Active XML Replication and Recovery

Debmalya Biswas
IRISA/INRIA
Campus Universitaire de Beaulieu
Rennes, France 35042
dbiswas@irisa.fr

Abstract

Active XML (AXML) systems provide an elegant platform to integrate the power of XML, Web services and Peer to Peer (P2P) paradigms by allowing (active) Web service calls to be embedded within XML documents. In this work, we focus on the replication aspect of AXML systems, and study the effect of peer disconnection (an inherent trait of P2P systems) on replication. To be more precise, given peer disconnection, for both eager and lazy replication strategies, we discuss the following: (a) the replication guarantees that can be provided, and (b) recovery procedures for peer disconnection/reconnection.

1 Introduction

Active XML (AXML) [1, 4] systems provide an elegant way to combine the power of XML, Web services and Peer to Peer (P2P) paradigms by allowing (active) Web service calls to be embedded within XML documents. An AXML system consists of the following main components:

- AXML documents: XML documents with embedded AXML service calls (defined below). For example, the AXML snippet in Fig. 1 is an AXML document with the embedded service call "getGrandSlamsWon".
- AXML services: Web services defined as queries over AXML documents. Note that an AXML service is also exposed as a regular Web service (with a WSDL [17] description file).
- AXML peers: Peers where the AXML documents and services are hosted. AXML peers also provide a user interface to query/update the AXML documents stored locally.

Replication is an integral part of most large scale systems including AXML (an AXML document replicated on more than one AXML peer). Replication leads to high throughput, low response times, high availability, etc. Till now, replication oriented research has mostly focused on:

- the storage strategy which specifies "what" and "where" to replicate, and
- replication strategies to efficiently propagate updates,

such that, the following (or a subset of) objectives are achieved:

- a particular property holds, e.g., serializability [9, 11],

```
<?xml version = "1.0" encoding = "UTF-8"?>
<ATPList date = "18042005">
   <player rank = 1>
      <name>
         <firstname>Roger</firstname>
         <lastname>Federer</lastname>
      </name>
      <citizenship>Swiss</citizenship>
      <points>475</points>
      <axml:sc mode = "merge"
serviceNameSpace = "getGrandSlamsWon"
serviceURL = "..." methodName = "getGrandSlamsWon">
         <axml:params>
            <axml:param name = "name">
            <axml:value>Roger Federer</axml:value>
            <axml:param name = "year">
            <axml:value>$year</axml:value>
         </axml:params>
         <grandslamswon year = "2003">
A, W</grandslamswon>
      </axml:sc>
   </player>
...
</ATPList>
```

Figure 1. Sample AXML document with embedded service call "getGrandSlamsWon".

- the number of exchanged messages is optimized [3],
- the system is resilient against failures, e.g., network partitioning [14], and
- high availability [13], throughput [10, 4], etc.

However, it is impractical to assume that the storage/replication strategy of a system (especially, a heterogeneous one) can always be dictated. Thus, in this work, we consider the following orthogonal problem: *Given a replicated system with a fixed storage/replication strategy and possibility of failure, study the replication guarantees that can be provided.* More precisely, we consider the replication guarantees that can be provided for both eager and lazy replication strategies, given the possibility of peer disconnection.

The replication strategies in literature can be broadly classified into two categories: (a) Eager [12]: With eager replication, any update on data d, performed as part of transaction τ at peer p, is applied at all the replicated peers hosting d before τ's commit. (b) Lazy [9, 11]: In this scheme, any update propagation is not performed as part of the update transaction. Rather, updates are propagated "as and when" convenient after the corresponding update transaction has committed. This leads to lower response times and higher throughput, however the data at replicated peers may not always be consistent.

The recovery aspect of replication is particularly relevant for P2P systems given their inherent problem of peer disconnection. Basically, we would like to provide precise answers to questions of the type:

- Who maintains the (continuously evolving) list of replicated peers, and how?
- How does the disconnection of a replicated peer affect replication? Note that a replicated peer may disconnect (and possibly never reconnect) during different stages of the replication process, e.g., before an update propagation has been initiated, during an update propagation (leading to only a subset of the replicated peers receiving the update), etc.
- What additional steps does a peer need to perform on reconnection from a replication perspective?

