Reducing Sub-transaction Aborts and Blocking Time Within Atomic Commit Protocols

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. Composed Web service transactions executed in distributed networks often require an atomic execution. Guaranteeing atomicity in mobile networks involves a lot more challenges than in fixed-wired networks. These challenges mostly concern network failures, e.g. network partitioning and node disconnection, each of which involves the risk of infinite blocking and can lead to a high number of aborts.

In this paper, we introduce an extension to existing atomic commit protocols, which decreases the time during which a resource manager that is involved in a web-service is blocked. In addition, our proposal reduces the number of sub-transaction aborts that arise due to message loss or due to conflicting concurrent transactions by distinguishing reusable and repeatable sub-transactions from aborting sub-transactions.

1 Introduction

The use of Web service transactions within fixed wired network structures is supported by multiple specification languages, e.g. BPEL4WS [1] or BPML [2]. Especially when Web service transactions are composed of several sub-transactions, it is often crucial that either all or none of the sub-transactions are executed. Atomic commit protocols are a standard technique to meet this requirement, i.e. for guaranteeing an atomic execution of nested transactions. However, in the context of Web services, sub-transactions may be dynamically invoked and therefore are not always known in advance. Approaches like the "WS-AtomicTransaction" standard ([3]) suggest using a modified 2-Phase-Commit-protocol (2PC, [4]), where each dynamically invoked sub-transaction registers at the coordinator by itself.

However, if parts of a Web service transaction should be processed within a mobile ad-hoc environment where mobile participants are suspect to disconnects and network partitioning may occur, the use of 2PC may lead to a long blocking

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

D. Bell and J. Hong (Eds.): BNCOD 2006, LNCS 4042, pp. 59–72, 2006.

time of mobile participants. Even if we assume the coordinator to be stable, a database that disconnects while executing a transaction blocks the data that the transaction has accessed until it gets the commit decision.[1]

In addition, a high number of disconnects and reconnects may lead to an unnecessarily high number of aborts since timeouts for reconnections are hard to estimate. Imagine a coordinator that waits for the last missing vote for commit. If the coordinator waits too long, concurrent transactions that access conflicting data cannot be processed. If the coordinator does not wait and aborts the transaction due to the missing vote, the database may reconnect just at this moment and the transaction was superfluously executed and aborted. Since estimations about message delivery times and transaction execution times within dynamic mobile ad-hoc networks are often significantly different to the observed times, the coordinator will very likely estimate wrong timeouts.

Solutions like timeout-based approaches ([5]) optimistically suggest to commit a transaction and apply compensation transactions to undo in the case of abort. However, 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, we focus on a transaction model, within which atomicity is guaranteed for distributed, non-compensatable transactions.

In this paper, we show how the suspend state can be used within 2PC to not only reduce the blocking times of databases, but also to reduce the number of aborts and reuse existing results as far as possible. Section 2 describes details of our assumed transaction model and introduces necessary requirements for guaranteeing atomic commit in a mobile environment. In Section 3, we propose a solution, which consists of the following key ideas: a non-blocking state for transactions that are ready to vote for commit; a flexible reaction to concurrency failures by distinguishing failures that require a transaction abort, failures that only require the repetition of a sub-transaction, and failures that allow the reuse of a sub-transaction; and finally, we show how a tree data structure that represents the execution status of all active sub-transactions can be used by the coordinator to efficiently coordinate the transaction.

2 Problem Description

This section describes our assumed transaction model and identifies the additional requirements that arise when using mobile devices within a transaction. Finally, we describe the goal, i.e., to reduce both blocking and the number of transaction aborts.

2.1 Transaction Model

Our transaction model is based on the Web Services Transactions specifications. However, since we focus on the atomicity property, we can rely on a much sim-

[1] Even validation-based synchronisation shows a blocking behavior, cf. Section 2.4.

pler transaction model, e.g., we do not need a certain Web service modeling or composition language like BPEL4WS [1] or BPML [2]. Therefore, we have designed our transaction model to consist only of the following objects, that are "application", "transaction procedure", "Web service", and "sub-transaction". In the following, we explain how we understand these terms as well as their relationship to each other.

