An Atomic Web-Service Transaction Protocol for Mobile Environments

Stefan Böttcher[1], Le Gruenwald[2], and Sebastian Obermeier[1]

[1] University of Paderborn, Computer Science
Fürstenallee 11; 33102 Paderborn; Germany
{stb, so}@uni-paderborn.de
[2] The University of Oklahoma, School of Computer Science
200 Telgar street, Room 116 EL; Norman, OK 73019-3032; USA
{ggruenwald}@ou.edu

Abstract. Atomicity is a widely required property of Web service transactions that are executed in distributed networks. Compared to fixed-wired networks, atomicity in mobile networks is much more challenging due to network failures, e.g. network partitioning and node failures, which make global knowledge concerning the operational status of devices difficult or even impossible to achieve. In comparison to existing atomic commit protocols that abort or block transactions when network failures occur, our atomic commit protocol extension significantly reduces the number of aborts. In addition, the approach guarantees atomicity for transactions that dynamically invoke sub-transactions during the commit protocol's execution.

1 Introduction

The application of database technology to a network of mobile devices involves many new challenges, including guaranteeing transaction atomicity in the presence of lost connections and node failures. Atomic transaction commitment is necessary to deal with concurrent transactions in complex systems like distributed databases and peer-to-peer systems. In addition, even Web service oriented architectures have the need for an atomic transaction execution in the context of cascading web service calls.

In mobile networks, problems like message delay, disconnection of nodes, and network partitioning may occur, which are not appropriately solved by standard atomic commit protocols such as 2PC, in case the coordinator fails. Consequently, there are proposals to increase coordinator stability by using multiple coordinators (e.g. [1] or [2]), but these protocols abort transactions in case of missing abort/commit votes of participating nodes after a timeout. Since message delay or network partitioning is likely to occur in mobile networks which make reasonable timeouts hard to determine, the use of these protocols results in an unnecessarily high number of aborts. Other solutions, e.g. timeout-based approaches ([3]), demand committed transactions to be undone by applying compensation transactions. Since committed transactions can trigger other operations, we cannot assume that compensation for committed transactions in mobile networks, where network partitioning makes nodes unreachable but still operational, is always possible. Therefore, in this paper, we focus on a transaction

[2] This material is based upon work supported by (while serving at) the National Science Foundation (NSF) and the NSF Grant No. IIS-0312746.

model, within which atomicity is guaranteed for distributed, non-compensatable transactions.

In addition, modern Web service architectures demand a more dynamic transactional model than the classical distributed transactional model, for which 2PC ([4]) is designed. This means, atomicity must be guaranteed even for transactions that invoke other transactions or Web services dynamically at any time during transaction execution. Therefore, a previously unknown set of sub-transactions must be guided to an atomic commit decision.

This paper proposes a solution, which is a useful extension to many existing atomic commit protocols including 2PC, 3PC, and Paxos Commit Protocol ([5]), as it is neither based on a specific architecture nor on a specific atomic commit protocol. For simplicity of presentation, however, the scope of this paper focuses on how 2PC can be improved by our protocol extension, and why our extension results in protocols that reduce the number of aborts and guarantee an atomic execution of non-compensatable, dynamically invoked Web service transactions.

The rest of this paper is organized as follows. In Section 2, we describe the transactional model and introduce necessary requirements for guaranteeing atomic commit in a mobile environment. In Section 3, we propose a solution, which introduces a non-blocking state for transactions, and finally, we propose a tree data structure used to represent the execution status of all active sub-transactions.

2 Problem Description

In this section, we describe the transaction model for which our proposed solution is designed. Furthermore, we identify the requirements that our protocol must fulfill and outline the underlying assumptions.

2.1 Transaction Model

Our transaction model has the goal to support atomic execution of Web services in a mobile ad-hoc network. Our transaction model is based on the concepts "application", "transaction procedure", "Web service", and "sub-transaction", as well as their relationship to each other.

An *application* AP may consist of one or more *transaction procedures*. A transaction procedure is a Web service that must be executed in an atomic fashion. Transaction procedures and Web services are implemented using local code, database instructions, and (zero or more) calls to other remote Web services. Since the invocation of a Web service depends on conditions and parameters, different executions of the same Web service may call different Web services and execute different local code.