To summarize, for replicated AXML systems, given the possibility of arbitrary and frequent peer disconnection, for both eager and lazy replication strategies, we

- specify recovery procedures for peer disconnection/reconnection, and
- discuss the replication guarantees that can be provided.

The rest of the paper is organized as follows: In section 2, we introduce the basic algebra for (non-replicated) AXML systems. Section 3 provides a variant of the algebra with "local" semantics. Section 4 considers recovery for AXML replication in detail, with sub-sections 4.1 and 4.2 focusing on eager and lazy replication strategies, respectively. Section 5 concludes the paper and provides some directions for future work.

2 AXML Semantics

In this section, we briefly introduce the semantics of AXML expression evaluation (slightly modified, but follows mainly from [6]). We view an XML tree as an unranked, unordered tree, where each leaf node has a label from $\mathcal{L}$, and each internal node has a label from $\mathcal{L}$ and an identifier from $\mathcal{N}$. Each tree resides on a peer $p \in \mathcal{P}$, and is referred to as $t@p$. An XML document is a tuple (t, d) where t is an XML tree and $d \in \mathcal{D}$ is a document name. We model a Web service as a tuple (p, s), where $p \in \mathcal{P}$ is the peer providing the service, and $s \in \mathcal{S}$ is the service name. We use $d@p$ and $s@p$ to refer to a document d and service s hosted on peer p, respectively.

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 p_1 and a service s_1, where $s_1@p_1$ identifies an existing Web service. The others are labeled $param_1, \cdots, param_n$, where n is the input arity of $s_1@p_1$.

Given this, let us assume that an AXML document $d_0@p_0$ contains a service call to a service $s_1@p_1$ as above. When the call is activated, the following sequence of steps takes place:

1. p_0 sends a copy of the $param_i$-labeled children of the sc node, to peer p_1, asking it to evaluate s_1 on those parameters.
2. p_1 eventually evaluates s_1 on this input, and sends back to p_0 an XML subtree containing the response.
3. When p_0 receives this subtree, it inserts it in d_0, as a child of the sc node.

Next, we introduce a simple algebra for AXML expressions, denoted ξ. Any tree $t@p$, document $d@p$ or service $s@p$ is in ξ. Also, let $q@p$ be a query of arity n defined at p, and let $t_1@p, t_2@p, \cdots, t_n@p$ be a set of trees at p. Then, $q@p(t_1@p, t_2@p, \cdots, t_n@p) \in \xi$. Let $t@p_1$ be a tree. Then, $send(p_2, t@p_1) \in \xi$, where $send(.)$ 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, $send(p_2, d@p_1) \in \xi$. ξ also allows to

specify the exact location(s) where a tree should arrive. The expression $send(n_2@p_2, t@p_1)$ says that t should be added as a child of the node $n_2@p_2$. $t(n@p) = t@p$ denotes the tree $t@p$ containing the node $n@p$.

We first define eval for tree expressions. Let $t@p_0$ be a tree, whose root is labeled $l \neq sc$, and let $t_1, \cdots, t_n$ be children of the root in t. Then,

Rule 1. $eval@p_0(t@p_0) = l(eval@p_0(t_1), eval@p_0(t_2), \cdots, eval@p_0(t_n))$

The evaluation copies t's root and pushes the evaluation to the children. On the same lines, the evaluation of query expression trees can be defined as follows:

Rule 2. $eval@p(q(t_1@p, \cdots, t_n@p)) = q(eval@p(t_1@p), \cdots, eval@p(t_n@p))$

Evaluating a local query expression tree amounts to evaluating the query parameters, and then evaluating the query (in the usual sense) on these trees. Next, we define the evaluation of send expressions as follows:

Rule 3. $eval@p_0(send(p_1, t@p_0)) = \Phi$
Rule 4. $eval@p_0(send(n_1@p_1, t@p_0)) = \Phi$

Evaluating a send expression tree at p_0, hosting t, returns at p_0 an empty result. However, as a side effect, a copy of $t@p_0$ is made, and sent to peer p_1. Sending $t@p_0$ to the location $n_1@p_1$ returns an empty result at p_0, and as a side effect, the result of $eval@p_0(t@p_0)$ is added as a child of $n_1@p_1$. From now on, we use the short-hand $send_{p_0 \to p_1}(e)$ to denote $eval@p_0(send(p_1, e))$. On the same lines, $send_{p_0 \to n_1@p_1}(e)$ is used to denote $eval@p_0(send(n_1@p_1, e))$. Next, we define the eval at some peer p, of a data expression of a remote tree.

