Chapter 9

Transactions

Transactions are a mechanism that allows endowing a sequence of actions with the properties of a single atomic (indivisible) action. Originally introduced in the context of database management systems, transactions are a powerful tool to help building complex applications, by ensuring that the system remains in a consistent state in all circumstances.

In this chapter, we examine how transactional mechanisms are being implemented and used in middleware systems. After a general introduction, the chapter briefly reviews the main concepts and techniques developed for implementing transactions, with special emphasis on distributed transactions. It goes on with a discussion of transactional frameworks for middleware systems, and concludes with a case study.

9.1 Motivations and Main Concepts

The notion of an *atomic action* is a powerful tool for overcoming the difficulties of concurrent programming. To illustrate this point, consider the case of multithreaded programs in a shared memory multiprocessor. Elementary hardware-implemented operations such as atomic exchange (exchanging the contents of a processor register and a memory location in a single, indivisible action) are the basic building blocks on which more elaborate synchronization mechanisms can be constructed and made available to application developers.

Elementary atomic operations have simple properties, which facilitate reasoning and allow ensuring that the mechanisms built upon these operations meet their specifications. For instance, the concurrent execution of two atomic operations A and B may only have two outcomes (in the absence of failures): it has the same effect as either A followed by B or B followed by A.

A composite action is defined as a sequence of actions; ultimately, any action consists of a sequence of elementary atomic operations.The main idea of transactions is to extend the properties of these elementary operations to composite actions.

A few notions are needed to specify the effect of actions. The universe (also called "the system") is composed of objects (in the sense of 2.2.2), each of which has a well-defined state (typically, the values of a set of variables). The global state of the universe is the union of the states of the individual objects. A process is a sequence of actions, each of which is defined as follows: given an initial state of the system and a (possibly void) input value, the effect of the action, when executed alone, is to bring the system in a final state in finite time, and to deliver a (possibly void) output value[1]. For a given action, the final state and the output value are specified functions of the initial state and the input value. This is essentially a state machine model, which we also use in 11.3.1.

Process execution may be subject to failures. A *failure* occurs when the behavior of the system (or some component of it) does not conform to its specification. Different classes of failures, according to their degree of severity, have been identified (see 11.1.3 for a detailed discussion). Here we only consider *fail-stop* failures: either the component works according to its specification, or it does nothing. We assume the existence of *stable storage*, i.e., storage that guarantees that the data committed to it are immune from failures[2]. Techniques for the implementation of stable storage are described in [Lampson and Sturgis 1979].

We now come to the notion of *atomicity*, a property of elementary actions that we wish to extend to composite actions. Atomicity has two facets.

- *Concurrency atomicity.* Consider concurrent actions, i.e., actions executed by concurrent processes. Concurrency atomicity is then expressed as follows: let A and B be two actions that are executed concurrently (which we denote by $A||B$). Then, in the absence of failures, the effect of this execution is either that of $A;B$ or that of $B;A$ (the effect is defined by the output values and the change of state of the system).

  A different formulation of this property is as follows. A composite action goes through intermediate states of the system (the final states of its composing actions). No such intermediate state may be visible to any concurrently executing action. Stated in this form, concurrency atomicity is called *isolation*.

- *Failure atomicity.* Consider an action A, which (when executed alone) makes the system go from state $S1$ to state $S2$ (which we denote by $S1\ \{A\}\ S2$). Then, if a failure may occur during the execution of A, the final state of the system is either $S1$ or $S2$. In other terms, the action is performed either entirely or not at all[3]. Again, no intermediate state may be visible.

  In addition to failures provoked by external causes, an action may decide to interrupt its execution, and to cancel all the changes made so far to the system state. This operation is called *abort*. If $S1\ \{A\}\ S2$, the effect of an abort in the execution of A is to restore the state of the system to $S1$, an operation called *rollback*.

Thus the atomicity property allows us to limit the uncertainty about the outcome of a possibly concurrent execution of a set of actions in the presence of failures. A simple form of transaction is defined as a sequence of actions that has the atomicity property (both for concurrency and failures).

Transactions have been historically introduced in the context of database management systems, which motivated the introduction of two additional properties.

[1]Two remarks are in order: (a) the effect of an action may be to create new objects; and (b) output delivery may take the form of an action in the "real world" (e.g., printing a form, or delivering cash).

[2]We exclude *catastrophes*, exceptional events that would cause an extensive loss of storage media.

[3]"Real actions", i.e., output actions operating in the real world (see note 1), pose a problem here, since their effect can not always be canceled. More on this issue in 9.3.

- *Consistency.* Consistency in a database is usually expressed as a set of *integrity constraints*, i.e., predicates that apply to its state. These constraints are meant to express invariant properties of the real world entities that are represented in the database (e.g., banking accounts, inventory, contents of a library, etc.). Since consistency needs to be preserved through the life of the database, a transaction is defined as an atomic (composite) action that brings the system from a consistent state to another consistent state. During the execution of a transaction, some intermediate states may be inconsistent; however, by virtue of atomicity, they are never visible outside the transaction. By definition, consistency is an application dependent notion.

- *Durability.* The durability (or permanence) property states that, if the transaction succeeds (i.e., if it concludes without failures and does not abort), the changes it made to the database are permanent. Of course, the state may be modified by subsequent transactions. The operation that marks the successful end of a transaction is called *commit*. Durability means that, after a transaction has committed, the changes that it made are immune to failures (catastrophes excluded).

The acronym ACID (Atomicity, Consistency, Isolation, Durability) is commonly used to qualify transactions in a database environment (note that "atomicity" in ACID has the restricted meaning of "failure atomicity", as "isolation" stands for "concurrency atomicity"). Thus if a database is initially in a consistent state, and all operations on it are included in (consistency preserving) transactions, it is guaranteed to remain in a consistent state, and all outputs are likewise consistent.

Transactions favor separation of concerns (1.4.2), since they allow the application developers to concentrate on the logic on the application (i.e., writing transactions that preserve consistency), while a *transaction management system* is responsible for ensuring the other transactional properties (atomicity, isolation and durability). These properties are guaranteed through the following methods.

- Concurrency atomicity (or isolation) is ensured by *concurrency control*, which uses locking (the main technique) or time-stamping.