We call the execution of a transaction procedure a global transaction T. The application AP is only interested in the result of T, i.e. whether the execution of a global transaction T has been committed or aborted. In case of commit, AP is also interested in the return values of the parameters of T.

The relationship between transactions, Web services, and sub-transactions is recursively defined as follows: We allow each transaction or sub-transaction T to dynamically invoke additional Web services offered by physically different nodes. We call the execution of such Web services invoked by the transaction T or by a sub-transaction T_i the sub-transactions $T_{si} \dots T_{sj}$ of T or of T_i, respectively. This invocation hierarchy can be arbitrarily deep.

Whenever $T_1, \ldots, T_n$ denote all the sub-transactions called by either T or by any child or descendant sub-transaction T_s of T during the execution of the global transaction T, atomicity of T requires that either all transactions of the set $\{T, T_1, \ldots, T_n\}$ commit or all of these transactions abort.

We assume that each Web service only knows the Web services that it calls directly, but not whether or not the called Web services call other Web services. Therefore, at the end of its execution, each transaction T_i knows which sub-transactions $T_{is_1} \ldots T_{is_j}$ it has called, but Ti, in general, will not know which sub-transactions have been called by $T_{is_1} \ldots T_{is_j}$. Furthermore, we assume that usually a transaction Ti does not know how long its sub-transactions $T_{is_1} \ldots T_{is_j}$ are going to run.

We assume that each sub-transaction consists of the following phases: a read-phase, a coordinated commit decision phase, and, in case of successful commit, a write-phase. During the read-phase, each sub-transaction performs write operations on its private storage only. After commit, during the write phase, write operations on the private storage are transferred to the database, such that the changes done throughout the read-phase become visible to other transactions after completion of the write-phase.

In the mobile architecture for which our protocol is designed, Web services are invoked by messages instead of invoking them by a synchronous call to a Web service for the following reason. We want to avoid that a Web service T_i that synchronously calls a sub-transaction T_j cannot complete its read phase and cannot vote for commit before T_j sends its return value. Therefore, we allow sub-transactions only to return values indirectly by asynchronously invoking corresponding receiving Web services, and not synchronously by return statements[1]. Since (sub-)transactions describe general services, the nodes that execute these (sub-) transactions may be arbitrary nodes and are not necessarily databases. We therefore call these nodes *resource managers (RM)*. Here we describe an example application that demonstrates the necessity of atomic execution of transactions

Example 1. An application needs to book a travel package containing a flight, a hotel, and a transfer from the destination airport to the hotel. As shown in Figure 1, the corresponding global transaction T, started by the Initiator I, invokes a travel agency Web service. The corresponding sub-transaction T_1 first chooses among the available flights and hotels and then starts the Web services T_2 to book the chosen flight and T_3 to book the chosen hotel. To organize the transportation for the traveler from the airport to the hotel, T_3 needs the Web service of a bus company to start a sub-transaction for organizing this transportation. It is obvious that no booking component is allowed to fail: if the hotel is no longer available or the flight cannot be booked, the passenger cannot be transported from the airport to the hotel upon arrival of the desired flight. Therefore, all sub-transactions are required to be performed in an atomic fashion.

[1] However, if the application needs synchronous calls, e.g. because of dependencies between sub-transactions, the intended behavior can be implemented by splitting T_i into T_{i_1} and T_{i_2} as follows. T_{i_1} includes T_i's code up to and including an asynchronous invocation of its sub-transaction T_j; and T_{i_2} contains the remaining code of T_i. T_j performs an asynchronous call to T_{i_2} which may contain return values computed by T_j that shall be further processed by T_{i_2}

One characteristic of our Web service transactional model is that the initiator and the Web services do not know every sub-transaction that is generated during transaction processing. Our model differs from other models that use nested transactions (e.g. [6], [7], [8]) in some aspects including but not limited to the following:

- Since network partitioning makes it difficult or even impossible to compensate all sub-transactions, we consider each sub-transaction running on an individual resource manager to be non-compensatable. Therefore, no sub-transaction is allowed to commit independently of the others or before the commit coordinator guarantees that all sub-transactions can be committed.
- Different from CORBA OTS ([7], [9]), we assume that we cannot identify a hierarchy of commit decisions, where aborted sub-transactions can be compensated by executing other sub-transactions.
- Different from the Web service transaction model described in [8], the Initiator of a transaction in our model does not need to know all the transaction's sub-transactions. We assume that the Initiator is only interested in the commit status and the result of the transaction, but not in knowing all the sub-transactions that have contributed to the result.
- A Web service may consist of control structures, e.g. if `<Condition> then <T1> else <T2>`.This means that sub-transaction executing this Web service may create other sub-transactions dynamically. These dynamically created sub-transactions also belong to the global transaction and must be executed in an atomic fashion.
- Communication is message-oriented, i.e., a Web service does not explicitly return a result, but may invoke a receiving Web service that performs further operations based on the result.

