A Framework for Distributed XML Data Management

Serge Abiteboul¹, Ioana Manolescu¹, and Emanuel Taropa¹,²

¹ INRIA Futurs & LRI, France, firstname.lastname@inria.fr
² CS Department, Yousei University, Korea

Abstract. As data management applications grow more complex, they may need efficient distributed query processing, but also subscription management, data archival etc. To enact such applications, the current solution consists of stacking several systems together. The juxtaposition of different computing models prevents reasoning on the application as a whole, and wastes important opportunities to improve performance. We present a simple extension to the AXML [7] language, allowing it to declaratively specify and deploy complex applications based solely on XML and XML queries. Our main contribution is a full algebraic model for complex distributed AXML computations. While very expressive, the model is conceptually uniform, and enables numerous powerful optimizations across a distributed complex process.³

1 Introduction

Distributed data management has been an important domain of research almost since the early days of databases [15]. With the development of the Web and the existence of universal standards for data exchange, this problem arguably became the most essential challenge to the database community. The problem as considered in distributed relational systems was already very complex. With the heterogeneity and autonomy of sources, it is now even more difficult.

The language Active XML based on the embedding of service calls inside XML documents has been proposed for distributed data management. Several works have shown that the exchange of such documents provides a powerful support for distributed optimization [1–3]. However, these works proposed isolated solutions for isolated tasks, and had to rely on features not present in the language. In this paper, we isolate the missing components and propose an extension that could serve as a unified powerful language for describing, in a very flexible manner the deployment and evaluation of queries in a collaborative manner. The aforementioned techniques, as well as standard distributed query optimization techniques, can all be described based on rewrite rules in the language.

To pursue the analogy with (centralized) relational database, Active XML as originally proposed, is a logical language for describing distributed computation,

³This work was partially supported by the French Government ACI MDP2P and the eDos EU project.
to which we associated a fixed simple evaluation strategy. Its extension proposed here is an algebraic counterpart that provides for more efficient evaluation.

One missing aspect from Active XML (as originally described) is the capability to control explicitly the shipping of data and queries, although we did use this feature [1]. We explicitly add it here, to allow delegating computations to other peers. We also explicitly introduce generic data and service, which are available on several sites; a particular flavor of this feature was used in [3].

This paper is organized as follows. Section 2 introduces AXML, and shows how the application could be deployed based on AXML. Section 3 holds our main contribution: an algebra for distributed computations, with associated equivalence rules and an optimization methodology. Section 4 concludes.

An application of our methodology to a real-life software distribution application is described in the full version of this paper [4].

2 Preliminaries: AXML documents and queries

In this section, we briefly introduce the features of the pre-existing ActiveXML model (AXML, in short) [5, 7]. We use the following notations:

- a set $D$ of document names. Values from $D$ are denoted: $d$, $d_1$, $d_2$ etc.
- a set $S$ of service names. Values from $S$ are denoted: $s$, $s_1$, $s_2$ etc.
- a set $P$ of peer identifiers. Values from $P$ are denoted: $p$, $p_1$, $p_2$ etc.
- a set $N$ of node identifiers. Values from $N$ are denoted: $n_1$, $n_2$ etc.

We assume given a finite set of peers, each of which is characterized by a distinct peer identifier $p \in P$. Intuitively, a peer represents a context of computation; it can also be seen as a hosting environment for documents and services, which we describe next. We make no assumption about the structure of the peer network, e.g., whether a DHT-style index is present or not. We will discuss the impact of various network structures further on.

2.1 XML documents, types, and services

We view an XML tree as an unranked, unordered tree, where each leaf node has a label from $L$, and each internal nodes has a label from $L$ and an identifier from $N$. Furthermore, each tree resides on exactly one peer identified by $p \in P$. We will refer to the tree as $t @ p$. An XML document is a tuple $(t, d)$ where $t$ is an XML tree, $d \in D$ is a document name. No two documents can agree on the values of $(d, p)$. We will refer to a document as $d @ p$.

We denote by $\Theta$ the set of all XML tree types, as expressed for instance in XML Schema [18], and we refer to individual types as $\tau, \tau_1, \tau_2$ etc.