- Failure atomicity is ensured by *recovery*, which in turn is based on logging techniques.

- Durability depends on failure atomicity, and relies on *stable storage*.

The transaction management system provides a simple interface to its users. While the syntax of the operations may vary according to the system, the basic interface is usually composed of the following operations.

- *begin_transaction*: start of a new transaction.

- *commit_transaction*: normal end of the transaction (changes are durably registered).

- *abort_transaction* (or *rollback_transaction*): discontinue the execution of the transaction and cancel the changes made by the transaction since its beginning.

A transaction is uniquely identified, usually through an identifier delivered by the transaction management system at the start of the transaction. In some systems, a transaction is associated with an object, which provides the above operations in the form of methods. Additional operations may be available, e.g., consulting the current state of the transaction, etc.

The above operations achieve *transaction demarcation*, i.e., delimiting the operations that are part of a transaction within the program being executed. Transaction demarcation may be explicitly done by the application programmer, or may be automatically generated from a declarative statement by the programming environment (more details in 9.6).

The topic of transactions is an important area in its own right, and is the subject of a vast amount of literature, including several reference books [Bernstein et al. 1987, Gray and Reuter 1993, Weikum and Vossen 2002, Besancenot et al. 1997]. In the context of this book, we concentrate on transactional aspects of middleware; our presentation of the fundamental aspects of transaction management is only intended to provide the necessary context, not to cover the subject in any depth.

The rest of this chapter is organized as follows. Section 9.2 examines concurrency control. Section 9.3 is devoted to recovery techniques. Distributed transactions are the subject of Section 9.4. Section 9.5 presents nested transactions and relaxed transactional models. Transactional middleware is introduced in Section 9.6, and illustrated by a case study in Section 9.7. A brief historical note (9.8) concludes the chapter.

9.2 Concurrency Control

If transactions were executed one at a time, consistency would be maintained, since each individual transaction, by definition, preserves consistency. However, one wishes to allow multiple transactions to execute concurrently, in order to improve performance, through the following effects.

- Concurrent execution allows taking advantage of multiprocessing facilities, both for process execution and for input-output.
- Concurrent execution potentially allows a better usage of resources, both physical and logical, if each transaction only needs a subset of the available resources.

The goal of *concurrency control* is to ensure isolation (or concurrency atomicity, as defined in 9.1) if several transactions are allowed to execute concurrently on the same set of data. The main concept of concurrency control is *serializability*, a property of a set of transactions that we examine in 9.2.1. The main mechanism used to enforce serializability is *locking*, which is the subject of 9.2.2. In this section, we assume that all transactions take place on a single site, and that they actually commit. Aborts and failures are examined in section 9.3, and distributed transactions are the subject of section 9.4.

9.2.1 Serializability

The theory of concurrency control has developed into a vast body of knowledge (see e.g., [Bernstein et al. 1987, Papadimitriou 1986, Weikum and Vossen 2002]). Here we only present a few results that are widely used in current practice.

An acceptable (or "correct") concurrent execution of a set of transactions is one that ensures isolation, and thus preserves consistency. The first issue in concurrency control is to find a simple characterization of correct executions. A widely accepted notion of correctness is that the concurrent execution be equivalent to *some* serial execution of this same set of transactions[4]. Here, "equivalent to" means "having the same effect as"; a more precise definition is introduced later in this section. Note that we do not specify a particular order for the serial execution: any order is acceptable. If each transaction preserves consistency, so does any serial execution of the set of transactions.

Consider a set of transactions $\{T_i\}$, operating on a set of objects $\{a, b, \ldots\}$. When executed alone, each transaction is a sequence of read and write operations on the objects (each of these individual operations is atomic). We do not consider object creation and deletion; these are discussed in 9.2.3. When the transactions $\{T_i\}$ are executed concurrently, the operations of the transactions are interleaved in a single sequence, in which the order of the operations of each individual transaction is preserved. Such a sequence is called a *schedule*. By our definition of correctness, an acceptable schedule (one generated by a correct concurrent execution of the set $\{T_i\}$) is one equivalent to a *serial schedule*, i.e., one corresponding to a serial execution of the transactions of $\{T_i\}$, in some (unspecified) order. A schedule that is equivalent to a serial schedule is said to be *serializable*.

In order to characterize serializable schedules, let us identify the situations in which consistency may be violated. A read (resp. write) operation performed by transaction T_i on object x is denoted $R_i(x)$ (resp. $W_i(x)$). Consistency may be violated when two transactions T_i and T_j ($i \neq j$) execute operations on a shared object x, and at least one of these operations is a write. Such a situation is called a *conflict*. Three types of conflicts may occur, characterized by the following patterns in the execution schedule.

- Unrepeatable read: $R_i(x) \ldots W_j(x) \ldots$. Then a subsequent read of x by T_i may deliver a different result from that delivered in a serial execution.

- Dirty read: $W_i(x) \ldots R_j(x) \ldots$. Then the result of $R_j(x)$ may be different from that delivered in a serial execution.

- Lost write: $W_i(x) \ldots W_j(x) \ldots$. Then the effect of the first write may be canceled by that of the second write.

Note that a conflict does not necessarily cause an inconsistency (this depends on whether T_i commits before or after the operation by T_j). [Berenson et al. 1995] thoroughly analyze various cases of anomalous behavior, including the three above situations.

By definition, for a given execution, transaction T_j *depends on* transaction T_i if, for some object x, there is a conflict between T_i and T_j, and the operation executed by T_i on x appears first in the schedule. This relation is captured by a *dependency graph*, in which vertices represent transactions, and edges (labeled by object names) represent conflicts. Edges are oriented by the order of conflicting operations (e.g., for any of the conflicts illustrated above, the edge would be oriented from T_i to T_j).

[4] Two remarks are in order: (1) This definition only provides a sufficient condition for correctness. In other words, some consistency-preserving executions may not be equivalent to a serial execution. (2) For efficiency reasons, weaker correctness conditions have been proposed. See an example (snapshot isolation) in 9.2.3.

We may now define a form of equivalence for schedules: two schedules are *conflict-equivalent* if they contain the same operations and if they have the same dependency graph (i.e., if each pair of conflicting operations appears in the same order in both schedules). A schedule is conflict-serializable if it is conflict-equivalent to a serial schedule[5].