Rule 5. $eval@p_1(t@p_2) = send_{p_2 \to p_1}(eval@p_2(t@p_2))$

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, and then send the result to p_1.

Given the above rules, we are in a position to define the evaluation of a tree $t@p_0$, whose root is labeled sc. We denote by $parList = [t_1, t_2, \cdots, t_n]$ the list of $param_i$-labeled children of the sc.

Rule 6. $eval@p_0(sc(p_1, s_1, parList)) = send_{p_0 \to sc@p_0}(send_{p_1 \to p_0}(q_1(send_{p_0 \to p_1}(eval@p_0(parList)))))$

where $eval@p_0(parList)$ stands for $[eval@p_0(t_1), \cdots, eval@p_0(t_n)]$. The second part of Rule 6 is best read from the innermost parenthesis to outer. To evaluate sc, p_0 first evaluates the parameters (innermost eval), then sends the result to p_1. Peer p_1 evaluates, in the usual sense, the query q_1 (the one which implements its service s_1), and sends the results back to p_0. Finally, p_0 inserts the results as a child of sc.

3 Local AXML

Clearly, the AXML semantics in section 2 is distributed (nested), that is, an AXML expression evaluation may lead to multiple AXML document updates at different peers. In this section, we show how the rules can be modified to acquire "local" semantics. The intuition is that with such local semantics, a distributed protocol for AXML systems (e.g., global concurrency control protocol) can be replaced by a local variant (sufficient if the individual peers implement locking locally without the need for a central/global concurrency control manager).

We assume that each tree t is unique, that is, there exists only one copy of t among all the peers (by extension, each node n is also unique). However, the location of t, at any point of time, is not fixed and it may move from one peer to another. To accommodate this, we replace the location identifier $@p$ with $@any$. The above replacement is based on the assumption that there exists an index IN (preferably DHT style) which keeps track of the current location of the trees. Basically, we have added an extra level of indirection: Given an expression of the form $eval@p(t@any)$, the current location of t is retrieved by querying IN and substituted, leading to the expression $eval@p(t@p_1)$ (assuming t is currently hosted by the peer p_1). In addition, we need to modify the evaluation semantics as follows:

Rule 3'. $send_{p_0 \to p_1}(t@p_0))$

Evaluating a send expression tree at p_0, hosting t, results in t being (physically) moved from p_0 (deleted from p_0) to p_1. Note that this is in contrast to creating a copy of t and sending it to p_1 (by the earlier semantics).

Rule 4'. $send_{p_0 \to n_1@any}(t@p_0)) = send_{p_0 \to n_1@p_1}(t@p_0)) = send_{p_1 \to p_0}(t_1@p_1)), send_{p_0 \to n_1@p_0}(t@p_0))$

where $t_1@p_1$ is the tree containing the node $n_1@p_1$. Given this, the tree t_1 is moved from peer p_1 to p_0. The final send is basically a local operation at p_0 (as the target tree t_1 is currently hosted by p_0).

Rule 5'. $eval@p_1(t@any) = eval@p_1(t@p_2) = eval@p_1(send_{p_2 \to p_1}(t@p_2))$

If the current location of t is $p_2 (\neq p_1)$, then t is moved to p_1 and evaluated locally at p_1.

Rule 6'. $eval@p_0(sc(p_1, s_1, parList)) = send_{p_0 \to sc@p_0}(send_{p_1 \to p_0}(q_1), (eval@p_0(q_1, parList)))$

Any changes, required in the semantics for parameter evaluation, are taken care of by the modified Rules 3', 4' and 5' above. Here, we only discuss the modified semantics for service s_1's evaluation. Peer p_1 sends the query q_1 (the one which implements its service s_1) to p_0. This ensures that the complete service call evaluation can be processed locally at p_0. Intuitively, the above leads to a local semantics by pulling the required document and query trees from their respective peers and evaluating the expressions locally, rather than "pushing" them to the peers where the target document and query trees are located.

