Distrib Parallel Databases (2009) 25: 165–192
DOI 10.1007/s10619-009-7033-z

Blocking reduction for distributed transaction processing within MANETs

Sebastian Obermeier · Stefan Böttcher · Martin Hett · Panos K. Chrysanthis · George Samaras

Published online: 24 February 2009

Abstract Atomic commit protocols for distributed transactions in mobile ad-hoc networks have to consider message delays and network failures. We consider ad-hoc network scenarios, in which participants hold embedded databases and offer services to other participants. Services that are composed of several other services can access and manipulate data of physically different databases. In such a scenario, distributed transaction processing can be used to guarantee atomicity and serializability throughout all databases. However, with problems like message loss, node failure, and network partitioning, mobile environments make it hard to get estimations on the duration of a simple message exchange.

In this article, we focus on the problem of setting up reasonable time-outs when guaranteeing atomicity for transaction processing within mobile ad-hoc networks, and we show the effect of setting up "wrong" time-outs on the transaction throughput and blocking time. Our solution, which does not depend on time-outs, shows a better performance in unreliable networks and remarkably reduces the amount of blocking.

Communicated by Amit Sheth.

S. Obermeier (✉) · S. Böttcher · M. Hett
University of Paderborn, 33102 Paderborn, Germany
e-mail: so@upb.de

S. Böttcher
e-mail: stb@upb.de

M. Hett
e-mail: mh1108@upb.de

P.K. Chrysanthis
University of Pittsburgh, Pittsburgh, PA 15260, USA
e-mail: panos@cs.pitt.edu

G. Samaras
University of Cyprus, 1678 Nicosia, Cyprus
e-mail: cssamara@cs.ucy.ac.cy

Keywords Mobile transaction processing · Concurrency control · Locking · Validation · Mobile ad-hoc networks · Atomic commit · Adjourn state

1 Introduction

As mobile devices get ubiquitous and grow in computational power, their management of interdependent data also becomes increasingly important. For example, consider the following application scenarios that make use of mobile ad-hoc networks.

Mobile market scenarios allow buyers and sellers to use their PDAs for searching, offering, and buying items. A sale transaction can involve items and services of different people located in different parts of the flea marked. In this case, to guarantee that buyer gets all items and services, atomicity is a required criterion. Furthermore, when the number of requested items and services is higher than the number of items available at the flea market, the sellers do not want to reserve their goods for a long time for a customer that might even back out of the purchase, i.e., an appropriate time-out mechanism for sale transactions is required.

Mobile gaming, e.g., mobile role playing games, let a player exchange and equip virtual characters with virtual items. When the game uses embedded local databases to store the game information, all steps of a player's action that affect one or more databases should be encapsulated by database transactions in order to guarantee consistent game data.

Disaster management, e.g., in case of forest fires, makes use of mobile devices to instruct fire fighters and to supply them with actual position data and burning densities. Coordinated actions of distributed fire fighters, e.g. coordinated rescue actions, require atomic decisions on the use of men and resources. In this scenario, the agreement of all participating fire fighters about the next action is crucial for the security of the fire fighters. As the application of transaction processing techniques allows giving guarantees for atomicity, the reduction of transaction blocking is desirable, for example when fire fighters must be re-grouped.

Scientific exploration, e.g., a Mars exploration, use mobile ad-hoc networks to coordinate multiple exploration rovers and to manage sensored data. In order to store, search, and compare gathered data of a rover with another rover's data, embedded databases are also useful. In order to guarantee the consistency of all databases in case database operations affect more than one database, atomicity is required. Blocking reduction is also important for the overall application performance.

Clearly, such mobile environments combine all the characteristics of a distributed database with the challenge of whimsical connectivity.

In order to maintain data consistency and integrity across the network and across the local databases of each device, in our examples, the use of atomic mobile distributed transactions is crucial and challenging.

Within fixed wired networks, atomic commit-protocols like Two-phase Commit protocol (2PC) [18] or Three-phase Commit protocol (3PC) [36], and a lot of variations of them (e.g., [2, 11, 22, 34, 35]), ensure the atomic execution of distributed transactions by using a central Coordinator. However, in the context of mobile ad-hoc networks in which mobile devices interact cooperatively via Web services, the protocol must be able to deal with a much more enhanced failure model and must be more flexible in unreliable environments. As disconnection times are unforeseeable, the use of standard lock-based concurrency control techniques and atomic commit protocols in mobile ad-hoc environments may lead to unbounded and unpredictable delays. Most of these techniques and protocols rely on time-outs to detect and handle failures. However, it is extremely difficult to set up reasonable time-outs, for example, for aborting a transaction when its Commit coordinator does not respond immediately during the execution of an atomic commit protocol such as 2PC or 3PC.

1.1 Contributions

The main contributions of this paper are:

- We propose a distributed transaction model based on Web services.
- We present the "Adjourn state", a non-blocking state that allows concurrent transactions to be processed while a distributed transaction is waiting for the Coordinator's voteRequest message to demand the commit decision. In contrast to the traditional Blocking (or Wait) state, the Adjourn state shows the following advantages:
  - It does not require the setup of a transaction time-out.
  - It does not block concurrent transactions.
  - It provides 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.
- We introduce a novel use of the Commit tree, i.e. a tree data structure that represents the execution status of all active sub-transactions, and show how the Commit tree can be used to efficiently coordinate the transaction.
- We present an experimental evaluation that demonstrates the advantage of the Adjourn state in a mobile and highly unreliable environment.

Beyond our previous contributions [30] and [7], this article further

- provides a detailed Web service transaction model,
- shows implementations of the Adjourn state for both locking and validation concurrency control,
- explains how sub-transaction dependencies and commit-status information shared in the Commit tree can be used to partially restart and re-use sub-transactions, and
- gives experimental results for the Adjourn state and evaluates the effect of setting up different time-outs regarding transaction throughput and overall blocking time.

1.2 Paper organization

The remainder of the article is organized as follows. In Sect. 2, we explain the underlying system model and the transaction model, and we point out its use in two concurrency control mechanisms, one pessimistic approach using locks, and one optimistic approach using backward oriented validation. In Sect. 2.3, we describe the blocking problem that both concurrency control mechanisms involve regarding the processing of concurrent transactions in a mobile environment, and the resulting difficulty in setting up reasonable time-outs for aborting non-progressing transactions is explained in Sect. 2.4.

Section 3 describes our solution, which distinguishes between a blocking and a non-blocking state—called the *Adjourn state*—while waiting for the atomic commit protocol to start. Thus, our solution makes setting up time-outs obsolete. We describe how to combine our solution with both concurrency control mechanisms, locking and validation. Furthermore, we show how the use of the Commit tree speeds up the commit decision phase by implicitly flattening the transaction invocation tree, and how the Commit tree can be used to optimize concurrency control issues.

Section 5 points out the experimental setup and describes the results of our experiments. Finally, Sect. 6 discusses related work and Sect. 7 summarizes our results.

2 System model

We consider mobile devices, each equipped with a local database. The mobile devices form a mobile ad-hoc network and offer their data by providing (web-) services. When a device uses an offered Web service of a participant P_i, one or more sub-transactions are invoked at the local database of P_i, which may invoke other sub-transactions on mobile devices P_k.

Due to the distributed character of our system model, one challenge when guaranteeing the atomicity property is to block resources as short as possible.

2.1 Transaction model

Our transaction model bases on the "Web Services Transactions Specification" [10]. However, due to our focus on the blocking problem when guaranteeing atomicity, we can use a much simpler transaction model in this paper, i.e., we do not need a certain Web service modeling or composition language like BPEL4WS [13]. Thus, our transaction model consists only of the objects "application", "Web service", "transaction procedure", "global transaction", and "sub-transaction". These terms are related to each other as explained in the first subsection. The second subsection summarizes some characteristic differences to other transaction models, and the third subsection describes an example execution illustrating our transaction model.

2.1.1 Basic components of our Web service transaction model

An *application* is a program that may consist of one or more *Web services*. A transaction procedure is a Web service that must be executed in an atomic fashion. Transaction procedures and Web services are program fragments including local code, database instructions, and (zero or more) instructions to call other remote Web services.

Note that we distinguish between transaction procedures and transactions as follows. A *transaction procedure* is a program fragment that shall be executed in an atomic fashion. In contrast, a *transaction*, which is either a global transaction or a sub-transaction, is an atomic execution of a transaction procedure. The same transaction procedure when called twice leads to different transactions that may even call a different number of sub-transactions.

