Atomicity for P2P based XML Repositories

Debmalya Biswas, Il-Gon Kim
IRISA-INRIA
Campus Universitaire de Beaulieu, Rennes, 35042 France
{dbiswas, ikim}@irisa.fr

Abstract

Over the years, the notion of transactions has become synonymous with providing fault-tolerance, reliability and robustness to database systems. However, challenges arise when we try to apply them to novel computing paradigms such as ActiveXML (AXML) systems. AXML provides an elegant platform to integrate the power of XML, Web services and Peer to Peer (P2P) paradigms by allowing (active) Web services calls to be embedded within XML documents. We propose a transactional framework which provides relaxed ACID properties to AXML systems. Relaxed atomicity is usually provided with the help of compensation. However, current compensation based models assume the existence of a pre-defined compensating operation. Also, compensation is assumed to be more or less peer (or service provider) dependent, i.e., the original and compensating services are provided by the same peer. We show how compensation for AXML transactions can be constructed dynamically at run-time and achieved in a peer independent manner. Finally, we consider the issue of peer disconnection, an inherent trait of P2P systems, and propose an innovative solution based on peer "chaining".

1. Introduction

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

- AXML documents: XML documents with embedded Web service calls. The embedded services may be AXML services (defined below) or generic Web services. For example, the XML snippet below is an AXML document with the embedded service call "getGrandSlamsWon".

```
<?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 = "replace" serviceNameSpace =
"getGrandSlamsWon" serviceURL = "…" methodName =
"getGrandSlamsWon">
         <axml:params>
            <axml:param name="name">
            <axml:value>Roger Federer</axml:value>
         </axml:params>
      </axml:sc>
   </player>
   …
</ATPList>
```

- AXML Services: Web services defined as queries/updates over AXML documents. Note that AXML services are also exposed as a regular Web service (with a WSDL description file).

- AXML peers: Nodes where the AXML documents and services are hosted. AXML peers also provide a user interface to query/update the AXML documents stored locally.

An embedded service call may be invoked (or materialized): 1) in response to a query on the AXML document (the invocation results are required to evaluate the query), or 2) periodically (specified by the "frequency" attribute of the AXML service call tag <axml:sc>). The invocation results may be static XML nodes or another service call. The service calls can have the modes: a) replace: the previous results are replaced by the current invocation results, or b) merge: the invocation results are appended as siblings of the previous invocation results.

A transaction can be considered as a group of operations encapsulated by the operations Begin and Commit/Abort having the following properties (ACID):

- Atomicity: Either all the operations are executed or none of them are executed. In case of failure (abort), the effects of any operation belonging to the transaction are canceled (roll-back).

- Consistency: Each transaction moves the system from one consistent state to another.

- Isolation: To improve performance, often several transactions are executed concurrently. Isolation necessitates that the effects of such concurrent execution are equivalent to that of a serial execution (serializability).

- Durability: Once a transaction commits, its effects are durable, i.e., they should not be destroyed by any system or software failure.

While transactions are synonymous with providing fault-tolerance, reliability and robustness to database systems, challenges arise when we try to apply them to a novel computing paradigm such as AXML. Characteristics of an AXML system, important from a transactional point of view, are as follows:

- Distributed: The distributed aspect follows from 1) the capability to invoke services hosted on remote peers, and 2) distributed storage of parts of an AXML document across multiple peers [2]. In case of distributed storage, if a query Q on peer AP_1 is interested in part of an AXML document stored on peer AP_2 then there are two options: a) the query Q is decomposed and the relevant sub-query sent to the peer AP_2 for evaluation, or b) the required fragment of the AXML document is copied to the peer AP_1 and the query Q evaluated locally (on AP_1). Both the above options require invoking a service on the remote peer and as such are similar in functionality to (1).

- Replication: AXML documents (or fragments of the documents) and services may be replicated on multiple peers [2].

