatives. We use the symbol  $\mathcal{A}$  to indicate a finite set of algebraic operators. In Section 6, we present a formal definition for the minimum forest of spanning trees of an AXML document, and describe how such a forest is generated and converted into materialization plans. It should be noted that a tree-based representation is interesting for several reasons, specially because it enables the optimizer to reduce the complexity of AXML planning by partitioning the problem into simpler and possibly independent tasks (using the Divide&Conquer heuristic [44]). Basically, AXML planning is encoded in the following structures.

**DEFINITION 23** *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 invoke and execute  $v$ , respectively, such that  $P^E \in L_v^E$  and  $P^C \in (L_v^C \cup P_v)$ . The term  $\widehat{IP}_v$  denotes the set of all possible invocation plans of  $v$ , according to  $L_v^C$  and  $L_v^E$ .*

**DEFINITION 24** *Let  $\Delta$  be a dependency graph. A materialization plan  $\mathcal{M}$  for  $\Delta$  is of the form  $\langle \Lambda, \mathcal{O}, \mathcal{L}, \succ, P_m \rangle$ , where:*

- $\Lambda$  is the minimum forest of spanning trees of  $\Delta$ ;
- $\mathcal{O}$  is a labelling function that associates every node in  $\Lambda$  with an operator in  $\mathcal{A}$ ;
- $\mathcal{L}$  is a mapping from each node  $v$  in  $\Lambda$  to invocation plans in  $\widehat{IP}_v$ ;

- $\succ$  associates with each node in  $\Lambda$  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  $\Lambda$  to exactly one invocation plan; otherwise,  $\mathcal{M}$  is abstract.

In a materialization plan, the *height* of a node indicates the size of the longest path from the node to a leaf node. This property can be: *simple* (denoted by  $h$ ), if it considers only simple edges; *collateral* ( $ch$ ), when it is based on paths that start with a collateral edge of the node and that may include transitive collateral calls; or *absolute* ( $ah$ ), which is the highest value between  $h$  and  $ch$ . A variation of this property is the *least height* of a node, which is the size of the shortest path from the node to a leaf. We prefix the height with “ $l$ ” to indicate such a variation (e.g.,  $lh$  denotes the least simple height). We assume the size of a path is given by the number of nodes on it, including the origin and the destination, and leaf nodes of the plan have  $h = 1$ . Unless stated otherwise, hereafter we refer to the simple height of the nodes by default.

Observe that the evaluation of a physical plan corresponds to a grounded invocation track of a materialization phase. This correspondence is important because it enables the optimizer to control the plan evolution, namely to correctly make the necessary updates to the plan during its evaluation. It is also worth noting that one can determine a materialization plan for some subset of service call nodes of a document, based on the corresponding dependency graph. In this case, we have a *subplan* of the document.

**Performance metric outline.** Comparing alternative materialization plans requires estimating their makespan. To calculate this metric, it is necessary to consider the sequential computations of the plan, as well as its peer assignment load. In workflow systems, the makespan of a physical plan is typically estimated by its *critical path* [18, 33, 63], namely the path of sequential executions with the larger completion time. Similarly, in AXML planning, sequential executions are mostly determined by invocation constraints. We can estimate the *static critical path* of each spanning tree of a physical plan by calculating the costs of both service execution and data transfers of its nodes (which is done recursively, from the leaves). Such a path is said “static” because the load of service executions of the involved peers is disregarded. On the other hand, the *dynamic critical path* accounts the sequential processing of service executions assigned to each peer. We describe the basic formula to estimate the costs of Web service invocations in [44], which are compounded with other costs, such as delegation costs, in the model presented in Section 7. We consider both

execution and communication costs may be weighted, to calibrate the cost model according to the *computation-to-communication ratio* (CCR) [33] of the AXML setting.

An issue in estimating the makespan of a materialization plan is that *delegating AXML materialization implies in a non-deterministic execution scenario*. Due to P2P collaboration, it is impossible to know the exact position of the service call invocations of an AXML document on the execution queue of each peer involved in the materialization process. Consequently, we cannot determine precisely the dynamic critical path of a dependency graph. This represents an important restriction on the invocation scheduling problem, since it prevents the direct use of current algorithms to schedule workflows tasks onto heterogeneous machines, such as proposed in [15, 18, 33, 46, 47, 63, 66]. These algorithms are usually based on estimates such as the “*earliest start time*” and the “*latest start time*” of a task on a machine, which cannot be obtained with exactitude in AXML settings. The main reason is that peers are not dedicated resources and some parts of the dependency graph may be evaluated in parallel without global synchronization of service invocations. Nevertheless, ignoring current invocation assignments during the optimization may yield schedules similar to those based on the “*Minimum Execution Time*” (MET) heuristic [18], which may cause significant load imbalance across peers. For example, if a peer outperforms the others in many services, a load-blind optimizer may exhaust such a peer with excessive assignments. Worst, this would also stretch the makespan with massive sequential processing. To take into account service assignment load during AXML planning, our cost model (presented in Section 7) defines the *energy factor* of a peer in a materialization plan, which is used to dynamically weight the costs of plan nodes.

**Optimization strategies.** Many different strategies can be used to generate and compare alternative materialization plans (based on makespan estimates) of an AXML document. Each strategy is determined by an algorithm to produce these plans, and by a method used to compute their makespan, as follows.

**DEFINITION 25** *Let  $MS$  be an objective function that estimates the makespan of materialization plans. An optimization strategy is an expression  $\langle \Upsilon, MS, \text{ceil} \rangle$ , where  $\Upsilon$  is an algorithm that finds a plan  $\mathcal{M}$  for a given dependency graph  $\Delta$ , and  $\text{ceil}$  is a performance ceiling such that  $MS(\mathcal{M}) \leq \text{ceil}$ . The optimization algorithm  $\Upsilon$  is optimal if  $MS(\mathcal{M})$  is minimal for any  $\Delta$  and for all possible plans of  $\Delta$ . Moreover, we say that  $\Upsilon$  is complete if  $\mathcal{M}$  is physical whenever such a plan exists for  $\Delta$ .*

A straightforward optimal strategy to find materialization plans is to adopt an exhaustive search, which yields all possible combinations of callers and executors, and of invocation sequences. However, such a strategy is often un-

feasible due to its high time complexity (see Equations 3 and 4), and heuristic approaches are necessary to prune the search space. For instance, the optimizer can use the *Divide&Conquer* (D&C) heuristic to split the problem into smaller disjoint parts [44], which correspond to the connected subgraphs of the dependency graph. Thereby, the total complexity is reduced from a product to a sum of the complexity of the problem parts. Suppose there are  $n$  connected subgraphs in a dependency graph  $\Delta$ , then the number of alternative physical plans is given by:

$$\#plans^{D\&C}(\Delta) = \sum_{i=1..n} \#plans(\Delta_i) \quad , \quad (5)$$

where  $\#plans(\Delta_i)$  denotes the number of alternative plans of each subgraph  $\Delta_i$ . Since each subgraph is evaluated independently, the D&C strategy is optimal only if there is no interference between the enactment of the subgraphs. Namely, if the corresponding materialization plans involve distinct location scopes. Otherwise, some optimization decisions of a materialization plan might disregard the performance penalties resulting from the evaluation of other plans. Nonetheless, the main problem of the D&C strategy is that shared dependencies and collateral calls often imply graphs with a few, yet complex connected subgraphs. Hence, in these cases, the optimizer cannot explore significant complexity reductions. In Section 6, we propose an algorithm that maximizes the number of subgraphs of an AXML document by virtually replicating shared nodes.

Another class of heuristic strategies consists in setting arbitrary plan configurations. For example, the optimizer can systematically: set the master peer as the caller of the service invocations; and choose each service executor from the best provider w.r.t. execution time (namely, using the MET heuristic). We call this particular strategy MMET, referring to “*Master caller and MET heuristic*”. In this case, the number of plan configurations that are analyzed for a dependency graph  $\Delta$  is bound to:

$$\#locationConfigs^{MMET}(\Delta) = \sum_{x=1..|V^*|} |L_x^E| \quad . \quad (6)$$

Although the MMET strategy may save significant optimization time, it does not explore P2P collaboration. Moreover, the MET heuristic ignores the invocation scheduling, and usually produces poor makespans [18]. Notice the MMET strategy is clearly suboptimal.

An important aspect of AXML planning is that, in general, *cost functions for makespan estimation are not monotonic*, in the sense defined in [25]. Namely, optimizing a subproblem may increase the overall makespan, as well as increasing the cost of a subproblem may lead to a better overall solution. For that reason, materialization plans need to be entirely specified to be properly compared. However, AXML settings are highly-dynamic scenarios, thus generating complete materialization plans may produce solutions

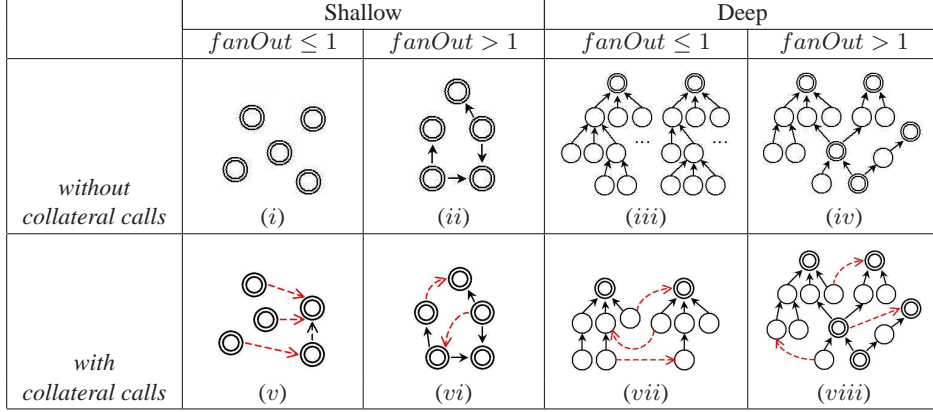


Figure 12. Main topologies of dependency graphs.

that are invalid at runtime. We propose an optimization strategy that interleaves AXML planning and materialization. We first identify independent tasks (spanning trees) of the graph, and then evaluate them separately. For each tree, we incrementally generate the corresponding plan based on the topological order of the service invocation nodes. The plan evaluation algorithm basically consists in climbing up a (partial) plan, from the leaf nodes to the root, considering subtrees of fixed heights. For each subtree, we analyze its materialization possibilities and generate a subplan, which is executed before resuming the AXML planning and materialization. It is worth mentioning that our strategy is based on a meta-heuristic that can be combined with other approaches to find efficient solutions for the subplans.

### 5.5 Main Problem Topologies

The shape of a dependency graph determines most of the opportunities for P2P collaboration, as well as the requirements of scheduling service invocations. To get a better understanding of the issues behind AXML materialization, we analyze the main graph topologies. We highlight three dimensions: (1) *the length of paths of sequential data transfers with temporary nodes*, which is related to the paths of invocation dependencies on the graph; (2) *the shared dependencies of the graph*, namely the nodes with  $fanOut > 1$ ; and (3) *the sequential collateral invocations of service call nodes*. We say that a dependency graph is *deep* if it has long paths of invocation dependencies with few persistent nodes, and *shallow* otherwise.

Based on these criteria, we have the graph topologies of Figure 12. Considering shallow graphs, the simplest topology is an empty dependency graph, shown in item (i); it consists essentially of a *bag-of-tasks*, without data transfers between service call nodes. Optimizing such a graph concerns mostly load balancing. A variant of this topology including collateral calls is illustrated by item (v), where

the optimizer has to schedule parallel sequences of service invocations without data transfers between service executions. Also in this case, service execution costs are the performance yardstick. Notice that with shared dependencies and/or some data flows between service invocations (*i.e.*, topologies (ii) and (vi)), the optimization goal remains load balancing, since only persistent nodes are handled. Yet, if persistent nodes do not represent large data transfers, then AXML delegation can be explored to improve parallelism in shallow graphs, thereby reducing their makespan.

Observe that shallow topologies are not expected to be frequent in AXML documents, because they do not encompass intensional parameters. Another class of problems is represented by deep graphs, which are more usual in AXML materialization. For these graphs, the optimizer can try to find fast links between the participating peers in order to reduce communication costs related to temporary nodes. The typical case is topology (iii), where the exit points of the graph denote independent materialization tasks with few required data transfers to the master peer. Interestingly, these tasks remain independent in topology (vii), despite the collateral edges connecting them. The reason is that collateral calls represent new instances of service invocations, which are “clones” of the referred nodes. On the other hand, the shared dependencies in topology (iv) determine some synchronization points between materialization tasks, which are not independent from each other.

The first attempt of our optimization strategy is to simplify the dependency graph by transforming it into a forest of (possibly deep) independent tasks. The resulting graph is close to the topology in (iii), which can be more easily evaluated by the optimizer. Such a topology favours parallel execution, along with the reduction of communication costs. Next, we present the proposed optimization strategy.

## 6 Optimizing AXML Materialization

Currently, the ActiveXML platform lacks a performance-oriented approach to the optimization of AXML documents with abstract service references. Basically, ActiveXML peers support only one materialization strategy, which is uniquely determined by explicit service call attributes and (possibly) typing control. In this Section, we propose a cost-based optimization strategy to improve AXML materialization by dynamically analyzing alternative materialization plans. We describe the overall optimization strategy in Section 6.1, and its main steps are detailed as follows. Section 6.2 presents techniques based on spanning trees to generate initial materialization plans from arbitrarily-complex dependency graphs. These plans are encoded with an algebra that enables the optimizer to perform incremental and collaborative AXML materialization. In Section 6.3, we outline an algorithm to partition a materialization plan into tasks, and a priority-based mechanism to order these tasks. In particular, we explore both the inter-task and the intra-task parallelism degree of a plan to sort its tasks. Going further, we propose in Section 6.4 an algorithm that scans a materialization plan in topological order and dynamically generates “good” physical plans based on cost metrics. Finally, we discuss task delegation and collaborative optimization in Section 6.5.