When an application AP invokes a transaction procedure, we call AP the *Initiator*, and we call the execution of the transaction procedure a *global transaction T*. The 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 is also interested in the return values of the parameters of T.

A global transaction may be distributed over several databases and may consists of one or more *sub-transactions*, each of which 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.

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 directly or indirectly invoked by the T the sub-transactions $T_j \dots T_k$ of T. The invocation hierarchy of sub-transactions can be regarded as an invocation tree of sub-transactions that may be arbitrarily deep. Within such an invocation tree of sub-transactions, the "root-node" represents the global transaction T_g, while all other nodes of the tree represent sub-transactions T_i of T_g. In this sense, each transaction T_i calling a sub-transaction T_j is represented by the "parent" node of the node representing the sub-transaction T_j within the invocation hierarchy, and, for simplicity, we also call T_i the parent transaction of T_j.

Each sub-transaction T_j that has been invoked by a parent transaction T_i is executed autonomously. T_j and T_i independently of each other send their commit decision or abort decision to the coordinator. Whenever $T_1 \dots T_n$ denote all the sub-transactions called by either a global transaction T_g directly or by any child or descendant sub-transaction T_s of T_g during the execution of T_g, atomicity of T_g requires that either all transactions of the set $\{T_g, T_1, \dots, T_n\}$ commit, or all of these transactions abort.

We assume that each Web service only knows the Web services that it calls directly, but does not know 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 $T_k \dots T_n$ it has called, but T_i, in general, will not know which sub-transactions have been called by $T_k \dots T_n$. Furthermore, we assume that usually a transaction T_i does not know how long its sub-transactions $T_k \dots T_n$ are going to run. This holds in particular for each global transaction.

In the mobile architecture for which our protocol is designed, Web services are invoked by asynchronous messages instead of invoking them by synchronous calls for the following reason. We want to avoid dependencies that occur if the completion of the read-phase of a sub-transaction T_i depends on the execution of another sub-transaction T_j. Figure 1 illustrates the situation that we want to avoid: The Web service T_i must wait for the return value of T_j before it can finish its execution.

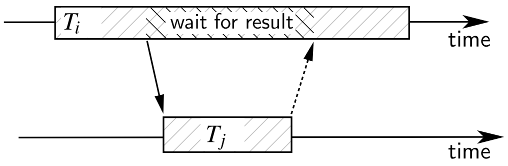

Fig. 1 Synchronous calls for Web services

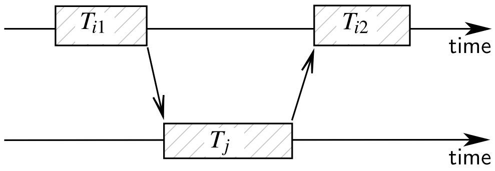

Fig. 2 Modeling synchronous Web service calls by asynchronous invocations

In contrast, we want each sub-transaction to be able to autonomously complete its read phase. Thus, we allow sub-transactions only to return values indirectly by asynchronously invoking corresponding receiving Web services, but not synchronously by return statements. However, our model also supports a synchronous Web service call of T_i to T_j by modeling the call as Fig. 2 illustrates: we split 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 additionally contains an asynchronous call to T_{i_2} that contains the return values computed by T_j that shall be further processed by T_{i_2}. In other words, a transaction T_i calling a sub-transaction T_j and waiting for T_j's return value to continue, as shown in Fig. 1, is replaced in our model with three transactions as shown in Fig. 2. Here, T_{i_2} is a sub-transaction of T_j, T_j is a sub-transaction of T_{i_1}.

Since (sub-)transactions describe general Web service executions, they may not necessarily access only relational databases, but can also access other resources stored on the mobile device. Thus, we also call these nodes *resource managers (RM)* and assume they support transaction processing.

2.1.2 Characteristic features of our Web service transaction model

We distinguish between program code, e.g., Web Services and transaction procedures, and concrete executions of the program code on the database (transactions). A transaction does not contain program code and Web services are never executions of program code. As 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. Thus, the same Web service can lead to different transactions on the database, e.g., when it is executed twice.

A special feature of our Web service transaction 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. [10, 16, 31]) 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-compensable. 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 [26, 31], 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 [10], the Initiator of a transaction in our model does not need to know all the transaction's sub-transactions in advance. We assume that each sub-transaction notifies the Initiator after completion of the read-phase by including a list of asynchronously invoked sub-transactions into its "read phase completed" notification message to the Initiator.
- A Web service may consist of control structures, e.g., if `<Condition> then <T1> else <T2>`. This means that an execution of a sub-transaction may create other sub-transactions dynamically. These dynamically created sub-transactions belong to the same global transaction and must participate in the atomic commit decision, as well.
- 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.

2.1.3 *An example executing illustrating out Web service transaction model*

Figure 3 shows an example execution of our Web service transaction model. The Initiator of the transaction invokes a Web service T_i that is offered by participant P_i. After the completion of the read-phase, an additional transaction is T_j offered by participant P_j is invoked. Furthermore, the Initiator is notified whenever a sub-transaction's read-phase is completed, as the dotted arrows indicate. We will see the benefit of this notification regarding a fast determination of all involved participants in Sect. 4. Furthermore, Fig. 3 shows how call-by-value-result invocations are implemented: When participant P_i invokes the sub-transaction T_j at P_j, it additionally adds a parameter $[T_k]$, which specifies the sub-transaction T_k to handle the return value. Thus, P_j invokes T_k with the requested return value after P_j has finished T_j's read-phase.

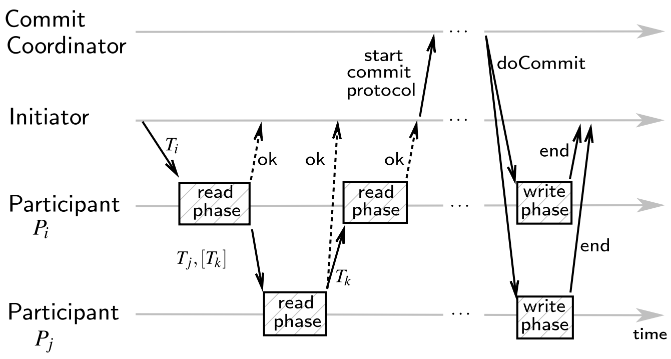

Fig. 3 Web service transaction execution sequence

Note that the transaction consists of the sub-transactions T_i, T_j, and T_k. The invocation hierarchy of the sub-transactions is $T_i \to T_j \to T_k$. However, the three sub-transactions are executed only by two physical participants, P_i and P_j. Although, in this case, T_i and T_k are run by the same participant P_i, their execution is independent of each other and P_i must sent two votes to the Coordinator during the atomic commit protocol, one vote for T_i and one vote for T_k. However, P_i can bundle these two votes into one message.

The Initiator starts the commit protocol when all sub-transactions have finished; how the Initiator determines this, is explained in Sect. 4. The commit protocol then needs several message exchanges in order to decide on the transaction's commit status.

2.2 Concurrency control

We describe two common concurrency control methods, namely *Two-phase locking (2PL)* and *validation*, which can be used in our Web service transaction model. Based on these synchronization schemes, we show how to combine our solution with each of these synchronization techniques in Sect. 3.

For each sub-transaction T_v, let $RS(T_v)$ denote the local data read by T_v and $WS(T_v)$ denote the local data written by T_v.

2.2.1 Two-phase locking

The *Two phase locking protocol* [17] consists of two consecutive phases for handling transaction locks:

- In phase 1, necessary locks are acquired, no lock is released.
- In phase 2, no lock must be acquired anymore, but locks may be released.

For transactions that obey 2PL, the serializability property is guaranteed according to [38]. However, in order to avoid cascading aborts and to guarantee recoverable histories, we require transactions to be *strict* [5], i.e., Phase 2 can only be entered when the resource manager has received the decision of the coordinator on the transaction's commit or abort status. In other words: Unless the transaction's commit decision is not known by a resource manager, the resource manager is not allowed to use the transaction's resources for other purposes. We assume for the remainder of the paper that the strictness property holds when 2PL is used.

2.2.2 Local concurrency control by backward validation

Optimistic concurrency control [20, 23], more precisely *backward oriented optimistic concurrency control* with parallel validation, does not use locking and allows parallel access to conflicting data tuples. However, after the read-phase has been finished, an additional *validation phase* follows in which concurrency conflicts are discovered. Only those transactions that pass the validation may, after a successful distributed commit decision, enter the write phase, during which they write their changes back to the database.