- Nested: The nested aspect is mainly with respect to the nested (recursive) invocation of services. a) Local nesting: The service call parameters may themselves be defined as service calls. As such, evaluating a service call may require evaluating the parameters' service calls first. Analogously, a service invocation may return another service call as its result leading to a nested invocation of service calls. 2) Distributed nesting: Invocation of a service S_X of peer AP_2 , by peer AP_1, may require the peer AP_2 to invoke another service S_Y of peer AP_3 (while executing S_X) leading to a nested invocation of services across multiple peers.

- Duration: The duration of AXML transactions, especially, those including generic Web services can be very long (in hours).

- Concurrent (simultaneous) access: The number of users accessing the system simultaneously can be very high.

- Availability: In true P2P style, we consider that the set of peers in the AXML system keeps changing with peers joining and leaving the system arbitrarily.

Given the above characteristics, we propose a transactional framework which provides relaxed ACID properties to AXML systems. Relaxed atomicity is usually provided with the help of compensation [3][4]. However, current compensation based models assume the existence of a pre-defined compensating operation (for each operation), which is invoked in case the effects of the original operation need to be canceled. Also, compensation is assumed to be more or less peer (or service provider) dependent, i.e., the original and compensating operations/services are provided by the same peer. We show how the compensating operations can be constructed dynamically and compensation achieved in a peer independent manner for AXML transactions. Finally, peer disconnection is an inherent trait of P2P systems. As far as we know, the issue of peer disconnection hasn't been considered explicitly in a transactional context. We outline an innovative solution based on "chaining" the involved peers to handle AXML peer disconnection.

The rest of the paper is organized as follows. In section 2, we discuss some related work. Section 3 is dedicated to the transactional framework for AXML systems. More precisely, we discuss dynamic compensation construction, nested and peer independent recovery, and the issue of peer disconnection in respective sub-sections of section 3. Section 4 concludes the paper and provides some directions for future work.

2. RELATED WORK

The notion of transactions has been evolving over the last 30 years. As such, it would be a vain effort to even try and mention all the related research here. Given this, we suffice to mention the transactional models which have been proposed specifically for the XML and Web services paradigms. Links to general transactional related work are provided in the text as and when required.

[5] and [6] consider lock-based concurrency control protocols customized for XML repositories. However, due to the "active" nature of AXML documents, lock-based protocols are not well suited for AXML systems.

[7] describes how compensating transactions can be modeled based on the active database concept of triggers, basically, as Event-Condition-Action (ECA) rules. [8] presents a forward recovery based transaction model. It introduces the concept of co-operative recovery (in the context of Web services). In [9], Pires et. al. propose a framework (WebTransact) for building reliable Web services compositions. [4] stresses the importance of Cost of Compensation and end-user feedback while performing compensation for Web services compositions. [10] and [11] discuss in detail the practical implications of compensation with respect to hierarchical Web Services Compositions. Broadly, given the process or workflow underpinnings of Web services compositions, the focus of the above works is towards process atomicity. However,

Web services or embedded service calls are only a part of AXML. As mentioned earlier, AXML systems provide an elegant integration of the XML (data), Web services (process) and P2P (infrastructure) platforms. As such, we also need to consider data related aspects, e.g., consistent query and update of AXML documents, etc. From a P2P perspective, transactions haven't received much attention till now as their commercial use has been mostly restricted to file (or resource) sharing systems where failure resilience equates to maintaining sufficient information (by the P2P client) so that a file download can be resumed (from the original or a different peer). However, the trend is slowly changing with a steady rise in the use of P2P systems for collaborative work [1][12] including the Grid [13]. In this paper, we consider the issue of peer disconnection from a transactional perspective.