We model a Web service as a tuple $(p, s)$, where $p \in P$ is the identifier of the peer providing the service, and $s \in S$ is the service name. The service is associated an unique type signature $(\tau_{in}, \tau_{out})$, where $\tau_{in} \in \Theta^n$ for some integer $n$, and $\tau_{out} \in \Theta$. We use $s @ p$ to refer to such a service; it corresponds to a (simplified) WSDL request-response operation [17].

When a Web service $s @ p$ receives as input an XML forest of type $\tau_{in}$, it reacts by sending, successively, one or more XML trees of type $\tau_{out}$. If the service may send more than one such tree, we term it a continuous service.
2.2 AXML documents

An AXML document is an XML document containing some nodes labeled with a specific label sc, standing for service calls. An sc node has several children. Two children, labeled peer and service, contain, respectively, a peer p1 ∈ P and a service s1 ∈ S, where s1 @ p1 identifies an existing Web service. The others are labeled param1, ..., paramn, where n is the input arity of s @ p.

Assume an AXML document d0 @ p0 contains a service call to a service s1 @ p1 as above. When the call is activated, the following sequence of steps takes place:

1. p1 sends a copy of the param-label children of the sc node, to peer p1, asking it to evaluate s1 on these parameters.
2. p1 eventually evaluates s1 on this input, and sends back to p0 an XML subtree containing the response.
3. When p0 receives this subtree, it inserts it in d0, as a sibling of the sc node.

XML supports several mechanisms for deciding when to activate a service call. This control may be given to the user via some interactive hypertext. Alternatively, a call may be activated only when the call result is needed to evaluate some query over the enclosing document [2], or in order to turn d0’s XML type in some other desired type [6]. It is also possible to specify that a call must be activated just after a response to another activated call has been received.

AXML also supports calls to continuous services. When such a call is activated, step 1 above takes place just once, while steps 2 and 3, together, occur repeatedly starting from that moment. In this paper, we consider that the response trees successively sent by p1 accumulate as siblings of the sc node [5]. If a service call sc2 must be activated just after sc2 and sc2 is a call to a continuous service, then sc1 will be activated after handling every answer to sc2. We consider all services are continuous.

Sc nodes may reference any WSDL-compliant Web service. Of particular interest for us are declarative Web services, whose implementation is a declarative XML query or update statement, possibly parameterized. The statements implementing such services are visible to other peers, enabling many optimizations.

Our goal is thus: given a set of AXML documents and declarative services, and a query Q, find alternative evaluation strategies (possibly involving new documents and services dynamically created) which compute the same answers for Q, and are potentially more efficient. We first make some extensions to AXML.

2.3 AXML extensions

We introduce generic documents and services, and define a notion of tree, document, and service equivalence. Then, we make some extensions to the syntax of sc elements, central in AXML, to allow for more communication patterns.

A generic document ed@any denotes any among a set of regular documents which we consider to be equivalent; we say ed is a document equivalence class. We consider a specific notion of document equivalence denoted ≡, suited for AXML. Two documents are equivalent iff their trees are equivalent. Two trees t1 and t2 are equivalent iff their potential evolution, via service call activations, will eventually reach the same fixpoint. This notion has been formally defined
in [5] for the purpose of studying confluence and termination for AXML; we use it here as a basis for optimization. We introduce generic services similarly [4].

We allow queries to refer to generic documents as well as regular ones, and sc nodes to refer to generic services as well as regular ones. The semantics of such queries and calls will be defined shortly.

We add to an sc element some optional form children, each of which contains a location to which the service results(s) should be forwarded. Each form element encapsulates a node identifier of the form \( n @ p \) where \( p \in \mathcal{P} \) and \( n \in \mathcal{N} \). The semantics is that the response should be added as a child of node \( n \), which resides on peer \( p \). If no form child is specified, a default one is used containing the ID of the sc's parent, just like in the existing AXML model.

