Abstract. Supporting atomicity of multi-site transactions in a distributed transaction management system is equated with long-duration delays, blocking, and loss of autonomy of the individual sites. The two-phase Commit protocol embodies these problems. The focus of this paper is on an alternative notion of relaxed atomicity where these difficulties are alleviated. Relaxed atomicity is characterized by an asynchronous process of recovering from uncoordinated local decisions as to whether to commit or abort a multi-site transaction, and finally coercing a unanimous outcome. The consequent atomicity notion is weaker than the standard all-or-nothing atomicity, as transactions with discrepant commit decisions are recovered semantically rather than physically. A formal model that unifies the two dual methods of semantic recovery, namely compensation and retry, is constructed. Due to the asynchrony introduced to the commit procedure, non-atomic executions of transactions occur, and it is required to isolate them from other transactions until they are semantically recovered. The required isolation property is defined, and a protocol that satisfies this property is presented.

1 Introduction

A cornerstone of the transaction concept is the notion of atomicity, which states that a transaction either completes entirely, or it is guaranteed to have no visible effect. A transaction that completes successfully is said to commit, otherwise the transaction aborts.

When transactions access data items at multiple sites of a distributed system, atomicity is accomplished by employing an atomic commit protocol. The two-phase commit (2PC) protocol [Gra78] is the most common atomic commit protocol, and it is widely used in distributed transaction management systems. In this protocol, a multi-site transaction is associated with a coordinator that gathers votes from the participating sites as to whether to commit or abort the transaction. Based on these votes, the coordinator makes a decision and transmits it to the participating sites. The receipt of this decision message must precede the release of all resources held by the transaction that is involved in the 2PC protocol. Consequently, all other transactions contending for the resources held by the transaction in question must wait until the decision message has been received. Since the 2PC protocol involves three rounds of messages (request for vote, vote and decision) the delay can be intolerable.

It is well known that there is no atomic commit protocol that is not blocking, in a distributed system that
is subject to (fail-stop) site failures, and failures in communication links [Ske82, BHG87]. Blocking is the undesirable phenomenon where transactions may be delayed for unbounded periods. In the context of 2PC, blocking implies that if the coordinator for a transaction, or a communication link to that coordinator, fails in a certain critical time, some other transactions at active sites are delayed until the failure is repaired.

Another severe difficulty arises when atomic commitment is considered in the context of multidatabase systems where several sites are integrated to create a cooperative environment (see [hdb90] and the references there). The goal of the integration is to support global transactions by dividing them into local subtransactions that are executed at the different sites. In a multidatabase, the individual sites comprising the integrated system may use heterogeneous database management systems. The sites may belong to distinct, and possibly competing business organizations (e.g., competing computerized reservation agencies). In such systems the local autonomy of the individual sites is crucial. It is undesirable, for example, to use a protocol where a site belonging to a competing organization can intentionally or innocently block the local resources. One of the flavors of local autonomy is defined as the capability of a site to abort any local (sub)transaction at any time before the (sub)transaction terminates. Employing the 2PC protocol, a site enters a prepared state if it votes to commit a transaction $T$ [Gra78]. Once in this state, a site becomes a subordinate of the external coordinator, and it can no longer unilaterally determine the fate of the local subtransaction of $T$. Therefore, local autonomy is sacrificed once a 2PC protocol is imposed on the integrated sites.

There has been some work to alleviate these problems by relaxing the guarantees of atomicity. Prominent previous proposals in the direction of relaxed atomicity are sagas [GM83, GMS87], and their generalization—multitransactions [GMGK+90]. These proposals do not consider the atomicity problem in distributed setting, but rather the similar problem of atomicity of long-duration transactions. Essentially, the idea in these proposals is to decompose the coarse unit of a transaction into finer subtransaction units. Subtransactions commit or abort independently, without coordination with other subtransactions of the same transaction. Resources held by the subtransactions are released as soon as the subtransaction terminates, without waiting for the termination of the entire transaction. Therefore, atomicity of the whole transaction is given up for a weaker notion referred to as semantic atomicity [GM83]. Each subtransaction, $T_i$, is associated with a compensating subtransaction, whose task is to undo semantically, rather than physically, the effects of $T_i$ in case the entire transaction aborts. Intuitively, compensating for $T_i$ can be thought of as performing (an approximation of) the inverse of the function performed by $T_i$, rather than physically restoring data items to their values prior to $T_i$‘s execution. Instead of the standard all-or-nothing atomicity, semantic atomicity guarantees that either all subtransactions commit—and then the entire transaction commits, or that all subtransactions that committed locally are compensated-for, if the entire transaction is to abort.