The proposed strategy is dynamic in the sense it partially materializes the plan before completing the optimization. Moreover, resource allocation and planning are performed at runtime. It is also adaptive, as defined in [28], since it makes the optimizer choices sensitive to changes in the P2P system at each step of the materialization process.

### 6.1 Dynamic Optimization Strategy

To materialize an AXML document, the optimizer has to deal with two major issues: a *huge search space* of materialization alternatives, and the *unpredictability* of the P2P setting. In a static approach to generate materialization plans, all the service calls, the interactions among them, their service providers, and communication costs are assumed to be known before the optimization starts. Clearly, this approach is not suitable for AXML materialization. Ideally, the optimizer should react to changes in the environment, and this should not be based solely on plan reoptimization, which is often quite expensive in an unstable setting. We propose an optimization strategy that exploits dynamic techniques to reduce complexity and to enable the system to adapt to both system performance and membership fluctuations. In our approach, materialization plans are not produced at once, and reoptimization is triggered only when really necessary. Not surprisingly, our techniques mostly rely on *splitting the dependency graph into smaller pieces*. However, the main question here is how to partition the materialization prob-

1	procedure <i>DynamicOptimize</i> ( $\Delta, k$ )
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 $P'_m$ in $\mathcal{N}$ for $t$
11	Delegate $t$ to $P'_m$
12	go to next task
13	end if
14	Split $t$ into $k$ -depth subplans
15	for each subplan $\mathcal{M}_x$ in $t$ in topological order 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 $T_i$
22	end for
23	end

Figure 13. Overall optimization algorithm.

lem such that important performance aspects, such as data flows between service invocations, are preserved and considered by the optimizer.

Inspired on 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 materialization planning and execution*, thus the peer optimizer can decide how to proceed after partial executions, when it may have more up-to-date information on the system status.

The basis of our optimization strategy is to work on the materialization of an AXML document by using its dependency graph, which explicitly shows all the invocation constraints of the embedded service calls. The optimizer unfolds this graph into a simplified tree-based structure, and then produces an *initial abstract plan*, whose location scope and invocation schedule are not determined. Such a plan is partitioned into materialization tasks, which can be further split into subplans of height  $k$  according to the topological order of the service call nodes. For each subplan, the optimizer annotates the execution and delegation scopes of its nodes, and then generates its alternative physical subplans. These equivalent subplans are then ranked based on some cost metrics, and the “best” (but not necessarily optimal) alternative is picked and evaluated. This process is repeated until all the tasks and their subplans are evaluated. Optionally, the optimizer can delegate some tasks to be completely evaluated by other peers. The overall algorithm of our op-

timization strategy is described in Figure 13. In the following, we detail these optimization steps.

## 6.2 Extracting Materialization Plans

To generate a materialization plan, the optimizer has to associate the service call nodes of the dependency graph with adequate evaluation operators, which will actually process each service request. For example, a service call may be invoked locally (*i.e.*, by the master peer) or delegated to another peer; each of these cases require a different operator to handle the service call node. These operators are interpreted by the optimizer, and their service call results are inserted into the AXML document. Nevertheless, a dependency graph is rather a complex structure to be directly processed by the optimizer, due mostly to shared dependencies and collateral calls. Therefore, we first encode the dependency graph using a simpler, tree-based structure that is more adequate for distributed evaluation. In this transformation process, we also attempt to apply a *Divide&Conquer* heuristic to reduce the problem complexity. In particular, we extract the *spanning trees* of the dependency graph, such that the optimizer can identify and evaluate independent tasks. Then, we use an algebra of materialization operators to generate an initial plan from these trees. Such a plan contains only *abstract operators* (which lack location scope information) and its purpose is to enable the optimizer to produce alternative physical plans.

**Extracting spanning trees.** Several classical algorithms can be used to build spanning trees from an arbitrary graph. For instance, the well-known Prim’s algorithm [11] 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 this start node. Each node in the border is added to the current spanning tree and expanded recursively. These steps are repeated until all the spanning trees of the graph are obtained. The roots of these trees are the seeds of the algorithm. In our case, only exit points of the dependency graph are used as seeds. Moreover, the border is built considering the opposite direction of simple edges, as stated next.

**DEFINITION 26** *Let  $\Delta$  be a dependency graph and  $v_x, v_y$  be two nodes in  $\Delta$ . The node  $v_y$  is reachable for spanning from  $v_x$  iff either  $v_y \rightarrow v_x$  or  $v_x \hookrightarrow v_y$  is in  $\Delta$ . Furthermore, the border of  $v_x$  consists of the set of all the nodes in  $\Delta$  that are reachable for spanning from  $v_x$ .*

If the dependency graph has only connected subgraphs that represent trees, then we can easily identify and evaluate its independent tasks. However, an AXML document usually involves shared dependencies and/or collateral calls,

which denote subgraphs that do not correspond to trees. To handle this, we have to build a flat representation of the dependency 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 (*i.e.*, which have service calls in common). We propose two special transformations to obtain such a representation:

- *node replication*, that is to replace the node by a set of exact copies of it, which represent the same instance of the corresponding service call. This transformation is applied to separate subgraphs connected by shared dependencies; and
- *node cloning*, which consists of adding new instances of a service call node to the dependency graph. It is used to represent collateral calls.

Figure 14 shows the algorithm used to flatten a dependency graph based on these two transformations. Basically, each shared dependency (namely, a node with  $fanOut > 1$ ) is replaced by  $fanOut$  replicated nodes, such that each replica inherits exactly one of the outgoing simple edges of the original node, and all of its incoming simple edges. For nodes that are pointed as collateral calls, we clone their entire subtrees for each incoming collateral edge. Replicating shared nodes has time complexity  $O(\otimes \times |E|)$ , while the bound for node cloning depends on the algorithm used to unfold spanning trees.

From a flat dependency graph, we can expand the spanning tree of each exit point based on the border criteria of Definition 26. The result of this process consists of the following set of (possibly related) trees.

**DEFINITION 27** *Let  $\Delta$  be a dependency graph, and  $V^{exit}$  the set of exit points of  $\Delta$ , where  $V^{exit} \subseteq V$ . The expression  $\Lambda = \langle ST, \varrho, \hookrightarrow \rangle$  represents the minimum forest of spanning trees (or MFST, for short) of  $\Delta$ , where  $ST$  is a set of unordered trees of  $\Lambda$ , such that:*

- *shared nodes of  $\Delta$  are properly replicated in  $ST$ ;*
- *collateral calls of  $\Delta$  are properly cloned in  $ST$ ; and*
- *for each node  $v_x$  in  $V^{exit}$  there is a tree  $st_x$  in  $ST$  that is rooted by  $v_x$  and which results from recursively expanding the border of  $v_x$ .*

Furthermore, the function  $\varrho$  associates replicated nodes in  $\Lambda$  with their original node in  $\Delta$ , and  $\hookrightarrow$  denotes a distinguished subset of edges in  $\Lambda$ , which correspond to the outgoing collateral edges of  $\Delta$ . The number of spanning trees of  $\Lambda$  is denoted by  $|\Lambda|$ .

Observe that  $ST$  is equivalent to  $\Delta_{flat}$ . We assume that tree nodes keep their properties from the dependency graph, such as the persistency flag and invocation status. Also, since the edges of a spanning tree are not directed, collateral

```

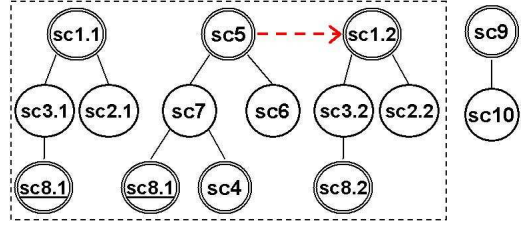
1 function FlattenDependencyGraph( $\Delta$ ):  $\Delta^{flat}$ 
2 {Transform shared nodes (due to shared dependencies
  and collateral calls) to disconnect subgraphs of  $\Delta$ .}
3 begin
4   let  $\Delta^{flat} = \Delta$ 
5   for each node  $v$  in  $\otimes^{flat}$ 
6     if  $fanOut(v) > 1$  then {shared dependency}
7       Replicate( $v, \Delta^{flat}$ )
8     end if
9   end for
10  for each edge  $v_x \hookrightarrow v_y$  in  $\overset{\leftarrow}{E}^{flat}$  do {collateral call}
11    Clone( $v_y, v_x \hookrightarrow v_y, \Delta^{flat}$ )
12  end for
13  return  $\Delta^{flat}$ 
14 end
15 procedure Replicate( $v, \Delta$ )
16 {Replicate node  $v$  in dependency graph  $\Delta$ 
  according to its outgoing simple edges.}
17 begin
18   for each  $v \rightarrow v_x$  in  $E$  do
19     let  $v_{rep}$  be a new replica of  $v$  {an exact copy of  $v$ }
20     Add  $v_{rep}$  to  $V$ 
21     Add  $v_{rep} \rightarrow v_x$  to  $E$ 
22     for each  $v_y \rightarrow v$  in  $E$  do
23       Add  $v_y \rightarrow v_{rep}$  to  $E$ 
24     end for
25     for each  $v \hookrightarrow v_z$  in  $\overset{\leftarrow}{E}$  do
26       Add  $v_{rep} \hookrightarrow v_z$  to  $\overset{\leftarrow}{E}$ 
27     end for
28     Delete  $v \rightarrow v_x$  from  $E$ 
29   end for
30 end
31 procedure Clone( $v_y, v_x \hookrightarrow v_y, \Delta$ )
32 {Clone subtree rooted at node  $v_y$  through the
  collateral call  $v_x \hookrightarrow v_y$  in graph  $\Delta$ .}
33 begin
34   let  $v_{clone}$  be a new clone of  $v_y$  {a new instance of  $v_y$ }
35   Add  $v_x \hookrightarrow v_{clone}$  to  $\overset{\leftarrow}{E}$ 
36   Unfold the spanning tree rooted at  $v_y$  into  $v_{clone}$ 
37   Delete  $v_x \hookrightarrow v_y$  from  $\overset{\leftarrow}{E}$ 
38   let  $IN$  be the set of incoming collateral edges of  $v_y$ 
39   if  $|IN| = 0$  then
40     Delete  $v_y$  from  $\Delta$ 
41   end if
42 end

```

**Figure 14. Algorithm to flatten (by node replication and/or cloning) a dependency graph.**

edges require proper distinction for their particular “fire after” semantics. In fact, we consider that collateral calls are not expanded as regular child nodes in the spanning trees, but as *annotations* on the service call nodes.

**DEFINITION 28** *Given a MFST  $\Lambda$  and two nodes  $v_y$  and  $v_z$  in  $\Lambda$ , we say that  $v_y$  is a collateral annotation on  $v_z$  iff  $v_z \hookrightarrow v_y$  in  $\Lambda$ .*



**Figure 15. MFST of the *SwapWorkspace* graph.**

Figure 15 shows the MFST obtained from the graph of Figure 8; replicated nodes have underlined text, and collateral annotations are denoted by dotted arrows. Cloned nodes have distinct IDs, which we represent by “scX.Y”, where X is the ID of the original node, and Y is the specific ID of the clone. Observe that the spanning trees of a dependency graph may be grouped into (possibly overlay) clusters, such that each cluster has all the trees with node replicas of a service call. More precisely, a cluster represents some connected subgraph of the dependency graph. For example, there is one cluster in the MFST of Figure 15, which is indicated by a dotted rectangle. Although there are not precedence constraints between the trees of a cluster, they are not independent from each other: replicated nodes express synchronization points in the evaluation of their respective spanning trees. Yet, the trees of a cluster become independent once one of the replicas is evaluated. We address these issues in Section 6.3.