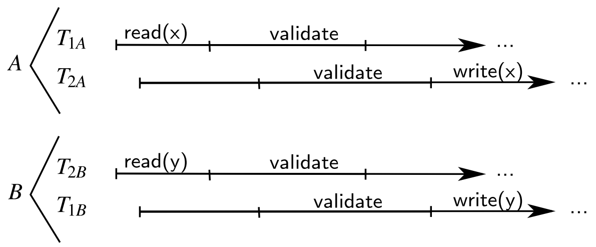

Fig. 4 Serializability for distributed transactions

A local sub-transaction T_o is called *older* than a local sub-transaction T_v running on the same database, if T_o starts its validation phase before T_v does.

During the validation phase, the resource managers checks whether for each older transaction T_o, one of the following conditions hold. If this is the case, transaction T_v validates to *true*. Otherwise, T_v validates to *false*.

1. T_o has completed its write phase before T_v has started.
2. T_o has completed its write phase after T_v has started, but before T_v has started its validation phase, and $(RS(T_v) \cap WS(T_o)) = \varnothing$.
3. T_o has not finished its write phase before T_v has started the validation, and
   (a) $(RS(T_v) \cup WS(T_v)) \cap WS(T_o) = \varnothing$, and
   (b) $(RS(T_o) \cap WS(T_v)) = \varnothing$.

Compared to [23], we additionally require condition 3(b) to be fulfilled in order to guarantee serializability for distributed transactions. While the concurrency control proposed by [23] correctly guarantees serializability for non-distributed transactions, it cannot guarantee serializability when distributed transactions are executed concurrently, as shown in the example of Fig. 4.

Figure 4 shows a schedule for the participants A and B, each of them running two sub-transactions concurrently. On participant A, a dependency T_{1A} before T_{2A} exists, while on participant B, a dependency T_{2B} before T_{1B} exists. If a global transaction T_1 consists of the sub-transactions T_{1A} and T_{1B} and a global transaction T_2 consists of the sub-transactions T_{2A} and T_{2B}, the serialization graph would contain a cycle, which violates serializability. The request for checking condition 3(b) prevents this cycle, since our proposed concurrency control protocol would abort both sub-transactions T_{2A} and T_{1B}.

2.3 Blocking behavior of locking and validation

Traditional validation [23] is usually considered a scheduling technique for synchronization of concurrent transactions that avoids blocking.

However, although validation does not directly block any resources, we argue that even the validation-based concurrency control shows a blocking behavior when 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 that a sub-transaction T_i, reading the tuples $RS(T_i)$ and writing the tuples $WS(T_i)$, is waiting for the Coordinator to request its vote.

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

Traditional validation (e.g., [23]) *would allow* any younger sub-transaction T_k with $WS(T_i) \cap (WS(T_k) \cup RS(T_k)) \neq \emptyset$ to enter the read-phase. However, since the tuple sets $WS(T_i)$ and $(WS(T_k) \cup RS(T_k))$ are not disjoint, the validation of T_k fails, resulting in an abort of T_k. Thus, validation prevents T_k to enter its write phase as well. Note that even a repetition of T_k would result in an abort as long as T_i waits for the Coordinator's decision.

This means that both techniques, locking and validation, show a similar behavior when dealing with atomic commit decisions for mobile networks: A transaction T_i that waits for the commit decision and that has accessed the tuples $WS(T_i) \cup RS(T_i)$ during its read-phase, is not allowed to unilaterally commit or abort the transaction. Thus, T_i prevents other sub-transactions T_k that write on the data T_i accessed or that read data T_i has written from being committed.

2.4 Problem description

Regardless of whether validation or locking is used, the following problem occurs when the database is still able to abort a transaction T_O, but the Commit coordinator is not reachable anymore. Then, the concurrency control prevents conflicting transactions T_O from being successfully executed. In other words, any delay in the commit phase of T_O has a blocking effect on concurrent conflicting transactions T_V. To solve this problem, [36] has introduced time-outs after which the database aborts the transaction T_O if it is still allowed to do so, i.e., if it has not sent its vote message.

However, especially in mobile networks, the question arises: "What is a reasonable time-out after which the database should abort the transaction T_O if it has not sent a vote message yet?". If the time-out is too large, it prevents concurrent and conflicting transactions T_N from a successful validation, since T_N will not pass the validation phase successfully due to the pending transaction T_O. If the time-out is too short, T_O may be unnecessarily aborted, e.g., when the delay is caused by the network or when the duration of the validation phase differs for the databases participating in the global transaction. Determining a reasonable time-out is difficult since it involves not only knowledge about the network conditions, e.g., device movement, message delivery times, message loss rates, etc., it must also consider the device's computing power and CPU utilization, and the varying duration of the validation phase for each mobile device. Therefore, our solution, which does not rely on such a time-out, is much easier to setup, and we will even see that it increases the overall transaction throughput and reduces the amount of blocking.

3 Adjourn state blocking reduction

Our solution consists of two parts, namely, the *Adjourn state* and the *Commit tree*. The Adjourn state avoids setting up participant time-outs for aborting a transaction by distinguishing between two states in which a database can wait for the coordinator's voteRequest message: the *blocking state* (defined in Sect. 3.1) and the non-blocking *Adjourn state* (introduced in Sect. 3.2). We describe the database's reaction regarding concurrent conflicting transactions for both, locking and validation-based concurrency control.

The second part of our solution—the Commit tree—deals with the identification of invoked sub-transactions. Furthermore, the Commit tree handles partial restarts of a sub-transaction instead of repeating the whole global transactions.

3.1 The blocking state

The database is allowed to switch unilaterally from the blocking state to the non-blocking Adjourn state as long as the vote has not been sent. However, the vote on the transaction can only be sent in the blocking state, and once a vote has been sent, the Adjourn state must not be entered anymore.

Both states, the blocking state and the non-blocking Adjourn state differ in the way whether or how the validation phase for a concurrent transaction is executed, and therefore show a different blocking behavior.

3.1.1 Blocking state for locking

If a locking-based concurrency control scheme is used and a sub-transaction T_i that is in the blocking state has acquired the set of read locks $RL(T_i)$ and the set of write locks $WL(T_i)$, another transaction T_k is not allowed to acquire a write lock wl with $wl \in RL(T_i)$, and T_k is not allowed to acquire any lock l with $l \in WL(T_i)$. Thus, a concurrent transaction T_k must wait until T_i is committed or aborted, or until T_i has proceeded to the Adjourn state, and thus has unlocked $RL(T_i)$ and $WL(T_i)$.

3.1.2 Blocking state for validation

While a successfully validated transaction T_v is in the blocking state, the validation of a newer transaction T_n against the older transaction T_v is done by T_n as described in Sect. 2.2.2. This means, transaction T_n is validated against T_v with the effect that whenever transaction T_n is in conflict with T_v, T_n is aborted.

3.2 The non-blocking Adjourn state

A transaction T_v that has successfully finished its read-phase may enter the non-blocking Adjourn state at any time after T_v has sent the result message to the Initiator and before T_v has sent the vote message. However, T_v must migrate from Adjourn state to blocking state before it may send its vote message to the coordinator. Remember that, after the vote message has been sent, the sub-transaction is not allowed to enter the Adjourn state anymore.

If validation is used, the database must further perform a *second adjourn-specific validation phase* before a transaction is allowed to leave the Adjourn state.

Definition 1 The Adjourn state of T_v is a state in which the resource manager RM executing T_v waits for the Commit coordinator's request to vote on T_v, but RM does not block the tuples in $WS(T_v) \cup RS(T_v)$, i.e., the tuples written or read by T_v.

In the following, we describe what happens when a concurrent transaction tries to perform conflicting accesses to the data contained in $WS(T_v) \cup RS(T_v)$.

3.2.1 Adjourn state for locking

If locking is used and a transaction T_v, which has acquired the set of read locks $RL(T_v)$ and the set of write locks $WL(T_v)$, enters the Adjourn state, the locks $RL(T_v)$ and $WL(T_v)$ are released. However, when the resource manager RM grants one or more locks from the set $RL(T_v) \cup WL(T_v)$ to another transaction T_k while T_v is in the Adjourn state, the RM checks whether or not

$$\begin{aligned} &WL(T_v) \cap (RL(T_k) \cup WL(T_k)) = \varnothing, \\ &\bigwedge RL(T_v) \cap WL(T_k) = \varnothing. \end{aligned}$$

If this check evaluates to false, there is a conflict between T_v and T_k. Therefore, the RM locally aborts T_v and the RM can either abort all other corresponding (sub-) transactions that belong to T_v, or try a repeated execution of the sub-transaction T_v if T_v is still repeatable.