The work in this direction of non-atomic transactions, as opposed to the traditional atomic transactions, has
not matured yet. The precise guarantees of such transaction models have not been examined to date. In particular, the attractive idea of using relaxed atomicity in a distributed setting has not been carefully examined. Due to the asynchrony introduced into the commit procedure, incomplete executions of non-atomic transactions occur. The most severe unrecognized problem arises when such executions become visible, thereby affecting other transactions. Isolating non-atomic executions means that a transaction should not be affected by both failed (or compensated-for) and successful subtransactions of the same transaction. (Being affected by a successful subtransaction that is later compensated-for is permitted, though). We refer to this requirement as isolation of recoveries and illustrate its importance with the following example. Consider a non-atomic transaction $T_1$ composed of two subtransactions: $T_{11}$ updating data item $x$ at one site, and $T_{12}$ updating data item $y$ at a different site. Further assume that $T_{11}$ succeeds whereas $T_{12}$ fails. This non-atomic execution of $T_1$ leaves the relation between $x$ and $y$ in an inconsistent state, a state that could not have been reached had there been no transaction failures. A transaction reading both $x$ and $y$ is exposed to this inconsistency.

Often, aborting a long-running transaction that consumes many resources may be too costly. Instead, it is preferable to retry the failed subtransactions and recover the transaction forward rather than backward. Additionally, forcing a global commit by retrying the failed subtransactions is a necessity when an autonomous site does not support a prepared state (see [BST90] for an example). Therefore, in a dual manner to compensation, it is useful to incorporate retry for recovery of non-atomic transactions. The duality of retry and compensation is rooted in the traditional redo/undo paradigms.

In this paper we present a formal model of non-atomic transactions that encompasses both compensation and retry. An abstraction that unifies the two dual methods under the same framework is constructed (Section 2). Section 2.2 discusses the important concepts of compensation and retry in more detail. The semantic atomicity of [GM83], which is based solely on compensation is a special case in our model. The problem of recovery isolation is cast in our model as a correctness criterion for evaluating non-atomic executions (Section 3). A protocol satisfying this criterion is presented and proved correct (Section 4). This protocol is a combination of a commit protocol similar in structure and message complexity to the 2PC protocol, and a marking algorithm. Section 5 includes several practical extensions and applications of the idea of relaxed atomicity.

2 Non-Atomic Transactions

For brevity, a non-atomic, multi-site transaction is referred to just as a transaction. A standard transaction is referred to as an atomic transaction.

A distributed database is a set of disjoint databases, where each database is a set of data items that is associated with a site.

A step is an atomic transaction that consists of a totally ordered sequence of accesses to data items at a single site. The success or failure of a step (i.e., whether it committed or aborted) is denoted by a binary polarity. There are two kinds of steps; forward steps and
recovery steps. Each forward step is associated with a recovery step. We use \( p, q, o \) to denote forward steps and \( r_p, r_q, r_o \) to denote their respective recovery steps. When we refer to either a recovery or forward step, we use \( s, t \). The notation \( s^b \) (where \( b \) is either 0 or 1) denotes that \( s \) has polarity \( b \). A polarity opposite to \( b \) is denoted \(~b\). If a recovery step, \( r_p \), succeeds then \( r_p \)'s polarity is opposite to \( p \)'s polarity. Otherwise, if \( r_p \) fails, the polarities of \( p \) and \( r_p \) are identical.

We assume the following regarding the interleaving of accesses to data items:

- Serializability and Strictness. The schedules of accesses to data items are serializable and strict \([BH87]\) at the step level. Strictness means that a step does not access a data item \( x \) before the previous step to access \( x \) terminates (commits or aborts).

Strictness is needed so that a step \( p \) is assigned a polarity before subsequent steps access the data items accessed by \( p \). The necessity of this requirement becomes clear later.

Steps accessing at least one data item in common are said to be conflicting. A recovery step accesses at least all data items accessed by its forward step. Therefore, a forward step and its recovery step conflict.

A transaction \( T_i \) is a partial order with ordering relation \( <_i \) where

- the elements of \( T_i \) are a fixed set of forward steps; and

- there is at most one step of \( T_i \) at a particular site.

We subscript step names to denote the transactions they belong to, or the transaction they correspond to in case of recovery steps. For example, \( p_i \) is a forward step of transaction \( T_i \), \( r_p_i \) is its corresponding recovery step, and \( s_i \) denotes either a forward step of \( T_i \) or a corresponding recovery step. Regardless of whether \( s_i \) is a forward or a recovery step, we say that \( s_i \) is a step of \( T_i \).

Observe that there are no intra-transaction conflicts among steps (except for the conflicts between steps and their recovery steps). This is because a transaction may have only one step at a particular site, and data items at different sites are disjoint.