We will refer to a document as \( d @ p \) or alternatively as \( d @ any \), and similarly for services. We will denote a service call in our extended AXML model as:

\[
\text{sc}(p_{\text{peer}}|\text{any}), \text{serv}, [\text{param}_1, \ldots, \text{param}_m], [\text{form}_1, \ldots, \text{form}_n])
\]

where \( p_{\text{peer}} \in \mathcal{P} \) is a peer providing the service \( \text{serv} \).

3 An algebra for extended AXML computations

3.1 AXML expressions

To model the various operations needed by our distributed data management applications, we introduce here a simple language of AXML expressions, denoted \( \mathcal{E} \). In the following, \( p, p_1, p_2 \) are some peers from \( \mathcal{P} \).

Any tree \( t @ p \) or document \( d @ p \) is in \( \mathcal{E} \). Also, let \( q @ p \) be a query of arity \( n \) defined at \( p \), and let \( t_1 @ p, t_2, \ldots, t_n @ p \) be a set of trees at \( p \). Then, \( q @ p(t_1 @ p, t_2, \ldots, t_n @ p) \in \mathcal{E} \).

Let \( t @ p_1 \) be a tree. Then, \( \text{send}(p_2, t @ p_1) \in \mathcal{E} \), where \( \text{send}() \) is an expression constructor. This expression denotes the sending of a piece of data, namely \( t \), from \( p_1 \) to \( p_2 \). Similarly, if \( d @ p_1 \) is a document, \( \text{send}(p_2, d @ p_1) \in \mathcal{E} \). The exact place where \( t \) (or \( d \)) arrives at peer \( p_2 \) is determined when evaluating the expression, as the next sections explains.

\( \mathcal{E} \) also allows to specify the exact location(s) where a tree should arrive. The expression \( \text{send}(n_2 @ p_2, t @ p_1) \) says that \( t \) should be added as a child of the node \( n_2 @ p_2 \). The expression \( \text{send}([n_2 @ p_2, n_3 @ p_3, \ldots, n_k @ p_k], t @ p_1) \) corresponds to the operation of sending the same tree to several destinations. Finally, \( \text{send}(d @ p_2, t @ p_1) \) states that \( t \) is installed under the name \( d \) as a new document at \( p_2 \) (where \( d \) was not previously in use on \( p_2 \)).

\( \mathcal{E} \) also allows sending queries (in the style of code shipping). Let \( q @ p_1 \) be a query. Then, \( \text{send}(p_2, q @ p_1) \in \mathcal{E} \), where \( \text{send}() \) is the same (slightly overloaded) expression constructor. This denotes the sending of the query \( q \) on peer \( p_2 \).

An expression can be viewed (serialized) as an XML tree, whose root is labeled with the expression constructor, and whose children are the expression parameters. An expression located at some peer, denoted \( e @ p \), is an XML tree.

3.2 Evaluating AXML expression trees

The expression language \( \mathcal{E} \) describes some computations to be performed. In this section, we define the evaluation of an expression tree \( e @ p \) where \( e \in \mathcal{E} \).
Intuitively, \( \text{eval}@p(e) \) may do one or more of the following: (i) return another XML tree (or, more generally, a stream of XML trees, where a stream is a flow of XML trees which accumulate, as children of a given node on some peer); (ii) return a new service; (iii) as a side effect, create one or more XML streams, accumulating under some well-specified nodes on one or more peers.

This is best illustrated by the following \( \text{eval} \) definitions, where \( p, p_i \) designates a peer, \( t_j @ p \) is a tree at peer \( p \), and \( n_i @ p \) a node at \( p \).

We first define \( \text{eval} \) for tree expressions. Let \( t @ p_0 \) be a tree, whose root is labeled \( l \in \mathcal{L} \), \( l \neq \mathcal{S} \mathcal{C} \), and let \( t_1, \ldots, t_n \) be the children of the root in \( t \). We define:

\[
\text{eval}@p_0(t @ p_0) = l(\text{eval}@p_0(t_1), \text{eval}@p_0(t_2), \ldots, \text{eval}@p_0(t_n))
\]  

(1)

The evaluation copies \( t \)'s root and pushes the evaluation to the children. Evaluating one XML tree (the expression tree on the left) yields the (partially evaluated) XML tree at right, into which the expressions to evaluate are smaller.