An *application* AP can 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 a Web service invocation can depend on conditions and parameters, different executions of the same Web service may result in different local executions and different calls to other Web services.

An *execution* of a transaction procedure is called a global transaction T. We assume that an application AP is 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 may be further interested in the return values of T's parameters.

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 $T_{si} \ldots T_{sj}$ of T or of T_i, respectively.

Whenever during the execution of the global transaction T, a child or descendant sub-transaction T_s of T, or T itself, invokes the sub-transactions $T_1 \ldots T_n$, the atomicity property of T requires that either all transactions $T, T_1 \ldots T_n$ commit, or all of these transactions abort.

Since we allow dynamic Web service invocations, we assume that each Web service only knows which Web services it calls directly, but not which Web services are invoked by the called Web services. This means that at the end of its execution, each transaction T_i knows which sub-transactions $T_{is_1} \ldots T_{is_j}$ it has called directly, but T_i, in general, will not know which sub-transactions have been called by $T_{is_1} \ldots T_{is_j}$. Moreover, we assume that usually a transaction T_i does not know how long its invoked sub-transactions $T_{is_1} \ldots T_{is_j}$ will run.

We assume that during the execution of a sub-transaction, a database enters the following phases: the *read-phase*, the *coordinated commit decision phase*, and, in case of successful commit, the *write-phase*. While executing the read-phase, each sub-transaction carries out write operations only on its private transaction storage. Whenever the coordinator decides to commit a transaction, each database enters the write phase. During this phase, the private transaction storage is transferred to the durable database storage, 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 must be invoked by messages instead of synchronous calls 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. For this reason, we allow invoked sub-transactions only to return values indirectly by asynchronously invoking corresponding receiving Web services and not synchronously by return statements.[2]

Since (sub-)transactions describe general services that do not exclusively concern databases, we also call the node executing a sub-transaction *resource manager (RM)*.

The following example shows the necessity of an atomic execution of transactions within our transactional model:

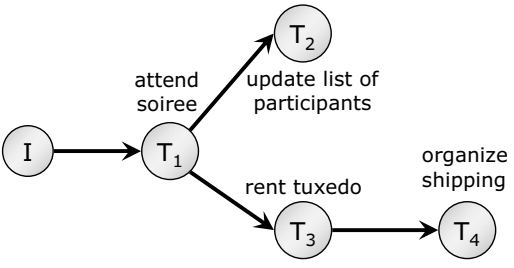

Fig. 1. The initiator I of a Web service transaction T and its sub-transactions $T_1 \ldots T_4$

Example 1. Assume a conference participant decides to attend the conference's soiree and invokes the corresponding Web Service. As shown in Figure 1, the corresponding global transaction T, started by the Initiator I, invokes the Web Service for attending the soiree. The corresponding sub-transaction T_1 then updates the participant list by invoking sub-transaction T_2 and detects that the participant has not brought a tuxedo. Therefore, a tuxedo must be rented and T_1 calls a sub-transaction T_3. To organize the shipping of the tux to the participant's hotel, T_3 calls the Web service T_4 of a shipping company. It is obvious that no booking component is allowed to fail: if the participant list is already full, if there are no tuxedos available, or if the shipping company cannot deliver the tuxedo to the participant on time, the participant cannot attend the soiree. Therefore, all sub-transactions are required to be performed in an atomic fashion.

This example shows one characteristic of our Web service transactional model: 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], [3]) in some aspects including, but not limited, to the following: Since compensation of a sub-transaction may not be possible if a mobile network is partitioned, we consider each sub-transaction to be non-compensatable. Therefore, no sub-transaction is allowed to commit independently of the others or before the commit coordinator guarantees – by sending the commit message – that all sub-transactions will be committed.