We envision that a control structure in the form of a program controls the invocation of steps of the same transaction at the different sites. This issue, however, is orthogonal to our concerns, and we model a transaction as a static partial order of steps. An ordering among two steps of the same transaction is called a dependency ordering. These dependency orderings model the logical precedence, flow of information, causality and synchronization constraints among steps of the same transaction that are imposed by its program. Regardless of the actual type of dependency modeled by \( p_i <_i q_i \), we say that \( p_i \) invokes \( q_i \).

### 2.1 Commit Protocol

A site decides whether a local forward step commits or aborts without coordination with other sites executing steps on behalf of the same transaction. Once this decision is made, all the local resources the step holds are released at once. A centralized beacon initiates a commit protocol by requesting these decisions from the sites that executed steps on behalf of the to-be-committed
transaction. The decisions are cast as votes in the first phase of the commit protocol. At a site, the request for a vote may be received from the beacon after the corresponding step has already terminated. The beacon gathers the votes and decides whether to commit or abort the transaction according to a decision rule whose nature is explained shortly. This global decision is conveyed to the different sites in a second phase of the commit protocol. In case of a discrepancy between a local decision and the global decision, a recovery step is executed at the local site. Namely, if the local decision was commit and the global one is abort, then the local step is compensated for. Conversely, a local step is retried if the global decision is commit and local decision was abort. The recovery steps ensure convergence to a unanimous outcome at all sites despite the uncoordinated local decisions. The concepts of compensation and retry are elaborated in Section 2.2.

Any rule governing the decision making by the beacon must conform to the unanimity requirement:

- **Unanimity.** If all votes are identical then the decision must be unanimous with the votes.

A decision rule can be either biased or balanced. In standard atomic commit protocols, the following biased decision rule is used:

- **Biased Decision.** If at least one of the sites votes to abort, then the decision is abort.

A balanced rule can be based on quorum, majority or other principles that conform with the unanimity requirement. For instance, a transaction may be considered successful if a certain subset of its steps succeed. Also, the invocation of a step may be triggered by the failure of another step in that transaction, and hence a biased decision is not applicable [KR88]. The specifics of the balanced decision rule are abstracted from our discussion. The main distinction between the two rules is the possibility of reversing a local abort decision by a retry step. This option is lacking in the biased case, and is present in the balanced case.

### 2.2 Semantic Recovery

Recovery by compensation or retry is a semantic recovery activity, in contrast to the standard recovery methods of undo and redo [BHG87] which are physical recovery methods. Using the standard methods, from the point at which the forward step terminates until the point recovery is performed, no other steps may access the data items updated by the forward step. Therefore, undoing or redoing the forward step amounts to restoring the physical before or after images, respectively, of the relevant data items. By contrast, during the interval between termination and semantic recovery, locks are not held on data items updated by the forward step. Consequently, subsequent steps may update these data items. Therefore, physical recovery is impossible as it would obliterate the effects of steps executed after the forward step and prior to the recovery step.

A recovery step does not exist by its own right; it is always regarded within the context of the forward step, and it is always executed after the forward step. A recovery step is driven by a program that is a derivative of the program of the forward step. The binding of a forward and a recovery step is explicit, and is realized as the recovery step gets as input a trace of the execution of
the forward step (in the form of the latter’s log records, for example).

We do not support in our model the recovery of recovery steps, since that would only distract us from the real issues, and since it can be guaranteed, with very high likelihood, that compensation and retry steps complete their execution and do not fail (see below).

It is acknowledged that designing compensating and retry steps is a complex task that relies heavily on the semantics of the application at hand [KLS90]. Semantic recovery is not applicable when steps cannot be compensated-for, or cannot be retried. Using relaxed atomicity in the context of these cases is discussed in Section 5.

2.2.1 Compensating Steps

A compensating step undoes the effects of its forward step semantically, without causing the cascading aborts [BHG87] of steps that follow the forward step. The intention of compensation is to leave the effects of transactions that follow the forward step intact, yet preserve database consistency. Ideally, a compensating step can be thought of as performing the inverse of the function performed by the forward step. However, steps may not always commute and therefore, compensation for a step \( p_i \) does not guarantee the physical undoing of all the direct and indirect effects of \( p_i \). Compensation does guarantee, however, that a consistent state is established based on semantic information. We emphasize that the state of the database after compensation took place may only approximate the state that would have been reached, had the forward step never executed. In [KLS90] we formally characterize the outcome of compensation based on the properties of the forward step and on properties of steps that follow it in the execution.

To illustrate the principles of compensation consider the following example. As part of a multi-site transaction \( T_1 \), a step at site \( S_1 \) made a successful reservation for a certain number of seats in a certain flight. \( S_1 \) votes to commit \( T_1 \) and releases its locks on the flight reservation information. It happens that a subsequent reservation by \( T_2 \) is rejected since \( T_1 \) reservations booked the last available seat. Finally, \( T_1 \)'s beacon decides to abort \( T_1 \) and cancel its reservations by executing a compensating step at site \( S_1 \). Had the reservation of \( T_2 \) executed alone, without \( T_1 \)'s forward and compensating steps, it would have resulted in successful reservations. Thus, the state after compensation took place is indeed consistent (i.e., there is no over-booking); however, it differs from the state resulting from an execution of \( T_2 \)'s reservation on its own.