**Algebra of materialization operators.** Having computed the MFST of a dependency graph, the optimizer has to turn the resulting trees into a materialization plan. Basically, this can be done by replacing tree nodes in the MFST by operators of an algebra  $\mathcal{A}$ , whose main requirements are the support for Web services invocation and for P2P collaboration. Also, such an algebra should allow the optimizer to incrementally evaluate a materialization plan. Based on these requirements, we propose the algebra described in Table 2. We distinguish three groups of operators:

- (i) the *abstract operators*  $\mu$  and  $\rho$ , which represent the possible combinations of executors and callers of service call nodes in a plan. These operators cannot be interpreted as service executions since they lack specific invocation details, such as the Web service endpoint. For example, the  $\mu$  operator has to be converted into some physical operator (e.g., **invoke**) in order to result into a service invocation;
- (ii) the *physical operators* **invoke**, **fetch** and  $\delta$ , which contain all the information required to invoke a Web service. Observe that  $\delta$  does not point directly to services that are requested in the AXML document. Instead, it represents the invocation of a basic Web service for P2P collaboration, which is going to handle

**Table 2. Algebra of operators for dynamic and decentralized AXML materialization.**

Operator	Description
$\mu(v)$	The <i>materialize</i> operator tells the optimizer to determine an invocation plan for the service call node $v$ . Namely, to choose both a caller and an executor for $v$ among the peers in its location scope ( $L_v^C$ and $L_v^E$ , respectively). <i>Pre-conditions</i> : $L_v^E$ is not empty; none of its descendant nodes in the plan is an auxiliary operator; and every descendant node has at least one provider in its execution scope.
$\rho(v)$	The <i>retrieve</i> operator is slightly different from $\mu(v)$ . Additionally, it informs the optimizer that $v$ is a shared dependency. That is, it behaves like $\mu(v)$ with the additional cache operation for future retrievals, unless $v$ is already evaluated. Otherwise, it retrieves the invocation plan of $v$ from the cache. <i>Pre-conditions</i> : either there is an entry for $v$ in the cache or the same pre-conditions as for $\mu(v)$ hold.
<b>invoke</b> ( $v, IP_v$ )	This operator is interpreted during evaluation as an invocation of $v$ from peer $P^C$ to execute the requested Web service at peer $P^E$ , such that $IP_v = \langle P^E, P^C \rangle$ is an invocation plan of $v$ . <i>Pre-conditions</i> : all the dependencies of $v$ are inactive.
<b>fetch</b> ( $v$ )	It informs the optimizer to look for previous invocation results of $v$ at the system cache before materializing it. It corresponds to a physical version of $\rho$ . In particular, it behaves as <b>invoke</b> with the caching feature. <i>Pre-conditions</i> : either there is a cache entry for the invocation result of $v$ (if $v$ is not inactive), or there is an invocation plan for $v$ in the cache and all the dependencies of $v$ are inactive.
$\delta(v, IP_v)$	The <i>delegate</i> operator asks peer $P^C$ , from the invocation plan $IP_v = \langle P^E, P^C \rangle$ such that $P^C \in L_v^C$ , to materialize (possibly with some further optimization) the subplan rooted at $v$ , by solving itself all the necessary intensional parameters and collateral calls. <i>Pre-conditions</i> : all the $\delta$ operators in its subplan, as well as all the <b>invoke</b> operators whose caller is the current master peer, are evaluated. Also, $P^C$ must be a neighbor (that is, $P^C \in \mathcal{N}$ ).
$\Theta(v)$	The <i>optimize</i> operator denotes a request for a remote peer ( <i>i.e.</i> , neighbor) to optimize the subplan rooted at $v$ , possibly including its materialization. <i>Pre-condition</i> : at least one peer in $\mathcal{N}$ supports $\Theta$ .
<b>locate</b> ( $v$ )	This auxiliary operator denotes a request for a neighbor to discover the execution scope of both the Web service of $v$ and the services of all the $\mu$ operators in the subtree of $v$ that have an empty execution scope. <i>Pre-condition</i> : at least one peer in $\mathcal{N}$ supports <b>locate</b> .
<b>pipe</b>	It is used to simplify updating the dependency graph with intensional answers, as explained in Section 4.4. This operator represents a pipeline that gathers the results of its children and transmits them to its parent node in the plan. Its Web service may be executed either locally or at some peer in $\mathcal{N}$ . <i>Pre-condition</i> : all of its children are ready.

one or more service requests of the document; and

- (iii) the *auxiliary operators*  $\Theta$ , **locate** and **pipe**, which are mainly used to decentralize the optimization process. Similarly to  $\delta$ , these operators point to some basic collaboration service. Initially, they may lack the target peer, but the optimizer must set them to some specific neighbor (or locally, in case of **pipe**) before their evaluation.

Each algebraic operator in  $\mathcal{A}$  is associated to a specific set of actions, according to its goal. To be evaluated by the optimizer (thus triggering its actions), the operator has to satisfy some pre-conditions, as described in Table 2. The semantics of evaluating an operator varies according to its type. In general, the evaluation of abstract operators does not trigger any service invocation, but only makes the optimizer to analyze their alternative physical plans in order to choose the “best” options. Such an analysis usually results in replacing the abstract operators by their physi-

cal counterparts. For auxiliary operators, evaluating them means choosing a peer (neighbor) and then invoking the corresponding Web service for P2P collaboration. Observe that both abstract and auxiliary operators do not cause any changes to the AXML document, except for  $\Theta$  when it includes plan materialization (that is, the subplan evaluation is completely delegated to other peer). On the other hand, evaluating physical operators results in invoking some service calls of the AXML document and updating its contents.

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. Table 3 enumerates the possible transformations obtained by evaluating algebraic operators. Notice all physical operators are either consumed or replaced by an intensional answer. Also,

**Table 3. Evaluation of algebraic operators.**

Original operator	Resulting operator(s)
$\mu(v)$	either <b>invoke</b> ( $v, IP_v$ ) or $\delta(v, IP_v)$
$\rho(v)$	<b>fetch</b> ( $v$ )   <i>cached plan of v</i>
<b>invoke</b> ( $v, IP_v$ )	<i>none</i>   <i>intensional answer</i>
<b>fetch</b> ( $v$ )	<i>none</i>   <i>intensional answer</i>
$\delta(v, IP_v)$	<i>none</i>   <i>intensional answer</i>
$\Theta(v)$	<i>subplan for v</i>   <i>none</i>   <i>intens. answer</i>
<b>locate</b> ( $v$ )	<i>subplan rooted with <math>\mu(v)</math></i>
<b>pipe</b>	<i>none</i>

the  $\Theta$  operator may be consumed if it includes materializing its delegated subplan. Furthermore, we assume the optimizer can apply the following basic transformation rules:

1. it inserts  $\Theta$  operators to partition the children of a node if they are too numerous;
2. it replaces a  $\mu$  operator by **locate** if the subplan of  $\mu$  has at least some pre-defined percentage of nodes missing the execution scope (if the peer failed to identify the providers of these nodes); and
3. it replaces an **invoke** operator by  $\delta$  if its caller (*i.e.*,  $P^C$ ) is neither the master peer nor the caller of its parent node.

These rules are meant to be explored at the discretion of the optimizer, in different phases of the optimization strategy. For example, rule 1 is used in early optimization phases to break the plan into pieces of reasonable size, while rule 3 is used to determine delegation points when generating physical plans. Rule 2 is related to *contingency planning*, which we discuss in Section 6.4.

It is worth mentioning that our algebraic approach is extensible, since one can add other operators to  $\mathcal{A}$ , as well as new rules to handle these operators. For example, the **invoke** operator could be further specialized into other physical operators, such as a synchronous and an asynchronous operator (for continuous Web services).

**Generating initial plans.** Although the MFST is a canonical representation, usually there are many different plan alternatives for its trees, according to the algebraic operators used to handle the service requests. Nevertheless, the optimizer can rely on abstract operators to rather perform a simple (and fast!) analysis to generate initial plans. Such an analysis assumes that each service call node yields either a  $\mu$  or a  $\rho$  operator. (This last operator is used if the node is a replica.) The initial plan is called *abstract* because its nodes consist essentially of abstract operators.

The *GenerateInitialPlan* algorithm of Figure 16 shows the steps to generate an initial abstract plan from a dependency graph. After computing the MFST, an initial plan is set as a carbon copy of it. Then, for each tree, nodes are labelled by abstract operators accordingly. Once all the nodes

```

1  function GenerateInitialPlan( $\Delta$ ):  $\mathcal{M}_i$ 
2  begin
3    FlattenDependencyGraph( $\Delta$ )
4    Compute the MFST  $\Lambda$  of  $\Delta$ 
5    let  $\mathcal{M}_i$  such that  $\Lambda_i = \Lambda$ 
6    {Determine the labelling function  $\mathcal{O}_i$ }
7    for each tree  $st$  in  $\Lambda_i$  do
8      for each node  $v$  in  $st$  do
9        if  $v$  is a replicated node then
10         Replace  $v$  by  $\rho(v)$ 
11        else Replace  $v$  by  $\mu(v)$ 
12        end if
13      end for
14      if the master peer cannot evaluate  $st$  then
15        let  $v_{root}$  be the root of  $st$ 
16        Replace  $v_{root}$  by  $\Theta(v_{root})$ 
17      end if
18    end for
19    {Both  $\mathcal{L}_i$  and  $\succ_i$  are left undetermined for now.}
20    return  $\mathcal{M}_i$ 
21  end

```

**Figure 16. Algorithm to generate initial plans.**

of a tree are associated with algebraic operators, the optimizer decides (at 14) whether or not it is going to evaluate the respective subplan. Notice the optimizer may choose to delegate a subplan in many cases; for the initial plan, we consider this happens when either the MFST has too many trees or the master peer is overloaded. Also, for simplicity, we assume delegated subplans do not contain replicated nodes (to avoid problems due to data coupling). At this phase, we neither set an invocation schedule ( $\succ$ ) nor determine the location scope ( $\mathcal{L}$ ) for the abstract plan, which will be progressively defined during the optimization process.

Figure 17 depicts the initial abstract plan for the dependency graph of Figure 8. Each node represents an algebraic operator, which is specified in the node label. Nodes may also have collateral annotations (the “*cp*” reference under the label, in Figure 17). Additionally, the optimizer may annotate nodes with some supportive information, such as node height. Persistent nodes are denoted by double-line rectangles. We divide the materialization plan into three areas. The central area is the “*main plan*”, which consists of all the spanning trees of the materialization plan, such that subplans rooted by replicated nodes are represented by either a  $\rho$  or a **fetch** operator. The subplans of replicated nodes are kept in the “*cached plans*” area, and the “*cp*” references point to subplans in the “*collateral plans*” area. The master peer of our example is  $P_1$ , according to Figure 1.

If the dependency graph does not change between subsequent materialization requests (or if changes are not significant), then the optimizer can store the initial abstract plan of an AXML document for further reuse. Also, the optimizer can propagate occasional updates on the graph to the

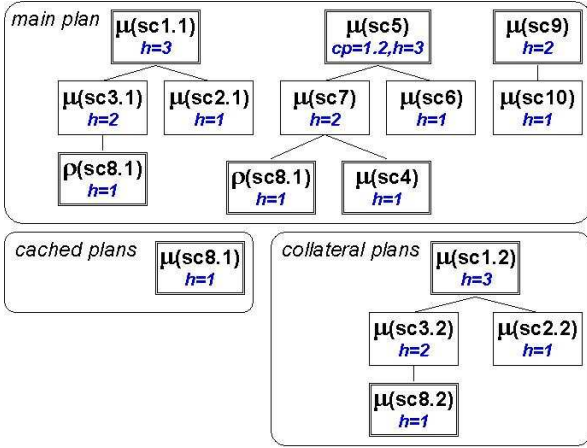


Figure 17. Initial abstract plan for the *Swap-Workspace* document.

plan according to the rules presented in Section 4.4, possibly by using **pipe** operators. For many changes, however, re-computing the initial plan from scratch may be less expensive than applying the respective updates.

### 6.3 Determining Materialization Tasks

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*. By materialization task, we consider a well-defined, self-contained goal of a materialization plan. To determine the tasks of a plan, we adopt an approach that focuses on its exit points, which correspond to the roots of the MFST. Also, for preliminary scheduling purpose, each task is associated with some evaluation priority, as defined next. Let  $\mathcal{T}$  be a finite set of tasks identifiers.

**DEFINITION 29** Given a plan  $\mathcal{M}$ , the expression  $\langle T, \pi \rangle$  denotes the set of materialization tasks of  $\mathcal{M}$ , where  $T$  is an injection of trees in  $\mathcal{M}$  into tasks in  $\mathcal{T}$ , and  $\pi$  associates each task  $t$  of  $\mathcal{T}$  with a priority level.

We consider that each tree of a materialization plan yields a task. Our choice for this criterion is motivated by two main reasons. Following Proposition 2, we know the materialization process converges at the exit points of a plan after a finite number of service invocations (assuming a snapshot semantics). Therefore, these nodes enable the optimizer to use an objective function to properly estimate the makespan of each task. Besides that, exit points are always first-level service calls, and they constitute the ultimate contents of the document materialization.

**Clustering tasks.** Due to shared dependencies (*i.e.*, nodes labelled by  $\rho$  or **fetch** operators), the trees of a materialization plan are not necessarily independent. That is, plan

```

1 function IdentifyTaskClusters( $\mathcal{M}$ ): Clusters
2 {Returns a collection of clusters indexed by shared
  nodes (or root nodes, for independent trees).}
3 begin
4   let Clusters =  $\emptyset$ 
5   for each tree st in  $\mathcal{M}$  do
6     let Rep be the set of replicated nodes of st
7     if Rep =  $\emptyset$  then {st is independent}
8     let  $v_{root}$  be the root of st
9     Create Clusters[ $v_{root}$ ]
10    Add st to Clusters[ $v_{root}$ ]
11  else {st has some shared nodes}
12    for each node v in Rep do
13      let node  $v_{origin}$  such that  $\rho(v) = v_{origin}$ 
14      if  $v_{origin} \notin Clusters$  then
15        Create Clusters[ $v_{origin}$ ]
16      end if
17      Add st to Clusters[ $v_{origin}$ ]
18    end for
19  end if
20 end for
21 return Clusters
22 end

```

Figure 18. Algorithm to identify clusters of materialization tasks.

trees can be grouped into overlay clusters, where the trees of each cluster share some service call results. Identifying the clusters of a plan can be done by a simple algorithm, as shown in Figure 18. Such a data coupling usually restricts (or, at least, complicates) the distributed processing of a materialization plan. Yet, once a replicated node is evaluated and its result is passed to the other replica accordingly, its corresponding cluster stops existing. Determining independent tasks has many advantages. In particular, it enables parallel execution and the decentralization of the optimization process, which fit quite well in P2P systems. Moreover, partitioning the plan may significantly reduce its optimization complexity, as discussed in Section 5.4. Hence, to overcome the data-coupling problem, we try to find a tasks evaluation order that would gradually increase the *parallelism potential* of the plan, thus allowing the optimizer to effectively explore P2P collaboration. For simplicity, let us assume each independent task of a plan is a cluster. Thus, we can estimate the parallelism potential of a plan  $\mathcal{M}$  as:

$$par_{\mathcal{M}} = |Clusters| . \quad (7)$$

where *Clusters* is the set of clusters of  $\mathcal{M}$ , including its independent tasks. Figure 19 describes the clusters found in the initial plan of Figure 17. Tasks in each cluster are represented by their root nodes. In this example, we have  $par_{\mathcal{M}} = 2$ . Observe this number may change after each task evaluation, according to the affected clusters.

$$\begin{aligned} Clusters[sc8.1] &= \{\mu(sc1.1), \mu(sc5)\} \\ Clusters[sc9] &= \{\mu(sc9)\} \end{aligned}$$

**Figure 19. Clusters of the *SwapWorkspace* plan.**

**Priority assignment.** To rank the tasks of a materialization plan, we consider the optimizer assigns priorities for them. Such a metric can be determined based on (possibly a combination of) several different task properties, such as:

- the number of service calls;
- the abstract critical path;
- the absolute height of the root node;
- the expected makespan, based on previous execution times;
- the number related clusters; and
- the number of replicated nodes.