[2] However, if a synchronous call within T_i to T_j is needed, e.g. because T_i needs return values from T_j, it is possible to achieve this behavior by splitting T_i into T_{i1} and T_{i2} as follows: T_{i1} includes T_i's code up to and including an asynchronous invocation of its sub-transaction T_j; and T_{i2} contains the remaining code of T_i. After T_j has completed its read phase, T_j performs an asynchronous call to T_{i2} which can contain return values computed by T_j that shall be further processed by T_{i2}.

Different from CORBA OTS ([7]), we assume that we cannot always identify a hierarchy of commit decisions, where aborted sub-transactions can be compensated by executing other sub-transactions.

A Web service T may be programmed by using control structures, e.g. `if <Condition> then <T1> else <T2>`. This means that resource managers executing a Web service T may dynamically invoke other sub-transactions $T_1 \ldots T_j$.

We assume a message-orientated communication. This means that a Web service does not explicitly return values but may pass parameters to other invoked Web services which perform further operations based on these results.

2.2 Requirements

Besides the main requirement to design an atomic commit protocol for guaranteeing the atomic execution of non-compensatable Web service transactions including sub-transactions in mobile networks, we identified the following additional requirements especially for mobile network protocols.

A resource manager failure or disappearance may occur at any time. This, however, must not have blocking effects on other resource managers.

A general problem when guaranteeing atomicity for transactions that are non-compensatable is that participating resource managers are blocked during protocol execution. The time that a distributed sub-transaction remains in a state within which it waits for a commit decision can be much longer in dynamic mobile environments than it is in fixed-wired networks, since link failures and node failures occur significantly more frequently. Furthermore, each sub-transaction T_{si} of a transaction T running on a resource manager RM that resides in a blocking state also involves the risk of the infinite blocking of other sub-transactions of T and sub-transactions of other transactions running on RM. Therefore, we need efficient mechanisms to reduce the time that a resource manager is in a blocking state and to unblock sub-transactions if other resource managers do not respond.

Within mobile networks, message delay or message loss is no exception. In case that the vote message of a resource manager is lost or delayed, traditional protocols like 2PC either wait until the missing vote arrives and block all participating resource managers in the meantime, or they abort the transaction which thereafter can be repeated as a whole. Nevertheless, a lost vote differs from an explicit vote for abort. On one hand, 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. On the other hand, if there are concurrent transactions that try to access data in a conflicting way, an abort is necessary for processing these concurrent transactions. Therefore, a requirement is to abort as few transactions as necessary.

There are situations, especially when network partitioning occurs, where blocking is proven to be unavoidable until the network is reconnected again ([8]). Of course, our solution cannot avoid this blocking, but it should reduce the chance that resource managers are blocked by minimizing the time period during which a failure could have a blocking effect on the resource managers.

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 use previous results to the greatest possible extent, so that an unnecessary repetition of a sub-transaction can be avoided in as many cases as possible.

It should be possible that the user can abort a transaction as long as the transaction has not been globally committed.

2.3 Further Assumptions

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

In case of resource manager failure, the atomicity property requires the following: Whenever a resource manager RM_i is unreachable after the 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, it is not a violation of the atomicity constraint if RM_i has not executed its sub-transaction T_i of T. However, if the resource manager RM_i recovers and returns to the network, RM_i must immediately execute or abort T_i, depending on whether the commit decision on T was commit or abort, before further participating in the network.

We assume that at least some (sub-)transactions are non-compensatable. We claim that this is a realistic assumption for mobile environments. Even in our simple example given above (Example 1), some sub-transactions (e.g. T_3 and T_4) can be considered to be non-compensatable since many of today's rental or shipping companies demand expensive cancellation fees. We cannot tolerate a commit protocol which must repeatedly change or cancel contracts.

An aborted (sub-) transaction cannot be compensated by the invocation of a different sub-transaction; in contrast to Corba OTS ([7]), in which a hierarchy of commit decisions allows this kind of compensation.