Compensating steps cannot voluntarily abort, nor are they subject to a system-initiated abort. Also, their completion is guaranteed despite system crashes by either resuming them from a save-point, or retrying them. Finally, a compensating transaction must be designed to avoid a logical error leading to abort. Consequently, it is guaranteed that, once compensation is initiated, it completes successfully. This stringent requirement is recognized in [GMS87, GM83, Vei89, GMGK+90, Rei89, KR88, KLS90]. The rationale behind this requirement is preserving some sense of atomicity. Initiating a compensating step is caused by the beacon's decision to abort the transaction to which the step belongs. In order to maintain at least relaxed atomicity, we claim this decision to be non-reversible. Requiring a com-
pensating step to succeed unconditionally implies that design of a compensating transaction is a complex and application-dependent task. The fact that the compensating step always executes after its forward step must be used to alleviate this difficulty. Essentially, the forward step should record enough information (e.g., as log records) for the recovery step to execute properly.

2.2.2 Retry Steps

Retry is initiated based on the premise that the forward step has failed. There are few possible reasons for a step failure:

- The site executing the step crashes.
- The step was aborted intentionally, in order to resolve a deadlock, or for other reasons.
- A logical error in the step's code led to its abort (e.g., division by zero, attempt to violate integrity constraint).

The most simple form of retrying a step is re-executing its program. Following the first two types of failures, re-execution of the step can either fail or succeed. In case of a logical error that is state-dependent, the error may occur again depending on the state of the database during the re-execution. Therefore, regardless of the cause of the failure, we cannot require a retry by re-execution to succeed unconditionally as was required for compensation.

A more sophisticated retry step can examine the log records of the forward step and determine the cause for the failure. Based on this analysis, the retry step may take the correct actions, thereby increasing its probability to succeed. Such a retry mechanism is similar to an exception handler whose actions are determined by the type of the failure. In addition, a retry step may invoke contingency actions if the failure analysis leads to the conclusion that mere re-execution is futile. If the semantics of a particular forward step are such that the unconditional success of this exact step is crucial to the success of the entire transaction, then retry should not be considered as a recovery option for such a step (see Section 5).

2.3 Executions

A complete execution $E$ over a set of transactions $T = \{T_1, \ldots, T_n\}$ is a partial order with ordering relation $\prec_E$ where

$$E = \bigcup_{i=1}^{n} T_i \cup rec,$$

where

$$rec \subseteq \bigcup_{i=1}^{n} \{rp_i | p_i \in T_i\}$$

That is, $E$ consists of the transaction steps in $T$ and recovery steps for a subset of these steps.

- Each step in $E$ has a polarity. Polarities are used below to encode the fate of a transaction in an execution.
  - $\prec_E \supseteq \bigcup_{i=1}^{n} \prec_i$.
  - For any two conflicting steps $s_i$, $t_j$, either $s_i \prec_E t_j$ or $t_j \prec_E s_i$.
  - For any pair $p_i$, $rp_i$, of forward and recovery steps, $p_i \prec_E rp_i$.

An execution is a prefix of a complete execution.

The fate of a transaction $T_i$ in an execution $E$ is the union of the set of the polarities of $T_i$'s recovery steps.
in $E$, with the set of polarities of steps of $T_i$ that have no recovery steps. Formally:

$$fate(T_i, E) = \{ b \mid rp_i^b \in E \} \cup \{ b \mid (rq_i^b \notin E) \}$$

A transaction $T_i$ has a unanimous fate in an execution $E$ if $| fate(T_i, E) | = 1$. That is, all polarities considered in the construction of $fate(T_i, E)$ are identical. This polarity is referred to as the fate of $T_i$, if indeed $T_i$ has a unanimous fate.

An execution $E$ is semantically atomic (SA) if for each transaction $T_i$:

- There is at least one step, $p_i \in T_i$ that has no recovery step in $E$; and
- $T_i$ has a unanimous fate.

Observe that if a recovery step fails, the execution does not preserve semantic atomicity since the transaction to which the step belongs cannot have a unanimous fate. We note without proof that a commit protocol (structured as prescribed in Section 2.1) that satisfies the unanimity condition ensures that eventually executions are SA. (Compliance with the unanimity condition satisfies the first requirement in the definition of an SA execution).

The property of semantic atomicity is not prefix-closed. Consequently, as a non-SA execution progresses and more recovery steps are executed it may become SA.