3.2.2 Adjourn state for validation

While T_v is in the non-blocking Adjourn state, the validation of a concurrent transaction T_n is done as follows: T_n is validated against all older transactions *except* those being in the Adjourn state when T_n started its validation phase. This means, T_v, which is in the Adjourn state, has no blocking effect on concurrent transactions T_n.

When T_v must leave the Adjourn state, i.e., when the Commit coordinator demands a binding vote on the transaction, T_v must be validated again in a second adjourn-specific validation phase. However, the scope of this second validation is different from the first validation phase:

This second validation of a transaction T_v is successful, if and only if the following condition holds for each transaction T_n that has started its validation while T_v has been in the Adjourn state:

$$\begin{aligned} &(RS(T_n) \cup WS(T_n)) \cap WS(T_v) = \varnothing, \\ &\bigwedge RS(T_v) \cap WS(T_n) = \varnothing. \end{aligned}$$

When this validation fails, T_v must either be aborted or can be locally restarted.

The reason for this concurrency check is the following: Although T_v entered its validation phase before T_n, i.e., T_v is older, T_n has not been validated against T_v. Since T_n may have already been committed, the validation of T_v against T_n must be either successful, or T_v must be aborted or locally restarted.

Note that the Adjourn state only delays the validation of T_n against T_v. However, the number of validation tests is exactly the same as with other commit protocols that use backward oriented concurrent validation.

3.3 Local restarts and re-use of sub-transactions

Whenever a local sub-transaction T_i executing a Web Service W_i is in Adjourn state and must be aborted due to a conflicting access of a concurrent transaction, the database can try to re-execute the Web service W_i as a sub-transaction T_i'. However, as sub-transactions are dynamically generated, T_i' may invoke different Web services than T_i. Since in our transactional model T_i and T_i' do not return any values, a repetition of T_i as T_i' results only in a repetition of those sub-transactions that have been invoked by T_i, but a repetition of T_i as T_i' will not result in a repetition of the sub-transaction that has invoked T_i.

Furthermore, we can optimize the repetition of the Web service W_i that has spawned a sub-transaction T_i that is repeated as T_i' as follows: T_i' does not need to execute a call to Web service W_s with parameters P_s when the same call has also been issued by T_i with exactly the same parameters. In this case, we can re-use the call of T_i to W_s in the repetition of W_i, i.e., in T_i', since W_s itself is responsible for local restarts in case of concurrency conflicts. Since the effects of the execution of W_s will become permanent after the completion of the atomic commit protocol and W_s never returns any value to an ancestor in the invocation tree, W_s only depends on the invocation parameter of W_i. The concrete execution of W_i, however, does not depend on the execution of W_s at all. Thus, whenever W_s is repeated with the same invocation parameters, this repetition does not have any effect on W_i. Therefore, the call of T_i to W_s can be re-used.

To summarize, if W_i must be repeated and the corresponding sub-transaction T_i has invoked W_s with parameter P_s, the repetition of W_i as T_i' can lead to the following possibilities regarding the calls to other Web services:

1. T_i' must issue a call to W_s with the same parameters P_s as T_i has done. This call does not need to be executed, T_i' can re-use the invocation done by T_i.
2. T_i' must issue a call to W_s with different parameters P_s'. This call must be executed.
3. T_i' does not need to call W_s anymore. Then, W_s can be aborted.
4. T_i' must issue a call to a new Web Service W_t that has not been invoked by T_i. Then, W_t must be treated as every other Web service that belongs to the global transaction.

If the repetition T_i' of a Web service W_i previously executed as T_i does not call any new Web service W_j and does not call any Web service W_k with parameters that differ from the parameters that were used when calling T_i, W_i can be locally repeated without having an effect on other Web services.

3.4 Entering the Adjourn state

Each transaction may decide for itself when it enters the Adjourn state. However, we propose to wait for a short delay in order to avoid unnecessary aborts. Our experiments have shown that waiting for the duration of a typical message delay gives the best results.

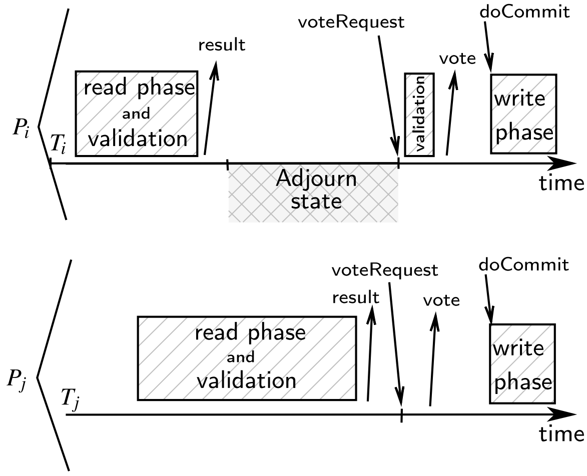

Fig. 5 Two sub-transactions executed at P_i and P_j

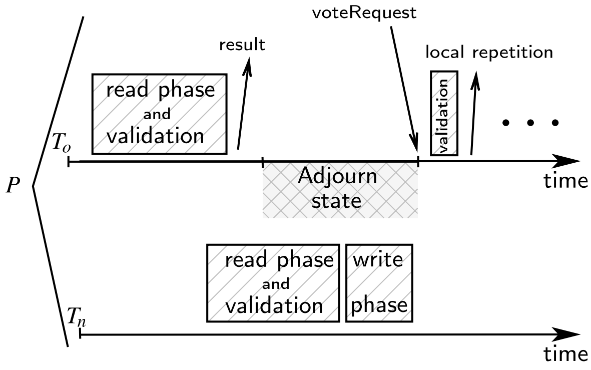

Fig. 6 Single resource manager running concurrent and conflicting transactions

Figure 5 shows an example application of the Adjourn state when validation-based concurrency control is used. It shows two resource managers P_i and P_j that try to commit the sub-transactions T_i and T_j, respectively. Both, T_i and T_j, belong to the same global transaction T. As the execution time of the read phase and the validation phase takes longer for P_j than for P_i, P_i has to wait for a longer period of time for the voteRequest message to arrive. However, P_i does not know about the delay of P_j. In order to avoid blocking of concurrent transactions after the validation phase, P_i migrates T_i into the Adjourn state and unblocks the occupied resources. After P_j has successfully sent the result, the coordinator demands the vote of P_i and P_j. Since T_i has entered the Adjourn state, P_i must perform the second validation phase as stated in Sect. 3.2, before T_i can leave the Adjourn state and can send the vote to the coordinator.

Figure 6 shows a single resource manager P executing the two independent sub-transactions T_o and T_n. While the older transaction, T_o, waits for the coordinator's voteRequest message, the newer concurrent local transaction, T_n, performs conflicting accesses to data tuples accessed by T_o. Therefore, after T_o received the voteRequest, the second adjourn-specific validation phase of T_o against T_n fails, which requires the repetition of T_o's read-phase. However, the delay within the coordination process of T_o has not led to a chain reaction of blocking of the concurrent transactions, e.g., T_n could be still committed.

3.5 Proof of correctness

The following theorem states that it is not possible that a sub-transaction T_k of a global transaction T is still in Adjourn state, when a sub-transaction T_i of T is committed.

Theorem 1 *Whenever a sub-transaction T_i of a transaction T is committed, i.e., T_i's write phase is executed, no sub-transaction T_k belonging to the transaction T can be in the Adjourn state anymore.*

Proof by contradiction. Assume, the coordinator's decision for T is commit, and the sub-transaction T_k is in Adjourn state. In order to decide for commit, the coordinator must have had received voteCommit messages of all participating sub-transactions, including T_k. However, in order to be able to send a "voteCommit" message, T_k must be in the blocking state. After having sent the voteCommit message, T_k is not allowed to migrate to the Adjourn state anymore, as described in Sect. 3.2. Therefore, without violating the protocol, there is no possibility for a sub-transaction T_k of T to be in Adjourn state, when another sub-transaction of T is committed. □

3.6 Number of messages

The Adjourn state is entered after the read-phase and the first validation phase have been successfully finished. After the coordinator has sent the voteRequest message, the database performs a second validation, and either immediately replies by sending the vote message, or it locally restarts the sub-transaction. In the failure-free case, each protocol with Adjourn state does not require additional messages compared to the corresponding protocol without Adjourn state. Of course, if a transaction must be locally restarted, additional messages for invoking sub-transactions may be necessary. However, this involves at most the same work and at most the same number of messages as restarting the global transaction as required by protocols without the Adjourn state.