As a consequence of (1), for any tree \( t @ p_0 \) containing no \( \mathcal{S} \mathcal{C} \) node, we have \( \text{eval}@p_0(t @ p_0) = t @ p_0 \); evaluating the expression simply returns the data tree.

Now consider what happens if we replace the (static) tree \( t @ p_0 \) with a stream of successive XML trees, accumulating as children of some node \( n @ p_0 \). Clearly, in this case, definition (1) applies for every tree in the stream, thus \( \text{eval} \) over the stream of trees returns another stream of (partially evaluated) trees.

Definition (1) covers a particular class of \( \text{eval}@p(t) \) expressions; we will define \( \text{eval} \) for the other cases gradually. For the time being, we turn to defining the evaluation of (a particular class of) query expression trees:

\[
\text{eval}@p(q(t_1 @ p, \ldots, t_n @ p)) = q(\text{eval}@p(t_1 @ p), \ldots, \text{eval}@p(t_n @ p))
\]  

(2)

Evaluating a local query expression tree amounts to evaluating the query parameters, and then evaluating the query (in the usual sense) on these trees.

Recall that all queries are continuous. If we take \( t_i @ p \) to be streams of trees arriving at \( p_i \), definition (2) captures the intuitive semantics of continuous incremental query evaluation: \( \text{eval}@p(q) \) produces a result whenever the arrival of some new tree in the input streams \( t_1, t_2, \ldots, t_n \) leads to creating some output. This generalization reasoning (from trees to streams of trees) applies to all remaining \( \text{eval} \) definitions, and we consider it implicit in the sequel.

We next define the evaluation of a simple class of \( \text{send} \) expressions.

\[
\text{eval}@p_0(\text{send}(p_1, t @ p_0)) = \emptyset
\]  

(3)

Evaluating a \( \text{send} \) expression tree at \( p_0 \), hosting \( t \), returns at \( p_0 \) an empty result. Intuitively, the message encapsulating the copy of \( t \) has left \( p_0 \), and moved to \( p_i \). However, as a side effect, a copy of \( t @ p_0 \) is made, and sent to peer \( p_i \). From now on, all evaluations of \( \text{send} \) expression trees are implicitly understood to copy the data model instances they send, prior to sending them.

**Notation** From now on, we will use the shorthand \( \text{send}_{p_1 \rightarrow p_2}(e) \) to denote \( \text{eval}@p_1(\text{send}(p_2, e)) \), where \( p_1, p_2 \in \mathcal{P} \) and \( e \in \mathcal{E} \), and we use \( \text{send}_{p_1 \rightarrow fw\text{List}}(e) \) to denote \( \text{eval}@p_1(\text{send}(\text{fw\text{List}}, e)) \), where \( \text{fw\text{List}} \) is a list of nodes.
If \( p_2 \neq p_0 \), \( \text{send}_{p_2\rightarrow p_1}(t@p_0) \) is undefined. The intuition is that \( p_2 \) cannot send something it doesn’t have. More generally, for any tree \( x@p_0 \), \( \text{send}_{p_2\rightarrow p_1}(x@p_0) \) is undefined if \( p_2 \neq p_0 \). Similarly, we define:

\[
\text{send}_{p_0\rightarrow p_1}@p_2,n_2@p_2,\ldots,n_n@p_2}(t@p_0) = \emptyset
\] (4)

Sending \( t@p_0 \) to the locations \( n_i@p_2 \) returns an empty result at \( p_0 \), and as a side effect, at each \( p_i \) the result of \( \text{eval}@p_i(t@p_2) \) is added as a child of \( n_i@p_2 \). We use \( t@p_1 \) to denote the copy of \( t@p_0 \) that has landed on \( p_1 \).