The first three criteria basically define the size of a task; the optimizer can be configured to evaluated tasks following either a “smallest-first” or a “biggest-first” policy, according to the system requirements. Observe that to estimate the size of a task, ideally the optimizer should consider the dynamic critical path of the plan. However, the plan operators are rather abstract at this phase of the optimization strategy, and this information is not available yet (trying to get it would be very costly). Besides these size-based criteria, more user-defined properties could also be used, such as the presence of certain Web services requests.

The last two bullets are related to parallelism potential. These clustering-based criteria reflect, respectively, the external and internal data coupling of a task. Notice that both size- and clustering-based criteria are important to efficiently schedule materialization tasks. Therefore, let  $t$  be a materialization task and  $Clusters_t$  be the set of clusters that contain  $t$ . We compute the priority level  $\pi$  of  $t$  as:

$$\pi(t) = \pi_{size}(t) + \sum_{c \in Clusters_t} \pi_{cluster}(t, c) , \quad (8)$$

where  $\pi_{size}$  is the size-based priority of  $t$ , and  $\pi_{cluster}$  is the cluster priority of  $t$  (for each cluster  $c$  in  $Clusters_t$ ). We assume that  $\pi_{size}$  is estimated by some arbitrary function, regarding some criteria such as those we enumerated. On the other hand, the cluster priority of a task is given by:

$$\begin{aligned} \pi_{cluster}(t, c) &= \omega_{inter} \times \pi_{inter}(c) + \\ &\quad \omega_{intra} \times \pi_{intra}(t, c) . \end{aligned} \quad (9)$$

The terms  $\omega_{inter}$  and  $\omega_{intra}$  denote weights to adjust parallelism priority, for inter-cluster and intra-cluster parallelism, respectively. These weights allow the optimizer to adapt priority assignment to different performance requirements and problem topologies. The *inter-task parallelism degree*  $\pi_{inter}$  of a cluster  $c$  indicates the potential of  $c$  for

parallel evaluation (*i.e.*, if one of the tasks of  $c$  is evaluated), and it is given by:

$$\pi_{inter}(c) = |c| - 1 , \quad (10)$$

where  $|c|$  is the number of tasks on  $c$ . Analogously, the *intra-task parallelism degree*  $\pi_{intra}$  of a task  $t$  in a cluster  $c$  represents the branches of  $t$  that could be parallelized once  $c$  has been solved. This parameter is given by the number of replicated nodes of  $c$  in  $t$ , since two replicated nodes cannot be in the same branch of a task.

Observe the optimizer has to start the plan evaluation by distributing the tasks that were chosen to be delegated to other peers. Hence, we consider the highest priority level (denoted by  $\pi_{\infty}$ ) is assigned to delegated tasks by default. Recall that we assume these tasks do not belong to any cluster. Moreover, tasks with the same priority level can be further ordered by some deciding criterion, such as task size.

**Blocking versus non-blocking tasks.** In a regular peer (*i.e.*, with only one processor), 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 tasks delegation. On the other hand, for parallel peers, the optimizer can directly explore the clusters of a plan to speedup its evaluation, since each task blocks the evaluation only within the scope of its clusters. It is worth mentioning this can be simulated with a multi-thread system in regular peers. Nevertheless, empirical results [37] have shown that managing several simultaneous connections to Web services usually penalizes performance. Notice that tasks delegation is a quite different technique, since the optimizer can use asynchronous Web services for collaboration, thus avoiding lasting open connections.

When parallelism is imperative, the optimizer may compute dynamically the priority levels of materialization tasks, since these parameters can significantly change after each task evaluation.

## 6.4 Dynamic Plan Generation

As we saw in Section 5.4, the search space of alternative materialization plans is dramatically large even for very small problem configurations. In general, exponential-complexity search problems cannot be solved for any but the smallest instances [45]. Also, breaking the optimization problem into pieces fosters P2P collaboration. Although materialization tasks are natural candidates as a plan splitting unit, usually they are not necessarily small enough to be efficiently optimized. Hence, we further split materialization tasks into subplans, which are used to finally generate and rank alternative physical plans.

Two main aspects rule generating physical plans in AXML optimization: the location scope ( $\mathcal{L}$ ) of the re-

quested Web services; and the invocation schedule ( $\succ$ ) of the plan operators. Basically, the optimizer yields different combinations of service providers and callers (according to both the execution and delegation scopes of each service call node), along with different invocations sequences, to obtain alternative physical plans. Our optimization strategy adopts a variation of the *workflow-based generation approach* [15] to produce these plans. This is opposed to a task-based generation approach (which is quite popular in grid systems [24, 51]), where the optimizer makes greedy decisions for each plan operator. Notice a task in this context is just a plan operator. Typically, a greedy optimization algorithm processes a plan node-by-node, picking the best alternative for each node according to some heuristics and localized performance parameters. Its main advantages are the reduced complexity and the adaptability to changes in the system. Moreover, intermediate results can be shipped as soon as possible. However, in the AXML setting, there is a strong performance correlation between a plan node and its dependencies, and greedy decisions are usually very inefficient. In a workflow-based approach, the whole problem is considered to produce the search space. The analysis of complete plans enables the optimizer to reason about overall performance, but it is really inadequate for AXML materialization due to the exponential nature of the problem. We propose a new optimization strategy that combines advantages from these two approaches.

**Dynamically producing  $k$ -depth subplans.** The main idea of our strategy is to exploit a hybrid search technique, which performs a greedy analysis on  $k$ -depth subtrees of each materialization task. The  $k$  parameter determines the absolute height of each subplan that is analyzed by the optimizer. This parameter usually has a significant impact on the problem complexity, thus we assume  $k$  is a small integer (say, chosen from 2 to 4 inclusive). For each subtree, the optimizer generates and ranks alternative physical subplans, considering the relationships between the operators of the subtree (e.g., data transfers and invocation sequencing). Once a good physical subplan is selected, it is executed before the optimization process is resumed. In Figure 13, the steps from line 14 to 20 describe the core of the proposed strategy.

Observe that, through our dynamic strategy, the optimizer can have a bird’s eye perspective of height  $k$  of the materialization plan. This way, it can still be sensitive to overall performance issues, while keeping the plan complexity manageable. Moreover, the optimizer is able to adapt a plan to changes in the system, such as peers membership fluctuations. Also, it can handle incomplete problem specifications. For example, the optimizer does not have to know the execution scope of all the plan operators from the beginning of the optimization process. Instead, it can try to increase its knowledge of the problem gradu-

ally, as the plan is evaluated. The  $k$  parameter can be either determined by the peer administrator or inferred (based on some heuristics) from the absolute height of the task root.

**Computing split points.** To partition a task, the optimizer has to compute its *split points*, namely the nodes that root the  $k$ -depth subplans of the task. The algorithm outline is shown in Figure 20. Loosely speaking, the optimizer walks the task in some arbitrary tree traversal (e.g., in pre-order or in post-order); for each task operator, if its height is a multiple of  $k$ , then the optimizer sets it as a split point. The task root is considered a split point by definition. Notice we have to use the least height of the plan operators to properly split the task. The *mod* (for modulus) function in lines 33 and 34 of Figure 20 returns the remainder of an integer division.

Furthermore, two node occurrences require special treatment to determine the split points of a task. The first occurrence is of replicated nodes, which are essentially task leaves that point to some cached plans. When considering a replicated node for task splitting, either its respective cached plan is already evaluated or it is waiting for evaluation. In the last case, the optimizer can partition the corresponding cached plan similarly to a task, except that the root node of the cached plan (i.e., the shared node) is not considered a split point unless its least height is a multiple of  $k$ . The intuition is that a replicated node has to be replaced by its cached plan. However, since a task may have several replicated nodes pointing to the same cached plan, the optimizer has to choose exactly which replica is going to be solved first (and be replaced by its cached plan). For replicated nodes in different subplans, this is done following the split points evaluation order. Within a subplan, the optimizer can analyze the resulting subplan complexity for each replica replacement (based on the formula presented in Section 5.4), and choose the less-impacting change. Before starting to split a task, the optimizer retrieves the unsolved replicated nodes and copy their cached plans into the task accordingly. If a cached plan is already evaluated, the optimizer just need to replace the replicated node by a **fetch** operator.

The second special occurrence is of collateral annotations. The optimizer deals with this occurrence according to the size of the collateral plan. Small collateral plans can be attached to the main plan, thus enabling the optimizer to consider collateral annotations to determine an efficient invocation schedule. When splitting a task, to assign node heights, the root of an attached plan is handled as a child node. If collateral plans are potentially large (more specifically, when the node has  $ch \geq k$ ), then the optimizer has to handle them independently. This is because we assume the entire collateral plan has to be executed *after* the node that triggers it. Since optimization and execution are interleaved in our strategy, the optimizer can evaluate large collateral plans (including determining their split points) only imme-

diately after their origin nodes. In a more relaxed execution environment, we can admit the optimizer processing both a plan operator and the dependencies of its collateral call concurrently. In such a scenario, all the collateral plans have to be attached to the main plan. However, we limit plan attachment to small collateral calls.

Figure 21 shows the split points of the task rooted by operator  $\mu(\text{sc5})$ , from the initial plan in Figure 17. In this example, we also include the collateral plan rooted by  $\mu(\text{sc1.2})$ . Nonetheless, we remark that  $\mu(\text{sc1.2})$  is not attached to  $\mu(\text{sc5})$ . For simplicity, nodes are annotated with *ch* only if they have a collateral annotation, and with *lah* only if  $lah \neq h$ . Split points are indicated by large red arrows. We assume  $\mu(\text{sc5})$  is the first task to be evaluated, thus the operator  $\rho(\text{sc8.1})$  is replaced by its corresponding cached plan. Moreover, we have that  $k = 2$ , and the split points are:  $\mu(\text{sc5})$ ,  $\mu(\text{sc7})$ ,  $\mu(\text{sc1.2})$ , and  $\mu(\text{sc3.2})$ . Notice that, if collateral dependencies can be evaluated concurrently, then the split points of the task of  $\mu(\text{sc5})$  are:  $\mu(\text{sc5})$ ,  $\mu(\text{sc7})$ , and  $\mu(\text{sc3.2})$ . Although these points do not seem to change much in this case, the optimization of their subplans is quite different. This is mainly because the optimizer will consider materialization alternatives for both  $\mu(\text{sc5})$  and  $\mu(\text{sc1.2})$  before executing them. Also, the descendants of  $\mu(\text{sc1.2})$  may be executed before  $\mu(\text{sc5})$ .

**Scheduling subplans evaluation.** Basically, the subplans of a materialization task must be evaluated in topological order from the leaf nodes, such that child nodes are inspected first and collateral calls are properly triggered. This task traversal usually can be easily followed. However, since an abstract task does not enforce a total order on sibling operators, some subplans can be processed concurrently. Therefore, the optimizer has to determine an invocation schedule ( $\succ$ ) for the task. This schedule does not need to be completely specified, since only split points have to be considered at this moment.

To focus the scheduling analysis on split points, the optimizer summarizes a materialization task with a *subplans guide* (or s-guide, for short), which is an access structure that expresses the dependency relationships between subplans. An s-guide contains only the split points of a task, along with their descendant relationships. Notice that two split points may be siblings in the s-guide even if their respective operators do not have this relationship in the materialization task. We consider two basic approaches to schedule the nodes of an s-guide. The first approach consists in determining an evaluation order on the children of each node of the s-guide. Such an order can be defined by some size-based heuristic, such as “*deepest first*” (e.g., based on the absolute height of subplan root in the task), similarly to the size-based criteria used for tasks priority assignment in Section 6.3. The second scheduling approach is based on a “*ready list*”, which contains the subplans that can be im-

```

1 function SplitTask(t, k): SplitPoints
2 {Returns the set of k-depth split points of task t.}
3 begin
4   let SplitPoints =  $\emptyset$ 
5   {Handle replicated nodes}
6   for each cluster c in Clusterst do
7     let Rep be the set of replicated nodes of c in t
8     if the cached plan of c is evaluated then
9       for each node v in Rep do
10        Replace the operator of v by fetch
11      end for
12    else
13      Choose a node vm in Rep as the master replica
14      Replace vm by the cached plan of c
15      for each node v in Rep do
16        if  $v \neq v_m$  then
17          Replace the operator of v by fetch
18        end if
19      end for
20    end if
21  {Handle collateral annotations}
22  let Colls be the set of nodes containing collateral
23  annotations in t
24  for each node v in Colls do
25    if  $ch_v < k$  then
26      let tcoll be the collateral plan of v
27      Attach tcoll to v
28    end if
29  end for
30  for each node v in t do {search for split points}
31    let remainder = 0
32    if v has an attached plan then
33      remainder =  $\text{mod}(lah_v, k)$ 
34    else remainder =  $\text{mod}(h_v, k)$ 
35    end if
36    if remainder = 0 then
37      Add v to SplitPoints
38    end if
39  end for
40  let vroot be the root of t
41  if  $v_{root} \notin \text{SplitPoints}$  then
42    Add vroot to SplitPoints
43  end if
44  return SplitPoints
45 end

```

Figure 20. Algorithm to compute split points.