The following result has been proven [Papadimitriou 1979]: *a schedule is conflict-serializable if and only if its dependency graph is acyclic.* This property gives a convenient characterization of a class of consistency-preserving schedules, and is the base of the most common concurrency control algorithms used in practice. A simple example follows.

Example 1. Consider two transactions T_1 and T_2, defined as follows:

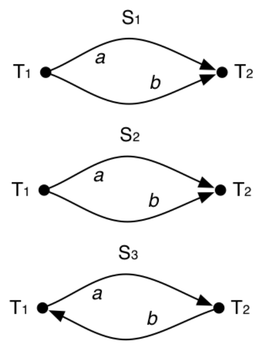
Transaction T_1	Transaction T_2
begin	
if $(A \geq X)$	*begin*
$\{A = A - X\}$	$A = (1+R)A$
else *abort*	$B = (1+R)B$
$B = B + X$	*commit*
commit	

A and B denote banking accounts (and the corresponding balances). Transaction T_1 transfers an amount X from A to B. Transaction T_2 applies an interest rate R to both accounts A and B. In fact, T_1 and T_2 are patterns for transaction types, in which A, B, X and R are parameters.

The integrity constraints are the following: (a) the balance of each account is non-negative; (b) the sum of all accounts is invariant after the execution of a transaction of type T_1; and (c) the sum of all accounts involved in a transaction of type T_2 is multiplied by $(1+R)$ after the execution of the transaction.

Now consider the concurrent execution of two transactions T_1 and T_2, with parameters a, b, x, r, and assume that, initially, all parameters are positive and $a \geq x$ (so that T_1 does not abort). Three possible schedules of this execution are shown below, together with their dependency graphs.

Schedule S_1	Schedule S_2	Schedule S_3
$begin_1$	$begin_1$	$begin_1$
$a = a - x$	$a = a - x$	$a = a - x$
$b = b + x$	$begin_2$	$begin_2$
$commit_1$	$a = (1+r)a$	$a = (1+r)a$
$begin_2$	$b = b + x$	$b = (1+r)b$
$a = (1+r)a$	$commit_1$	$commit_2$
$b = (1+r)b$	$b = (1+r)b$	$b = b + x$
$commit_2$	$commit_2$	$commit_1$



Schedule S_1 is serial $(T_1; T_2)$. S_2 derives from S_1 by exchanging two write operations on different objects; since this does not cause a new conflict, S_2 is equivalent to S_1

[5]Note that conflict-equivalence is a restricted notion of equivalence, introduced because of its easy characterization. Thus a schedule may be serializable, but not conflict-serializable. However, the two notions coincide if a transaction always reads an object before writing it, a frequent situation in practice [Stearns et al. 1976].

and thus is serializable. S_3 has a write-write conflict causing the first write on b to be lost, violating consistency. The dependency graph of S_3 contains a cycle.

We now have a simple characterization of serializable transactions. A mechanism that allows enforcing serializability is introduced in the next section.

9.2.2 Locking

In order to enforce serializability of concurrent transactions, one needs a mechanism to delay the progress of a transaction. Locking is the most frequently used technique. In order to allow concurrent access to a shared object in read-only mode, two types of locks are defined: exclusive and shared (this distinction is inspired by the canonical synchronization scheme of readers and writers [Courtois et al. 1971]). To acquire a lock on object o, the primitives are $lock\_exclusive(o)$ and $lock\_shared(o)$. An acquired lock is released by $unlock(o)$. The following rules apply for access to shared objects within a transaction.

- A transaction needs to acquire a lock on a shared object before using it (shared lock for read-only access, exclusive lock for read-write access).
- A transaction may not lock an object on which it already holds a lock, except to upgrade a lock from shared to exclusive.
- No object should remain locked after the transaction concludes, through either commit or abort.

The first two rules directly derive from the readers-writers scheme. The last rule avoids spurious locks, which may prevent further access to the locked objects. A *well-formed* transaction is one that respects these rules.

Locks are used to constrain the allowable schedules of concurrent transactions. Two rules are defined to this effect.

- An object locked in shared mode by a transaction may not be locked in exclusive mode by another transaction.
- An object locked in exclusive mode by a transaction may not be locked in any mode by another transaction.

A *legal* schedule is one that respects these rules. To ensure legality, locking operation that would break these rules is delayed until all conflicting locks are released.

Two-Phase Locking

While legal schedules of well-formed transactions ensure the valid use and correct operation of locks, all such schedules are not guaranteed to be serializable. Additional constraints are needed. The most commonly used constraint, called *two-phase locking* (2PL), is expressed as follows.

After a transaction has released a lock, it may not obtain any additional locks.

A transaction that follows this rule therefore has two successive phases: in the first phase (growing), it acquires locks; in the second phase (shrinking), it releases locks. The point preceding the release of the first lock is called the maximum locking point.

The usefulness of 2PL derives from the following result [Eswaran et al. 1976].

> Any legal schedule S of the execution of a set $\{T_i\}$ of well-formed 2PL transactions is conflict-equivalent to a serial schedule. The order of the transactions in this serial schedule is that of their maximum locking points in S.

Conversely, if some transactions of $\{T_i\}$ are not well formed or 2PL, the concurrent execution of $\{T_i\}$ may produce schedules that are not conflict-equivalent to any serial schedule (and may thus violate consistency).

Example 2. Consider again the two transactions T_1 and T_2 of Example 1. Here are two possible implementations of T_1 (T_{1a} and T_{1b}), and one implementation of T_2, using locks (for simplicity, we have omitted the test on the condition $x \leq a$, assuming the condition to be always true).

Transaction T_{1a}	Transaction T_{1b}	Transaction T_2
$begin_{1a}$	$begin_{1b}$	$begin_2$
$lock\_exclusive_{1a}(a)$	$lock\_exclusive_{1b}(a)$	$lock\_exclusive_2(a)$
$a = a - x$	$a = a - x$	$a = (1 + r)a$
$unlock_{1a}(a)$	$lock\_exclusive_{1b}(b)$	$lock\_exclusive_2(b)$
$lock\_exclusive_{1a}(b)$	$unlock_{1b}(a)$	$b = (1 + r)b$
$b = b + x$	$b = b + x$	$unlock_2(b)$
$unlock_{1a}(b)$	$unlock_{1b}(b)$	$unlock_2(a)$
$commit_{1a}$	$commit_{1b}$	$commit_2$

T_{1b} and T_2 are 2PL, while T_{1a} is not.