4 Commit tree

As described in Sect. 3.3, a sub-transaction T_i might be restarted as T_i' in case of concurrency conflicts. In this case, T_i and other sub-transactions T_j that have been invoked by T_i and are either invoked by T_i' with different parameters or are not at all invoked by T_i, can be aborted. In order to abort these sub-transactions, the coordinator must learn about the invocation hierarchy. For this purpose, the invocation hierarchy and the commit status of the involved sub-transactions is stored in a data structure called "Commit tree". To generate the Commit tree, each participant that sends a result to the Initiator attaches the IDs of all invoked sub-transactions. The Initiator then creates the *Commit tree* for a transaction and passes it to the coordinator, which is responsible to maintain the tree in case of restarts.

Note that each sub-transaction is required to send a result message to the Initiator of the transaction after read-phase completion. However, a sub-transaction does not send a message to a sub-transaction that is a parent sub-transaction in the invocation hierarchy (with the exception of the parent being the Initiator).

Each Commit tree belongs to exactly one global transaction and stores the following variables:

1. the global transaction ID,
2. a tree structure containing Commit tree nodes,
3. a list unassignedN of unassigned nodes that correspond to sub-transactions T_c that have sent a result before their parent sub-transactions, i.e., the sub-transactions calling those transactions T_c, have sent the result, and
4. a list openSubTransactions of known transaction IDs, for which the result has not yet been received by the Initiator.

Furthermore, each Commit tree node stores

1. the sub-transaction ID of the sub-transactions T_c represented by this node,
2. the ID of the resource manager running the sub-transaction,
3. the caller transactionID of the parent sub-transaction, and
4. 0 or more IDs of invoked sub-transactions.

When the Initiator has ascertained that the Commit tree is complete, i.e., the Commit tree has an empty list openSubTransactions, it passes the Commit tree to the Commit coordinator. Based on the current status of the Commit tree and based on a timer, the coordinator sends the following messages to the participating resource managers:

1. voteRequest: when the Commit tree has been received from the Initiator, i.e., all results have arrived at the Initiator and the list openSubTransactions is empty,
2. doCommit when all participants have voted for commit,
3. doAbort: when at least one participant has voted for abort, and
4. doAdjourn: when after a time-out some votes are still missing.

4.1 An example of the coordinator's Commit tree

To ensure that all sub-transactions $T_1, \ldots, T_n$ invoked by a sub-transaction T_i are known to the coordinator, the initiator must process the result messages sent by each participant.

Figure 7 shows an example Commit tree. Each result message of a sub-transaction T_i includes the result data, the ID of T_i, the ID of the parent sub-transaction that has invoked T_i, a list of sub-transactions that have been invoked by T_i, the global transaction ID *TID* to which T_i belongs, and a sequence number. When the initiator receives the result of T_1, the node T_1 is created. Since the sub-transaction T_1 has invoked the sub-transactions T_2 and T_3, the vote for commit of the sub-transactions T_2 and T_3 is also required to commit the whole transaction. Therefore, these nodes are added

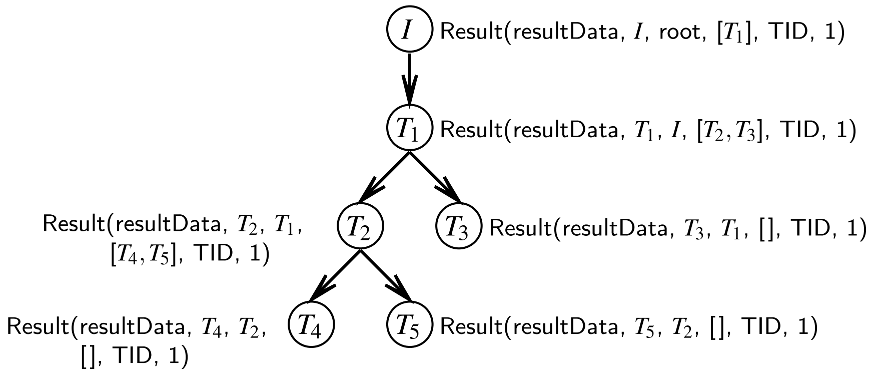
Result(resultData, I, root, [T_1], TID, 1)
Result(resultData, T_1, I, [T_2,T_3], TID, 1)
Result(resultData, T_2, T_1, [T_4,T_5], TID, 1)
Result(resultData, T_3, T_1, [], TID, 1)
Result(resultData, T_4, T_2, [], TID, 1)
Result(resultData, T_5, T_2, [], TID, 1)
I
T_1
T_2
T_3
T_4
T_5

Fig. 7 Commit tree example

to the Commit tree as well. The initiator builds this Commit tree dynamically and determines when all sub-transactions needed for starting the atomic commit protocol execution are finished. Since the information about invoked sub-transactions is sent along with a result message of the parent transaction and the parent's result can be received later than the child's result, it may be the case that a sub-transaction's result cannot be immediately assigned to a node connected in the Commit tree. In this case, the result is stored in a list of unassigned nodes and this node is connected in the Commit tree after the corresponding parent sub-transaction's result has arrived.

4.2 Commit tree modification by the result operation

Whenever a participant has successfully finished its read-phase of a sub-transaction T_i, the following result message is sent to the Initiator in order to invoke the initiator's RESULTRECEIVED method, which is described in Algorithm 1:

```
resultReceived(Object resultData,
  ID subtransactionID, ID callerID,
  ListOf(ID) invokedSubT,
  ID globalTID, int sequenceNr)
```

The optional parameter "resultData" contains T_i's result, while the "subtransactionID" indicates the ID of T_i. The callerID is the ID of the participant that has invoked T_i. The list "invokedSubT" contains the IDs of all sub-transactions invoked by T_i, while the "globalTID" is the ID of the global transaction to which T_i belongs. Furthermore, the result message contains a sequence number, which is increased if the sub-transaction is restarted and a second result message must be sent.

If the sub-transaction was not successful and has been aborted, the participant does not send a "resultReceived" message. Instead, it notifies the Initiator about this abort. Depending on the transaction's implementation, the Initiator might choose a different Web service to fulfill the global transaction.

Algorithm 1 outlines the implementation of the Initiator's resultReceived operation, which is executed on the Commit tree whenever a resultReceived message is

Algorithm 1 Implementation of resultReceived

```
1:  procedure RESULTRECEIVED(Object resultData, ID subtransactionID, ID callerID,
                                  ListOf(ID) invokedSubT, ID globalTID, int sequenceNr)
2:      if isPreVoteValid(sequenceNr) then
3:          markOutdatedAndAbortUnusedST(subtransactionID, callerID,
                                                        invokedSubT, globalTID);
4:          N :=createNode(subtransactionID, callerID, globalTID, invokedSubT)
5:          openSubTransactions.del(subtransactionID)
6:          if (ParentNode:=getNode(callerID)) == null then
7:              unassignedNodes.add(N)  ▷ If parent's result has not arrived yet, put back node
8:          else
9:              ParentNode.addChild(N)
10:             assignNodes(invokedSubT, N)      ▷ Try to assign nodes from unassignedNodes
11:         end if
12:         openSubTransactions.add(invokedSubT)
13:     end if
14: end procedure
```

received. First, the Initiator uses the sequence number to check that no newer message was processed earlier (line 2). This may be the case when sub-transactions are repeated and certain invocations must be repeated due to different invocation parameters (cf. Sect. 3.3). Therefore, a node N_{old} that represents the sub-transaction subtransactionID may already exist within the Commit tree, but is no longer valid. Thus, the procedure markOutdatedAndAbortUnusedST(...) marks N_{old} as outdated (line 3).

When a sub-transaction is repeated, its invoked sub-transactions may change. To identify sub-transactions that are no longer needed for the commit decision, the IDs of the sub-transactions invoked by N_{old} are compared with the actual invoked sub-transaction parameters invokedSubT of the new result message. Those sub-transactions that are invoked by N_{old} but not needed for commit anymore are aborted and deleted from the Commit tree (line 3).

After this, a new node N is created (line 4) and the parent-child relationships between N and the nodes representing other sub-transactions are managed (lines 4-11). In addition, a list openSubTransactions is updated where transactions are stored whose votes have not yet arrived (line 12).