3. AXML Atomicity

The possible operations on AXML documents are queries, updates, inserts and deletes (update operations with action types "replace", "insert" and "delete", respectively). The operations can be submitted locally or by invoking the query/update services (AXML services) provided by an AXML peer. We do not differentiate between the two modes and use the terms operation and service interchangeably throughout the paper. *We consider a transactional unit as a set of update/query operations (services).*

3.1. Dynamic Compensation

Compensation based models, in the event of a failure, preserve atomicity by executing a compensating operation which is responsible for semantically undoing the effects of the original operation. For example, the compensation of "Book Hotel" is "Cancel Hotel Booking". Here, it helps to recall that compensation is not equivalent to the traditional "undo"; rather, it is another forward operation (transaction) which moves the system to an acceptable state (which maybe different from the initial state [14]). Compensation is achieved by executing the compensating operations in the reverse order of the execution of their respective forward operations. Usually, the compensation handlers for a service call are pre-defined statically on the lines of exception/fault handlers. However, static definition of compensation handlers is not feasible for AXML systems. We consider the issue in detail in the rest of the section.

The compensation for an insert (AXML update operation with action type "insert") is delete and vice versa. Similarly, the compensation for an update (AXML update with action type "replace") is another update which reinstates the old data values. To illustrate, let us consider the following AXML document:

ATPList.xml:

```
1:<?xml version = "1.0" encoding = "UTF-8"?>
2:<ATPList date = "18042005">
3:    <player rank = 1>
4:        <name>
5:            <firstname>Roger</firstname>
6:            <lastname>Federer</lastname>
7:        </name>
8:        <citizenship>Swiss</citizenship>
9:         <axml:sc mode = "replace" serviceNameSpace =
"getPoints" serviceURL = "…" methodName =
"getPoints">
10:            <axml:params>
11:               <axml:param name = "name">
12:                <axml:value>Roger Federer</axml:value>
13:            </axml:params>
14:            <points>475</points>
15:        </axml:sc>
16:         <axml:sc mode = "merge" serviceNameSpace =
"getGrandSlamsWonbyYear" serviceURL = "…"
methodName = "getGrandSlamsWonbyYear">
17:            <axml:params>
18:               <axml:param name = "name">
19:               <axml:value>Roger Federer</axml:value>
20:               <axml:param name = "year">
21:                              <axml:value>$year (external
value)</axml:value>
22:            </axml:params>
23:                    <grandslamswon year = "2003">A,
W</grandslamswon>
24:                    <grandslamswon year = "2004">A,
U</grandslamswon>
25:        </axml:sc>
26:    </player>
          …
…</ATPList>
```

AXML update operations (analogous to XQuery updates [15]) can be divided into two parts: 1) the <location> query to locate the target nodes, and 2) the actual update actions. The <location> query evaluation may involve service call materializations, and as such, updates to the AXML document. The data (nodes) required for compensation cannot be predicted in advance and would need to be read from the log at run-time. For example, let us consider an AXML delete operation and its compensation as shown below:

Delete operation:

```
<action type = "delete">
```

<location>Select p/citizenship from p in ATPList//player where p/name/lastname = Federer; </location>
</action>

Compensating operation:

<action type = "insert">
<data> <citizenship>Swiss</citizenship> </data>
<location>Select p/points/.. from p in ATPList//player where p/name/lastname = Federer;</location>
</action>

where the <location> and <data> of the compensating insert operation are the parent (/..) of the deleted node and the result of the <location> query of the delete operation, respectively. Thus, the delete operations as well as the results of the <location> queries of the delete operations need to be logged to enable compensation. Note that the above compensation mechanism does not preserve the original ordering of the deleted nodes. For ordered documents, the situation is slightly more complicated and formulation of the compensating operation would depend on the actual semantics of the insert operation. For example, the situation is simplified if the insert operation allows insertion "before/after" a specific node [15].