### 3 Isolation of Recoveries

The concept of atomicity is intended to mask failures by creating a virtual failure-free system in a failure-prone environment. When relaxing atomicity, as was done in the previous section, we must make sure that failures do not become visible. In standard transaction models, visibility is modeled by the reads-from conflict relation. (see [BHG87] for the exact definition). Since we do not deal with reads and writes, every conflict among steps of different transactions represents the fact that the effects of the preceding step, $t_j$, are visible to the subsequent step, $p_i$. Recall that causality and logical precedence are modeled by the dependence orderings within a transaction. Hence, it is appropriate to model the further propagation of the effects of $t_j$ within $T_j$ along these orderings. The notion of propagation and visibility of effects is made formal by defining the follows relation.

Let $\rightarrow$ denote the transitive closure of the $<$ relation. A forward step $p_i$ follows a step $t_j$ ($i \neq j$) in an execution $E$ if:

- $t_j < E p_i$, and there is no $s_k$ such that $t_j < E s_k < E p_i$. Or, if
  - $q_i$ follows $t_j$, and $q_i \rightarrow p_i$.

As long as a transaction does not have a unanimous fate, its execution is incomplete and its failure may be visible to other transactions. We must isolate transactions with non-unanimous fate until all the recovery steps are executed and semantic atomicity is obtained.

An execution isolates recoveries if there is no forward step of a particular transaction that follows steps of opposing polarities of another transaction. Formally:

Let $e_j^T$ and $e_j^F$ be two steps of $T_j$ that have opposing polarities in an execution $E$. Then an execution $E$ isolates recoveries (IR) iff whenever $p_i$ follows $s_j$, then $p_i$ does not follow $t_j$.

In an IR execution it is possible to follow a forward
step, p before a successful rp actually reverses p’s effects. The class of IR executions excludes, however, executions in which a global state resulting from an incomplete execution of a transaction is observed by other transactions. Thus, a notion of virtual atomicity is enforced.

To illustrate, consider the sample executions depicted in Figure 1. In this figure a step of transaction Ti executing at site sj is denoted Tij. The notation CTij denotes the compensating step specific to the forward step Tij. The <E relation is depicted by arrows. In this figure all the executions are not IR.

The example in Figure 1(a) is of particular interest. By following CT12, T22 “knows” that the beacon of T1 has decided to abort T1. On the other hand, the effects of T11 are visible at site S1, and thus affect T21 and transitively T22. T22 is exposed to the asynchrony in the process of recovering T1, and consequently may observe an inconsistent state.

4 The Polarized Protocol

In this section, we present a protocol, called the polarized protocol, that ensures that executions isolate recoveries. The protocol is executed by schedulers at each site that collectively ensure the IR property. The protocol implements the ‘follows’ relation which is crucial to the generation of IR executions. It does so by marking data items with polarities of steps that access them, and propagating these markings along conflict and dependency orderings.

First, we present the general polarized protocol which applies regardless of the type of decision rule the beacon employs. The protocol, expressed as a set of rules, and some explanatory comments are included in Section 4.1. Section 4.2 focuses on the problem of discarding markers. A correctness proof is presented in Section 4.3 and an optimization for the simpler case of a biased decision is presented in Section 4.4

4.1 The Protocol

The following type and data structures are used in the protocol:

- A marker is an ordered pair (i, b), where i is a transaction identifier and b is a polarity of a step within that transaction.
- For each data item x the protocol maintains the following set:
  access(x), the set of all markers (i, b) such that x is accessed by a step Pj.
- For each step Pj the protocol maintains the following set:
  follow(Pj), the set of all markers (i, b) such that Pj follows qj.

Initially, for all data items x, and for all steps Pj, access(x) = ∅ and follow(Pj) = ∅, respectively.

Regarding the first rule below, the set subtraction is effective only for a successful recovery step that removes
the marker of its corresponding forward step. The second rule propagates markers only in cases of conflicts among steps. Dependence orderings may not be based on data conflicts, but still take part in 'follows' chains. The third rule takes care to reflect these dependency orderings in the follow sets. Since dependency orderings are inter-site, this rule implicitly assumes the communication needed for the remote invocation.

1. Marking. Whenever a forward or a recovery step \( s_i \) terminates with polarity \( b \), then for all data items \( z \) accessed by \( s_i \):
   \[
   \text{access}(z) := (\text{access}(z) - \{(i, \bar{b})\}) \cup \{(i, b)\}
   \]

2. Access and Propagation. Whenever a step \( p_i \) requests access to data item \( x \):
   \[
   \text{if } (\exists j : (j, b) \in \text{follow}(p_i) \land (j, \bar{b}) \in \text{access}(x)) \text{ then reject } p_i's \text{ request}
   \]
   \[
   \text{else } \text{follow}(p_i) := \text{follow}(p_i) \cup \text{access}(x)
   \]