2.2 Requirements

The main requirement is to design an atomic commit protocol for guaranteeing the atomic execution of non-compensatable Web service transactions including sub-transactions in mobile networks. In comparison to protocols designed for fixed-wired networks, we can identify the following additional requirements for protocols for mobile networks:

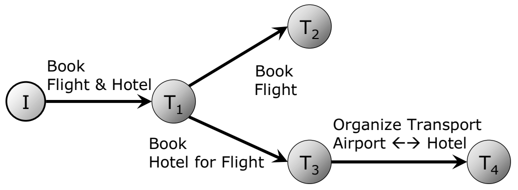

Fig. 1. The initiator I of a Web service transaction T and its sub-transactions $T_1, \dots T_4$

- Resource managers may fail or disappear at any time. This, however, must not have a blocking effect on other resource managers.
- If the vote of a resource manager is lost or delayed, traditional protocols like 2PC either wait for the missing vote and block all participating resource managers, or abort the transaction which thereafter can be repeated as a whole. However, since a lost vote differs from an explicit vote for abort, such a general abort may not be necessary for many sub-transactions, especially if there is no other concurrent transaction that tries to get a lock on the same data that the sub-transactions are accessing.

- Our contribution should be an extension to existing atomic commit protocols, such that a concrete protocol can be chosen depending on the applications' needs.
- Our protocol extension should reduce the number of aborts that occur due to timeout or loss of votes.
- It should be possible that the user can abort a transaction as long as the transaction has not been globally committed.

2.3 Underlying Assumptions

Our atomic commit protocol extension is based on the following assumptions:

- Atomicity requires the following in case of resource manager failures. Whenever a resource manager RM_i is unreachable after a commit decision on a transaction T was reached, i.e. RM_i has failed or is separated from the network for an indefinitely long time, we do not consider this as a violation of the atomicity constraint, no matter whether or not RM_i has executed its sub-transaction T_i of T. However, if the resource manager RM_i recovers and returns to the network, RM_i is allowed to participate further only after having executed or aborted T_i, depending on whether the commit decision on T was commit or abort.
- Increasing the stability of the coordinator process itself is not a topic of this proposal. There are many proposals for handling coordinator failures, e.g. to run special termination protocols or to increase coordinator availability by using more than one coordinator (e.g [1], [5], or [2]). Since our proposal is designed as an extension to one of these protocols, the user can choose a protocol depending on the desired coordinator stability. Therefore, we do not discuss coordinator failures further and assume that the coordinator process does not fail.
- At least some (sub-)transactions are non-compensatable. We claim that this assumption is realistic for mobile environments. Even in our simple example given above (Example 1), some sub-transactions (e.g. T_2 and T_3) can be considered to be non-compensatable since many of today's airlines and some hotels demand expensive cancellation and rebooking fees. We cannot tolerate a commit protocol which must repeatedly change or cancel flights if there is no hotel available for the initially booked transportation.
- A sub-transaction T_i that invokes a sub-transaction T_{s_i} does not need to invoke another sub-transaction after T_{s_i} has aborted. Instead, if any sub-transaction T_{s_j} aborts, the global transaction T to which T_{s_j} belongs must be aborted.

3 Solution

This section describes the solutions to our main requirements, i.e., how to guarantee atomicity for Web service transactions and to reduce the number of transaction aborts compared to standard protocols like 2PC or 3PC in case of message loss. To reduce the number of aborts, we introduce a new non-blocking suspend state in Section 3.1. Then, Section 3.3 proposes a data structure called "commit tree", which represents the commit status of all sub-transactions involved in a global transaction, and which helps to guarantee atomicity with both reduced blocking times and reduced aborts.