4 AXML Replication

Till now, we have studied AXML semantics without replication. In this section, we provide "recovery" semantics for AXML systems which allow replication, that is, there may exist more than one copy of a tree t on the peers $p_1, p_2, \cdots, p_n$ (the trees $t@p_1, t@p_2, \cdots, t@p_n$ are equivalent). Given this, the peers $p_1, p_2, \cdots, p_n$ are also referred to as the replicated peers of t. While replication leads to enhanced performance and throughput, the main challenge is with respect to keeping the replicated copies in sync, that is, an update on a tree t at any of the the peers hosting t, needs to be propagated to all the other replicated peers of t. More precisely, we discuss how replication guarantees can be provided in the event of a failure (especially, disconnection of a replicated peer) during the update propagation phase.

We consider the primary-secondary configuration for AXML replication. In this configuration, a peer, among the replicated peers of a tree t, is designated as the *primary* of t (denoted pr_t), and the remaining are referred to as *secondaries* of t. Basically, with this configuration, an update on a tree t can only occur at pr_t, while a query based on t can be answered by any of the replicated peers of t. Thus, the primary is responsible for propagating any updates to the secondaries. We assume that a primary pr_t retains a list of the secondaries of t, referred to as $list - sect_t$. Further, each peer $p \in list - sect_t$ is aware of pr_t; but is unaware of the other peers in $list - sect_t$.

We consider the evaluation of an AXML expression e, $eval@p(e)$, as a transactional unit. We discuss AXML replication with both eager and lazy semantics in detail in the sequel.

4.1 Eager

With eager replication, an update on a tree $t@p$ as part of the transaction $\tau = eval@p(e)$, is propagated to the other replicated peers of t within the same transaction τ. To accommodate the update propagation part within an AXML expression evaluation (transaction), we modify rules 4 and 6 (section 2) as follows:

Rule 4". $send_{p_0 \to n_1@p_1, n_2@p_2, \cdots, n_k@p_k}(t@p_0) = \Phi$

As before, sending $t@p_0$ to the locations $n_i@p_i$ returns an empty result at p_0, and as a side effect, at each p_i, the result of $eval@p_0(t@p_0)$ is added as a child of $n_i@p_i$. We use the short-hand $send_{p_0 \to fwList}(e)$ to denote $eval@p_0(send(fwList, e))$, where $fwList$ is a list of nodes.

Rule 4a". $sendwa_{p_0 \to fwList}(t@p_0) = send_{p_0 \to fwList}(t@p_0), send_{p(fwList) \to p_0}(ack)$

where $p(fwList)$ denotes the set of peers hosting (trees of) the nodes in $fwList$. Here, the peers in $p(fwList)$, after having performed the updates, send an acknowledgment back to p_0.

Rule 6". $eval@p_0(sc(p_1, s_1, parList)) = sendwa_{p_0 \to list - sec_{t(sc@p_0)}}(send_{p_1 \to p_0}(q_1(send_{p_0 \to p_1}(eval@p_0(parList)))))$

Basically, the modification allows p_0 to propagate the invocation results (of $sc@p_0$) to a set of peers (secondaries of the affected tree $t(sc@p_0)$). Given this, a transaction $\tau = eval@p_0(e)$ *commits* only after p_0 has received acknowledgments from all the corresponding secondaries. Note that p here is also the primary of t, that is, $p = pr_t$. Next, we discuss the possible failures during the update propagation part, and their recovery semantics to provide the following replication guarantee:

Eager replication guarantee. At any point of time, evaluation of an AXML query expression e based on tree t produces the same result, irrespective of the (replicated) peer (of t) where e was evaluated.

Secondary disconnection. We consider a transaction τ = $eval@p(e)$ which has updated the tree $t@p$. Now, let us consider propagation of $t@p$'s update, and assume that a secondary $p_1 \in list - sec_t$ has disconnected. As a result, p would not receive the acknowledgment from p_1. Given this, p attempts forward recovery by following a timeout mechanism. If p does not receive an acknowledgment after t secs (configurable), it retries the send. If p does not receive an acknowledgment from p_1 even after m retries (again, configurable), it deletes p_1 from $list - sec_t$.

Secondary reconnection. A peer p_1, on reconnecting, does the following (before performing any updates or answering queries): For each hosted tree t, if p_1 was a secondary of t before disconnection, then p_1 tries to contact pr_t. If successful,

1. p_1 synchronizes the state of $t@p_1$ with $t@pr_t$.
2. pr_t adds p_1 to $list - sec_t$.