If all results are present, the list openSubTransactions is empty and the atomic commit protocol, which requires votes for commit of all nodes in the Commit tree, can be started. After the resource managers sent their binding votes, the objects accessed by the transaction are blocked. To ensure that in case of a resource manager failure no infinite blocking of the other resource managers occurs, the coordinator starts a timer. If the time is over and some votes are missing, the coordinator sets the commit status stored in each node of the Commit tree to the Adjourn state, proposes the Adjourn state to T_i and to all sub-transactions belonging to T_i, and requests the votes for the sub-transaction once more.

4.3 Commit tree modification by repetition

If a sub-transaction T_r must be repeated as T_r', the result message with updated parameters must be sent to the Initiator and to the Commit coordinator, and the parameter "sequenceNr" must be increased.

The coordinator replaces the node for T_r with the updated parameters of T_r', and notifies sub-transactions that are not needed anymore, i.e. sub-transactions that were invoked by T_r, but have not been invoked by T_r'. Furthermore, the coordinator demands the votes of those sub-transactions that are additionally invoked by T_r'. Whenever a sub-transaction can be re-used instead of being repeated, the Commit tree does not need to be changed for the re-used sub-transaction.

4.4 Benefits of combining Commit tree and Adjourn state

The use of the Commit tree in combination with the Adjourn state shows several advantages for transaction processing. As the Commit tree allows to identify all sub-transactions that are invoked during transaction execution, it can be combined with dynamic transactional models like our Web service transactional model, which allows to invoke sub-transactions even during transaction execution.

In combination with the Adjourn state, the Commit tree allows participants to save energy and to speed up transaction processing for the following reason. Assume, for example, a concurrency conflict has occurred for a sub-transaction T_i. In this case, a repetition of T_i as T_i' including all of its invoked sub-transactions is necessary. However, if during the execution of T_i' some sub-transaction invocations are equal to those performed by T_i, we can re-use the invocations done by T_i and do not need to re-invoke them. This results in time and energy savings.

Furthermore, the Commit tree allows the transaction coordinator to identify and abort sub-transactions that are not needed anymore, for example if sub-transactions have been invoked by T_i but not by T_i'.

5 Experimental evaluation

We evaluated transaction processing in an unreliable mobile environment, which means, participants often disconnect for a short time and come back or, equivalent to this, a lot of messages are lost. Within such a scenario, the longer the blocking of a transaction is (blocking in terms of preventing other transactions from being committed) the greater the risk that another participant will disconnect and does not receive the voteRequest message. Therefore, it is more frequent that the coordinator cannot decide for commit, and that the coordinator must abort the transaction after a time-out than in fixed-wired networks. In our experiments, we especially focus on two parameters that influence transaction execution and that are characteristic for a mobile network: disconnections and message delay. The adjustment of other parameters such as transmitting power or movement models will finally affect these two parameters. Therefore, we identified "disconnection time and length" and "message delay" as the key parameters that influence the transaction execution. In other words, the more unreliable the network is, the more disconnections, message delays, and message losses occur.

For the simulation, we define a set of scenarios, which differ in both the network events such as disconnections and message delays and the spawned transactions. For each scenario, we start simulating transaction processing at the time when the global transaction is started, and observe the transaction execution until the time when the atomic commit protocol is invoked, i.e. when the coordinator demands the vote. In order to be able to compare the blocking state (which requires a time-out) and the Adjourn state (which does not require a time-out), we simulate the blocking state with various time-outs. We measure the number of successful transaction executions and the overall blocking time of both the Adjourn state and the blocking state. The concrete parameters of the generated scenario are described in the following subsection.

5.1 Simulation model

To simulate a varying reliability of our environment, we executed 7 simulations runs, which started when the sub-transaction was sent to each resource manager. We have simulated a logical clock in order to compare different approaches. One time unit corresponds to the shortest database operation (e.g., the shortest read phase). The used message delivery time (including routing, stack access, powering up wireless device for sending, etc.) is assumed to be fast, i.e. between 0.2 and 2 time units. Each of these 7 runs are stopped after 1000 time units plus additional 60 time units in order to let the time-out based protocols finish the last sub-transactions.

In each of these runs, we let 200 resources execute the same 135 global transactions. Each of these global transactions consists of 4 to 8 sub-transactions, resulting in 830 sub-transactions in total, where each sub-transaction uses exactly one resource.

Most of the sub-transactions have short read phases (randomly selected between 1 and 5 time units), but some transactions contain long read phases (randomly selected between 4 and 25 time units) that delay the transaction's commit.

As the number of committed transactions of atomic commit protocols for mobile networks primarily depends on the number of disconnections, we have focused on an evaluation based on disconnection of nodes. Thus, the simulated number of disconnections reflects the combination of other criteria like field size, movement speed, node density, and node sending power, which influence the number of disconnections. For example, the lower the number of nodes within a fixed field, the greater the number of disconnections.

The 7 runs differ in the number of disconnections of participants. A disconnected participant cannot communicate with other participants as long as the disconnection lasts. In the first run, we let no participant disconnect. In the second run, we randomly add 1000 disconnections to the participants (exponentially distributed). Each of these disconnections has a length of 25 to 50 time units. After 1000 time units, we stop the experiment and count the number of successfully committed transactions.

Predicting an optimal transaction time-out is difficult for the blocking state, thus, we simulated the blocking state using different transaction-time-outs (25, 50, 100, 250, 500, and 750 time units). Each participant starts its time-out after it has sent the result to the coordinator. When the time-out has expired and a participant has not received a voteRequest from the coordinator, the participant aborts the transaction in order to unblock the occupied resources. Furthermore, we measure the total blocking times of all participants.

5.2 Results

Figure 8 shows the transaction throughput for the Adjourn state and for the blocking state when validation is used. On the x-axis, the different simulation runs, which vary in the number of resource manager disconnections, are shown. Besides the Adjourn state, Fig. 8 shows different curves for the blocking state, each of which represents the used transaction time-out. For example, the curve "Block 100" describes a simulation run in which each participant aborts a transaction if the participant has not received the coordinator's voteRequest message for 100 time units after the first validation succeeded.

We have repeated each simulation run five times in order to see whether there is an impact by chance. As we have observed that chance has an impact only on short running experiments, we have chosen the duration of each experiment long enough to get repeatable results.

Our experiments confirm that setting up a time-out that maximizes the throughput is difficult and depends on the concrete network reliability. In contrast, the Adjourn state, which does not require such a time-out, shows an average throughput in reliable networks, but is superior to each time-out of the blocking state in unreliable networks.

Besides the transaction throughput, the blocking time is an important criterion for the concurrency control. Figure 9 shows the sum of the blocking times of all

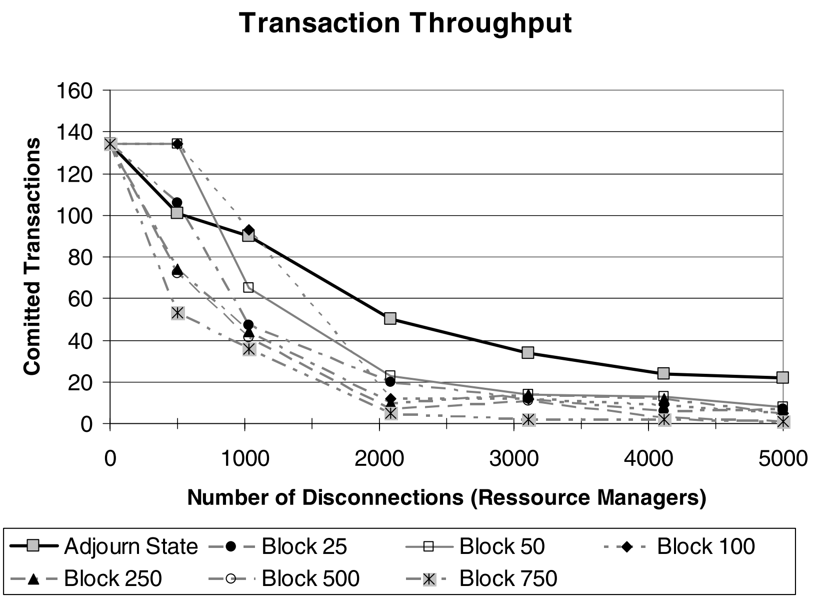

Fig. 8 Transaction throughput (Adjourn state and blocking state)