3.1 The Non-Blocking Suspend State

While waiting for the transaction's commit decision, protocols like 2PC or 3PC remain in a wait state as long as votes are missing, and they block the resource managers that have voted for commit ([10]). In order to reduce this blocking, our protocol extension suggests an additional suspend state as follows:

For each sub-transaction T_i, let $RS(T_i)$ denote the data read by T_i and $WS(T_i)$ denote the data written by T_i. Then, we define the suspend state for the sub-transaction T_i in the following way:

Definition 1. *The suspend state of T_i is a state in which the resource manager RM executing T_i waits for a decision from the commit coordinator on T_i, but does not block the tuples $WS(T_i) \cup RS(T_i)$.*

If another transaction T_k is executed while T_i is suspended, RM checks whether

$$WS(T_i) \cap (RS(T_k) \cup WS(T_k)) \neq \varnothing \quad \vee \quad RS(T_i) \cap WS(T_k) \neq \varnothing$$

If this is the case, there is a conflict between T_i and T_k, and therefore, RM locally aborts T_i and can either abort the global transaction T or try a repeated execution of the sub-transaction T_i.

3.2 Using the Suspend State to Reduce the Number of Aborts

Figure 2 shows an automaton containing the resource managers' states. Each resource manager executes its read-phase and sends a vote for commit or for abort to the coordinator like in 2PC. In case of having sent a vote for commit, the resource manager enters the wait state and waits for the coordinator's commit decision. In contrast to 2PC, the coordinator has, besides aborting or committing the transaction, a third possibility, i.e., to suspend the transaction.

The suspend state is proposed by the coordinator if after a timeout some votes are still missing. In this situation, traditional 2PC would abort the transaction because participating resource managers have locks on read or written tuples, and waiting for the missing votes implies blocking these tuples, thereby preventing concurrent transactions that access conflicting data from completion. In fixed-wired environments, 2PC's transaction abort makes sense because a resource manager whose vote is missing has most likely failed. In mobile environments, however, the vote message can get lost more easily or the devices may be disconnected for a short time due to the mobile character of the network. In this case, an abort of the whole transaction does not make sense. Instead, the suspend state can be used to unblock all participants until one of the used resources is needed by a

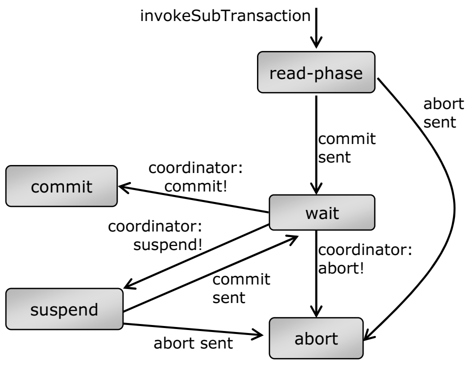

Fig. 2. Automaton showing the states and the received messages of a resource manager

concurrent transaction or all votes of the sub-transactions belonging to the same global transaction have arrived at the coordinator.

A resource manager waits in the non-blocking suspend state until either one of the following events occurs:

- the coordinator demands again a vote on the sub-transaction or
- the coordinator aborts the sub-transaction or
- a concurrent transaction causes an abort of the sub-transaction due to access conflicts on the tuples accessed by the sub-transaction.

In case that the user or the application program wants to abort the transaction, the initiator sends an abort message to the coordinator, which is allowed to abort the transaction anytime before the commit decision has been reached. Our protocol definition implies that each resource manager is able to give its vote on the transaction. If, however, not all resource managers respond immediately, the coordinator may advise the resource managers to go into the suspend state while waiting for the missing votes.

2PC, in contrast, blocks a resource manager from the time when it finishes its read-phase, i.e., when it is ready to vote for commit, until the time when it receives the commit decision.

3.3 Atomicity of Web service Transactions

This section explains how atomicity can be guaranteed for the Web service transactions that we have introduced in Section 2.

The main problem of ensuring atomicity for Web service transactions is that the coordinator does not know all sub-transactions. In order to inform the coordinator about all the invoked sub-transactions, we outline the sub-transaction's ID management in Section 3.4 and propose a data structure called commit tree to dynamically store the completion status of each sub-transaction involved in a transaction in Section 3.5.