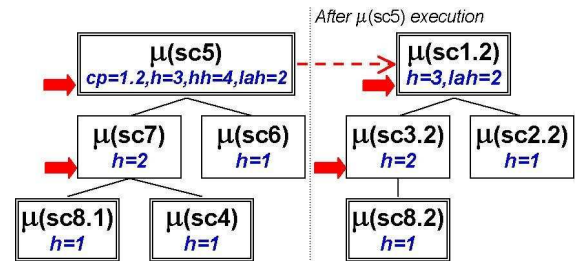


Figure 21. Split points of the materialization task rooted by  $\mu(\text{sc5})$ , assuming that  $k = 2$ .

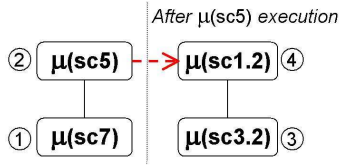


Figure 22. S-guide with evaluation order.

mediately evaluated. Initially, the ready list contains all the leaf nodes of the s-guide. Once a subplan is evaluated, its parent node in the s-guide is added to the ready list. The optimizer makes scheduling decisions only for the subplans on this list; to choose the next subplan to be evaluated, the optimizer can apply some size-based heuristic on these subplans. Although this approach relies on localized decisions, since it ignores the overall task makespan, it can improve intra-task parallelism and workload distribution.

Figure 22 shows the s-guide for the task rooted by  $\mu(\text{sc5})$  of our running example, along with the collateral plan of  $\mu(\text{sc5})$ . Circled numbers indicate the subplans evaluation order; nonetheless, only the topological order can be observed in this example, since this task does not have concurrent subplans.

**Locating Web services.** In AXML documents, Web services requests may be specified with abstract references, which need to be converted into some concrete service endpoints in order to generate physical materialization plans. The location scope of a plan is an essential input of our optimization problem. It determines a two-dimensional search space, since: (i) abstract references may be converted into many alternative addresses of Web service providers; and (ii) peers can collaborate to materialize an AXML document. Basically, the optimizer can retrieve this information from an internal peer catalog, a catalog server in the network (such as a UDDI server [52]), and from other peers. The optimizer tries to annotate operators with their respective execution and delegation scopes, primarily accessing the peer catalog and registered catalog servers. If some operators miss this information, then the optimizer can explore a dynamic discovery method, using the **locate** operator to gather the missing scopes from neighbors. This enables the optimizer to perform basic *contingency planning*. That is, the optimizer can try to automatically recover from failing in determining the providers of some Web services, by collaborating with other peers.

Since P2P systems are highly dynamic, the location scope of a materialization plan should be preferably provided by *late binding*. In our optimization strategy, this is done incrementally, for each subplan of a task, in some arbitrary tree walk. By restricting the location scope to subplans, we enable the optimizer to defer retrieving Web services addresses until they are really necessary. Hence, the

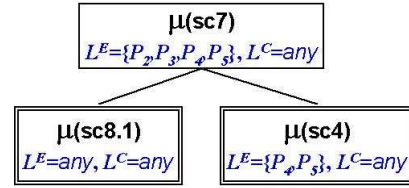


Figure 23. Abstract subplan rooted by  $\mu(\text{sc7})$  annotated with location scope.

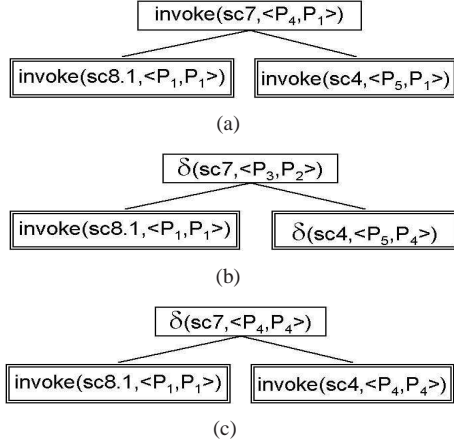
optimizer can use more fresh (and reliable!) information.

When annotating the operators of a subplan with location scope, the optimizer replaces every  $\mu$  operator that misses this information by a **locate** operator. After visiting the entire subplan, it checks if **locate** operators can be grouped by subtrees. This analysis aims at reducing communication costs. Moreover, the optimizer may decide to start evaluating another subplan (or other operators of the current subplan) while it waits for the result of a **locate** operator. Also, if too many operators of a subplan miss the location scope, it is quite likely that the optimizer will face difficulties evaluating the subplan, due to the lack of supportive information (e.g., cost parameters and statistics). To solve this problem, either peers may embed supportive information in **locate** results, or the optimizer may delegate the entire subplan to another peer. In Figure 23, we show the subplan rooted by  $\mu(\text{sc7})$  annotated with the location scope according to the services distribution of Figure 1(b); we assume any peer can invoke all the requested Web services.

**Generating and ranking physical subplans.** Having specified the location scope of an abstract subplan, the optimizer is able to enumerate and analyze the costs of its physical alternatives. This is a key phase of our optimization strategy, and it mainly concerns reasoning about resource planning and invocation scheduling.

Algorithms for job scheduling usually rely on a *scheduling list* [33], which is essentially a sequence of nodes ordered by priority. To define the schedule, the first node is removed from the list and allocated to some available resource, repeatedly until the list is empty. The scheduling list can be either *static* or *dynamic*, according to whether the optimizer recomputes the priorities of unscheduled nodes after each allocation. Also, there are several different policies to assign priorities to nodes, most of them focused on the critical path of a plan. Notice that defining an invocation schedule depends on the available resources. Therefore, we consider the optimizer first chooses the peers that are going to participate in the materialization process, and then attacks the problem of finding a good invocation sequence based on the critical path.

Nevertheless, resource planning is also affected by the order in which physical operators are processed. For example, if two concurrent operators are assigned to be executed



**Figure 24.** Some alternative physical conversions for the subplan rooted by  $\mu(\text{sc7})$ .

at the same peer, they are probably going to be processed sequentially. Even if the peer offers the best execution cost for each operator, the overall performance may be worst than assigning one of the operators to another peer (and possibly profiting from parallel execution). To tackle this problem without having to previously define an invocation schedule, we rely on a cost model that considers the workload of operators assigned to peers (see Section 7 for further details). In general, the optimizer performs the following steps:

1. Generate the search space of *partial subplans*, whose nodes are set only with the execution scope;
2. For each partial subplan, generate the search space of physical subplans by setting the delegation scope of the algebraic operators; and
3. For each physical subplan, generate the search space of alternative invocation schedules.

However, we are considering decentralized execution environments, where an invocation schedule cannot be globally enforced. Hence, we focus our strategy on resource planning, assuming the optimizer chooses the “best” physical subplan and then determines a good invocation schedule only for this subplan. Observe that, by starting with partial plans based only on service providers, the optimizer is able to make use of some heuristics (e.g., the “Context” heuristic [44]) to prune the search space of physical subplans, as in the eager enumeration approach presented next.

An abstract subplan works as a template for generating alternative physical subplans. A straightforward (and mostly inefficient) approach to produce these subplans consists in exhaustively enumerating the search space, such that the optimizer yields all the possible combinations of location scope and invocation schedule. To generate a physical subplan, first the optimizer converts every  $\mu$  operator into an **invoke** operator by picking a definite invocation plan for the corresponding service call node from its  $\widehat{IP}$ . Once the

1	procedure <i>EagerEnumeration</i> ( $sp, X$ )
2	{Enumerates the search space of the abstract subplan $sp$ by applying a two-level cost analysis on its top $X$ partial plans.}
3	begin
4	let $Phys = \emptyset$ {set of physical subplans}
5	Generate all the partial subplans of $sp$
6	Rank partial subplans by cost of potential transfers
7	let $PP$ be the set of top $X$ partial subplans of $sp$
8	for each subplan $p$ in $PP$ do
9	Generate physical subplans of $p$ into $Phys$
10	end for
11	Rank subplans of $Phys$ by makespan
12	Let $\mathcal{M}_{best}$ be the best subplan in $Phys$
13	Generate an invocation schedule for $\mathcal{M}_{best}$
14	end

**Figure 25.** Eager enumeration of alternative physical plans.

location scope of the subplan is determined, the optimizer applies a transformation rule that searches for *delegation points*, namely the **invoke** operators whose caller is neither the master peer nor the caller of the parent node. This rule replaces **invoke** operators by  $\delta$  accordingly. Figure 24 shows some alternative physical conversions for the subplan rooted by  $\mu(\text{sc7})$ . The optimizer uses a cost model to rank these alternative physical subplans by their makespan.

Although exhaustive algorithms are often unfeasible for complete materialization plans, they may become useful in our optimization strategy, since the optimizer deals with subplans of reduced size. Furthermore, with our dynamic approach, the optimizer can use intermediary results to reduce error propagation in cost prediction, thereby improving the cost analysis of plan operators.

Another approach is based on an *eager enumeration* of the search space, where the optimizer generates alternative physical subplans only for the top  $X$  partial plans, as shown in Figure 25. The optimizer relies on a two-levels cost model. In the first level, only the execution scope of the operators is considered, and the optimizer tries to reduce the costs of transferring invocation results to the master peer, based on the set of distinct peers of each partial subplan. This selects subplans involving a few peers, which have a fast link to the master peer. Although clearly suboptimal, this heuristic may be efficient when communication costs are predominant. Notice that other criteria could be used to filter partial plans. In the second level, only the top  $X$  partial subplans are used to generate physical subplans, which are fully analyzed and ranked by their makespan. An eager approach is interesting when the size of the search space is critical even for low values of  $k$ .

It is worth mentioning that many different algorithms can be exploited to produce the search space of physical subplans. Yet, our work rather puts emphasis on breaking the

```

1 procedure EvaluatePhysicalSubplan( $v_{root}$ )
2 {Evaluates subplan rooted by  $v_{root}$  operator.}
3 begin
4 let Children be the set of child nodes of  $v_{root}$ 
5 for each node  $v_{child}$  in Children in order of  $\succ$  do
6   EvaluatePhysicalSubplan( $v_{child}$ )
7 end for
8 if  $P_{v_{root}}^C = P_m$  or  $\mathcal{O}(v_{root}) \in \{\delta, \Theta\}$  then
9   Evaluate  $v_{root}$ 
10 end if
11 end

```

Figure 26. Algorithm to evaluate subplans.

optimization problem to reduce the search space, based on the structure of materialization plans. Still, most of these algorithms can perform efficiently with our strategy, since it enables them to handle smaller problems.

**Evaluating physical subplans.** An important feature of our strategy is that planning and execution are interleaved during the AXML materialization process. After selecting a physical subplan, the optimizer evaluates its operators in a bottom-up traversal, following the specified invocation schedule, as described in the algorithm of Figure 26. Recall that we consider a decentralized execution model where parts of a plan may be delegated to other peers. Therefore, the optimizer actually evaluates only:

- *local operators*, namely those whose caller is the master peer; and
- *delegation points*, which are represented by either  $\delta$  or  $\Theta$  operators.

Moreover, although delegated points may be nested in a subplan (e.g., the  $\delta$  operators in Figure 24(b)), we assume the optimizer does not reason about transitive delegation, and all of these points are supposed to return their result to the master peer. Nonetheless, it is worth mentioning that this restriction does not prevent peers from deciding to delegate parts of a subplan coming from another peer.

Evaluating a plan operator involves the steps shown in Figure 27. Basically, the optimizer builds the required inputs, invokes the corresponding service call, gathers the result, and updates both the materialization plan and the AXML document accordingly. Also, the optimizer has to check on the result to verify whether or not it contains intensional answers. If it is the case, the optimizer must update the subplan with the new service call nodes. To simplify the plan update, the optimizer can employ **pipe** operators based on the techniques presented in Section 4.4 for pipelined graphs. Namely, the optimizer generates an initial abstract plan for the intensional answer, and inserts each task found in the answer into the current subplan by connecting the new task roots as children of a **pipe** operator.

Notice that an intensional answer may significantly change the subplan, specially its height and consequently

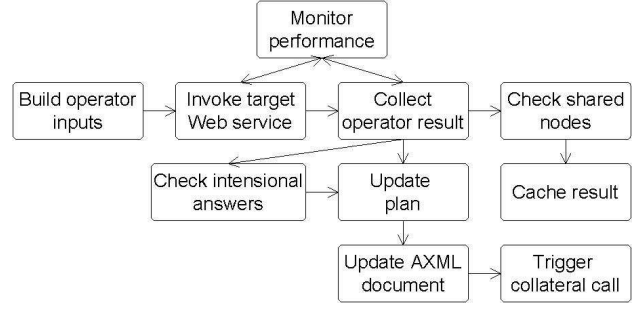


Figure 27. Steps of evaluating an operator of a physical subplan.

its complexity. Hence, the optimizer may have to review the subplan splitting to accommodate the new operators. Moreover, some optimization decisions may become wrong in the presence of new plan operators, and the optimizer may have to reconsider the chosen location scope and invocation schedule. Yet, in our strategy such an analysis does not ripple to the whole materialization plan. When an operator evaluation results in some intensional answer, the changes affect only the subplan that is being evaluated since the rest of the plan consists of abstract operators only. This avoids unnecessary re-optimization.