We now define \( \text{eval} \) at some peer, of a data expression of a remote tree.

\[
\text{eval}@p_i(t@p_2) = \text{send}_{p_2\rightarrow p_1}(\text{eval}@p_2(t@p_2))
\] (5)

We assume \( p_1 \neq p_2 \), thus \( p_1 \) initially doesn’t have \( t \). In order for \( p_1 \) to get the evaluation result, \( p_2 \) is asked to evaluate it \(^4\), and then send it at \( p_1 \). Overall, \( p_2 \) has received the expression tree \( t@p_2 \) as some local tree, has replaced this local tree with the result of \( \text{eval}@p_2(t) \), and has sent this result to \( p_1 \). After this send evaluation, the local send expression tree on \( p_2 \) becomes \( \emptyset \), by (3). Setting a tree to \( \emptyset \) amounts to deleting it, thus, \( p_2 \)’s set of documents and services is unchanged after the evaluation. The overall effect on \( p_1 \) is that the expression tree \( \text{eval}@p_1(t@p_2) \) has been replaced with the desired evaluation result.

We now have the ingredients for defining the evaluation of a tree \( t@p_0 \), whose root is labeled \( \text{sc} \). We denote by \( \text{parList} = [t_1, t_2, \ldots, t_n] \) the list of \( \text{param} \)-labeled children of the \( \text{sc} \), and by \( \text{fwList} \) the list of their \( \text{form} \)-labeled siblings.

\[
\text{eval}@p_0(\text{sc}(p_1, s_1, \text{parList}, \text{fwList})) = \\
\text{send}_{p_0\rightarrow p_1}(f_1(\text{send}_{p_0\rightarrow p_1}(\text{eval}@p_0(\text{parList}))))
\] (6)

where \( \text{eval}@p_0(\text{parList}) \) stands for \( [\text{eval}@p_0(t_1), \text{eval}@p_0(t_2), \ldots, \text{eval}@p_0(t_n)] \).

The second part of the definition (6) is best read from the innermost parenthesis to the outer. To evaluate \( \text{sc} \), \( p_0 \) first evaluates the parameters (innermost \( \text{eval} \)), then sends the result to \( p_1 \), as denoted by \( \text{eval}@p_0(\text{send}(p_1, \ldots)) \). Peer \( p_1 \) evaluates, in the usual sense, the query \( q_1 \) (the one which implements its service \( s_1 \)), and sends the result to the locations in the forward lists.

We do not need to define the evaluation of a tree \( t@p_0 \), whose root is labeled \( \text{sc} \), at some peer \( p_1 \neq p_0 \); this case is already covered by definition (5).

The evaluation at some peer \( p_1 \), of a query defined at another peer \( p_2 \), is:

\[
\text{eval}@p_1(q(t_1@p_2, t_2@p_2, \ldots, t_n@p_2)) = \\
\text{eval}@p_1(\text{send}_{p_2\rightarrow p_1}(q)) \cdot \text{send}_{p_2\rightarrow p_1}(\text{send}_{p_2\rightarrow p_1}(t_1@p_2, t_2@p_2, \ldots, t_n@p_2))
\] (7)

This states that \( p_2 \) should send both \( q \) and its arguments to \( p_1 \), as shown by the two \( \text{send}_{p_2\rightarrow p_1} \), and \( p_1 \) can then evaluate locally as per definition (2).

What happens when evaluating a send expression of some query?

\[
\text{eval}@p_1(\text{send}(p_2, q@p_1)) = \text{send}_{p_1\rightarrow p_2}(q@p_1) = \emptyset
\] (8)

\(^4\) This is performed at \( p_2 \) by applying successively definitions (1), (5) and (6), see next.
Evaluating the send expression tree erases it from \( p_t \) and, as a side effect, deploys query \( q \) on peer \( p_2 \) as a new service. Rather than giving it an explicit name, by a slight abuse of notation, we may refer to this service as \( \text{send}_{p_1 \rightarrow p_2} (q@p_1) \).