Otherwise (if p_1 was the primary of t before disconnection, or p_1 cannot contact pr_t), p_1 has the following couple of options: Basically, p_1 may not be able to contact pr_t if the primary has changed (detailed later while discussing primary disconnection) during the disconnection-reconnection period of p_1.

- Initiate a flooding of the P2P network to locate pr_t. Rather than flooding the whole network, if it is feasible for each secondary to be aware of the other secondaries as well, then p_1 may try to locate pr_t via the other secondaries (before initiating flooding, if required).
- p_1 deletes t from its repository and stops being a replicated peer of t.

Primary disconnection. As before, we consider a transaction $\tau = eval@p(e)$ which updates tree $t@p$, and $p = pr_t$ gets disconnected during the update propagation phase. Further, let us consider two secondaries $p_1 \neq p_2 \in list - sec_t$. Given this, we first analyze the problem scenario. Note that the receive of messages is not instantaneous. As a result, p_1 may receive the propagated update of t before p_2 (or vice versa). At this stage, a query based on t, would produce different results depending on whether it was posed at p_1 or p_2. An alternative (for p_1 and p_2) is to wait for a commit confirmation of τ from p, before answering queries with the updated state of t. However, this leads to an infinite cycle of update and acknowledgment (confirmation) messages.

As such, we follow another alternative: Given a query q based on t at a replicated peer p_1 of t, we require that p_1 first check if the transaction τ corresponding to the last propagated update on t has already committed at pr_t (or not), before answering q. Basically, the commit of a transaction updating t at pr_t, implies that the update on t has been propagated and applied by all the alive secondaries of t ($\in list - sec_t$). Clearly, p_1 does not need to check again if it already knows that τ has committed at pr_t (it performed the check for a previous query, and there haven't been any further update propagations with respect to t since then). To accommodate the above additional check (if required) on the affected trees of a query q, we extend Rule 2 as follows:

Rule 2". $eval@p(q(t_1@p, \cdots, t_n@p)) =$
$Check - Status@p(t_1@p, \cdots, t_n@p)$,
$q(eval@p(t_1@p), \cdots, eval@p(t_n@p))$

While performing the above check, if p_1 detects that pr_t has disconnected (that is, it cannot contact pr_t), then it does the following:

- Assume the role of a coordinator and initiate flooding to detect the replicated peers of t. At this stage, another peer p_2 may also have detected the disconnection of pr_t, and initiated flooding. Given this, both p_1 and p_2 will eventually receive each others' flood messages. Here, we assume that p_1 and p_2 negotiate, and only one of them continues as the coordinator.
- The state of t on the replicated peers is synchronized (possibly updated to the latest).
- Execute a leader election algorithm among the replicated peers.
- The elected leader becomes the new primary pr_t, and its $list - sec_t$ is assigned the list of replicated peers (excluding pr_t).

Finally, we reiterate the query mechanism. Given a query q based on t at a replicated peer p_1 of t, if there exists an "unchecked" update on t, then check the status of the transaction τ at pr_t corresponding to the last propagated update on t. Then, we have the following possibilities:

- τ has committed, answer q based on the updated state of t.
- τ hasn't committed yet, answer q based on the previous updated state of t.
- pr_t has disconnected: Perform the recovery steps as discussed above. In the meantime, answer q based on the previous updated state of t. Another alternative would be to wait for the new primary pr_t to be resurrected, and then answer based on the latest synchronized state of t. However, this might lead to long answering periods for queries. Thus, the alternative to choose is basically a choice between lower response time or (possibly) more current data.

Primary reconnection. Analogous to the secondary reconnection scenario, where a secondary p_1 is not able to contact pr_t.

To summarize, Rules 1, 2", 3, 4", 4a", 5 and 6" provide the complete (including recovery) semantics of an *eager* AXML replication system.

4.2 Lazy

Here also, any update on a tree t at its primary pr_t, needs to be propagated to the secondary peers of t. However, the

propagation is not performed as part of the expression evaluation transaction. Rather, the updates are propagated "as and when" convenient after the corresponding transaction has committed at the primary. This allows the primary to proceed with the next expression evaluation (transaction) without having to wait for the corresponding secondaries' acknowledgments, increasing the system throughput. However, for an update on a tree t, if pr_t disconnects before the update has been propagated to any of the peers in $list-sec_t$, then the update is lost forever. As such, we discuss recovery semantics for the following replication guarantee:

Lazy replication guarantee. For a pair of propagated updates u_1 and u_2 on tree t at secondary peers $p_1 \neq p_2 \in list-sec_t$, if $u_1(u_2)$ occurs before $u_2(u_1)$ at p_1, then $u_1(u_2)$ also occurs before $u_2(u_1)$ at p_2.