For AXML insert operations, we assume that the operation returns the (unique) ID of the inserted node. As such, the compensating operation (for the insert operation) is a delete operation to delete the node having the corresponding ID. An AXML replace operation is usually implemented as a combination of a delete and update operation, i.e., delete the node to be replaced followed by insertion of a node (having the updated value) at the same position. Compensation for a replace operation is shown below:

Replace operation:

<action type = "replace">
<data> <citizenship>USA</citizenship> </data>
<location>Select p/citizenship from p in ATPList//player where p/name/lastname=Nadal; </location>
</action>

decomposes to:

<action type = "delete">
<location>Select p/citizenship from p in ATPList//player where p/name/lastname=Nadal; </location>
</action>
<action type = "insert">
<data> <citizenship>USA</citizenship> </data>
<location>Select p/citizenship/.. from p in ATPList//player where p/name/lastname=Nadal; </location>
</action>

Compensating operation:

<action type = "delete">
<location>Select p/citizenship from p in ATPList//player where p/name/lastname=Nadal; </location>
</action>
<action type="insert">
<data><citizenship>Swiss</citizenship></data>
<location>Select p/citizenship/.. from p in ATPList//player where p/name/lastname=Nadal; </location>
</action>

Now, let us consider compensation for AXML query operations. Traditionally, query operations do not need to be compensated as they do not modify data. However, AXML query evaluation, due to the possibility of service call materializations, is capable of modifying the AXML document, e.g., insertion of the result nodes (and deletion of the previous result nodes in "replace" mode). There are two possible modes for AXML query evaluation: lazy and eager. Of the two, lazy evaluation is the preferred mode and implies that only those embedded service calls (in an AXML document) are materialized whose results are required for evaluating the query. *As the actual set of service calls materialized is determined only at run-time, the compensating operation for an AXML query cannot be pre-defined statically (has to be constructed dynamically)*. Given that the required insertion (deletion) of the result nodes are achieved using AXML Insert (Delete) operations, the compensating operation for an AXML query operation can be formulated as discussed for the AXML update operations earlier. The following couple examples (query operations A and B) illustrate the above aspect.

Query operation A:

<action type = "query">
<location>Select p/citizenship, p/grandslamswon from p in ATPList//player where p/name/lastname=Federer: </location>
</action>

Lazy evaluation of the above query would result in the materialization of the embedded service call *"getGrandSlamsWonbyYear" (and not "getPoints")* leading to the following AXML document:

```
1:<?xml version = "1.0" encoding = "UTF-8"?>
2:<ATPList date = "18042005">
3:    <player rank = 1>
            …
```

```
25:                <grandslamswon year = "2005">A, F</grandslamswon>
26:        </axml:sc>
27:    </player>
          …
…</ATPList>
```

The only change in the above AXML document, with respect to ATPList.xml, is the addition of line 25 (lines 4-24 are the same as ATPList.xml). Thus, the compensation for the above query operation would be a delete operation to delete the node "<grandslamswon year = "2005">A, F</grandslamswon>".

However, if the query were defined as follows:

Query operation B:

```
<action type = "query">
   <location>Select p/citizenship, p/points from p in ATPList//player where p/name/lastname=Federer;
   </location>
</action>
```

Lazy evaluation of query B would result in the materialization of the embedded service call *"getPoints" (and not "getGrandSlamsWonbyYear")* leading to the following AXML document:

```
9:        <axml:sc mode = "replace" serviceNameSpace = "getPoints" serviceURL = "…" methodName = "getPoints">
10:          <axml:params>
11:             <axml:param name = "name">
12:              <axml:value>Roger Federer</axml:value>
13:          </axml:params>
14:          <points>890</points>
15:       </axml:sc>
```

The only change in the above AXML document, with respect to ATPList.xml, is in line 14 (the value of points has changed from 475 to 890). Thus, the compensation for the above query operation would be a replace operation to change the value of the node "<points>890</points>" back to 475.