3. Propagation by Dependence. Whenever a step \( p_i \) requests to invoke a step \( q_i \):
   \[
   \text{if } (\exists j : (j, b) \in \text{follow}(p_i) \land (j, \bar{b}) \in \text{follow}(q_j)) \text{ then reject } p_i's \text{ request}
   \]
   \[
   \text{else } \text{follow}(q_i) := \text{follow}(q_i) \cup \text{follow}(p_i)
   \]

Executions that are not IR are excluded by checking the conditions of the second rule and the third rule. A step is prevented from accessing data items marked by markers of the same transaction with opposing polarity. If a step attempts to access a data item \( x \) that would violate this condition, the access request is rejected in the second rule. Rejection implies that the step can either fail or be delayed. Delaying is useful only if it is possible that \( x \)’s offending marker will be removed. Such a removal may occur when a successful recovery step replaces the offending marker with the opposite marker as prescribed in the first rule. Dependency orderings that violate the IR criterion are similarly rejected in the third rule.

Observe that the rules of the protocols check local conditions and prescribe local actions, and hence, a local scheduler can implement the protocol without additional inter-site communication.

4.2 Discarding Markers

For the condition checking in rules 2 and 3 to be performed fast and for space efficiency, it is required to keep the \( \text{access}(x) \) sets finite. Therefore, it is necessary to discard markers. Discarding markers should be done carefully, since discarding a marker too early can lead to incorrectly passing the condition of the second and third rules, thereby generating a non-IR execution. Such an undesirable scenario can arise if a step \( p_j \) follows \( q_j^k \), and accesses a data item whose \((i, \bar{b})\) marker was removed prematurely. As far as correctness goes, the precondition for discarding markers is formulated as follows.

A marker \((i, b)\) of a transaction \( T_i \) (with a unanimous fate), can be discarded if all \( T_j \) whose steps follow \( q_j^k \) are no longer active.

A transaction is active until it can no longer initiate new steps at new sites (i.e., at least until the beacon initiates the commit protocol for that transaction). Once \( T_j \) becomes inactive, steps \( p_j \) can no longer cause the problem outlined above. Implementing this transition requires cooperation among sites and hence extra communication. It should be emphasized, however, that this additional message exchange is needed only for the purposes of discarding markers, and it can be decoupled from the execution and commit procedure of a partic-
ular transaction. Specifically, discarding markers can be done periodically, as a garbage collection activity, thereby amortizing the communication cost over a set of transactions.

We describe the message exchange between a beacon and a set of sites under the polarized protocol. The purpose of this exchange is to discard markers of transactions executed at these participating sites and coordinated by that beacon. This message exchange is executed periodically, and not for every transaction separately. The additional rules for this exchange are described next.

4. At a participant. For transactions $T_i$, whose local recovery steps have terminated successfully with polarity $b$:
   \[
   \text{if } \neg(\exists j: (i, b) \in \text{follow}(p_j)) \text{ then send SAFE ack message to beacon}
   \]
   \[
   \text{else send UNSAFE ack message to beacon}
   \]

5. At the beacon. When the beacon has received SAFE/UNSAFE acks from all participants of $T_i$ that executed recovery steps for $T_i$:
   \[
   \text{if all the participants sent a SAFE ack then notify all participants to discard $T_i$'s markers}
   \]
   \[
   \text{else notify only the sites sending UNSAFE to discard $T_i$'s markers}
   \]

Having received all the SAFE/UNSAFE acks for a transaction $T_i$, the beacon has all the information needed to determine whether it is safe to discard $T_i$'s markers. First, the beacon is certain that $T_i$ has a unanimous fate, since the successful termination of all of $T_i$'s recovery steps was acknowledged. It is safe to discard $T_i$'s markers provided that there is no transaction $T_j$, whose step follows a step of $T_i$ with a polarity opposite to the final fate of $T_i$. The existence or absence of such a transaction $T_j$ is encoded in the SAFE/UNSAFE ack messages. Recall that markers are retained to prevent such $T_j$ from spawning additional steps at new sites. Therefore, the sites detecting such a $T_j$ (i.e., sites sending an UNSAFE ack) can discard $T_i$'s markers, and only sites sending SAFE acks have to retain $T_i$'s markers.

It is assumed that follow sets of steps are maintained as long as the corresponding transaction is active. Finally, it should be noted that discarding markers at a site need not be performed as a synchronous action. By the time a site is notified that it can discard markers, their presence or absence is of no consequence, whatsoever.

4.3 Correctness

We are in position now to state several results concerning the protocol. The polarized protocol is a reactive algorithm that reacts to events in the course of processing a multi-site transaction. The execution of transactions is governed by the protocol, and hence only a certain class of executions is allowed under the polarized protocol. Our objective is to show that these executions isolate recoveries. We assume that the commit protocol outlined in Section 2.1 is used and hence executions are eventually semantically atomic. The following propositions assume a semantically atomic execution $E$, that is generated under the polarized protocol. The propositions are direct consequence of the above rules.