So far, we have defined \( \text{eval} \) on expressions involving precise documents and queries. We now turn to the case of generic documents and queries. We have:

\[
\text{eval}(p(\text{expr}(d@\text{any}))) = \text{eval}(p(\text{expr}(\text{eval}(p(\text{pickDoc}(d@\text{any}))))))
\]

(9)

where \( \text{expr} \) is some \( \mathcal{E} \) expression, and the functions \( \text{pickDoc} \), present on all peers, return the name of some document from the equivalence class \( d@\text{any} \). A similar rule applies for generic services \([4]\). Definition 9 states that \( p \) should find the name of a regular document corresponding to the equivalence class \( d@\text{any} \), then proceed to evaluate \( \text{expr} \) where references to \( d@\text{any} \) have been replaced with that name. The implementation of an actual \( \text{pick} \) function at \( p \) depends on \( p \)'s knowledge of the existing documents and services, \( p \)'s preferences etc. \([4]\).

We have so far specified a procedure for expression evaluation: for any \( e \in \mathcal{E} \), to evaluate \( e@p \), identify the definition among (1)-(9) which fits \( e \)'s topmost node and \( p \) apply this definition, and so on recursively down \( e \)'s structure until a plain data tree is obtained at \( p \). This strategy extends the basic AXML one, to deal with the AXML extensions we introduced in Section 2.3. As we have argued, however, this strategy will not necessarily lead to best performance.

### 3.3 Equivalence rules

In this section, we explore equivalent, potentially more efficient strategies for evaluating an expression tree \( \text{eval}(e@p) \), where \( p \in \mathcal{P} \) and \( e \in \mathcal{E} \).

We call state of an AXML system over peers \( p_1, p_2, \ldots, p_n \), and denote by \( \Sigma \), all documents and services on \( p_1, p_2, \ldots, p_n \). Evaluating an expression \( e@p \) over an AXML system in state \( \Sigma \) brings it to a possibly different state, which we denote \( \text{eval}@p(e)(\Sigma) \). We say two expression evaluations \( e_1@p_1 \) and \( e_2@p_2 \) are equivalent, denoted \( e_1@p_1 \equiv e_2@p_2 \), if for any AXML system state \( \Sigma \),

\[
\text{eval}@p_1(e_1)(\Sigma) = \text{eval}@p_2(e_2)(\Sigma).
\]

Our first equivalence rule refers to query delegation:

\[
\text{eval}@p_1(q(t@p_1)) \equiv \text{send}_{p_2 \rightarrow p_1} ((\text{send}_{p_1 \rightarrow p_2} (q))(\text{send}_{p_2 \rightarrow p_1} (t)))
\]

(10)

This rule says that evaluating a query \( q(t) \) at \( p_t \) gives the same result as: sending \( q \) and \( t \) to another peer \( p_2 \), evaluating \( q(t) \) at \( p_2 \), and sending back the results to \( p_t \). The rule derives from the definitions (2), (4) and (8).

A second very useful rule refers to query composition/decomposition. Let \( q, q_1, q_2, \ldots, q_n \) be some queries, such that \( q \) is equivalent to the composed query \( g_1(q_2,q_3,\ldots,q_n) \) (in the sense defined in Section 2.3). We have:

\[
\text{eval}@p(q@p) \equiv \text{eval}@p(q,\text{eval}@p(q_1@p),\ldots,\text{eval}@p(q_n@p))
\]

(11)

Intuitively, the rule states that \( \text{eval} \) distributes over query composition. It is a direct consequence of the query equivalence hypothesis, and of the definition (2).

The query decomposition and query delegation rules, together, capture many existing distributed query optimization techniques, as Example 1 illustrates.
Example 1 (Pushing selections). Let $q_1$ be a query equivalent to $q_1(\sigma(q_2))$, where
$\sigma$ is some logical selection, and $q_1$ and $q_2$ are chosen so that $\sigma$ has been pushed
down as far as possible. Denoting by $q_3$ the query $\sigma(q_2)$, we have $q \equiv q_1(q_3)$. Let $t \oplus p_2$ be a tree, and $p$ be some peer other than $p_2$. We have:
\[
eval@p(q(t \oplus p_2)) = eval@p(q_1(q_3(d @ p_2))) \equiv (1)
\]
\[
eval@p(q_1(eval@p(q_2(t \oplus p_2)))) \equiv (0)
\]
\[
eval@p(q_1(send_{p_2}(eval @ p_2(q_3(t \oplus p_2))))) = (2) eval@p(q_1(send_{p_2}(q_3(t \oplus p_2))))
\]

The definition or rule used at each step above is shown by a subscript. The first $eval$ designates the evaluation of $q$ on the remote tree $t$. Definition (7) suggests sending the whole tree $t$ to $p$ and evaluating there. However, the last $eval$ above delegates the execution of $q_3$ (which applies the selection) to $p_2$, and only ships to $p$ the resulting data set, typically smaller. □

Other classical distributed optimizations may be similarly derived.