Intuitively, updates on a replicated tree t may not occur at the same time on its secondaries, however they occur in the same order. This is particularly significant for programs which rely on a stream of inputs, e.g., AXML continuous services [2], (user) session guarantees [15, 16], etc.

Secondary disconnection. Here, the secondaries, after performing a propagated update do not send acknowledgments. As such, the corresponding primary is not in a position to detect secondary disconnection, and perform the requisite forward recovery (sub-section 4.1). To overcome this, in addition to Rule 4, we need the following "extended" send:

Rule 4b. $sende_{p_0 \rightarrow fwList}(t@p_0, sucList) =$
$Check - Alive(p(n_1@p_1)), send_{p_0 \rightarrow n_1@p_1}(t@p_0),$
$assign(sucList, sucList \cup n_1@p_1),$
$sende_{p_1 \rightarrow (fwList - n_1@p_1)}(t@p_0),\ send_{p_{last} \rightarrow p_0}(sucList),$
$assign(fwList, sucList)$

Initially, $fwList = n_1@p_1, n_2@p_2, \cdots, n_k@p_k$ and $sucList = \Phi$. Then, its semantics are given as follows:

1. Select the first location $i = 1, n_i@p_i \in fwList$.
2. Check if p_i is still connected (e.g., using ping messages). If p_i has disconnected, then $fwList = fwList - n_i@p_i$, and go back to Step 1.
3. Otherwise, $sucList = sucList \cup n_i@p_i$, and recursively invoke $sende$ with the remaining $fwList - n_i@p_i$. On termination ($fwList = \Phi$), the last peer p_{last} sends the $sucList$ back to p_0, upon which p_0 can update its $fwList$ for any future send's. Basically, $sucList$ denotes the set of (alive) peers, to which the message could be transmitted successfully.

Further, for a triplet $n_x@p_x, n_y@p_y, n_z@p_z \in fwList$ with the following sequence ρ of operations: $Check - Alive(p_y)$, $send_{p_x \rightarrow n_y@p_y}(t@p_0)$, $sucList = sucList \cup n_y@p_y$, $Check - Alive(p_z)$, and $send_{p_y \rightarrow n_z@p_z}(t@p_0)$; ρ needs to occur as an "atomic" unit, that is, either all of them succeed or none. Otherwise, we may have a situation where a secondary p_y disconnects after receiving a propagated update (from p_x), but before propagating it further (to the next secondary p_z).

Given this, for each evaluation $eval@p_0(sc(p_1, s_1, parList))$ (Rule 6), its results can be propagated to the secondary peers of $sc@p_0$ ($\in list - sec_{sc@p_0}$) by using the $sende$ primitive with $fwList = list - sec_{sc@p_0}$. Note that the update propagation ($sende$) here is outside the transaction context, that is, a transaction still corresponds to an expression evaluation given by Rules 1 - 6 (section 2). Thus, a primary pr_t queues any pending updates on t, and propagates ($sende$) the next pending update only after receiving the $sucList$ corresponding to the previously propagated update on t (if any, and updating $list - sec_t$). This implicitly ensures that, for a tree t, its secondary peers receive any updates on t in the same order.

Primary disconnection. As mentioned before, if a primary pr_t disconnects before a pending update on t could be propagated, then that update is lost. Here, we consider the scenario where pr_t disconnects after initiating the propagation ($sende$), but before receiving the $sucList$ from the last secondary p_{last}. Given this, p_{last} detects pr_t's disconnection, and initiates the recovery procedure as given for the eager replication strategy (sub-section 4.1). That is, (i) Initiate flooding to detect the replicated peers of t. (ii) Synchronize t's state on the replicated peers. (iii) Elect the new leader pr_t and assign $list - sect_t$.

Primary/Secondary reconnection. The steps to follow on reconnection of a primary/secondary are the same, as given for the eager scenario (sub-section 4.1).