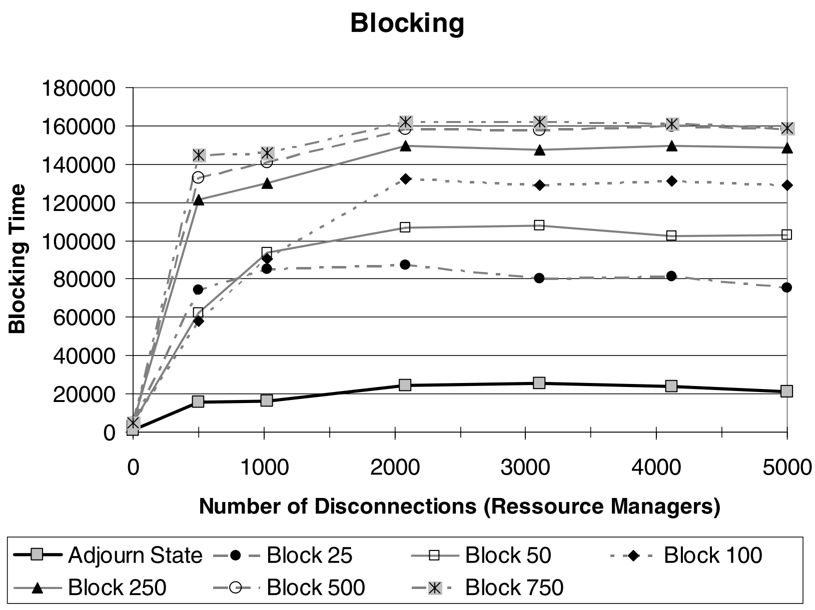

Fig. 9 Overall transaction blocking time (Adjourn state and blocking state)

resources for the Adjourn state and for the blocking state. We can see that the Adjourn state blocks the resources significantly less than each time-out of the blocking state. Again, the overall blocking time of using the blocking state highly depends on the concrete time-out value.

Although we have described solutions for both concurrency control schemes validation and locking, we have omitted the visualization of the results for locking as both concurrency control schemes lead to almost the same results. The reason for the identical behavior of locking and validation regarding transaction throughput and blocking is as described in Sect. 2.3: both concurrency control schemes lead to a blocking behavior on concurrent transactions *after* the successful completion of the read phase. Even concurrency schemes such as timestamp synchronization [25] suffer from the same problem of blocking after successful read phase completion. That is why we can expect similar evaluation results.

5.3 Evaluation summary

To summarize, our experimental results have shown that the Adjourn state concurrency control enhancement blocks remarkably less than using the traditional blocking state. Additionally, the Adjourn state achieves a significantly higher transaction throughput in unreliable networks with a lot of disconnections.

Furthermore, our tests have confirmed the difficulty in setting up a database time-out that increases the transaction throughput and reduces the amount of blocking.

As we argue in Sect. 2.3, the concrete concurrency control technique (e.g., locking or validation) has no influence on the blocking of distributed transactions after the read phase has been finished. Our experiments have proven this, as we got almost the same results regardless of whether validation or locking has been used as concurrency control technique.

This justifies the use of the Adjourn state even in mobile networks with moderate reliability, since Adjourn state protocols do not expose the user to the risk of setting up a "wrong" time-out that leads to a performance degradation.

6 Related work

Related work on distributed transaction processing can be classified according to the following two criteria: is the transaction execution flat or hierarchical/nested, and are transactions considered compensable?

6.1 Transaction invocation

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*. Today, Web services and their description languages (e.g. BPEL4WS [13] or BPML [3]) 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 [10] does. Note that our contribution is different from [10] in several aspects. For example, [10] has a "completion protocol" for registering at the Coordinator, but does not propose a non-blocking state—like our Adjourn state—to unblock transaction participants while waiting for other participants' votes. In addition, our Adjourn 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 [16]. Common with these transaction models, we have a global transaction and sub-transactions that are created during transaction execution and cannot be foreseen. However, our model allows sub-transactions to be removed if they are not invoked anymore by a newer execution of their parent sub-transactions.

Corba OTS [26, 31] 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 propose hierarchical commit decisions, since this implies that the upper nodes of the execution hierarchy must wait for the commit decision of all descendant nodes. In a mobile environment, where node failures are likely, our solution allows to spread the commit decision as fast as possible by flattening the invocation tree even before the atomic commit protocol starts.

6.2 Compensation

The main difference to other transactional models, e.g., [16, 32, 33], is that we consider all sub-transactions to be non-compensable for the following reason. Committed transactions can trigger other operations, thus, we cannot assume that compensation for committed transactions in mobile networks is always possible, since network partitioning makes nodes unreachable but still operational. When the compensation transaction will not reach the node, however, a model relying on compensation cannot give hard global atomicity guarantees as defined in [21]. Thus, we focus on a transaction model, within which atomicity is guaranteed for distributed, non-compensable transactions.

Models like [32] or [33] allow a transaction to define the level of consistency which the transaction leaves behind. In contrast, our solution does not need to adjust the level of consistency. We can guarantee atomicity without leaving states of inconsistency, even within mobile environments without base stations, where nodes that have consistent data may crash or permanently remain in a separated network partition.

The approaches [14] and [15] suggest the suspend state, which unblocks resources, as well. However, these approaches are 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. This means, our model assumes that a Coordinator cannot distinguish whether another node has failed or is still operational in another partition, and therefore compensation cannot be used. In addition, our transactional model is more powerful since it allows dynamically invoking Web service transactions, which must not be known in advance.

Another approach that relies on compensation of transactions is [35], which proposes 2PC optimizations, e.g. heuristics for committing transactions when messages are lost. However, inconsistencies may occur in case of network partitioning, for example, when some databases do not immediately receive the compensation decision or when the coordination process fails. Furthermore, the approach involves the difficulty of setting up time-outs as well.

Distributed transactions may also occur in the context of mobile agents (e.g., [12, 40]). In this context, the execution code is shipped to the resource managers. Our key ideas for guaranteeing atomicity of distributed transactions can be adapted to this context as well.

Other contributions focusing on agents, e.g., [28], propose to shift the coordination workload to fixed parts of the network by using *participant-agents*, which are executed on base stations and responsible for sending the votes and accepting the commit decision. However, our extension is also usable for completely mobile networks that do not have a fixed infrastructure.

While our solution reduces blocking of resource managers before the commit protocol starts, blocking that occurs during the atomic commit protocol execution is inevitable in asynchronous networks. This follows from contribution [37], which has proven that blocking during atomic commit protocol execution cannot be avoided if it cannot be determined whether a node has failed or is still working in another network partition. To enhance the Coordinator's availability of the atomic commit protocol, other contributions propose the use of protocols with more than one Coordinator. [34], for example, suggests the use of backup Coordinators in 2PC; [19] uses Paxos Consensus [24] to get a consensus on the commit decision; and [6, 8] allow "controlled failures" by proposing a combination of 2PC, 3PC [36], and Paxos Consensus. Since our Adjourn state affects the phase before the atomic commit protocol starts, it can also be combined with these protocols. Furthermore, [29] proposes a different termination technique called Bi-State-Termination, which allows concurrent transactions T_c to commit even if they are in conflict with a transaction T_b that waits for the coordinator's commit decision by executing T_c on two states, one representing T_b is committed, the other representing T_b is aborted. Again, as our Adjourn state effects only the phases before the commit protocol starts, it is compatible and can be combined with this approach.

6.3 Concurrency control

To omit locking, concurrency control mechanisms like multiversion concurrency control [4, 39], timestamp-based concurrency control [25], or optimistic concurrency control [20, 23] have been proposed. However, these approaches do not solve the problem of setting up time-outs when the database has to abort a transaction. Our proposed Adjourn state does not rely on such time-outs, and merges nicely with these concurrency control mechanisms since it is an "on demand" strategy for giving concurrent transactions access to resources that have been used by transactions which are still waiting for the commit protocol to be invoked.

Reference [27] proposes a concurrency control schema that combines multiversion concurrency control with a timestamp mechanism for transaction processing in mobile environments containing both mobile devices and fixed-wired databases. It uses an additional lock type called "verified-lock", which is a special lock type issued by the mobile devices for handling transaction termination at the fixed-wired databases. In contrast, our solution aims at a completely mobile environment and does not need to rely on these fixed-wired databases.

Compared to our previous contribution [7], the Adjourn state proposed in this paper is developed for the combination of optimistic concurrency control and atomic commit protocols. Furthermore, we give experimental results and discuss the concurrency control mechanism. Since the Adjourn state can be combined with a dynamic transactional model, the Coordinator must get to know the participating sub-transactions. An approach that allows the Coordinator to keep track of all dynamically invoked sub-transactions is described in [9].

Our approach is based on the same optimistic principle as [1]. However, the Adjourn state differs from [1] as the Adjourn state does not block resources while a transaction has sent the result of its read phase to the Initiator and waits for the vote command.