The stability of the coordinator process itself is not a topic of this proposal. There are many contributions that handle coordinator failures, e.g., by running special termination protocols or by increasing coordinator availability with multiple coordinators (e.g [9], [10], or [11]). We only propose an improvement which is compatible to each of these commit protocols. The concrete protocol can be chosen depending on the desired coordinator stability. Therefore, we do not discuss coordinator failure in this contribution.

2.4 Blocking Behavior of Locking and Validation

For synchronization of concurrent transactions in fixed-wired networks, validation is usually considered to be a scheduling technique that avoids blocking. However, we argue that even validation-based synchronization shows a blocking behavior if used in combination with an atomic commit protocol. More precisely, in case of link failures or node failures, locking and validation are equivalent regarding their blocking behavior in the following sense. Assume a sub-transaction T_i, reading the tuples $t(R, T_i)$ and writing the tuples $t(W, T_i)$, has voted for commit and is waiting for a global commit decision.

Two-phase locking would not allow any sub-transaction T_k with $t(W,T_i) \cap (t(W,T_k) \cup t(R,T_k)) \neq \emptyset$ to get the required locks and would therefore block T_k and prevent the completion of T_k´s read phase.

Validation (e.g. [12]) would allow any later sub-transaction T_k with $t(W,T_i) \cap (t(W,T_k) \cup t(R,T_k)) \neq \emptyset$ to enter the read phase. However, since the tuple sets $t(W,T_i)$ and $(t(W,T_k) \cup t(R,T_k))$ are not disjoint, validation aborts T_k during its validation phase and thereby prevents T_k to enter its write phase.

This means that both techniques show a similar behavior when dealing with atomic commit decisions for mobile networks: A transaction T_i that has voted for commit, is waiting for a global commit decision, and has accessed the tuples $t(W,T_i) \cup t(R,T_i)$ prevents other sub-transactions T_k to gain access to the same data. This means that T_i prevents T_k from being committed while T_i only waits for the commit decision.

To reduce both, the blocking time and the probability of an abort, our solution distinguishes between two states within which a transaction waits for the global commit decision: Besides a blocking state "wait for global commit", we introduce the non-blocking "suspend state".

3 Solution

In the following, we describe the solution to the requirements, i.e., how to guarantee atomicity for Web service transactions and how to reduce the time of blocking and the number of transaction aborts compared to standard protocols like 2PC or 3PC. To reduce the time of blocking to a minimum extent, we distinguish between two states where a transaction is waiting for a commit: a blocking state (defined in Section 3.1) and a new non-blocking suspend state (introduced in Section 3.2). In Section 3.3, we explain how the suspend state can be used to reduce the number of transaction aborts and how previous results can be reused to a maximum extent. Finally, Section 3.5 explains by means of a "commit tree" how the coordinator learns of all sub-transactions that are dynamically invoked.

3.1 The Wait for Global-Commit State

We define the wait for global-commit state for the sub-transaction T_i reading the tuples $t(R,T_i)$ and writing the tuples $t(W,T_i)$ in the following way:

Definition 1. *The wait for global-commit state of T_i is a state in which a resource manager waits for a final decision of a commit coordinator on T_i to commit and blocks the tuples $t(W,T_i) \cup t(R,T_i)$. If another transaction T_k is executed while T_i is in the wait for global-commit state, T_k is not allowed to write on the tuples $t(R,T_i)$, and it is not allowed to read or to write on the tuples $t(W,T_i)$. The concurrent transaction T_k must wait until T_i is back in the suspend state or is committed or aborted and T_i has unlocked $t(R,T_i)$ and $t(W,T_i)$.*

3.2 The Non-blocking Suspend State

While waiting for the transaction's commit decision, protocols like 2PC or 3PC remain in a wait state and block the client that has voted for commit ([13], [4]). To reduce the duration of this blocking, our protocol extension suggests an additional *suspend state*, such that a transaction waiting for the global transaction decision can be in either of two states: in the non-blocking suspend state or in the blocking wait state.