3.4 The Sub-Transactions´ ID Management

Since a global transaction T may consist of many invoked sub-transactions, we propose the use of sub-transaction IDs to distinguish all sub-transactions of T. To ensure that the coordinator gets knowledge of all invoked sub-transactions belonging to T, we require that each vote message sent by a sub-transaction T_i to the coordinator informs the coordinator about all sub-transactions $T_{s_1}, \ldots, T_{s_k}$ that are called by T_i. This means that each sub-transaction T_i includes a list of IDs of all its invoked sub-transactions $T_{i_1}, \ldots, T_{i_k}$ in its vote message. For this purpose, we have included a parameter `ListOfInvokedSubTransactions` in the vote message which has the following format:

```
VoteMessage V(bool commit, ID subtransactionID, ID callerID, ID globalTID,
              ListOf(ID) ListOfInvokedSubTransactions, int sequenceNr)
```

Since the atomic decision of the global transaction T must include those invoked sub-transactions, the coordinator must also wait for the votes of those newly added resource managers. This behavior is supported by the commit tree data structure, for which an example is given in the next section and which is generally defined in Section 3.6.

The other parameters have the following meaning: `commit` tells the coordinator whether or not the sub-transaction execution was successful and contains the value of either abort or commit. `subtransactionID` is the sub-transaction's own ID whereas `callerID` is the ID of the parent transaction and `globalTID` is the ID of the global transaction T to which the sub-transaction belongs. Finally, `sequenceNr` is needed to identify the latest version of the vote message in order to handle message delays if the vote message is sent more than once.

Furthermore, each sub-transaction T_{s_i} belongs to exactly one global transaction T, and it is called by exactly one caller T_s, i.e., the application program or another sub-transaction. Since each sub-transaction T_s must inform the coordinator about the sub-transaction IDs of all its sub-transactions $T_{s_1}, \ldots, T_{s_k}$, we have decided to provide a sub-transaction T_{s_i} with all the information when it is called. Therefore, the sub-transaction ID of T_{s_i} is generated by the resource manager running the calling parent transaction T_s and is passed in a parameter `subtransactionID` to T_{s_i} when T_{s_i} is invoked. The following example shows the use of IDs:

```
invokeSubTransaction( subtransactionID, callerID, globalTID,
       <WebService name and parameters>  )
```

The parameter `callerID` contains the ID of T_{s_i} and the parameter `globalTID` contains the ID of the global transaction T to which T_{s_i} and T_s belong. The parameters list `<WebService name and parameters>` contains the name of the called Web service and the parameters used for the Web service call.

3.5 An Example of the Coordinator´s Commit Tree

To ensure that all sub-transactions $T_{s_i}, \ldots, T_{s_j}$ invoked by a sub-transaction T_i are known to the coordinator, the coordinator must process the parameter `ListOfInvokedSubTransactions`, passed in the vote message of T_i and update the set of required votes for the global transaction T before processing T's commit decision. Before we describe the general use of the commit tree data structure, we give an example (c.f. Figure 3) that shows how we ensure that the coordinator can only come to a commit decision after it has knowledge of all necessary votes.

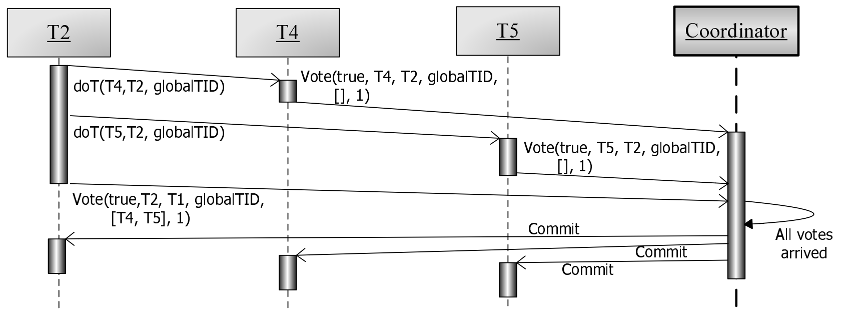
T2
T4
T5
Coordinator
doT(T4,T2, globalTID)
Vote(true, T4, T2, globalTID, [], 1)
doT(T5,T2, globalTID)
Vote(true, T5, T2, globalTID, [], 1)
Vote(true,T2, T1, globalTID, [T4, T5], 1)
Commit
Commit
Commit
All votes arrived

Fig. 3. Sequence diagram of example commit tree construction

Figure 3 shows part of a sequence diagram of an example execution: During the read-phase, a sub-transaction T2 needs the service doT(...) and generates the ID "T4" to invoke the Web service with the required parameters `subtransactionID` (T4), `CallerID` (T2) and `globalTID`. In our example, sub-transaction T4 has finished earlier than T2 and sends the vote message to the coordinator. Since T4 has not invoked any sub-transactions, the fifth parameter denoting the set of the called sub-transactions is empty. T2, however, includes the IDs of its invoked sub-transactions, T4 and T5, in its set of the called sub-transactions. This makes the coordinator to require the votes of T4 and T5, which, in this example, have arrived earlier. When the coordinator has received all these votes, it decides on the global commit decision.