Here are two possible (legal) schedules S_a and S_b, respectively generated by the concurrent execution of T_{1a}, T_2 and T_{1b}, T_2.

$S_a(T_{1a}, T_2)$	$S_b(T_{1b}, T_2)$
$begin_{1a}$	$begin_{1b}$
$lock\_exclusive_{1a}(a)$	$lock\_exclusive_{1b}(a)$
$a = a - x$	$a = a - x$
$unlock_{1a}(a)$	$lock\_exclusive_{1b}(b)$
$begin_2$	$unlock_{1b}(a)$
$lock\_exclusive_2(a)$	$begin_2$
$a = (1 + r)a$	$lock\_exclusive_2(a)$
$lock\_exclusive_2(b)$	$a = (1 + r)a$
$b = (1 + r)b$	$b = b + x$
$unlock_2(b)$	$unlock_{1b}(b)$
$lock\_exclusive_{1a}(b)$	$lock\_exclusive_2(b)$
$b = b + x$	$b = (1 + r)b$
$unlock_2(a)$	$commit_{1b}$
$commit_2$	$unlock_2(b)$
$unlock_{1a}(b)$	$unlock_2(a)$
$commit_{1a}$	$commit_2$

Schedule S_a violates consistency, since T_2 operates on an inconsistent value of b. Schedule S_b is equivalent to the serial schedule $T_{1b}; T_2$

There are two variants of two-phase locking. In *non-strict 2PL*, a lock on an object is released as soon as possible, i.e., when it is no longer needed. In *strict 2PL*, all locks are released at the end of the transaction, i.e., after commit. The motivation for non-strict 2PL is to make the objects available to other transactions as soon as possible, and thus to increase parallelism. However, a transaction may abort or fail before having released all its locks. In that case, an uncommitted value may have been read by another transaction, thus compromising isolation, unless the second transaction aborts. Strict 2PL eliminates such situations, and makes recovery easier after a failure (see 9.3.2). In practice, the vast majority of systems use strict 2PL.

Deadlock Avoidance

Using locks raises the risk of *deadlock*, a situation of circular wait for shared resources. For example, transaction T_1 holds an exclusive lock on object a, and attempts to lock object b; however, b is already locked in exclusive mode by transaction T_2, which itself tries to lock object a. More generally, one defines a (directed) *wait-for graph* as follows: vertices are labeled by transactions, and there exists an edge from T_1 to T_2 if T_1 attempts to lock an object already locked by T_2 in a conflicting mode. A cycle in the wait-for graph is the symptom of a deadlock.

There are two approaches to deadlock avoidance, *prevention* and *detection*. A common prevention method is that all transactions should lock the objects in the same order. However, this is not always feasible, since in many applications transactions are independent and do not have an advance knowledge of the objects they need to lock. In addition, imposing a locking order (independent from the transaction's semantics) may be overly restrictive, and may thus reduce concurrency.

A more flexible approach, dynamic prevention, has been proposed by [Rosenkrantz et al. 1978]. At creation time, each transaction receives a time-stamp, which defines a total order among transactions. Conflicts are resolved using a uniform precedence rule based on this order. Consider two transactions T_1 and T_2, which both request a shared object x in conflicting modes, and suppose T_1 has acquired a lock on x. Two techniques may be used to resolve the conflict:

- Wait-Die: if T_2 is "older" than T_1, it waits until the lock is released; otherwise, T_2 is forced to abort, and restarts later on.
- Wound-Wait: if T_2 is "older" than T_1, it preempts object x (breaking the lock); otherwise, it waits until the lock is released. After preemption, T_1 must wait for T_2 to commit or to abort before trying to re-acquire a lock on object x.

Thus, a wait-for edge is only allowed from an older to a younger transaction[6] (in wait-die) or in the opposite direction (in wound-wait). This avoids starvation (waiting indefinitely to acquire a lock). Wound-wait is usually preferable since it avoids aborting transactions.

Contrary to prevention, deadlock detection imposes no overhead when deadlocks do not occur, and is often preferred for performance reasons. Deadlock detection involves

[6]To guarantee this property, a transaction must keep its initial time-stamp when restarted after abort.

two design decisions: defining a deadlock detection algorithm; defining a deadlock resolution method. Deadlock detection amounts to finding a cycle in the wait-for graph; in a distributed system, this involves information interchange between the nodes, so that the deadlock may be detected locally at any site. A typical algorithm (introduced in the R* system [Mohan et al. 1986]) is described in [Obermarck 1982]. Once detected, a deadlock is resolved by aborting one or more "victim" transactions. A victim is chosen so as to minimize the cost of its rollback according to some criterion (e.g., choosing the "youngest" transaction, or choosing a transaction local to the detection site, if such a transaction exists).

Two-phase locking, in its strict version, is the most commonly used technique to guarantee a consistent execution of a set of transactions.

9.2.3 Additional Issues

In this subsection, we present a few complements regarding concurrency control.

Object Creation and Deletion

In the above presentation, we assumed that the database contained a fixed set of objects, accessible through read or write operations. Object creation and deletion raises additional problems. Consider the following transactions:

$T_1 : \ldots$ create(o_1) in set S; $\ldots$ create(o_2) in set S; $\ldots$
$T_2 : \ldots$ for all x in S do {read(x)$\ldots$} $\ldots$

Suppose the concurrent execution of T_1 and T_2 generates the following schedule:

$\ldots$ create(o_1) in set S; for all x in S do {read(x)$\ldots$}; create(o_2) in set S; $\ldots$

Although the schedule appears to be conflict-serializable (in the order $T_1; T_2$), this is not the case, because object o_2 "escaped" the "for all" operation by T_2. Such an object, which is potentially used before being created, is called a *phantom object*. A symmetrical situation may occur with object deletion (the object is potentially used after having been deleted).