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

Definition 2. *The suspend state of T_i is a state in which the resource manager RM executing T_i waits for a decision of the commit coordinator on T_i, but does not block the tuples $t(W, T_i) \cup (R, T_i)$.*

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

$$t(W, T_i) \cap (t(R, T_k) \cup t(W, T_k)) \neq \emptyset \texttt{ or } t(R, T_i) \cap t(W, T_k) \neq \emptyset$$

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.

3.3 Using the Suspend State to Reduce the Number of Aborts

Figure 2 shows an automaton that demonstrates all possible state transitions of a resource manager. Each resource manager enters the suspend state after having executed its read-phase and having sent a pre-vote message on the successful completion of the sub-transaction to the commit coordinator. The commit coordinator considers this pre-vote as a vote that does not bind the resource manager to a commit decision; instead, it only shows the resource manager's successful completion of the read-phase. Note that in comparison to 3PC, a resource manager is not bound to its pre-vote, but may still decide to abort the transaction as long as it is in suspend state. Similar to 2PC and 3PC, whenever a resource manager decides to abort a sub-transaction, it informs the coordinator, which then sends abort messages to the global transaction and all its other sub-transactions.

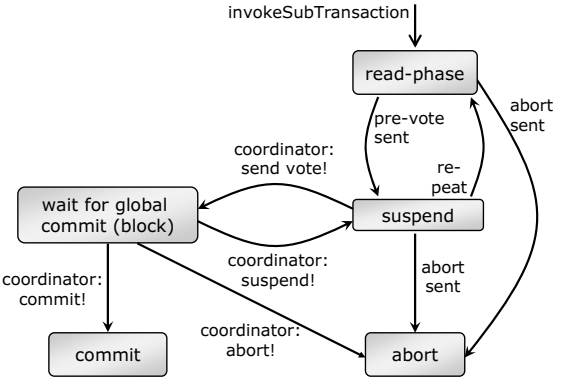

Fig. 2. Automaton showing the states and the received messages of a resource manager

When the coordinator has received the pre-votes of all resource managers, i.e., all resource managers have pre-voted for commit, the coordinator demands the resource managers to vote on the commit status of the transaction. This vote, in contrast to the pre-vote, is binding and the resource managers proceed to a blocking state where they wait for the global commit decision, i.e., the resource managers are not allowed to abort the transaction themselves while in blocking "wait for global-commit" state. The second collection of votes is needed because a resource manager may come to the decision to abort the sub-transaction while waiting in the suspend state. The coordinator collects these binding vote decisions on the sub-transactions and returns "abort" to the sub-transactions, if at least one RM's vote on its sub-transaction was "abort", and it returns "commit" to the sub-transactions, if it received the binding votes of all sub-transactions and all sub-transactions voted for commit. If the coordinator determines that vote messages are missing after a certain time, it can advise the RMs to put their sub-transactions back to the non-blocking suspend state instead of committing or aborting the sub-transaction. Note that this reduces the number of aborts compared to time-out-based 2PC, in which the coordinator decides to abort a transaction if votes do not arrive before time-out.

After a resource manager has successfully finished its read-phase, it sends the pre-vote message to the coordinator and proceeds to suspend. The resource manager waits in this non-blocking state until one of the following events occur:

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

After sending its vote to the coordinator, each resource manager proceeds to the "wait for global-commit state". Since it is now bound to its votes, concurrent transactions are blocked until the coordinator either decides on commit or abort or, after a timeout caused by missing votes, advises the sub-transaction to go into the suspend state again.

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 is reached.

The benefit of the suspend state lies in the reduction of blocking to only those cases where all pre-votes are present at the coordinator and all are commit. Even then, blocking only occurs for one message exchange cycle, i.e., while the coordinator asks for and retrieves the binding votes. Our protocol definition implies that all resource managers are able to give their vote on the transaction immediately. If, however, not all resource managers respond immediately, the coordinator may again advise the resource managers to go into 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. The more the duration of the read-phases varies

for different sub-transactions belonging to the same global transaction, the more the suspend state reduces the blocking time compared to 2PC.