7 Summary and conclusion

In this article, we have introduced a Web service transaction model suitable for mobile networks, which allows a Web service to dynamically invoke other Web services to fulfill its own service. We have discussed two concurrency control mechanisms for this Web service model, locking and validation, and we have demonstrated the blocking effect that both concurrency control schemes involve. Furthermore, we explained the difficulty in setting up time-outs for aborting transactions.

Our solution, the Adjourn state, does not rely on time-outs and allows to repeat Web services in case of concurrency failures. Furthermore, we have shown an optimization for Web service repetition that, under certain conditions, can re-use Web service calls instead of repeating them. The Adjourn state is mainly designed for wireless and mobile environments in which the transaction load is moderate and the number of disconnections high. If this applies to certain instances of wired systems as well, these systems may also benefit from our protocol, as it also works in fixed-wired environments.

We have described the Commit tree, which allows the transaction's Commit coordinator to keep track of the commit-status of all participating sub-transactions. Furthermore, we have shown how the Commit tree can be used to unblock participants by migrating them back to the Adjourn state if some participants must repeat their Web service.

We have evaluated our proposed scheme experimentally using simulation. Our experiments have proven the difficulty that traditional protocols involve when setting up time-outs for mobile networks with unpredictable reliability. Our experiments have also demonstrated that using the Adjourn state in unreliable environments can lead to an increased transaction throughput of committed transactions of a factor up to 2.5 compared to the use of traditional time-outs. Furthermore, the Adjourn state significantly reduces the amount of data blocking.

In the future, we plan to combine the Adjourn state with protocols using multiple coordinators.

References

1. Al-Houmaily, Y., Chrysanthis, P.K., Levitan, S.P.: An argument in favor of the presumed commit protocol. In: Proceedings of the 13th International Conference on Data Engineering, pp. 255–265 (1997)
2. Al-Houmaily, Y.J., Chrysanthis, P.K.: 1-2pc: the one-two phase atomic commit protocol. In: Proceedings of the 2004 ACM Symposium on Applied Computing, Nicosia, Cyprus, March 14–17, pp. 684–691 (2004)
3. Arkin, A., et al.: Business process modeling language, bpmi.org. Final draft, BPMI. org (2002)
4. Bernstein, P.A., Goodman, N.: Multiversion concurrency control—theory and algorithms. ACM Trans. Database Syst. **8**(4), 465–483 (1983)
5. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
6. 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)
7. Böttcher, S., Gruenwald, L., Obermeier, S.: Reducing sub-transaction aborts and blocking time within atomic commit protocols. In: 23rd British National Conference on Databases (BNCOD), Belfast, Northern Ireland, UK, pp. 59–72 (2006)

8. Böttcher, S., Gruenwald, L., Obermeier, S.: A failure tolerating atomic commit protocol for mobile environments. In: Proceedings of the 8th International Conference on Mobile Data Management (MDM 2007), Mannheim, Germany (2007)
9. Böttcher, S., Obermeier, S.: Dynamic commit tree management for service oriented architectures. In: Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS), Funchal, Madeira, Portugal (2007)
10. 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)
11. Chrysanthis, P.K., Samaras, G., Al-Houmaily, Y.J.: Recovery and performance of atomic commit protocols in distributed database systems. In: Kumar, V., Hsu, M., (eds.) Performance of Database Recovery Mechanism, pp. 370–416. Prentice Hall, New York (1998)
12. Covaci, S., Zhang, T., Busse, I.: Java-based intelligent mobile agents for open system management. In: Proceedings of the 9th International Conference on Tools with Artificial Intelligence (ICTAI '97), p. 492. IEEE Computer Society, Washington (1997)
13. Curbera, F., Goland, Y., Klein, J., Leymann, F., et al.: Business process execution language for web services. v1.0. Tech. rep., BEA, IBM, Microsoft (2002)
14. Dirckze, R.A., Gruenwald, L.: A toggle transact. management technique for mobile multidatabases. In: CIKM '98, pp. 371–377. ACM Press, New York (1998). doi:10.1145/288627.288679
15. Dirckze, R.A., Gruenwald, L.: A pre-serialization transact management technique for mobile multidatabases. Mob. Netw. Appl. **5**(4), 311–321 (2000). citeseer.ist.psu.edu/dirckze00preserialization.html
16. Dunham, M.H., Helal, A., Balakrishnan, S.: A mobile transaction model that captures both the data and movement behavior. Mob. Netw. Appl. **2**(2), 149–162 (1997). citeseer.ist.psu.edu/article/dunham97mobile.html
17. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM **19**(11), 624–633 (1976). doi:10.1145/360363.360369
18. Gray, J.: Notes on data base operating systems. In: Flynn, M.J., Gray, J., Jones, A.K., et al. (eds.) Advanced Course: Operating Systems. Lecture Notes in Computer Science, vol. 60, pp. 393–481. Springer, Berlin (1978)
19. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (2006). doi:10.1145/1132863.1132867
20. Härder, T.: Observations on optimistic concurrency control schemes. Inf. Syst. **9**(2), 111–120 (1984). 10.1016/0306-4379(84)90020-6
21. Kifer, M., Bernstein, A., Lewis, P.M.: Database Systems: An Application Oriented Approach. Pearson Addison-Wesley, Reading (2005)
22. Kumar, V., Prabhu, N., Dunham, M.H., Seydim, A.Y.: Tcot-a timeout-based mobile transaction commitment protocol. IEEE Trans. Commun. **51**(10), 1212–1218 (2002). doi:10.1109/TC.2002.1039846
23. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. **6**(2), 213–226 (1981). doi:10.1145/319566.319567
24. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998). doi:10.1145/279227.279229
25. Leu, P.J., Bhargava, B.K.: Multidimensional timestamp protocols for concurrency control. In: Proceedings of the Second International Conference on Data Engineering, pp. 482–489. IEEE Computer Society, Washington (1986)
26. Liebig, C., Kühne, A.: Open source implementation of the CORBA object transaction service. http://xots.sourceforge.net/ (2005)
27. Madria, S.K., Baseer, M., Kumar, V., Bhowmick, S.S.: A transaction model and multiversion concurrency control for mobile database systems. Distrib. Parallel Databases **22**(2–3), 165–196 (2007)
28. Nouali, N., Doucet, A., Drias, H.: A two-phase commit protocol for mobile wireless environment. In: Williams, H.E., Dobbie, G. (eds.) Sixteenth Australasian Database Conference (ADC2005). CRPIT, vol. 39, pp. 135–144. ACS, Newcastle (2005)
29. Obermeier, S., Böttcher, S.: Avoiding infinite blocking of mobile transactions. In: Proceedings of the 11th International Database Engineering & Applications Symposium (IDEAS), Banff, Canada (2007)
30. Obermeier, S., Böttcher, S., Hett, M., Chrysanthis, P.K., Samaras, G.: Adjourn state concurrency control avoiding time-out problems in atomic commit protocols (poster). In: Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE), Cancun, Mexico (2008)
31. Object Management Group: Trans. service spec. 1.4. http://www.omg.org (2003)

32. Pitoura, E., Bhargava, B.K.: Maintaining consistency of data in mobile distributed environments. In: International Conference on Distributed Computing Systems, pp. 404–413 (1995). citeseer.ist.psu.edu/pitoura95maintaining.html
33. Rakotonirainy, A.: Adaptable transaction consistency for mobile environments. In: DEXA Workshop, pp. 440–445 (1998). citeseer.ist.psu.edu/410658.html
34. Reddy, P.K., Kitsuregawa, M.: Reducing the blocking in two-phase commit with backup sites. Inf. Process. Lett. **86**(1), 39–47 (2003)
35. Samaras, G., Britton, K., Citron, A., Mohan, C.: Two-phase commit optimizations in a commercial distributed environment. Distrib. Parallel Databases **3**(4), 325–360 (1995). doi:10.1007/BF01299677
36. 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, pp. 133–142. ACM Press, New York (1981)
37. Skeen, D., Stonebraker, M.: A formal model of crash recovery in a distributed system. In: Berkeley Workshop, pp. 129–142 (1981)
38. Ullman, J.D.: Principles of Database Systems, 2nd edn. Computer Science Press, New York (1982)
39. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann, San Francisco (2001)
40. Ye, D.Y., Lee, M.C., Wang, T.I.: Mobile agents for distributed transactions of a distributed heterogeneous database system. In: DEXA 02, pp. 403–412. Springer, London (2002)