The problem of phantoms is solved by a technique called *multilevel locking*, in which objects are grouped at several levels of granularity (e.g., in files, directories, etc.), and locks may be applied at these levels. Besides read and write locks, intentional locks (*intent_read*, *intent_write*, ...) may be applied at the intermediate levels. The following rules apply: (1) Before locking an object, a transaction must have obtained an intentional lock, in a compatible mode, on all the groups that contain the object; and (2) An object may only be created or deleted by a transaction T if T has obtained an *intent_write* lock on a higher level group including the object. This ensures that object creation and deletion will be done in mutual exclusion with potential accesses to the object.

Time-stamp Based Concurrency Control

The principle of time-stamp based concurrency control is to label each transaction with a unique time-stamp, delivered at creation, and to process conflicting operations in time-stamp order. Thus the dependency graph cannot have cycles, and the serialization order is determined a priori by the order of transactions' time-stamps.

Processing conflicting operations in time-stamp order may involve aborting a transaction if it attempts to perform an "out-of-order" operation, i.e., if the order of the conflicting operations does not agree with that of the corresponding time-stamps.

In the basic variant of the method, each object x holds two time-stamps: R_x, the time-stamp of the most recent transaction (i.e., the one with the largest time-stamp) that has read x, and W_x, the time-stamp of the most recent transaction that has written x. Suppose that a transaction T, time-stamped by t, attempts to execute an operation $o(x)$ on an object x, that conflicts with an earlier operation on x. The algorithm runs as follows:

- If $o(x)$ is a read, then it is allowed if $t > W_x$; otherwise, T must abort.
- If $o(x)$ is a write, then it is allowed if $t > max(W_x, R_x)$; otherwise, T must abort.

In both cases, if the operation succeeds, R_x or W_x are updated.

The case of the write-write conflict may be optimized as follows: if $R_x < t < W_x$, then the operation succeeds, but the attempted write by T is ignored since a most "recent" version exists. In other words, "the last writer wins". This is known as Thomas' write rule [Thomas 1979].

Multiversion concurrency control [Bernstein and Goodman 1981, Bernstein et al. 1987] is a variant of time-stamp based concurrency control that aims at increasing concurrency by reducing the number of aborts. The above rules for conflict resolution are modified so that a read operation is always allowed. If a write operation succeeds (according to the rule presented below), a new version of the object is created, and is labeled with the transaction's time-stamp. When a transaction time-stamped by t reads an object x, it reads the version of x labeled with the highest time-stamp less than t, and t is added to a read set associated with x. When a transaction T, time-stamped by t, attempts to write an object x, it must abort if it invalidates a previous read of x by another transaction. Invalidation is defined as follows: let W be the interval between t and the smallest write time-stamp[7] of some version of x. If there is a read time-stamp (a member of the read set of x) in the interval W, then the attempted write invalidates that read (because the version read should have been the one resulting from the attempted write).

Snapshot isolation [Berenson et al. 1995] is a further step towards improving performance by increasing concurrency, building on the notion of multiversion concurrency control. When a transaction T starts, it gets a start time-stamp $ts(T)$ and takes a "snapshot" of the committed data at that time. It then uses the snapshot for reads and writes, ignoring the effect of concurrent transactions. When T is ready to commit, it gets a commit time-stamp $tc(T)$, which must be larger than any other start or commit time-stamp. T is only allowed to commit if no other transaction with a commit time-stamp in the interval $[ts(T), tc(T)]$ had a write-write conflict with T. Otherwise, T must abort.

While this method improves performance, since there is no overhead in the absence of write-write conflicts, it is subject to undesirable phenomena called skew anomalies. Such anomalies occur when there exists constraints involving different objects, such as

[7] If no such time-stamp exists, $W = [t, \infty]$.

$val(o_1) + val(o_2) > 0$, where $val(o)$ denotes the value of object o. If two transactions T_1 and T_2 run in parallel under snapshot isolation, T_1 updating o_1 and T_2 updating o_2, then the constraint may be violated although no conflict is detected. The inconsistency would have been detected, however, under a serial execution.

[Fekete et al. 2005] propose techniques to remedy this situation, thus making snapshot isolation serializable. Snapshot isolation is used in several commercial database systems. It is also used as a consistency criterion for data replication (see 11.7.1)

9.3 Recovery

The function of recovery in a transaction management system is to guarantee atomicity[8] and durability in the face of failures. Recall that storage is organized in two levels: non-permanent (or volatile), such as main memory, and permanent, such as disk storage. Response to failures thus raises two issues: (1) how to recover from failures that only affect volatile storage, such as system crash or power supply failure; and (2) how to make the system resistant to media failures (excluding catastrophes). The latter problem is treated by data replication, which is the subject of 11.7. In this section, we only consider recovery from volatile storage loss. Note that the problem of processing an abort (rolling back a transaction) is a special case of failure recovery.

A failure may leave the system in an inconsistent state. Restoring the system to a consistent state is not always feasible, as shown by the following example. Consider a transaction T_1 which reads an object written by a transaction T_2. Suppose that T_1 commits, and that T_2 aborts (or is affected by a failure) at a later time. Then rolling back T_2 makes the value read by T_1 non-existent. But since T_1 has committed, its effect (which may depend on the read value) cannot be undone. A similar situation occurs if T_1 writes an object that was previously written by T_2.

Therefore, a desirable property is that the system be *recoverable*, i.e., that the failure of a transaction cannot invalidate the result of a committed transaction. This property may be translated in terms of schedules: a schedule is *strict* if, for any pair of transactions T_1 and T_2 such that an operation op_2 of T_2 precedes a conflicting operation op_1 of T_1 on the same object, the commit event of T_2 also precedes[9] op_1. Strictness is a sufficient condition for recoverability. Note that strict two-phase locking (9.2.2) only allows strict schedules, since locks (which prevent conflicting operations) are only released at commit time. In practice, most transaction systems use strict two-phase locking and therefore are recoverable.

The problem of transaction recovery is a complex one, and we only outline the main approaches used to solve it. For an in-depth analysis, refer to [Haerder and Reuter 1983] and to the specialized chapters of [Gray and Reuter 1993] and [Weikum and Vossen 2002]. In the rest of this section, we examine the issues of *buffer management* (how to organize information exchange between the two levels of storage in order to facilitate recovery), and *logging* (how to record actions so as to be able to recover to a consistent state after a failure).

[8]in the sense of “failure atomicity” (9.1).

[9]As a consequence, if T_2 aborts or fails, it does so before T_1 executes op_1.