In addition, in our protocol resource managers may fail or disappear without having a blocking effect on other sub-transactions, while in 2PC, in case of a resource manager failure before reaching a commit decision, the participants are blocked until the transaction is aborted due to timeout. Since timeouts must be sufficiently long to allow the execution of all sub-transactions in 2PC, the blocking time can be significantly long. However, if final votes are missing or delayed in our protocol extension, the coordinator can advise the sub-transactions to go into the non-blocking suspend state after a very short timeout, and sub-transactions do not have to stay in a blocking state until all resource managers have reconnected and voted.

3.4 Abort and Repetition of Sub-transactions

The second key idea of the suspend state, besides reducing the time of blocking, is to reduce the number of transaction aborts. For this purpose, we distinguish three different cases that arise due to an abort of a sub-transaction T_i.

1. T_i cannot be restarted because T_i is not committable or shall not be repeated anymore.
2. T_i is restarted and invokes the same sub-transactions $T_{si} \ldots T_{sj}$ with the same values for the actual parameters $P_{si} \ldots P_{sj}$ during the repeated execution of the read-phase. This case includes the situation where both executions of the sub-transaction do not invoke any other sub-transaction.
3. T_i is restarted, i.e. executed as T_i', and T_i' invokes different sub-transaction calls during execution of its read phase. In this case, the sub-transactions called by T_i and T_i' or their parameter values differ.

Only the first case requires the global transaction to be aborted, whereas the other types of sub-transaction abort can be solved by a repetition of the aborted sub-transaction T_i, which may involve calls to other sub-transactions $T_{si} \ldots T_{sj}$ as well. Now, we discuss each of these three cases in more detail.

***T_i* cannot be Restarted.** There are different reasons why a restart of an aborted transaction does not make sense:

- the transaction abort is caused by the user,
- the commit coordinator requires T_i to abort because another sub-transaction belonging to the same global transaction T required T to abort,
- the abort is caused by the execution of T_i on a resource manager RM itself, i.e., the result of running T_i on RM is that T_i cannot be committed.

In each of these cases, T_i and the global transaction T must be aborted. Note that although we have a hierarchy of invoked sub-transactions, we do not have a hierarchy of commit decisions: If a leaf node is not able to repeat the aborted sub-transaction with a chance of commit, it must vote for abort and the complete transaction T must be aborted and started again.

Restart with Identical Sub-transaction Calls and Parameters. The restart of a sub-transaction T_i is useful if T_i is in suspend state and a concurrent transaction C_k accesses at least one tuple accessed by T_i in a conflicting access mode. The resulting abort of T_i does not necessarily mean that the whole transaction T, of which T_i is a sub-transaction, must be aborted and restarted. Instead, we monitor which sub-transactions T_{si} are called by T_i with which parameters P_{si}[3] for each transaction T_i. Whenever T_i has called T_{si} with parameters P_{si}, and T'_i, i.e. the restarted version of T_i, needs to call a sub-transaction T'_{si} with exactly the same parameters P'_{si}, i.e. $P'_{si} = P_{si}$, we have the following optimization opportunity: We can omit a new invocation of T_{si} for the following reason. If T_{si} is still in suspend state, the result of a new execution of T_{si} will be the same. If, however, the tuples accessed by T_{si} change and T_{si} leaves the suspend state, the resource manager that executes T_{si} detects this conflict and starts T'_{si} as the repeated execution of T_{si}. The same argument applies to all sub-transactions that T_{si} has called. Since a sub-transaction does not directly return values, but calls receiving Web services instead, T'_i is not affected by the time when T'_{si} is executed.