In contrast to 2PC, which handles flat transactions instead of nested transactions, the dynamic call of Web services supported by our protocol requires that the votes, which the coordinator needs for a global commit decision, can be determined only during the protocol's execution. In order to store the commit status and vote of each sub-transaction, we propose a dynamic data structure, called commit tree, and we define the initiator of a transaction to be the root of this commit tree.

Figure 4 shows an example commit tree (we have omitted the globalTIDs since all sub-transactions belong to the same global transaction). When the coordinator has received the initiator I's vote which includes the list [T1] for the parameter of the invoked sub-transactions, the root node is generated to represent the commit status of I. When the coordinator has received the vote of T1, the node T1 is created. Since the sub-transaction T1 invoked the sub-transactions T2 and T3, the commit of the sub-transactions T2 and T3 is also required to commit the whole transaction. Therefore, this information is added to the commit tree. The coordinator builds this commit tree dynamically and determines whether all votes needed for continuing the protocol's execution have arrived. Since the information about invoked sub-transactions is sent along with a vote message of the parent transaction and the parent's vote can arrive later than a sub-transaction's vote, it may be the case that an arriving sub-transaction vote cannot be assigned to a parent transaction, like the vote messages of T4 and T5 in the example of Figure 3. In this case, the sub-transaction's vote is stored and assigned after the corresponding parent sub-transaction's vote has arrived.

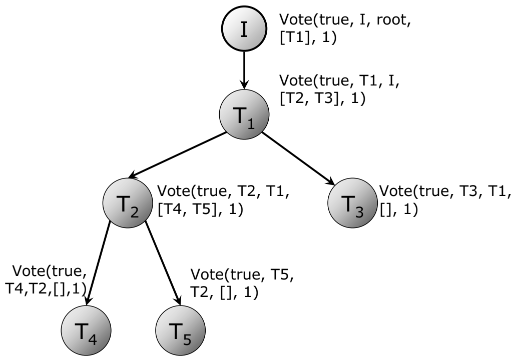
I
Vote(true, I, root, [T1], 1)
T_1
Vote(true, T1, I, [T2, T3], 1)
T_2
Vote(true, T2, T1, [T4, T5], 1)
T_3
Vote(true, T3, T1, [], 1)
Vote(true, T4,T2,[],1)
Vote(true, T5, T2, [], 1)
T_4
T_5

Fig. 4. An example commit tree

3.6 The Coordinator´s Commit Tree

The commit tree for a transaction is an unordered tree with additional parameters to dynamically store votes. Each commit tree corresponds to exactly one global transaction and stores the following variables: the global transaction ID; a tree structure containing commit tree nodes; a list `unassignedN` of unassigned nodes corresponding to sub-transactions that voted for commit before their parents voted for commit; and a list openSubTransactions of transaction IDs, for which the vote was not received by the coordinator. Furthermore, each commit tree *node* stores: the sub-transaction ID; the ID of the resource manager running the sub-transaction; the parent transactionID; and $0 - n$ IDs of invoked sub-transactions.

Depending on the status of the commit tree and based on the timer, the coordinator sends the following messages to the participating resource managers:

doCommit requires the recipient to commit the sub-transaction. the message is sent, when all participants have voted for commit, i.e. the list openSub-Transactions is empty.

doAbort requires the recipient to commit the sub-transaction. This message is sent, when at least one participant has voted for abort.

doSuspend requires the recipient to proceed to the suspend state. This message is sent, when some votes are still missing after a timeout.