9.3.1 Buffer Management

In order to exploit access locality, access to permanent storage uses a buffer pool in volatile storage, which acts as a cache, usually organized in pages[10] (unit of physical transfer). A read operation first looks up the buffer pool. If the page is present, the read does not entail disk access; if not, the page is read from disk unto the buffer pool, which may cause flushing the contents of a page, if there is no free frame in the pool. A write operation modifies the page in the pool (after reading it if it was not present). The contents of the modified page may be written on disk either immediately or at a later time, depending on the policy (more details further on).

Two main approaches are followed regarding updates in a database. In the direct page allocation, each page has only one version on disk, and all updates (either immediate or delayed) are directed to this page. In the indirect page allocation, updates are done on a different version of the page, called a *shadow*. Thus the original version of the page still exists, together with the most recent version (and possibly all intermediate versions, depending on the strategy adopted). To actually perform the update, one makes the (most recent) shadow to become the current version (an operation called *propagation*), and this may be done some time after the update. The advantage of indirect allocation is that it is quite easy to undo the effect of a sequence of updates, as long as the changes have not been propagated. The drawback is a performance penalty in regular operation (in the absence of failures), due to the overhead of maintaining the shadows and the associated data structures.

One may now identify three views of the database: (1) the current view, i.e., the most recent state, which comprises modified, unsaved pages in the buffer pool and up to date pages on disk; (2) the materialized view, i.e., the view after a system crash causing the loss of the buffer pool; this view consists of the propagated pages on disk; and (3) the physical view, again after a system crash, consisting of both propagated pages and unpropagated shadow pages on disk. If direct page allocation is used, the physical and materialized views coincide.

Two strategies may now be defined regarding the management of page updates in a transaction.

- Atomic: all the updates made by a transaction (from begin to commit) are performed in a single (indivisible) operation. Technically, this requires indirect page allocation (using indirection allows global propagation to be done by updating a single pointer, an atomic operation).

- Not Atomic: some, but not all, of the pages updated by a transaction may be present in the materialized view. This happens with direct page allocation, since there is no simple way to write the contents of a set of updated pages as a single indivisible operation.

While Atomic achieves indivisible global write, its drawbacks are those of indirect page allocation. On the other hand, Not Atomic is more efficient in normal operation, but necessitates to perform an undo algorithm after a system crash (see 9.3.2).

[10]Following the common usage, we distinguish between *pageframes* (containers) and *pages* (contents).

Regarding buffer pool management, the main issue is: "when is the contents of a modified page actually written to disk?". Writing a modified page to disk may be caused by two events: either its frame in the buffer pool is allocated to another page, or an explicit write decision is made by the transaction management system. Thus the options are the following:

- STEAL: A page that has been modified by a transaction may be "stolen" (its frame is allocated to another transaction) before the first transaction has committed (the page must then be written to disk). This reduces the space requirements of the buffer pool.
- NO STEAL: The pages modified by a transaction remain in the buffer pool at least until the transaction commits.
- FORCE: Before a transaction commits, all the pages it has modified must have been written to disk.
- NO FORCE: Some pages modified by a transaction may remain (non reflected on disk) in the buffer pool after the transaction has committed. This reduces I/O costs if such pages are needed by another transaction. The drawback is a more complex recovery procedure in the case of system failure.

Combining the three above binary choices leads to eight different policies. The (STEAL, NO FORCE) combination is frequently used, since it favors performance and allows flexible buffer management. System-R [Gray et al. 1981], the first full scale implementation of a relational database, used (ATOMIC, STEAL, NO FORCE), with shadow pages. Many current transaction managers are based on ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) [Mohan et al. 1992], which uses (NOT ATOMIC, STEAL, NO FORCE). ARIES emphasizes performance in normal operation, using direct page allocation rather than shadow paging, at the expense of a more complex recovery algorithm (see next section). This choice is justified, since failures are infrequent in current systems.

9.3.2 Logging

Logging is a general technique used to restore the state of a system after a failure which caused the contents of the volatile storage to be lost. A *log* is a sequence of records written on stable storage in the append-only mode. Each record registers a significant event that changed the state of the system. Typically, a record contains the date of the event, the state of the modified element before and after the change (possibly using a form of `diff` for economy of space), and the nature of the operation that performed the change.

In a transaction management system, the log may register the change of a logical element (a database record) or a physical element (a page); some systems register both classes of events, depending on their nature, thus constructing a "physiological" (physical + logical) log.

After a system failure, the log is used for two main purposes:

- To REDO the changes performed by a transaction that has committed at the time of the failure, but whose changes have not been written to the disk in entirety (this may occur with the NO FORCE option). This guarantees durability.

- To UNDO the changes performed by a transaction that has not committed at the time of the failure. This guarantees atomicity. UNDO also applies to aborted transactions whose initial state has not yet been restored.

In addition to logging, the recovery system may use *checkpointing* to register periodically a consistent state. This limits the number of log records to be processed after a failure, since one only needs to restart from the last checkpoint. The difficulty lies in capturing a *consistent* state of the system, which may be a complex and costly process in itself, specially in a distributed system.

The log, which is on stable storage, must contain enough information to execute the necessary UNDO and REDO operations after a failure that caused the loss of the contents of volatile storage. This is ensured by the technique of *Write Ahead Logging* (WAL), which enforces two conditions:

- A transaction is only considered committed *after* all its log records have been written to stable storage (condition for REDO).

- All log records corresponding to a modified page must be written to stable storage *before* the page is itself written to permanent storage (condition for UNDO).

Recovery proceeds in three phases. An analysis of the situation first determines the earliest log records to be used for the next two phases. A REDO (forward) phase, applied to all transactions, including those committed, ensures that all updates done before the failure are reflected on permanent storage. Then an UNDO (backward) phase ensures that the effect of all uncommitted transactions is undone. Both REDO and UNDO are idempotent; thus, if a failure occurs during one of these phases, the operation may simply be restarted without additional processing.

While the principle of log-based recovery is fairly simple, implementing a WAL-based recovery system is a challenging proposition. Note that recovery interacts with concurrency control; in that respect, *strict* two-phase locking facilitates recovery, because it prevents dirty reads (9.2.1). A detailed description of the WAL-based recovery system of ARIES may be found in [Mohan et al. 1992]. A summary description is in [Franklin 2004].