Proposition 1. If $q_j$ follows $s_i^k$ because of a conflict on $x$ at a certain site, and at that site $T_i$'s markers have not been discarded by the time $q_j$ accesses $x$, then $(i, b) \in \text{follow}(q_j)$. $\square$

Proposition 2. If $q_j \xrightarrow{p_j}$, then $\text{follow}(q_j) \subseteq \text{follow}(p_j)$. $\square$
Proposition 3. A marker \((i,b)\) is discarded at site, only if there is no active transaction \(T_j\) such that \((i,\bar{b}) \in \text{follow}(o_j)\) at any other site. □

Proposition 4. All markers in the follow set of a step are of the same polarity. □

Proposition 5. If \(o_j\) follows \(t_i^k\) because of a conflict on a data item, and \(T_i\)'s fate is \(\bar{b}\), then \((i,b) \in \text{follow}(o_j)\).

Proof. Let \(o_j\) and \(t_i\) conflict on \(x\). Since \(T_i\) has a unanimous fate, a recovery step that corresponds to \(t_i\) must exist in the execution. Thus, the marker \((i,b) \in \text{access}(x)\) is discarded by the recovery step that corresponds to \(t_i\). Therefore, since \(o_j\) follows \(t_i\), \(o_j\) must precede this recovery step, and hence the marker would not be removed prior to \(p_j\)'s access to \(z\). Consequently, \((i,b) \in \text{follow}(o_j)\). □

Theorem 1. The polarized protocol ensures that executions isolate recoveries. □

Proof. The proof is by contradiction. We assume that \(p_j\) follows \(s_i^k\) and \(t_i^k\), and derive a contradiction. By the definition of the 'follows' relation, there are steps \(q_j\) (\(o_j\)) that follow \(s_i^k\) (\(t_i^k\)) because of conflicts on data items \(x\) (\(y\)) at site \(S_1\) (\(S_2\)), and \(q_j \overset{i}{\to} p_j\) (\(o_j \overset{i}{\to} p_j\)). (One of \(q_j, o_j\) may be \(p_j\) itself). By assumption, the execution is SA, and hence \(T_i\) has a unanimous fate \(b\) or \(\bar{b}\). Without loss of generality it may be assumed that the final fate of \(T_i\) is \(b\) (the dual case is symmetric). By Proposition 5, \((i,\bar{b}) \in \text{follow}(o_j)\). The proof proceeds by considering two cases. First, we assume that site \(S_1\) had already discarded the marker \((i,b)\) when \(q_j\) accessed \(x\). By proposition 3, the \((i,b)\) marker could have been discarded only if \((i,\bar{b}) \notin \text{follow}(o_j)\) at site \(S_2\). However, \((i,\bar{b}) \in \text{follow}(o_j)\). A contradiction is derived. In the second case, we assume that the marker \((i,b)\) was not discarded before \(q_j\) accessed \(x\) at site \(S_1\). Then, by Proposition 1, \((i,b) \in \text{follow}(q_j)\), and by Proposition 2, \((i,b) \in \text{follow}(p_j)\). However, since \((i,\bar{b}) \in \text{follow}(o_j)\), by Proposition 2, \((i,\bar{b}) \in \text{follow}(p_j)\), too. This contradicts Proposition 4. □

4.4 A Biased Decision

If a biased decision is employed by the beacon, then the marking scheme can be simplified. Only one type of marker is needed, \((i,1)\), and all the information needed can be deduced from the absence or presence of this marker. Moreover, there is simple way of discarding the markers that incurs no extra messages. These optimizations are described by the rules below.

1. **Marking.** Whenever a forward step \(p_i\) terminates successfully, then for all data items \(x\) accessed by \(p_i\):\
   \[
   \text{access}(x) := \text{access}(x) \cup \{(i,1)\}
   \]

2. **Access and Propagation.** Whenever a step \(p_i\) requests access to data item \(x\):
   - if \(\forall j : ((j,1) \in \text{follow}(p_i)) \equiv ((j,1) \in \text{access}(x))\)
   - then \(\text{follow}(p_i) := \text{follow}(p_i) \cup \text{access}(x)\)
   - else reject \(p_i\)'s request

3. **Propagation by Dependence.** Whenever a step \(p_i\) requests to invoke a step \(q_i\):
   - if \(\forall j : ((j,1) \in \text{follow}(p_i)) \equiv ((j,1) \in \text{follow}(q_i))\)
   - then \(\text{follow}(q_i) := \text{follow}(q_i) \cup \text{follow}(p_i)\)
   - else reject \(p_i\)'s request

4. **Unmarking.** Whenever a recovery step \(r_{p_i}\) terminates with a polarity 0, or whenever a commit decision is received from the beacon, then for all data items \(x\) accessed by \(p_i\):
   \[
   \text{access}(x) := \text{access}(x) \setminus \{(i,1)\}
   \]

This simplified protocol has a reduced communication cost and complexity, however, it is unnecessarily restrictive. That is, it generates only a subset of the possible
IR executions. Specifically, if a step of $T_j$ follows $q_i^1$, then all other steps of $T_j$ should also follow steps of $T_i$ with a polarity 1. For instance, $T_j$ cannot spawn a step at a site where $T_i$ has not executed at all. The marking does not distinguish among data items accessed by steps of $T_i$ and marked by a 0, and data items that have not been accessed by $T_i$ at all.