As shown by the above examples, *static compensation definition is not feasible for query operations, and as such, need to be constructed dynamically at run-time.*

3.2. Nested and Peer Independent Recovery

On the lines of Java and Business Process Execution Language for Web Services (BPEL4WS) [16], we assume the existence of multiple fault handlers corresponding to the embedded service calls in an AXML document. For example, the embedded service call "getGrandSlamsWon" defined with fault handlers would be as follows:

```
<axml:sc … methodName="getGrandSlamsWon">
   <axml:params>
      <axml:param name="name">
      <axml:value>Rafel Nadal</axml:value>
   </axml:params>
   <axml:catch faultName="A" faultVariable="…"><!-- handle the fault --></axml:catch>
   <axml:catch faultName="B" faultVariable="…"><!-- handle the fault --></axml:catch>
   <axml:catchAll><!-- handle the fault --></axml:catchAll>
</axml:sc>
```

<!-- handle the fault --> part can be either some Java code or constructs like <axml:retry times="" wait=""><axml:sc …></axml:sc></axml:retry> which allow specifying the number of times a service invocation can be retried and the duration to wait before retrying. The optional <axml:sc …> allows retrying the invocation using a replicated peer (provided replication is supported).

Next, we discuss a nested recovery protocol for AXML transactions. We need some additional notations before presenting the protocol. The peer at which a transaction T_A is originally submitted is referred to as its origin peer. Peers whose services are invoked while processing T_A are referred to as the participant peers of T_A. On submission of a transaction T_A at a peer AP_1 (its origin peer), the peer creates a transaction context TC_{A1}. The transaction context, managed by the transaction manager, is a data structure which encapsulates the transaction id with all the information required for concurrency control, commit and recovery of the corresponding transaction.

We outline the nested recovery protocol with the help of an example scenario as shown in Fig. 1. Fig. 1 shows a scenario where the peer AP_5 fails while processing the service S_5 as part of transaction T_A. Given this, nested recovery can be achieved as follows:

1. AP_5 aborts the transaction TC_{A5} and sends "Abort T_A" messages to the peers whose services it had invoked (AP_6) and the peer which had invoked the service S_5 (AP_3).
2. The peer AP_6, on receiving the message "Abort T_A", aborts TC_{A6}.
3. The peer AP_3, on receiving the message "Abort T_A", tries to recover using the (application specific) fault handlers defined for the embedded service call S_5.
4. If there are no matching fault handlers, AP_3 follows the same course of action as AP_5, i.e., abort TC_{A3} and send "Abort …" messages to the peers whose services it had invoked as part of processing S_3 (AP_4) and the peer which had invoked the service S_3 (AP_1).