If the result of an operator evaluation corresponds to some shared node, then it has to be properly loaded into the cache. To manage keeping these results in the cache, the optimizer maintains for each cached plan a counter of *hanging references*, which represent the non-evaluated  $\rho$  and **fetch** operators of the main plan. Recall these operators stand for the replicated nodes of the plan. Initially, each counter is set with the number of respective replicated nodes. Whenever a replicated node is evaluated, its corresponding counter is decreased. The result of a shared node is kept in the cache while its counter is not zeroed. Some additional checking may also be applied to an operator result; for example, the optimizer may want to validate the result against some expected datatypes before updating the materialization plan and AXML document.

During an operator evaluation, the optimizer monitors the performance of both service invocation and result transfer. It may ask for subplan re-optimization in case the performance significantly surpasses the expected costs. Alternatively, the optimizer may exploit some rescheduling technique, similarly to query scrambling [53], to hide evaluation delays. Furthermore, if some error situation arises, the optimizer may resubmit an operator for evaluation (considering the user allows it). For example, an operator re-evaluation may occur due to service execution error, communication timeout, or insufficient/incorrect result (w.r.t. some predefined criteria, such as expected datatypes).

Once an operator is successfully evaluated, the optimizer must check if it has a collateral call to be triggered.

Nonetheless, this is done only for local operators, since we assume that delegating subplans includes processing their collateral calls remotely.

**Foreseeing intensional answers.** An advantage of our optimization strategy is that intensional answers affect only the current subplan evaluation. However, we consider that the optimizer reacts to intensional answers just when they occur. Namely, it does not try to foresee whether a Web service may return AXML data, using this information for performance prediction. An alternative for the optimizer to encompass intensional answers in the physical subplans enumeration is to look at the expected result type of the required Web services, as proposed in [3].

Our strategy can be extended to support this feature by exploiting the idea of enabling condition behind collateral calls. That is, likewise a collateral call, an intensional answer represents service call nodes that have to be invoked after their origin call. Thus, the optimizer may insert some “special collateral calls” into the materialization plan to represent intensional answers. An additional enabling condition must be specified for these special calls, to guarantee that they are going to be triggered only if they are actually returned as the evaluation result of their origin node. Notice this may significantly inflate in advance the size of a materialization plan. Although our dynamic optimization strategy is quite adequate for large plans, handling intensional answers *a priori* is rather a complex subject, and we leave a deeper analysis of this issue as future work.

## 6.5 Delegating AXML Optimization

The algebra proposed in Section 6.2 contains some operators specially tailored for P2P collaboration. They represent different collaboration possibilities to process service calls of an AXML document, namely of delegating: plan optimization ( $\Theta$ ); plan evaluation ( $\delta$ ); and Web service location (**locate**). These collaboration operators are executed remotely by their target peers, and their results are properly merged into the materialization plan by the master peer. From the perspective of the master peer ( $P_m$ ), evaluating these operators involves three basic phases:

1. *Target selection*, that is when  $P_m$  identifies possible collaborators, and selects a target peer among them to handle the subplan rooted by the delegated operator;
2. *Collaboration contracting*, when  $P_m$  builds the delegated subplan (including all the necessary input data), and sends it to the chosen target peer; and
3. *Result delivery*, when the target peer sends the result of the delegated subplan back to  $P_m$ .

Next we discuss issues involved in each of these phases.

**Selecting target peers.** This phase focuses on providing an execution scope for the collaboration operator. Observe that

both the execution and the delegation scope of a collaboration operator may differ from those of its service call node. By default, the caller of a collaboration operator is always the master peer of its subplan, since we consider the master peer cannot enforce transitive delegation (*i.e.*, the delegation scope is empty). That is, we assume:

$$L_{op}^C = \{P_m\}, \text{ if } op \in \{\delta, \Theta, \text{locate}\}. \quad (11)$$

Here we denoted the delegation scope  $L_{op}^C$  of an operator  $op$  similarly to the notation used for service call nodes, as defined in Section 5.

Going further, collaboration operators request P2P-specific Web services, which are provided by AXML-enabled peers. Therefore, their execution scope is mostly determined by the peers in  $\mathcal{N}$ . A particular case is that of  $\delta$  operators, which is used mostly to reduce data transfer costs. The executor of a  $\delta$  operator is chosen essentially from the delegation scope of its service call node, assuming the corresponding peers support collaboration. Namely, given  $\delta(v, IP_v)$ , we have that:

$$P_\delta^C = P_m \text{ and } P_\delta^E = P_v^C, \quad (12)$$

where  $P_v^C \in (L_v^C \cap \mathcal{N})$ .

For  $\Theta$  and **locate** operators, there is seldom a direct relationship between the execution scope of the collaboration operator and its service call node. Thus, the execution scope is usually arbitrarily chosen by the master peer based on some QoS metrics. For instance, the optimizer may use SLA (Service-Level Agreement) specifications of the P2P-collaboration Web services to prune target candidates, similarly to the approach proposed in [35]. Target selection can be either: *static*, if it is based on existing statistics and costs; or *dynamic*, when the optimizer polls specialized servers on the network for fresh information on target candidates. Dynamic selection may become necessary in scenarios such as ad-hoc P2P systems. Nevertheless, it is usually quite expensive due to the communication costs of its required control flows, and it must be carefully explored (*i.e.*, mostly in case of missing or highly-outdated statistics). Although our optimization strategy can support both static and dynamic target selection, for simplicity we considered only the static approach. Namely, we assume  $P_m$  always chooses targets from  $\mathcal{N}$ .

Since a materialization subplan may contain several collaboration operators, the optimizer can decide to set target peers either for the entire subplan or just before evaluating each operator. Furthermore, target selection may require some P2P negotiation to ensure the chosen peer is available (and willing!) to receive and process the delegated subplan. This additional step consists basically in contacting the target peer to confirm its participation.

**Collaboration contracting.** Once a target peer is chosen, the master peer has to properly build the delegated subplan. That is, the optimizer has to produce a subplan containing:

- the collaboration operator, along with all its (active) descendant nodes in the materialization plan;
- the data elements that should be passed as input of service call nodes. To retrieve these elements, the optimizer may have to evaluate XPath expressions for non-concrete parameters; and
- the collateral calls, along with their dependencies (if any).

We assume all local nodes are evaluated before sending a delegated subplan. That is, only their results are embedded into the subplan. Furthermore, since usually there is a mismatch between a materialization plan and the AXML tree of its service call nodes, a delegated subplan follows the plan structure shown in Figure 17. Such a structure organizes the plan into three areas, and it aims at avoiding unnecessary node replication. This is very important to reduce communication costs.

Also, some sideways information can be passed embedded into a delegated subplan. For example, to avoid delegated subplans to be endlessly forwarded, we assume a *hops counter* goes along on each subplan. In this case, a hop represents each time a subplan (or a part of it) is delegated to another peer. This information is related to the *delegation trace* of a subplan, which indicates all the peers that have ever processed it.

Before sending a delegation subplan to its target peer, the optimizer has to serialize it (in the AXML format) and marshal the resulting AXML data into a SOAP envelope. A serialized subplan contains both a header and a body section. The subplan header keeps general properties of the subplan, such as its delegation trace and some optimization hints (e.g., previously estimated plan costs), whereas the body encodes the plan operators and their input data. On the target peer side, a subplan has to be unmarshaled and parsed before resuming the materialization process. Notice these operations incur processing costs which the optimizer has to consider when analyzing materialization alternatives, as described in Section 7.

**Result Delivery.** When the master peer receives the result of a delegated subplan, it has to parse the serialized SOAP response, insert the embedded service results into the AXML document (if necessary), and update the materialization plan accordingly. The materialized contents that are embedded into the result of a subplan consists of a forest of nodes. The master peer uses a “origin” parameter to identify the respective service calls in the AXML document.

While the **locate** operator aims at retrieving only supportive information for the delegated subplan, both  $\delta$  and  $\Theta$  operators may involve some AXML materialization. In case the result contains materialized contents, it is basically composed by the results of:

- the children nodes of the collaboration operator;

- the operators related to persistent service call nodes; and
- all the collateral calls triggered in the subplan.

Recall that persistent nodes are essentially first-level service calls and shared dependencies. Also, notice that although temporary results are not required in the result of a delegated subplan, we consider the results of their collateral calls are sent back to the master peer.

**Horizontal plan partitioning.** The Split algorithm breaks materialization tasks in depth, to attack the optimization problem incrementally. However, if some plan operator has too many children, such a technique is not effective. To overcome this drawback, the optimizer may also partition a materialization subplan horizontally by inserting some collaboration operators. In particular, the  $\Theta$  operator enables the master peer to decentralize the optimization of a plan.

Inserting  $\Theta$  operators in a subplan is similar to procedure used to update a dependency graph with *pipe* nodes. The operators that are going to be remotely evaluated are connected as children of the  $\Theta$  operator.

To decide when horizontally partition a subplan, the optimizer can either use the resulting complexity of the subplan or some fixed horizontal splitting parameter  $k_h$ . Furthermore, instead of asking a target peer to return only one physical plan as the result of a  $\Theta$  operator, the master peer may consider getting a set of alternative solutions. Peers may also agree in setting a limit for the maximum number of inspected alternatives for a delegated subplan, thus limiting the expected optimization time.

## 7 Cost Analysis of AXML Materialization

Although heuristics are very useful when dealing with complex optimization problems, in order to compare the performance of alternative materialization subplans, the optimizer requires objective metrics. In this Section, we present a set cost formula to model the performance of AXML materialization. The proposed cost model considers relevant aspects, such as:

- heterogeneous machines and communication links;
- equivalent Web services;
- subplans delegation;
- parallel execution;
- invocation dependencies and collateral calls; and
- processing workload of peers.

As we discussed in Section 5.4, the performance of alternative physical plans is represented by their makespan. This metric represents the clock-time spent from the start of the materialization process to the moment when the master peer gathers into the AXML document the results of all its embedded service call nodes. Intuitively, the makespan is

determined by accounting the costs of Web services invocations/executions, and of communication messages between peers. To estimate the makespan of an entire plan, these costs must be properly combined according to the plan operators and their inter-relationships.

On the other hand, the optimization strategy of XCraft is based on dynamic and incremental plan generation. The optimizer has to analyze plans that contain abstract operators. To cope with this incremental approach, in XCraft we adopt a *multi-leveled cost analysis* based on three distinct cost models. In the first level, the optimizer estimates the costs of a materialization plan in terms of its complexity (*i.e.*, the size of its search space). For this purpose, we use the complexity bounds determined in Section 5.4. This enables the optimizer to avoid processing plans with too expensive analysis.

With the second-level cost model, the optimizer can limit the plans analysis by using some heuristic criteria. In particular, we assume it estimates costs for partial plans. In this case, the optimizer considers only the execution scope of plan operators. The idea is to emphasize the proximity (in terms of communication costs) of peers that are candidates to execute the requested Web services, with respect to the master peer. This way, the optimizer can rank partial plans before generating their physical alternatives. Finally, the third-level cost model consists of a comprehensive response-time analysis of plan operators.

We describe in Section 7.1 basic cost ingredients. The overall formula used in the second-level cost analysis is presented in Section 7.2, while Section 7.3 details the main components of the third-level cost model.

## 7.1 Representing Heterogenous Scenarios

Traditional cost models usually take into account very detailed information on the machines, such as I/O and CPU operations costs. Since P2P systems are quite heterogeneous and with autonomous peers, gathering this information is seldom possible. In [44], we modeled the basic costs of invoking a Web service by focusing on the response time of its major operations. According to [44], costs are computed from the client viewpoint. The response time of a service call  $v$  is denoted by  $ccost(v, P_i, P_j)$ , where  $P_i$  is client peer and  $P_j$  is the executor of  $v$ .

We consider that costs are given in time units and that peers may have different performance capabilities (CPU clock, RAM memory, etc.). For simplicity, we assume they are sequential machines, namely they can execute only one service call at a time and requested service executions have to join a queue at each peer. The *execution queue* of a peer  $P$  is an ordered list of service executions denoted by  $q_P$ . We assume execution queues are infinite and work on a “first arrived, first served” basis. The term  $|q_P|$  represents the

length of  $q_P$ , that is the number of service executions waiting in  $q_P$ .

The proposed cost model is sensitive only to *inter-operator parallelism*, such that a single service execution cannot be split among peers. Also, service executions are *non-preemptive*, namely they cannot be interrupted to be resumed by another peer. We denote by  $ET(v, P)$  the average execution time of the service call node  $v$  at peer  $P$ . The terms  $callSize(v)$  and  $resSize(v)$  represent the size in bytes of the input and output parameters of  $v$ , respectively.

We expect peers interconnected through heterogeneous links. Therefore, the costs of transferring  $X$  bytes of data from peer  $P_i$  to  $P_j$  costs is:

$$net(X, P_i, P_j) = \frac{X}{B(P_i, P_j)} \quad , \quad (13)$$

where  $B(P_i, P_j)$  denotes the bandwidth of the link from  $P_i$  to  $P_j$ . Notice that  $B(P_i, P_j)$  may be different from  $B(P_j, P_i)$ . It is worth mentioning that both the input parameters and the result of a service call may involve large data transfers. For example, in Figure 2(a), the input parameter of the call `sc9` is expected to be a PDF file with possibly a few Mega bytes, though its result is only an excerpt of the input.

## 7.2 Heuristic Cost Analysis

This cost analysis points out materialization plans involving fewer peers, as well as peers that are close (in terms of communication costs) to the master peer. It is worth noting this heuristic selection tends to stretch out the makespace of the resulting physical plans. This happens because communication costs are more weighted, and parallel executions may be ignored. Nonetheless, in very large search spaces or in scenarios with high communication costs, this preliminary cost analysis can be helpful.