The following rules allow for powerful optimizations of data transfers, and can be derived easily from the definitions of $send$ evaluation:

\[
send_{p_1 \rightarrow p_2}(eval@p_1(send(p_1, t @ p_0))) \equiv send_{p_0 \rightarrow p_2}(t @ p_0) \tag{12}
\]
\[
\begin{align}
\text{eval@p(e}_1(e_2(send_{p_1 \rightarrow p}(t @ p_1)), e_3(send_{p_1 \rightarrow p}(t @ p_1)))) & \equiv \\
\text{eval@p(e}_1(e_2(send_{p_1 \rightarrow p}(t @ p_1, d @ p)), e_3(d @ p))) & \tag{13}
\end{align}
\]

Rule (12), read from right to left, shows that data in transit from $p_0$ to $p_2$ may make an intermediary stop to another peer $p_1$. Read from left to right, it shows that such an intermediary halt may be avoided. While it may seem that rule (12) should always be applied left to right, this is not always true [4]!

In rule (13), subexpressions $e_2$ and $e_3$ need to transfer $t @ p_1$ to $p$. If $t$ is transferred for the needs of $e_2$ and stored in a document $d @ p_1$, $e_3$ no longer needs to transfer $t$, and can use $d @ p$ directly. The rule holds assuming that the evaluation of $e_3$ is only enabled when $d$ is available at $p$, which breaks the parallelism between $e_2$ and $e_3$’s evaluations. This may be worth it if $t$ is large.

Another powerful rule concerns delegation of expression evaluation:

\[
eval@p(e) \equiv eval@p_1(send(p, eval@p(e))) \tag{14}
\]

Some specific rules apply to trees rooted in $sc$ nodes:

\[
\begin{align}
\text{eval@p}(sc(p_1, s_1, parList, f wList)) & \equiv \\
\text{eval@p_2}(send_{p_2}(sc(p_1, s_1, parList, f wList))) & \tag{15}
\end{align}
\]
\[
\begin{align}
\text{eval@p}(q @ p)(sc(p_1, s_1, parList @ p, f wList)) & \equiv \\
\text{send}_{p_1 \rightarrow f wList}(eval@p_1((send_{p_1 \rightarrow p}(q @ p)) (q_1(send_{p_1 \rightarrow p}(parList @ p)))))) & \tag{16}
\end{align}
\]

Rule (15) shows that the peer where an $sc$-rooted tree is evaluated does not impact the evaluation result. Notice there is no need to ship results back to $p_1$, since results are sent directly to the locations in the forward list $f wList$.

Rule (16) provides an interesting method to evaluate a query $q$ over a $sc$-rooted tree. Here, $sc$ refers to service $s_1 @ p_1$, implemented by the query $q_1$. The idea is to ship $q$ and the service call parameters to $p_1$, and ask it to evaluate $q$ directly over $q_1(parList)$. We call this rule pushing queries over service calls.
4 Concluding remarks

The work presented here follows the footsteps of previous works on distributed query processing [12,15], and is particularly related to query optimization in mediator systems [10,16] and in peer-to-peer environments [3,8,9,11]. Our work brings the benefits of declarativeness and algebra-based optimization to AXML, a language integrating queries and data in a single powerful formalism. Our algebra can be seen as a formal model for mutant query plans [13], extended to continuous XML streams. AXML algebraic optimization has first been explored in [14].

Our ongoing work focuses on refining our algebraic formalism, extending it to AXML type-driven rewriting [6], designing and implementing in the AXML system efficient and effective distributed optimization algorithms.

References