9.4 Distributed Transactions

We have assumed, up to now, that a transaction takes place on a single site. We now consider *distributed transactions*, in which the objects may reside on a set of nodes connected by a network.

In a distributed system, the transaction manager is organized as a set of cooperating local managers, each of which manages the data present on its node. These managers need a mutual agreement protocol, in order to preserve global consistency. The main subject of agreement is the decision on whether to commit or to abort, since this decision must be consistent among all nodes involved in the transaction. In other words, distributed commitment should be *atomic* (all nodes commit, or none). In addition to commitment proper, an atomic commitment protocol can be used for concurrency control, through a method called commitment ordering [Raz 1992]. Atomic commitment is therefore a central issue for distributed transactions. In the following subsections, we specify the problem and examine its main solutions.

9.4.1 Atomic Commitment

On each node of the distributed system, a process executes the local transaction manager, and communicates with similar processes on the other nodes. We assume the following properties.

- Communication is *reliable* (messages are delivered unaltered) and *synchronous* (upper bounds are known for message transmission time and ratio of processor speeds at the different nodes, see 4.1.2). In addition, we assume that communication or process failures cannot partition the network.
- Processes may fail (in the fail-stop mode, see 11.1.3). A failed process may be repaired and be reintegrated in the transaction. A correct process is one that never failed.
- Each node is equipped with stable storage, which survives failures.

The assumption of reliable, synchronous communication is needed to ensure that an agreement protocol can run in finite time (see details in 11.1.3).

The atomic commitment protocol may be started by any process involved in the distributed transaction. Each process then casts a vote: YES if it is ready to commit locally (make permanent its updates to the local data), NO otherwise. The protocol must satisfy the following requirements.

- Validity: The decision must be either *commit* or *abort*.
- Integrity: Each process decides at most once (once made, a decision cannot be revoked).
- Uniform agreement: all processes that decide (be they correct or not) make the same decision.
- Justification: If the decision is *commit*, then all processes voted YES.
- Obligation: If all processes have voted YES, and if all processes are correct, the decision must be *commit*.
- Termination: each correct process decides in finite time.

A few remarks are in order about these specifications. The "atomic" property is ensured by *uniform* agreement, in which the decision of faulty processes is taken into account. This is because a process may perform an irreversible action, based on its decision, before failing. Agreement must therefore include all processes. Note that the two possible outcomes are not symmetric: while unanimity is required for *commit*, it is not required that all processes have voted NO to decide *abort* (in fact, a single NO vote is enough). The implication in the Justification requirement is not symmetric either: the outcome may be *abort*, even if all processes have voted YES (this may happen if a process fails between the vote and the actual decision). The Obligation requirement is formally needed to exclude trivial solutions, such as all processes always deciding *abort*.

We now present some protocols for atomic commitment.

9.4.2 Two-phase Commit

The two-phase commit protocol, or 2PC, has been initially proposed by [Gray 1978]. One of the processes is chosen as *coordinator*, and it starts[11] by requesting a vote from the other processes (participants). Each participant (and the coordinator itself) sends its vote (YES or NO) to the coordinator. When the coordinator has collected all votes, it decides (*commit* if all votes are YES, *abort* otherwise), and sends the decision to all participants. Each participant follows the decision and sends an acknowledge message to the coordinator. In addition, all events (message sending and receiving) are logged. This is summarized on Figure 9.1.

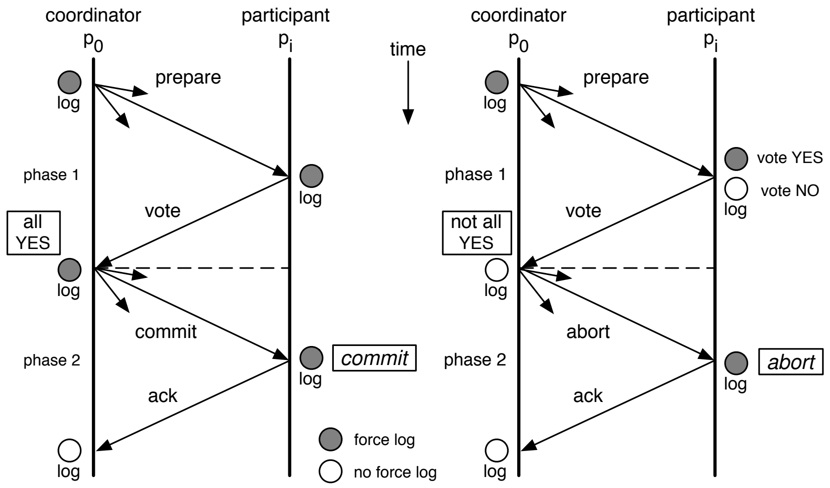


Figure 9.1. Two-phase commit: basic protocol

Logging may be forced (i.e., it takes place immediately and therefore blocks execution until finished) or non forced (it may be delayed, for example piggybacked on the next forced log).

Note that the coordinator and each participant start a round-trip message exchange, and can therefore use timeouts to detect failure. Recall that communication is assumed to be synchronous; call δ the upper bound for message propagation, and ϵ and T estimated upper bounds for processing a vote request and a vote reply, respectively. Then timeouts may be set at the coordinator and at each participant site, as shown on Figure 9.2.

If the coordinator has detected a participant's failure, it must decide *abort* (Figure 9.2a). If a participant has detected the coordinator's failure (Figure 9.2b), it first tries to determine whether a decision has been made, by consulting the other participants. If it fails to get an answer, it starts a protocol to elect a new coordinator.

[11] As noted above, the protocol may be initiated by any process, which then needs to send a "start" message to the coordinator.