Let be the set of distinct peers involved in a materialization plan  $\mathcal{M}$ . The heuristic cost of  $\mathcal{M}$  is:

$$hcost(\mathcal{M}) = \sum_{\forall P_i \in DP_{\mathcal{M}}} net(ARS, P_m, P_i) \quad , \quad (14)$$

where:

- $DP_{\mathcal{M}}$  is the set of distinct peers in the execution scope of  $\mathcal{M}$ ;
- $P_m$  is the master peer of  $\mathcal{M}$ ; and
- $ARS$  is a constant for the average result size of service calls in  $\mathcal{M}$ .

Notice that usually the heuristic costs of an entire plan can be quickly estimated.

This heuristic cost model is used to rank plans previously to a detailed analysis. For instance, the optimizer may apply this analysis first, and then calculate detailed costs only for  $n$  best alternative plans.

### 7.3 Costs of Plan Operators

Given a materialization plan  $\mathcal{M}$ , the cost of an operator  $op \in \mathcal{M}$  is estimated as:

$$\begin{aligned} cost(op) &= ccost(op) + dcost(op) \\ &+ \sum_{\forall op_{cc}, op \hookrightarrow op_{cc}} cost(op_{cc}) \quad , \quad (15) \end{aligned}$$

where:

- the term  $ccost(op)$  is the client-side cost, as provided in [44];
- the term  $dcost(op)$  is the cost of the dependencies of  $op$ ; and
- $op_{cc}$  indicates collateral calls.

Remember plan operators correspond to service call invocations, thus the cost analysis is uniform for different operator types. Therefore, the overall cost of a materialization plan  $\mathcal{M}$  can be calculated recursively as:

$$overall\_cost(\mathcal{M}) = \sum_{\forall r_i \in ROOT_{\mathcal{M}}} cost(r_i) \quad , \quad (16)$$

where  $ROOT_{\mathcal{M}}$  is the set of root operators of  $\mathcal{M}$ .

**Invocation dependencies.** Since peers have processing queues, if two service executions are mapped the same peer, they run sequentially. Thus, to properly estimate the costs of invocation dependencies, it is necessary to consider the processing workload of peers. We restrict this analysis to the children of each operator in a materialization plan.

Given a plan operator  $op$ , we take the set  $DP'_{op}$  of distinct peers involved in the evaluation of its dependencies. Notice  $DP'_{op}$  does not concern the location scope of  $op$ . Then, for each peer  $P_i$  in  $DP'_{op}$ , we calculate its total processing load as:

$$total\_load(P_i) = \sum_{\forall op_j \in Children(op, P_i)} cost(op_j) \quad , \quad (17)$$

where  $Children(op, P_i)$  is the set of invocation dependencies of  $op$  that are executed by  $P_i$ . From this result, we can estimate the cost of the dependencies of the operator  $op$  as:

$$dcost(op) = \max_{\forall P_i \in DP'_{op}} (total\_load(P_i)) \quad . \quad (18)$$

It is worth noting that if operators run in blocking mode, then we have:

$$dcost(op) = \sum_{\forall op_j \in Children(op)} cost(op_j) \quad , \quad (19)$$

where  $Children(op)$  is the set of all the dependencies of  $op$ . In this case, parallel executions are disregarded.

**Peer energy factor.** For P2P systems with low bandwidth rates, the optimizer may tend to sacrifice parallel execution

for the sake of avoiding data transfers. In this context, peers may be overloaded with service assignments. Thus, for fair costs ranking, the optimizer has to consider performance penalties from peers workload. When estimating the costs of a plan operator  $op$ , we define the energy factor of a peer  $P_i$  in  $DP'_{op}$  as:

$$ef(P_i) = \frac{Bogop_{P_i}}{\sum_{\forall op_j \text{ with } P_i \text{ in } IP_{op_j}} ET(op_j, P_i)} \quad , \quad (20)$$

where  $Bogop_P$  is the BogoMips [58] speed of  $P_i$ . The energy factor can be used to add some extra response time for service executions at  $P_i$ , which are due to its performance penalties. Another (simpler and more imprecise) way to estimate this factor would be computing the inverse of the number of plan operators assigned to the peers.

**Delegation costs.** Delegating a subplan encompasses the costs of: (i) sending the plan along with its input data; (ii) evaluating the plan at the remote peer; (iii) returning the plan along with its persistent results back to the master; and (iv) updating the AXML document. This means the costs of  $\delta$  operators can be computed by Equation 15, with small changes in the estimation of input and result sizes. In particular, the input size must take into account the results of dependencies that were previously evaluated.

## 8 XCraft Architecture

We present a service-oriented optimizer architecture called XCraft, which enables dynamic and decentralized materialization of AXML documents, and supports the proposed optimization strategy. XCraft works in a multi-thread fashion, as a facade component of the ActiveXML peer; it interacts with the AXML document repository, the services and statistics catalogs, and the Service Call Handler.

**Main XCraft modules.** Figure 28 shows the main modules of the XCraft optimizer, which 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). 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 both the execution and the delegation scope of the current subplan.

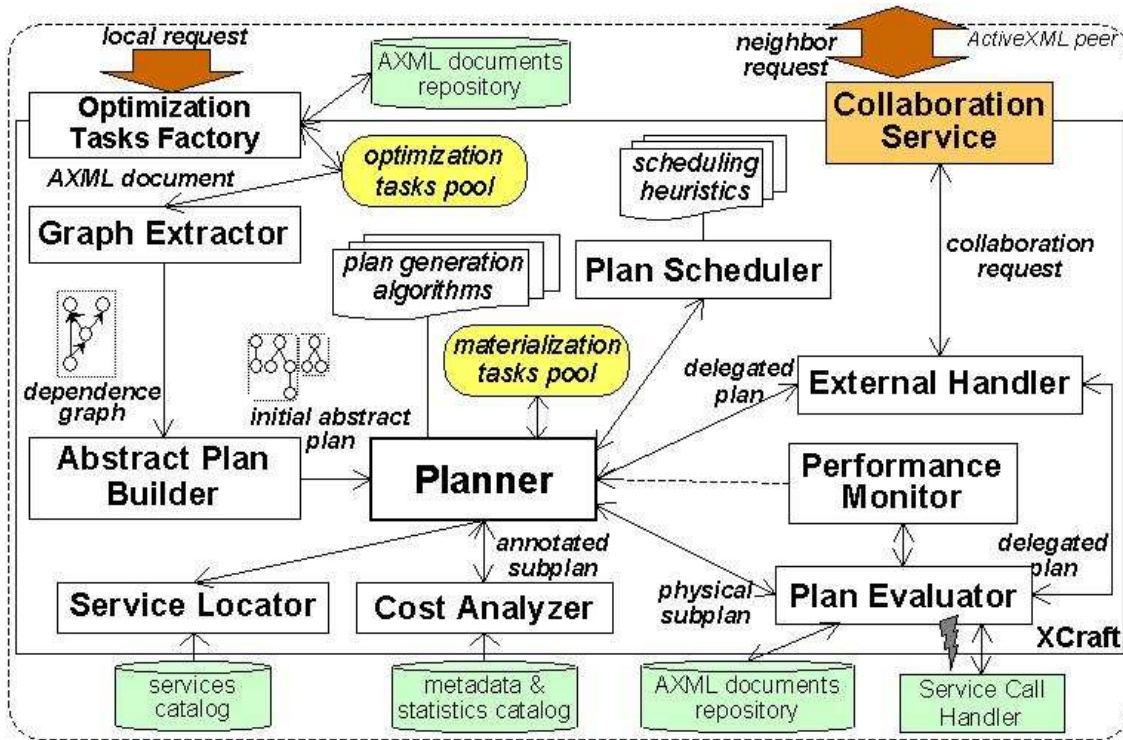


Figure 28. System architecture of the XCraft optimizer.

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. The Planner asks the *Cost Analyzer* to estimate the makespan of physical subplans, registering the subplan with the best makespan during the search. Then, it sends the overall best subplan to the *Plan Evaluator*, which executes it and returns the results, possibly along with intensional answers.

As the evaluation proceeds, service call results are gathered and merged into the AXML document. Furthermore, the *Performance Monitor* watches over operators evaluation, and it can trigger subplan re-optimization if necessary. Both the Planner and the Plan Evaluator may also get plans coming from the *External Handler*, which processes requests from other peers. These requests are received by the *Collaboration Service*, which implements the interface of the basic Web services for P2P collaboration. For collaboration requests, evaluation results are sent back to the origin peer in the Collaboration Service reply.

For simplicity, we omitted in Figure 28 two modules of the XCraft architecture: the *Plan Cache*, where the optimizer keeps shared plans and their results; and the *Optimizer Profile Loader*, which sets relevant properties to configure the behavior of the XCraft internal modules (e.g., the heuristic to be used by the Plan Scheduler).

**Configurable optimizer profiles.** There are many variables to be considered by the XCraft optimizer when materializing an AXML document, such as:

- the subplans depth used by the splitting algorithm;
- the scheduling heuristic;
- if re-optimization is allowed;
- if peer can forward delegated subplans to other participants; and
- the heuristic used to generate the search space of alternative physical subplans.

To handle all these options, XCraft uses the notion of *optimizer profile*, that is a set of properties that control the optimizer behavior. The profile is loaded at launch time, but it can be updated during the peer life cycle. It provides flexibility to adapt to different application requirements.

**Basic optimization services.** Cost parameters and statistics are provided by some basic optimization services, which are usually available at any ActiveXML peer. Although the optimizer may collect some missing costs and statistics at runtime, much of the supportive information that is required during the optimization process must be gathered in advance. Furthermore, plan operators such as  $\Theta$  and  $\delta$  are implemented as basic Web services for collaboration, thus enabling peers to exchange evaluation plans encoded as AXML data.

```

<serviceDefinition type="query"
  axml:docName="UnionService"
  xmlns:axml="http://www.activexml.net/AXML">
  <parameters>
    <param name="_documentIn"/>
  </parameters>
  <definition>
    <query> <![CDATA[
      {select e1
        from e1 in SigmodRecord//articlesTuple}
      union
      {select e2
        from e2 in {_documentIn}//articlesTuple};
    ]]> </query>
  </definition>
</serviceDefinition>

```

**Figure 29. Declarative Web service with union operation.**

```

<?xml version="1.0" encoding="UTF-8"?>
<serviceDefinition type="query"
  axml:docName="JoinService"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:axml="http://www.activexml.net/AXML">
  <parameters>
    <param name="_documentIn"/>
  </parameters>
  <definition>
    <query> <![CDATA[
      select e1
      from e1 in SigmodRecord//articlesTuple,
           e2 in {_documentIn}//articlesTuple
      where e1/title/text() = e2/title/text();
    ]]> </query>
  </definition>
</serviceDefinition>

```

**Figure 30. Declarative Web service with join operation.**

## 9 Experimental Results

We have implemented and tested the proposed optimization strategy in the ActiveXML system [13]. We extended the ActiveXML peer (version 4-Beta) with the XCraft optimizer components presented in Section 8. 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 [31].

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 [48]. The specifications of these declarative services are described in Figures 29 and 30. They take a single input parameter named “\_documentIn”, which is combined (either by a union or a join operation) with a locally stored file. Notice the “axml:docName” parameter (of the “serviceDefinition” element) indicates the name of the declarative service. In the ActiveXML platform, the query of declarative Web services is written with the X-OQL language [64].

We used three heterogeneous machines under different workloads, as described in Figure 31. Processing power is represented by BogoMips [58]. *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 32 shows these peers connections.

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 corre-

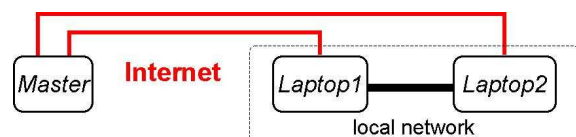
Peer	O.S.	BogoMips	RAM
<i>Master</i>	Debian GNU/Linux	2957.31	512Mbytes
<i>Laptop1</i>	MS Windows™XP	1718.18	512Mbytes
<i>Laptop2</i>	Linux SuSe™	1198,77	512Mbytes

**Figure 31. Hardware of deployed ActiveXML peers.**

sponds 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 [54].

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 improve the adaptivity capabilities of the optimizer.

It is also worth noting that, in P2P and grid systems, the communication costs from transferring data between two nodes that are delegated to the same peer are usually as-



**Figure 32. P2P network used in experiments.**



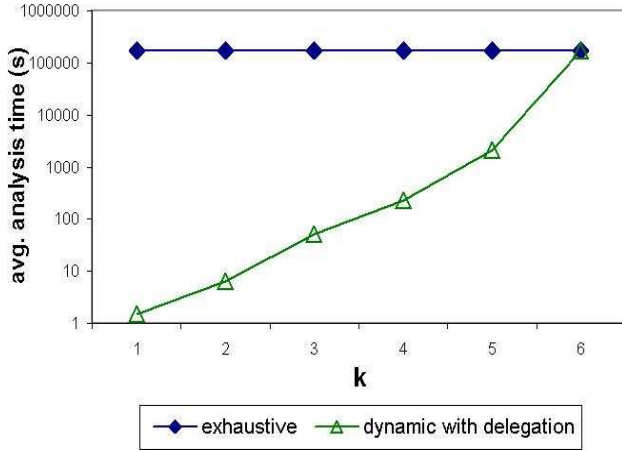


Figure 36. Simulated optimization time.

sults with the exhaustive search strategy, and results of using the Divide&Conquer heuristic to identify independent materialization tasks. Observe 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. We simulated this time by considering the optimizer spends an average of 0.5 millisecond to generate and analyze an alternative plan. We came to this value by observing experimental results obtained with small AXML documents. Although larger documents tend to require more time to be generated and analyzed, this average metric sets a good performance reference, as shown in Figure 36. We considered the same AXML document used in Figure 35.

## 9.2 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 37. 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 that contain batch-pipelined tasks (*i.e.*, with at most one dependency). We vary the height of nested service calls in the document from 2 to 6.