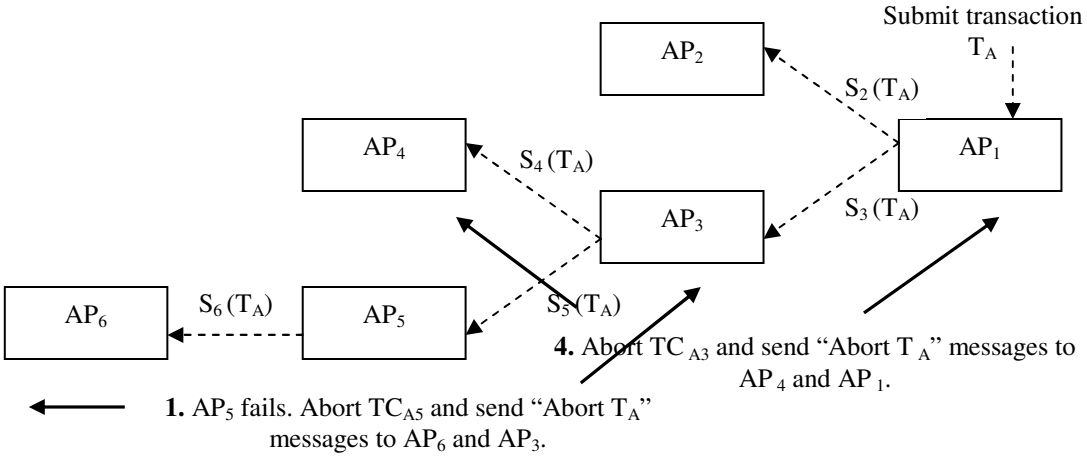

Fig. 1. Nested recovery protocol for AXML Transactions

This backward propagation of the fault continues till it finally reaches the origin peer (in which case, the whole transaction is aborted). The above recovery protocol is nested, i.e., the intermediate peers AP_3 and AP_1 have the option of performing forward recovery using the application specific fault handlers or backward recovery by propagating the failures to their parents (send "Abort T_X" messages). The preferred option would depend on the "cost" of forward versus backward recovery. *For AXML systems, the number of XML nodes affected (traversed) is usually a good measure of the cost of an operation (forward or compensating).* Note that the number of affected nodes would remain the same even if a different peer is selected to redo (forward recovery) the operation (a different peer, in this case, can only be a peer containing a replicated copy of the affected AXML document). Given this, we ignore the cost aspect and consider forward recovery as the preferred solution and undo only as much as required (as considered in the protocol above).

A peer independent variation of the above recovery protocol would be as follows: In the above protocol, the original peers (peers which had originally executed the services) are responsible for their compensation as well. Now, let us assume that a peer AP_X, processing the invocation of a service S, also returns the definition of the compensating service CS_{SX} of S along with the invocation results. The compensating service CS_{SX} is defined as "a service capable of compensating the modifications at AP_X which occurred as a result of processing the service S". The compensating service definitions can also be sent to the origin peer directly. Given this, a peer trying to perform recovery (say, the origin peer) can directly invoke the compensating services (CS_{SX}) on their original peers (AP_X). The original peers do not even need to be aware that the services they are executing are actually compensating services. The intuition is to free the original peers form the burden of compensation as much as possible.

3.3. Peer Disconnection

Peer disconnection is an inherent trait of P2P systems. Related P2P research relies on ping (or keep-alive) messages to detect peer disconnection. Resilience is provided against disconnection via redundancy, e.g., if a peer, from which a file is being downloaded, gets disconnected then the download can be resumed from some other peer sharing the same file. We consider peer disconnection from a transactional point of view and illustrate our solution with the help of the scenario in Fig. 2 (a slightly modified version of Fig. 1). *The main objective of the proposed solution is to minimize loss of effort by detecting the disconnection as soon as possible and reuse already performed work as much as possible.* The actual steps to be executed to handle peer disconnection vary based on the peer which got disconnected and the peer which detected the disconnection.

The list of active peers is denoted as follows: $[AP_X \rightarrow AP_Y]$ implies an invocation of AP_Y's service by AP_X. Parallel invocation of AP_Y and AP_Z s' services by AP_X is denoted as $[AP_X \rightarrow [AP_Y] \parallel [AP_Z]]$. Finally, super peers (trusted peers which do not disconnect) are highlighted by an * following their identifiers (AP_X*).

(a) *Leaf node disconnection* (peer AP_6 gets disconnected and the disconnection is detected by its parent AP_3): AP_3 follows the nested recovery protocol discussed earlier.