Therefore, the repeated invocation of a previously called sub-transaction T_{si} can be omitted in the execution of T'_i if the invocation parameters P_{si} have not changed. In this case, we call T_{si} a *re-used sub-transaction.*

After T'_i has executed its read-phase, the resource manager again proceeds to the suspend state. However, if the coordinator message to give a vote on the transaction arrives while RM still executes T'_i, RM can inform the coordinator to unblock other waiting resource managers until the repetition is completed.

To summarize, if all sub-transaction calls are re-used or if no sub-transaction was invoked, a renewed sending of the pre-vote is not necessary and the resource manager can continue as in the first execution.

Restart with Different Sub-transaction Calls and Parameters. If we allow repetition, a problem may arise if the invoked sub-transactions $T_{si} \dots T_{sj}$ differ in a repeated execution of a sub-transaction T_i. Therefore, a resource manager not only remembers invoked sub-transactions, but also the invocation parameters $P_{si} \dots P_{sk}$.

If T_i is restarted as T'_i and needs to invoke the sub-transactions $T'_{si} \dots T'_{sj}$, it checks whether it can re-use the sub-transaction calls $T_{si} \dots T_{sj}$ of T_i. If this is not the case or if some calls are different, those sub-transactions $T'_{si} \dots T'_{sj}$ that do not find re-usable sub-transactions must be executed again. Furthermore, a sub-transaction T_{si} which is no longer needed for the execution of T'_i can be locally aborted. A local abort of T_{si} means that T_{si} and all of its sub-transactions are aborted, but the global transaction T and all other sub-transactions not being a descendant of T_{si} are not aborted. The advantage is that the global transaction T and all other sub-transactions of T do not have to be repeated.

Furthermore, the sub-transaction T'_{si} now belongs to T and the coordinator must be informed to wait for the pre-vote of this newly invoked sub-transaction

[3] The parameters include the name of the transaction procedure or web-service and all its actual parameters.

T'_{si} instead of waiting for T_{si}. This information is passed from T_i to the coordinator by a renewed sending of the pre-vote with updated parameters to the coordinator at the end of the sub-transaction execution of T'_i.

In addition, a resource manager that is going to restart a sub-transaction T_i can inform the coordinator about this restart, so that a possible abort vote of a descendant sub-transaction T_{si} will not cause an immediate global abort. Instead, the coordinator can wait for the pre-vote of T'_i, i.e. the restarted version of T_i, to check whether T_{si} is still needed at all.

3.5 The Coordinator's Commit Tree

The coordinator's *commit tree* is a data structure that allows the coordinator to determine which votes are missing for a commit decision. The tree structure is used to represent dependencies between sub-transactions.

To ensure that the commit tree gets knowledge of all invoked sub-transactions belonging to T, we require that each pre-vote message, sent by a sub-transaction T_i to the coordinator, not only contains the commit vote, but also informs the coordinator about all sub-transactions $T_{s1} \dots T_{sk}$ that are called by T_i. The coordinator then creates the nodes $T_{s1} \dots T_{sk}$ and adds these nodes as child nodes of the commit tree node containing T_i. Since the coordinator needs the pre-votes of all commit tree nodes, it must also wait for the pre-votes of $T_{s1} \dots T_{sk}$.

When a resource manager has repeated the execution of a sub-transaction T_i as T'_i according to Section 3.4, the resource manager sends a renewed pre-vote to the coordinator, which includes the invoked sub-transactions T'_{si} and the IDs of the re-used sub-transactions T_{si}^{reused}. The coordinator then replaces the subtree with root node T_i with T'_i. Each sub-transaction $T_j \in T_{si}$ that is not needed anymore, i.e. $T_j \notin T_{si}^{\text{reused}}$, is locally aborted and deleted from the commit tree. The re-used sub-transactions T_{si}^{reused} and the new sub-transactions T'_{si} are appended as child nodes to T'_{si}.

4 Related Work

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

Our contribution differs from the Web service transaction model of [3] in several aspects. For example, [3] uses a "completion protocol" for registering resource managers 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. However, since the suspend state can be used as a protocol extension, it can also be combined with [3].