Theorem 2. The protocol for the biased decision ensures that executions isolate recoveries. □

The significant difference in the proof of this theorem is that we need to treat separately the cases of a final fate of commit (1), and abort (0) for $T_i$. If $T_i$'s final fate is 1, then it has no step with polarity 0 (biased decision), and hence IR is vacuously guaranteed. If the final fate of $T_i$ be 0, the proof proceeds similarly to the proof of Theorem 1.

The diagrams in Figure 2(a), and 2(b), depict the transitions in the marking of a data item $x$ with respect to $T_i$, for a balanced, and a biased, decision rule, respectively. The state 'neutral' corresponds to $(i, b) \not\in \text{access}(x)$, and the states labeled $b$ correspond to $(i, b) \in \text{access}(x)$.

5 Extensions

The concept of relaxed atomicity relies on the methods of compensation and retry. As was mentioned earlier, these methods are not applicable universally, and are based on semantics of the applications at hand. For instance, transactions involving real actions [Gra81] (e.g., firing a missile or dispensing cash) may not be compensatable. The adjustment for transactions involving non-compensatable steps is to retain their locks and delay real actions until a commit decision message is received from the beacon (as in standard two-phase commit) in all sites performing these actions. All other sites running compensatable steps on behalf of the same transaction can still benefit from the early lock release of our modified commit protocol.

A general way of integrating arbitrary steps (which may not be suitable for compensation or retry) into our model, is described next. Each step can be divided into three portions: a compensatable portion (CP), a pivot portion (PP), and a retriable portion (RP). The execution of such a step would proceed as follows: The CP is executed first, and following its termination all locks it has acquired are released at once. The PP is executed second and its termination is coordinated by a 2PC protocol among all the pivots of the steps of the same transaction. While waiting for the 2PC decision message to arrive, the RP is initiated. Locks acquired by the PP or the RP are released only after a decision message is received locally. If the decision is to abort, then the RP is aborted and both RP and PP are undone using standard recovery since their locks have not been released. Additionally, the CP is compensated-for. If, however, the decision is to commit, then PP’s locks are released, and if RP happen to have failed it is re-executed until it succeeds.

The work we have reported in [LKS91] is a conservative instance of the idea of relaxed atomicity developed here. There, a commit protocol with a biased decision rule, and compensating steps are used. Besides recovery and atomicity aspects, the work in [LKS91] concentrates on concurrency control issues. In particular, a site releases locks held by a step only after voting in the first
phase of the commit protocol. Therefore, in spite of the early locks release used in the commit protocol, forward transactions are serializable globally (in contrast to the local serializability assumption of Section 2). Another feature of the protocol there, is that markers are discarded without additional messages. This feature is based on the locking discipline assumed to be followed by transactions.

6 Conclusion

The importance of the work on relaxed atomicity is emphasized in light of the inevitable blocking phenomenon that is typical of standard atomicity. Moreover, relaxed atomicity is motivated by the growing interest in distributed system integration; an area where standard atomicity stands in sharp contrast to the crucial autonomy of the integrated components.

The criterion of recovery isolation gives transactions a degree of isolation from inconsistencies arising from failures and their asynchronous recoveries. In an IR execution, effects of both committed and aborted steps of the same transaction are allowed to be exposed, thereby avoiding the prohibitive cost of a distributed atomic commitment. However, it is ensured that transactions observe only effects of sets of steps with identical polarity, thus hiding the non-atomic execution of transactions. Similarly, in serializable executions [BH87], concurrency among operations of transactions is allowed, thereby enhancing throughput. Yet, the effects of concurrency are hidden from transactions by excluding undesirable executions.

It is interesting to examine the trade-off between the overhead incurred by the marking scheme and the degree of restrictions imposed by the corresponding protocol. This trade-off is demonstrated by the two protocols presented in Section 4.

The work reported in this paper is just a foundation. Additional research is needed in both theoretical as well as practical aspects of relaxed atomicity.

Acknowledgements. We would like to thank Rajeev Rastogi and Sharad Mehrotra for coming up with the idea of decomposing a transaction into a compensatable portion, pivot, and retrievable portion.
References