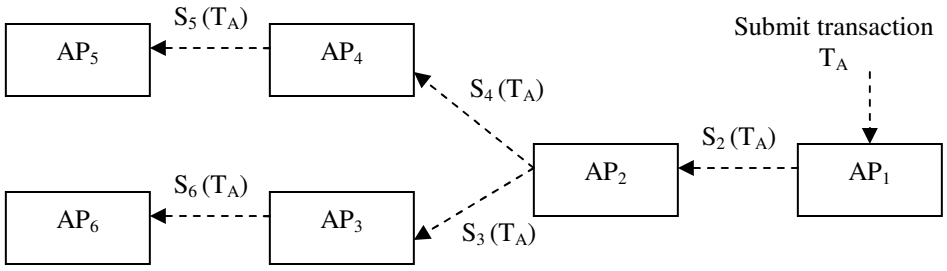

Fig. 2. Illustration for the peer disconnection scenario

(b) *Parent disconnection detected by child node* (peer AP_3 gets disconnected and the disconnection is detected by its child AP_6): Let us assume that AP_6 detects the disconnection of AP_3 while trying to return the results of processing service S_6 to AP_3. Traditional recovery would lead to AP_6 (aborting) discarding its work and actual recovery occurring only when the disconnection is detected by peer AP_2. A more efficient solution can be achieved if AP_3 passes the list of active peers $[AP_1^* \rightarrow AP_2 \rightarrow [AP_3 \rightarrow AP_6] \parallel [AP_4 \rightarrow AP_5]]$ also while invoking the service S_6 of AP_6. Given this, as soon as AP_6 detects the disconnection of AP_3, it can send the results directly to AP_2 (informing AP_2 of the disconnection as well). Once AP_2 becomes aware of the disconnection, it follows the nested recovery protocol discussed in the previous sub-section. Further, let us assume that AP_2 attempts forward recovery by invoking the service S_3 on a different peer (say, AP_X). In a general scenario, it might be very difficult to reuse the work already performed by AP_6. However, if we assume that S_6 was basically an invocation to materialize an input parameter of S_3 (recall that input parameters can also be defined as service calls) then it might be possible to reuse AP_6's work by passing the materialized results directly while invoking S_3 on AP_X. Finally, it is very likely that even AP_2 might have disconnected. Given this, AP_6 can try the next closest peer (AP_1) or the closest super peer (also, AP_1 in this case) in the list.

(c) *Child disconnection detected by its parent* (peer AP_3 gets disconnected and the disconnection is detected by its parent AP_2): Let us assume that AP_2 detects the disconnection of AP_3 via ping (or keep-alive) messages. As in the previous scenario, a more efficient recovery can be achieved if AP_2 is aware of the list of active peers $[AP_1^* \rightarrow AP_2 \rightarrow [AP_3 \rightarrow AP_6] \parallel [AP_4 \rightarrow AP_5]]$, especially, AP_6. In addition to attempting recovery using the nested recovery protocol, AP_2 can use the information about the children peers (of AP_3) to see if any part of their work can be reused. Even if reuse is not possible, AP_2 can at least use the information to inform the descendents (of AP_3) about the disconnection. This would prevent them from wasting effort (doing work which is ultimately going to be discarded).

(d) *Sibling disconnection* (peer AP_3 gets disconnected and the disconnection is detected by sibling AP_4): For data intensive applications, it is often the case that data is passed directly between siblings (rather than sibling A - parent - sibling B). In an AXML scenario, this is particularly relevant for subscription based continuous [1] services which are responsible for sending updated (streams of) data at regular intervals. Thus, a sibling would be aware of another sibling's disconnection if it doesn't receive data at the specified interval. Given such detection, AP_4 can use the list of active peers $[AP_1^* \rightarrow AP_2 \rightarrow [AP_3 \rightarrow AP_6] \parallel [AP_4 \rightarrow AP_5]]$ to notify the parent (AP_2) and children (AP_6) of AP_3 about its disconnection. Following this, AP_2 and AP_6 follow the protocol as outlined in steps (b) and (c), respectively.

The steps for the rest of the cases follow analogously. Another interesting aspect is the effect of peer disconnection on compensation. Compensation might lead to peer disconnection having an adverse affect even after the actual processing has completed. In fact, it might not be possible to guarantee atomicity as long as peer disconnection is possible. Here, we can use the notions of Spheres of Atomicity [17] to check if atomicity is guaranteed, e.g., atomicity may still be guaranteed for a transaction if all the involved peers (for that transaction) are super peers. The notion of peer independent compensation (discussed earlier) is also very helpful given the possibility of peer disconnection.

4. Conclusion

In this work, we proposed a transactional framework for AXML systems. AXML systems integrate XML, Web Services and P2P platforms, leading to some novel challenges which are not addressed by transactional models specific to any of the above. We considered the recovery aspect and proposed a compensation based recovery model for AXML systems. We showed in detail how compensation for AXML transactions can be constructed dynamically and introduced the notion of peer independent compensation. We also considered the issue

of peer disconnection and outlined a solution based on chaining the participant peers for a more efficient recovery.