Modification of the Commit Tree by the Vote Operation

Algorithm 1 below outlines the implementation of the coordinator's vote operation which is executed on the commit tree whenever a client's vote message arrives at the coordinator. First, the coordinator uses the sequence number to check that no newer message was processed earlier, e.g., due to message delay (line 3). Thereafter, a new node N is created and the parent-child relationships between N and the nodes representing other sub-transactions are managed (lines 5-10). In addition, the list `openSubTransactions` is updated where sub-transactions, the votes of which are necessary but have not yet arrived, are stored.

```
(1)   preVote(boolean commitStatus, ID subtrsID, ID callerID,
(2)            ID globalTID, ListOf(ID) invokedSubT, int sequenceNr) {
(3)   if (isVoteValid(sequenceNr)) {
(4)      if (commitStatus==false) abortTransaction();
(5)      N := createNode(subtrsID, callerID, globalTID, invokedSubT);
(6)      openSubTransactions.delete(subtrsID);
(7)      if( (ParentNode := getNodeByID(callerID)) == nil)
(8)           unassignedN.add(N)
(9)      else{  ParentNode.addChild(N)
(10)             assignNodes(invokedSubT, N)
(11)         }
(12)     openSubTransactions.add(invokedSubT)  } }
```

Algorithm 1: Coordinator's implementation of the preVote procedure

If all votes are present, the list `openSubTransactions` is empty and the global decision can be made. To ensure that in case of a resource manager's failure no infinite blocking occurs, the coordinator starts the timer. If the time is over and some votes are still missing, the coordinator sets the status stored in each node of the commit tree back to suspend, proposes the suspend state to T and to all sub-transactions belonging to T, and demands the votes for the transaction once more. Demanding for votes is repeated until a maximum number defined by the application is exceeded.

4 Related Work

We can distinguish contributions to the field of atomicity and distributed transactions according two main criteria: first, whether their transactional models use flat transactions or support nested transaction calls, second, whether transactions and sub-transactions are regarded as compensatable or non-compensatable. Our contribution is based on a transactional model that allows nested transaction calls and assumes sub-transactions to be non-compensatable.

The requirement to allow sub-transactions to invoke other sub-transactions originated from business applications. Within such a business application, the atomicity constraint is to complete all "sub-transactions" of a *workflow* ([11]).

Today, Web services and their description languages (e.g. BPEL4WS [12] or BPML [13]) are more and more used to implement nested Web service transactions, which are called *Web services orchestration*.

However, these languages do not provide a coordination framework to implement atomic commit protocols. For this purpose, our contribution can be combined with these description languages, like the "WS-Atomic-Transaction" proposal ([8]) does. Note that our contribution is different from [8] in several aspects. For example, [8] has a "completion protocol" for registering at the coordinator, but does not propose a non-blocking state – like our suspend state – to unblock transaction participants while waiting for other participants' votes. In comparison, our suspend-state may even be entered repeatedly during the protocol's execution.

Besides the Web service orchestration model, there are other contributions that set up transactional models to allow the invocation of sub-transactions, e.g. the Kangaraoo Model [6]. However, this model can not give global atomicity guarantees, as defined in [14], if compensation for all sub-transactions is not possible.

In Corba OTS ([7], [9]), the term "suspend" is also used for describing a state. However, the Corba "suspend" differs from our model because our suspend state is a non-blocking state for a mobile environment. Regarding the transactional model, Corba OTS uses a hierarchy of commit decisions, where an abort of a sub-transaction does not necessarily lead to an abort of the global transaction. Instead, the calling sub-transactions can react on this abort and use other sub-transactions, for example. Although we also assume that Web services invoke other Web services and the coordinator uses a tree structure to maintain information about commit votes, we do not have this hierarchical commit decisions, since in the presence of non-compensability, this implies waiting for the commit decision of all descendant nodes.

The idea of using a suspend state is also proposed by [15]. However, this approach is intended for the use within an environment with a fixed network and several mobile cells, where disconnections are detectable and therefore transactions can be compensated. In contrast, our assumed environment, which allows ad-hoc communication, demands a more complex failure model that takes network partitioning into consideration.

While our solution in combination with 2PC reduces blocking of resource managers during the commit protocol's execution, there is one case left where infinite blocking can occur: in case of network partitioning and coordinator failure during the vote phase. [16] has proven that this blocking cannot be avoided in asynchronous networks if we are not able to distinguish whether a node has failed or is still working in another network partition. However, our proposal reduces the chance that this blocking occurs by reducing the duration of the vote phase, in which the resource managers are blocked. To enhance the coordinator's availability, other contributions propose the use of protocols with more than one coordinator ([1],[2],[5]). Nevertheless, since all these protocols require a resource manager to send a vote on the sub-transaction, the protocols can be extended to support the suspend state by adding a separate vote phase to the protocols.