5 Conclusion and Future Work

In this work, we studied replication from a recovery perspective. For both eager and lazy replication strategies, we outlined recovery procedures to handle peer disconnection/reconnection. We also discussed the replication guarantees that can be provided in such a scenario. While the proposed algebra is with respect to AXML systems, we believe that it is also applicable to more general XML/P2P based systems as well.

The AXML implementation is available at http://forge.objectweb.org/projects/activexml/. The work in this paper is part of ongoing work to provide a transactional framework for AXML systems [7, 8]. Our future work

includes studying the effect of other transactional aspects, especially, logging on replication. Logging XML data is very expensive. As such, we would like to optimize AXML logging as much as possible. Towards this end, we are investigating if we can use the concept of "confluence" [5] in conjunction with replication and recovery.

Acknowledgment. I would like to thank Blaise Genest and Mahantesh Surgihalli for their helpful suggestions which helped to improve the work in this paper considerably. This work is supported by the ANR DOCFLOW and CREATE ACTIVEDOC projects.

References

[1] Active XML. http://www.activexml.net

[2] Active XML User Guide. http://www.activexml.net/AXML%20Guide.pdf

[3] D. Agrawal, A. Abbadi, and I. Stanoi. Using broadcast primitives in replicated databases. In: proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS), Amsterdam, The Netherlands, 1998, pp. 148-155.

[4] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML Documents with Distribution and Replication. In: proceedings of the 22nd ACM International Conference on Management of Data (SIGMOD), San Diego, CA, USA, 2003, pp. 527-538.

[5] S. Abiteboul, T. Milo, and O. Benjelloun. Positive Active XML. In: proceedings of the 23rd ACM Symposium on Principles of Database Systems (PODS), Paris, France, 2004, pp. 35-45.

[6] S. Abiteboul, E. Taropa, and I. Manolescu. A Framework for Distributed XML Data Management. In: proceedings of the 10th International Conference on Extending Database Technology (EDBT), Munich, Germany, 2006, pp. 1049-1058.

[7] D. Biswas, and B. Genest. Decomposing Minimal Observability for Transactional Services. Submitted Sep 2007, http://perso.crans.org/g̃enest/BG07.ps

[8] D. Biswas, and I.-G Kim. Atomicity for P2P based XML Repositories. In: proceedings of the 2nd IEEE International Workshop on Services Engineering (SEIW), Istanbul, Turkey, 2007, pp. 369-376.

[9] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In: proceedings of the 18th ACM International Conference on Management of Data (SIGMOD), Philadelphia, PA, USA, 1999, pp. 97-108.

[10] E. Cohen, and S. Shenker. Replication Strategies in Unstructured Peer to Peer Networks. In: proceedings of the ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), Pittsburgh, PA, USA, 2002, pp. 177-190.

[11] K. Daudjee, and K. Salem. Lazy Database Replication with Ordering Guarantees. In: proceedings of the 20th IEEE International Conference on Data Engineering (ICDE), Boston, MA, USA, 2004, pp. 424-435.

[12] B. Kemme, R. Jimenez-Peris, M. Patino-Martinez, and G. Alonso. Improving the scalability of fault tolerant database clusters. In: proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, 2002, pp. 477-484.

[13] G. On, J. Schmitt, and R. Steinmetz. The Effectiveness of Realistic Replication Strategies on Quality of Availability for Peer-to-Peer Systems. In: proceedings of the 3rd IEEE International Conference on Peer-to-Peer Computing (P2P), Linkping, Sweden, 2003, pp. 57-64.

[14] P. Padmanabhan, and L. Gruenwald. Managing Data Replication in Mobile Ad-Hoc Network Databases. In: proceedings of the 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), Atlanta, GA, USA, 2006, pp. 1-10.

[15] M. Surgihalli. Consistent Access to Replicated Web Service Registries. M. Sc. Thesis, Memorial University of Newfoundland, NL, Canada, Sep 2006, http://www.cs.mun.ca/m̃ahan/MScthesis.pdf

[16] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welsh. Session guarantees for weakly consistent replicated data. In: proceedings of the Internaltional Conference on Parallel and Distributed Information Systems (PDIS), Austin, TX, USA, 1994, pp. 140-149.

[17] Web Services Description Language (WSDL) Specification. http://www.w3.org/TR/wsdl