Currently, the "chaining" mechanism is restricted to the parent, children and sibling peers. We are exploring the feasibility of extending the same to uncles, cousins, etc. Our future work also includes implementation and a formal study of the proposed protocols. The implementation part involves integrating the transactional framework into the AXML implementation [18]. The objectives of the formal study are two-fold. The first (and obvious) objective is to prove the correctness of the above protocols formally. The second objective is to analyze the interdependence between the protocols. Related research tends to focus on the A, C, I and D transactional properties independently with strong assumptions about each other. As a result, the interplay between the properties is ignored. Note that a property may have both a constraining as well as relaxing effect on the other. However, such relaxation needs to be performed in a controlled manner so that the overall consistency of the system is not affected (which leads to the need for a formal analysis).

Acknowledgements

This work is supported by the INRIA projects ARC-ASAX and RNRT-SWAN. We would like to thank Krishnamurthy Vidyasankar and Stefan Haar for their helpful suggestions which helped to improve the work in this paper considerably.

References

[1] Active XML (AXML) systems, http://www.activexml.net/.

[2] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu and T. Milo, "Dynamic XML Documents with Distribution and Replication", In proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 527 - 538.

[3] H. Garcia-Molina and K. Salem, "Sagas", In proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, pp. 249 - 259.

[4] D. Biswas, "Compensation in the World of Web Services Composition", In proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC), 2004, pp. 69 - 80.

[5] Kuen-Fang Jea, Shih-Ying Chen and Sheng-Hsien Wang, "Concurrency Control in XML Document Databases: XPath Locking Protocol", In proceedings of the 9th International Conference on Parallel and Distributed Systems (ICPADS), 2002, pp. 551 - 556.

[6] M. P. Haustein and T. Härder, "Adjustable Transaction Isolation in XML Database Management Systems", In proceedings of the 2nd International XML Database Symposium (XSym), 2004, pp. 173 - 188.

[7] R. Karlsen and T. Strandenæs, "Trigger-Based Compensation in Web Service Environments", In proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS), 2003, pp. 487 - 490.

[8] F. Tartanoglu, V. Issarny, A. Romanovsky and N. Levy, "Coordinated Forward Error Recovery for Composite Web Services", In proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS), 2003, pp. 167-176.

[9] Paulo F. Pires, Marta L.Q. Mattoso, and Mário Roberto F. Benevides, "Building Reliable Web Services Compositions", LNCS 2593, 2003, pp. 59 - 72.

[10] K. Vidyasankar and G. Vossen, "Multi-level Model for Web Service Composition", In proceedings of the 2nd International Conference on Web Services (ICWS), 2004, pp. 462 - 471.

[11] D. Biswas and K. Vidyasankar, "Spheres of Visibility", In proceedings of the 3rd IEEE European Conference on Web Services (ECOWS), 2005, pp. 2 - 13.

[12] Environment for the development and Distribution of Open Source software (EDOS), http://www.edos-project.org/.

[13] C. Turker, K. Haller, C. Schuler and H.-J. Schek, "How can we support Grid Transactions? Towards Peer-to-Peer Transaction Processing", In proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR), 2005, pp. 174 - 185.

[14] Henry F. Korth, E. Levy and A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions", In proceedings of the 16th International Conference on Very Large Databases (VLDB), 1990, pp. 95 - 106.

[15] G. Ghelli, C. Re and J. Simeon, "XQuery!: An XML query language with side effects", http://www.di.unipi.it/~ghelli/papers/XQueryBangTR.pdf.

[16] Specification: Business Process Execution Language for Web Services (BPEL4WS). http://www-106.ibm.com/developerworks/library/ws-bpel/.

[17] Gustavo Alonso, and Claus Hagen, "Exception Handling in Workflow Management Systems", IEEE Transactions on Software Engineering, Vol. 26, No. 10, 2000, pp. 943 - 958.

[18] Active XML (AXML) implementation, http://forge.objectweb.org/projects/activexml/.