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

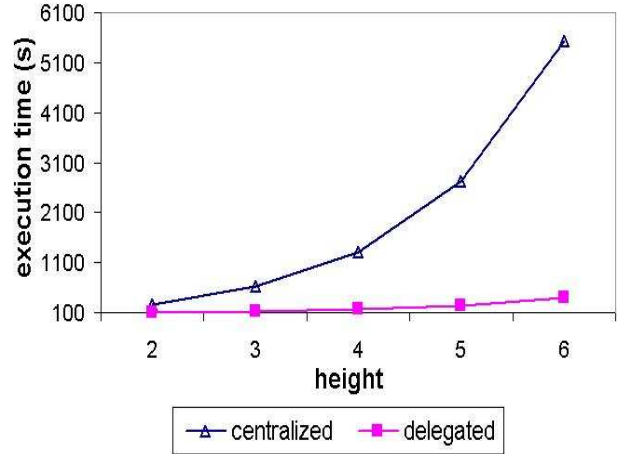


Figure 37. Performance gains obtained by subplans delegation.

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.

## 10 Related Work

Materializing AXML documents is quite 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. These invocation constraints correspond to some basic control flow patterns, namely sequence, parallel split, and synchronization [54]. 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 seldom meaningful in workflow systems.

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 [15, 33, 54]. As in scheduling workflow tasks for grid computing [15, 38, 56], we are interested in determining an efficient assignment of tasks (Web service executions) to distributed resources (peers). However, in grid systems usually tasks are assigned to *sites* whose infrastructure encapsulates many servers, aiming mainly for load balance [15, 27, 42, 56]. Still, planning workflows in distributed heterogeneous systems is an NP-complete problem [33], which remains a research challenge. Likewise, optimizing AXML materialization is a hard problem, with additional complications from the volatility of a P2P scenario.

Allocating resources and scheduling tasks to efficiently execute workflows is indeed an important issue. Current

planners [15, 27, 42, 56, 61] are essentially concerned with heuristics to schedule tasks and algorithms to improve locality of required data files. Nonetheless, tailored for grid computing, these planners are often based on static analysis [15, 27, 42, 61]. In AXML materialization, besides the performance and membership fluctuations of the system, the optimizer has to be prepared for occasional changes in the materialization plan due to intensional answers. On the other hand, planners that are based on dynamic strategies do either greedy [56] or opportunistic [38] resources selection. Since they work with local decisions, they usually cannot explore avoiding unnecessary data transfers. Even when planners are dynamic or adaptive [27, 38, 42, 56], they consider either centralized or hierarchical coordination, and rely on re-optimizations to react to changes.

Notice that, although decentralization has become a key feature in both P2P and grid computing, current systems do not support a decentralized planner. Our results highlight promising performance gains achieved by a decentralized approach.

AXML documents are similar to decision flows [30] in the sense that their materialization is *attribute-centric*, namely it aims at determining the values of certain data elements. Yet, conversely to [30], our strategy is dynamic and enables decentralized evaluation.

Previous work on AXML optimization mostly addressed typing control [39], XML query processing [3], and data and Web services replication [7]. Mechanisms to generate alternative strategies for AXML materialization, including basic cost formula for performance prediction, was first presented in [44]. XCraft is built upon these ideas, and focuses on the problem of efficiently producing and evaluating materialization plans in dynamic P2P systems. Recently, Abiteboul *et. al* [9] proposed an algebraic framework to generate AXML materialization alternatives, with emphasis on Web services that can be described by queries. In XCraft, we consider issues related to handling search complexity, resources heterogeneity and P2P membership dynamics when generating materialization plans.

## 11 Conclusions

Materializing an AXML document corresponds to a general case of finding an efficient assignment of inter-related tasks to heterogeneous machines, which is an extremely hard optimization problem. Nevertheless, this became a current challenge for many information integration systems based on Web services, such as P2P systems and grid computing. In this paper, we presented an optimization strategy for AXML materialization, which widely explores dynamic techniques, thus scaling well for decentralized and ad-hoc systems. We believe this work goes beyond the context of AXML documents, and contributes to the efficient instan-

tiation of abstract workflows, specially in highly-dynamic and heterogeneous systems. Also, with a decentralized architecture for collaborative optimization, we highlighted an important issue that has been neglected in most of the current systems.

In XCraft, since we assumed the cost analysis of materialization subplans is exhaustive, the algorithm used to generate these subplans is quite sensitive to the choice of the height of the planning step (*i.e.*, the  $k$  parameter). To diminish this shortcoming, we are currently developing methods based on stochastic algorithms and local search, such as the techniques proposed in [63]. The overall idea is to incrementally refine an arbitrary initial solution until some condition is satisfied (*e.g.*, some bounded period of time or performance improvement percentage), while allowing the optimizer to randomly move in the search space based on some probability function for plan acceptance. As stated in [45], these metaheuristics are usually very suited for searching good solutions using non-monotonic cost functions (as in AXML materialization).

There are many interesting paths to pursue the ideas raised in this paper. We are considering to extend the optimization strategy to support contingency planning for service call failures, that is to generate branching plans taking some or all of the possible alternative evaluations into account, as presented in [23] for Web services composition. Also, materialization plans are a very graphical and intuitive representation that can be explored to monitor AXML materialization, possibly allowing the user to interfere in the process “on the fly”. Finally, we have observed that planning and scheduling Web service invocations are quite affected by resources availability. Hence, an interesting research perspective consists in investigating non-intrusive techniques for resources provisioning in P2P systems.

**Acknowledgements.** The authors thank CNPq agency for partially funding this work. Gabriela Ruberg is also supported by the Central Bank of Brazil. The contents of this document express the viewpoint of the authors, and do not represent the position of either these institutions. We specially thank Ioana Manolescu and Serge Abiteboul for fruitful discussions (and text revisions) on preliminary versions of this work. Finally, we thank the Gemo team, at INRIA-Futurs, for the ActiveXML open-source prototype.

## References

- [1] S. Abiteboul, B. Alexe, O. Benjelloun, B. Cautis, I. Fundulaki, T. Milo, and A. Sahuguet. An electronic patient record “on steroids”: Distributed, peer-to-peer, secure and privacy-conscious. In *VLDB*, pages 1273–1276, 2004.
- [2] S. Abiteboul, B. Amann, J. Baumgarten, O. Benjel-

- loun, F. Dang-Ngoc, and T. Milo. Schema-driven customization of Web services (demo). In *VLDB*, 2003.
- [3] 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.
- [4] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: A data-centric perspective on Web services. In *Web Dynamics*, pages 275–300, 2004.
- [5] S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. In *ACM PODS*, pages 35–45, 2004.
- [6] S. Abiteboul, O. Benjelloun, and T. Milo. The active xml project: an overview. Gemo Tech. Report 331, 2005.
- [7] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *ACM SIGMOD*, 2003.
- [8] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and querying a peer-to-peer warehouse of XML resources. In *Semantic Web and Databases Workshop*, 2004.
- [9] S. Abiteboul, I. Manolescu, and E. Taropa. A framework for distributed XML data management. In *EDBT*, pages 1049–1058, 2006.
- [10] S. Abiteboul, B. Nguyen, and G. Ruberg. *Building an Active Content Warehouse.*, chapter III, pages 63–95. Idea Group Publishing, 2006.
- [11] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [12] M. N. Alpdemir, A. Mukherjee, A. Gounaris, N. Paton, P. Watson, A. Fernandes, and D. Fitzgerald. OGSA-DQP: A Service for Distributed Querying on the Grid. In *EDBT, LNCS 2992*, pages 858–861, 2004.
- [13] ActiveXML home page. <http://www.activexml.net>.
- [14] J. Blythe, E. Deelman, and Y. Gil. Automatically composed workflows for grid environments. *IEEE Intelligent Systems*, 19(4):16–23, 2004.
- [15] J. Blythe, S. Jain, E. Deelman, A. Mandal, and K. Kennedy. Task Scheduling Strategies for Workflow-based Applications in Grids. In *IEEE Int. Symposium on Cluster Computing and Grid (CC-Grid)*, 2005.
- [16] L. Bouganim, F. Fabret, C. Mohan, and P. Valduriez. Dynamic query scheduling in data integration systems. In *ICDE*, pages 425–434, 2000.
- [17] W. B. Bradley and D. P. Maher. The NEMO P2P service orchestration framework. In *HICSS*, 2004.
- [18] 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.
- [19] M. H. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. V. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. P. Sycara. DAML-S: Web service description for the Semantic Web. In *International Semantic Web Conference*, pages 348–363, 2002.
- [20] 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.
- [21] W. K.-W. Cheung, J. Liu, K. H. Tsang, and R. K. Wong. Towards autonomous service composition in a grid environment. In *ICWS*, pages 550–557, 2004.
- [22] R. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *ACM SIGMOD*, pages 150–160, 1994.
- [23] L. A. G. da Costa, P. F. Pires, and M. Mattoso. Automatic composition of Web services with contingency plans. In *ICWS*, pages 454–461, 2004.
- [24] 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.
- [25] A. Doan and A. Y. Halevy. Efficiently ordering query plans for data integration. In *ICDE*, pages 393–402, 2002.
- [26] F. Ferreira, C. J. P. de Lucena, and D. Schwabe. A Peer-To-Peer platform based on Semantic Web Services. In *WWW (Posters)*, 2003.
- [27] 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.

- [28] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.*, 23(2):7–18, 2000.
- [29] L. Huang, D. W. Walker, Y. Huang, and O. F. Rana. Dynamic Web service selection for workflow optimization. In *AHM*, 2005.
- [30] 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.
- [31] JDSL - the data structures library in Java. <http://www.cs.brown.edu/cgc/jdsl/>.
- [32] T. Jim and D. Suci. Dynamically Distributed Query Evaluation. In *ACM PODS*, pages 413–424, 2001.
- [33] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.
- [34] L. Liu, C. Pu, and D. D. A. Ruiz. A systematic approach to flexible specification, composition, and restructuring of workflow activities. *J. Database Manag.*, 15(1):1–40, 2004.
- [35] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck. Web Service Level Agreement (WSLA) Language Specification. IBM Software Group, Tech. Report, January 2003.
- [36] B. Medjahed and A. Bouguettaya. A dynamic foundational architecture for semantic Web services. *Distributed and Parallel Databases*, 17(2):179–206, 2005.
- [37] N. C. Mendonça and J. A. F. Silva. An empirical evaluation of client-side server selection policies for accessing replicated Web services. In *SAC*, pages 1704–1708, 2005.
- [38] L. A. Meyer. *Estrategias para o Escalonamento Dinamico de Workflows em Grid. (in Portuguese)*. PhD thesis, COPPE/UFRJ, 2006.
- [39] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. D. Ngoc. Exchanging intensional XML data. In *ACM SIGMOD*, pages 289–300, 2003.
- [40] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. T. Schlosser, I. Brunkhorst, and A. Löser. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In *WWW*, pages 536–543, 2003.
- [41] OWL-S: Semantic Markup for Web Services. <http://www.w3.org/Submission/OWL-S/>.
- [42] Pegasus home page. <http://pegasus.isi.edu>.
- [43] R. Rajamony and M. Elnozahy. Measuring client-perceived response times on the WWW. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.
- [44] N. Ruberg, G. Ruberg, and I. Manolescu. Towards cost-based optimization for data-intensive Web service computations. In *SBBD*, pages 283–297, 2004.
- [45] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2003.
- [46] R. Sakellariou and H. Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. *IPDPS*, 2(2):111b, 2004.
- [47] R. Sakellariou and H. Zhao. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Scientific Programming*, 12(4):253–262, 2004.
- [48] ACM SIGMOD Record articles database. Available at <http://acm.org/sigmod/record/xml/>.
- [49] Simple Object Access Protocol (SOAP) 1.1. <http://www.w3.org/TR/SOAP>.
- [50] D. Suci. Distributed query evaluation on semistructured data. *ACM Trans. Database Syst.*, 27(1):1–62, 2002.
- [51] C. Team. DAGMan (Directed Acyclic Graph Manager) meta-scheduler for condor. University of Wisconsin-Madison. <http://www.cs.wisc.edu/condor/dagman/>.
- [52] Universal Description, Discovery, and Integration of Business for the Web (UDDI). <http://www.uddi.org>.
- [53] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD Conference*, pages 130–141, 1998.
- [54] 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.
- [55] P. K. Vargas, I. de Castro Dutra, and C. Geyer. Application partitioning and hierarchical application management in grid environments. Tech. Report ES-657/04, COPPE/UFRJ, 2004.

- [56] 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.
- [57] K. Verma, K. Gomadam, A. P. Sheth, J. A. Miller, and Z. Wu. The METEOR-S approach for configuring and executing dynamic Web processes. LSDIS Tech. Report, University of Georgia, June 2005.
- [58] D. W. The quintessential Linux benchmark: All about the BogoMips number displayed when Linux boots. *Linux Journal*, 21, 1996.
- [59] The World Wide Web Consortium. <http://www.w3.org/>.
- [60] The Web Services Activity Report. <http://www.w3.org/2002/ws>.
- [61] M. Wiczcerek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the ASKALON grid environment. *SIGMOD Record*, 34(3):56–62, 2005.
- [62] Web Services Definition Language (WSDL). <http://www.w3.org/TR/wsdl>.
- [63] M.-Y. Wu, W. Shu, and J. Gu. Efficient local search for DAG scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):617–627, 2001.
- [64] X-OQL homepage. Available at <http://activexml.net/xoql/>.
- [65] XML Path Language (XPath) Version 1.0. <http://www.w3.org/TR/xpath>.
- [66] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for Web services composition. *IEEE Trans. Software Eng.*, 30(5):311–327, 2004.