Besides the Web service orchestration model, there are other contributions that set up transactional models to allow the invocation of sub-transactions, e.g. [6] or [14]. Common with these transaction models, we have a global transaction and sub-transactions that are created during transaction execution and cannot be foreseen. The main differences to these nested transactional models is that we consider all sub-transactions to be non-compensatable.

The use of a suspend state is also proposed by [15]. However, these approaches are intended for use within an environment with a fixed-wired network and several mobile cells, where disconnections are detectable and therefore transactions are considered to be compensatable.

Our work is based on [16], but goes beyond this in several aspects, e.g., we distinguish three different types of transaction aborts and provide technologies for reusing and locally restarting sub-transactions. In addition, we use the suspend state not only to treat network failures, but also in the failure-free case in order to further reduce the blocking time during failure-free transaction execution.

Corba OTS ([7]) uses the term "suspend" for a concept which 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 can be compensated by other sub-transactions. However, in the presence of non-compensability, this implies waiting for the commit decision of all descendant nodes. In a mobile environment where node failures are likely, we neither propose to wait nor to block the participants until the commit decision has reached all participants.

5 Summary and Conclusion

We have presented two key ideas for guaranteeing atomicity for web-service transactions in a mobile context that also reduce the time of blocking and the number of aborts. The first idea is to use the suspend state for a transaction that has finished its read phase while the coordinator waits for the votes of other sub-transactions of the same global transaction. Being in suspend state, a sub-transaction can still be aborted by the resource manager if the resource manager decides to grant the resources used by this sub-transaction to other concurrent transactions to prevent them from blocking.

Secondly, for reducing the number of aborts in case of conflicts or missing votes, we identify those aborted sub-transactions that are repeatable or reusable, instead of only aborting and restarting all sub-transactions of the global transaction. Additionally, we have introduced 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 2PC. Note however, that our contributions are applicable to a much broader set of protocols. For example, the extension of an atomic commit protocol by a

non-blocking suspend state is not limited to 2PC, but appears to be compatible with a variety of other atomic commit protocols, e.g. [9], [10], or [11].

Finally, our protocol extension can nicely be combined with various concurrency control strategies including validation and locking. Although a proof of serializability for any schedule of concurrent transactions is beyond the scope of this paper, we have evidence that serializability can be guaranteed, and we plan to report about this on a forthcoming paper.

References

1. 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)
2. Arkin, A., et al.: Business process modeling language, bpmi.org. (Technical report)
3. 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)
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. Kumar, V., Prabhu, N., Dunham, M.H., Seydim, A.Y.: Tcot-a timeout-based mobile transaction commitment protocol. IEEE Trans. Com. **51** (2002) 1212–1218
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. Object Management Group: Trans. Service Spec. 1.4. http://www.omg.org (03)
8. Skeen, D., Stonebraker, M.: A formal model of crash recovery in a distributed system. In: Berkeley Workshop. (1981) 129–142
9. Reddy, P.K., Kitsuregawa, M.: Reducing the blocking in two-phase commit with backup sites. Inf. Process. Lett. **86** (2003) 39–47
10. Gray, J., Lamport, L.: Consensus on transaction commit. Microsoft Research – Technical Report 2003 (MSR-TR-2003-96) **cs.DC/0408036** (2004)
11. 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)
12. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. **6** (1981) 213–226
13. 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
14. Pitoura, E., Bhargava, B.K.: Maintaining consistency of data in mobile distributed environments. In: Intl. Conf. on Distributed Computing Systems. (1995) 404–413
15. Dirckze, R.A., Gruenwald, L.: A toggle transact. management technique for mobile multidatabases. In: CIKM '98, New York, USA, ACM Press (1998) 371–377
16. Böttcher, S., Gruenwald, L., Obermeier, S.: An Atomic Web-Service Transaction Protocol for Mobile Environments. In: Proceedings of the 2nd EDBT Workshop on Pervasive Information Management, Munich, Germany (2006)