5 Summary and Conclusion

In this paper, we have presented two key ideas for guaranteeing atomicity for Web service transactions in a mobile context that also reduce the number of transaction aborts. The first idea is to use the suspend state for a transaction when, after a timeout, the coordinator still waits for the votes of some of its sub-transactions. In the suspend state, the sub-transaction still can be aborted by the resource manager when the resource manager decides to grant the resources used by this sub-transaction to other concurrent transactions to prevent them from blocking. This is especially useful in mobile environments where devices are more likely to fail and to disconnect. If votes of some devices are lost, waiting in the suspend state involves no risk of long-term blocking the resources accessed by the transaction. Second, we have presented the commit tree as a data structure that can be used to implement the coordinator's management of transaction atomicity for a dynamically changing set of sub-transactions. We have embedded our atomic commit protocol in a Web service transactional model, the characteristics of which is that sub-transactions must not be known in advance. We have furthermore presented all key solutions as an extension to the 2PC protocol, however, our contribution is applicable to a much broader set of commit protocols. Finally, our protocol extension merges nicely with a variety of concurrency control strategies including validation and locking. Although a serializability proof for arbitrary schedules of concurrent transactions running under such an integrated protocol is beyond the scope of this paper, we have evidence that serializability of concurrent transactions can be guaranteed, and we plan to report on this on a forthcoming paper.

References

1. Reddy, P.K., Kitsuregawa, M.: Reducing the blocking in two-phase commit with backup sites. Inf. Process. Lett. **86** (2003) 39–47
2. Böse, J.H., Böttcher, S., Gruenwald, L., Obermeier, S., Schweppe, H., Steenweg, T.: An integrated commit protocol for mobile network databases. In: 9th International Database Engineering & Application Symposium IDEAS, Montreal, Canada (2005)
3. Kumar, V., Prabhu, N., Dunham, M.H., et al.: Tcot-a timeout-based mobile transaction commitment protocol. IEEE Trans. Comput. **51** (2002) 1212–1218
4. Gray, J.: Notes on data base operating systems. In Flynn, M.J., Gray, J., Jones, A.K., et al., eds.: Advanced Course: Operating Systems. Volume 60 of Lecture Notes in Computer Science., Springer (1978) 393–481
5. Gray, J., Lamport, L.: Consensus on transaction commit. Microsoft Research – Technical Report 2003 (MSR-TR-2003-96) **cs.DC/0408036** (2004)
6. Dunham, M.H., Helal, A., Balakrishnan, S.: A mobile transaction model that captures both the data and movement behavior. Mobile Networks and Applications **2** (1997) 149–162
7. OMG: Transaction Service Specification 1.4. http://www.omg.org/ (2003)
8. Cabrera, L.F., Copeland, G., Feingold, M., et al.: Web Services Transactions specifications – Web Services Atomic Transaction. http://www-128.ibm.com/developerworks/library/specification/ws-tx/ (2005)
9. Liebig, C., Kühne, A.: Open Source Implementation of the CORBA Object Transaction Service. http://xots.sourceforge.net/ (2005)
10. Skeen, D.: Nonblocking commit protocols. In Lien, Y.E., ed.: Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, ACM Press (1981) 133–142
11. Workflow Management Coalition. (http://www.wfmc.org/)
12. Curbera, F., Goland, Y., Klein, J., Leymann, F., et al.: Business Process Execution Language for Web Services, V1.0. Technical report, BEA, IBM, Microsoft (2002)
13. Arkin, A., et al.: Business process modeling language, bpmi.org. (Technical report)
14. Kifer, M., Bernstein, A., Lewis, P.M.: Database Systems: An Application Oriented Approach. Pearson Addison-Wesley (2005)
15. Dirckze, R.A., Gruenwald, L.: A pre-serialization transaction management technique for mobile multidatabases. Mobile Networks and Applications **5** (2000) 311–321
16. Ancilotti, P., Lazzerini, B., Prete, C.A., Sacchi, M.: A distributed commit protocol for a multicomputer system. IEEE Trans. Comput. **39** (1990) 718–724