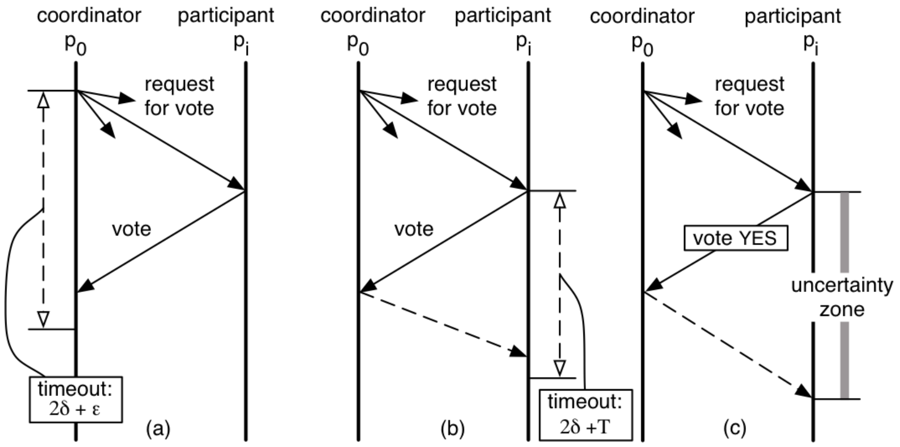


Figure 9.2. Two-phase commit: failure detection

When the coordinator or a participant restarts after a failure, it uses the log records to determine the state of the transaction, and to act accordingly. For instance, if it finds itself in phase 1, it restarts the transaction; if it finds itself in phase 2, it sends the logged decision to the participants; if the last log record is the end of the transaction, there is nothing to do.

Note that a participant that voted YES enters an "uncertainty zone" until it has received a decision from the coordinator, or until timeout (Figure 9.2c). This is because another participant may have already made either a *commit* or an *abort* decision (none is excluded for the time being). This may lead to a *blocking* scenario: suppose p_i has voted YES and is in the uncertainty zone; suppose that the coordinator made a decision, sent it to some participants, and then failed; suppose, in addition, that these latter participants also failed. Then a decision has actually been made, but p_i, although being correct, has no means to know it, at least until after the coordinator has been repaired (if the coordinator has logged the decision on stable storage before failing, the decision can be retrieved and resent to the participants).

Even if it is eventually resolved (either by retrieving the decision or by calling a new vote), a blocking situation is undesirable, because it delays progress and wastes resources that are held by blocked processes. Therefore various solutions have been devised to avoid blocking. Two of them are presented in the following subsection.

9.4.3 Non-Blocking Commitment Protocols

The first solution [Skeen 1981] proposed for avoiding blocking is the three-phase commit (3PC) protocol. If all participants have voted YES, the coordinator sends a PREPARE message to all participants; if not, it decides *abort*, like in 2PC.

When a participant receives a PREPARE message, it enters a "prepared to commit" phase, and replies with an ACK message to the coordinator. If the coordinator has received ACK from all processes, it decides *commit* and sends this decision to all participants. This protocol is summarized on Figure 9.3. Note that, if the communication system does not lose messages, acknowledgments are redundant.

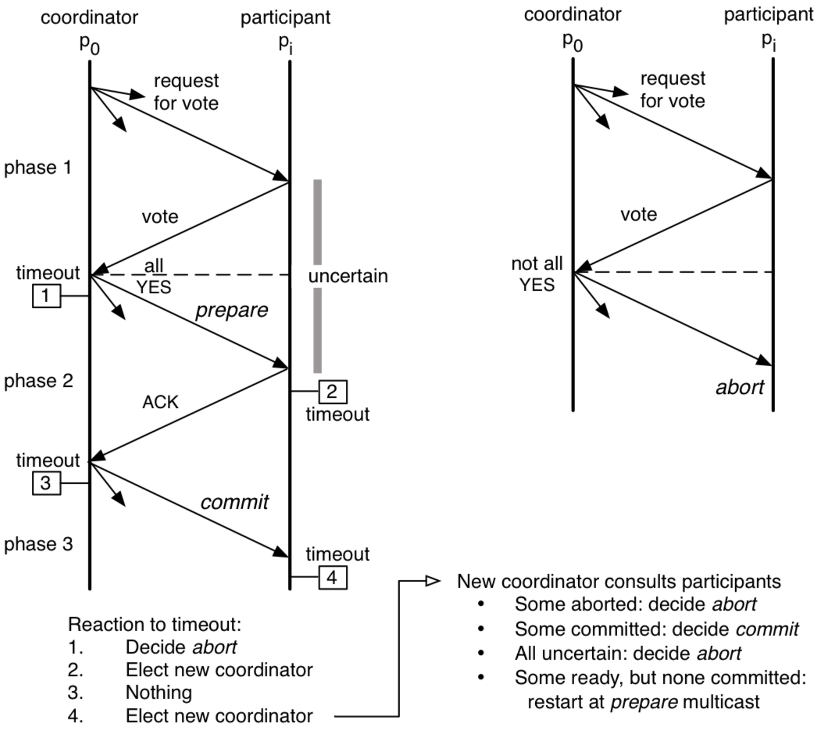


Figure 9.3. Three-phase commit

Why does this protocol eliminate blocking? The role of the "prepared to commit" phase is to reduce a participant's uncertainty as to the outcome of the transaction. After a participant p_i has voted YES, there is no situation in which p_i is uncertain and some process (say p_j) has already decided *commit*. Otherwise said: in 2PC, a process that decides *commit* "knows" that all have voted YES; in 3PC, a process that decides *commit* "knows", in addition, that all know that all have voted YES (because they all received the PREPARE message).

Dealing with failures is more complex than in 2PC, since there are more different possible states for the system. The situation is summarized on Figure 9.3.

A second, more elegant solution [Babaoğlu and Toueg 1993], consists in using a multicast protocol with strong properties, again with objective of reducing uncertainty among the participants. The algorithm is directly derived from 2PC: in the program of the coordinator, replace "send the decision (*commit* or *abort*) to the participants" by "broadcast the decision to the participants using UTRB (Uniform Timed Reliable Broadcast)". UTRB is a reliable multicast protocol (11.3.2) that has two additional properties:

- Timeliness: there exists a known constant δ such that, if the multicast of message m was initiated at real time t, no recipient process delivers m after real time $t + \delta$.
- Uniform Agreement: if any of the recipient processes (correct or not) delivers a message m, all correct recipient processes eventually deliver m.

The first property derives form the synchrony assumption for communication. The second property is the one that eliminates uncertainty, and thus prevents blocking: as soon as a participant has been notified of a decision by the coordinator, it "knows" that all correct participants will also receive the same information. If a participant times out without having received a decision, it can safely decide *abort* by virtue of the timeliness property.

9.4.4 Optimizing Two-phase Commit

As is apparent from the above description of 2PC (9.4.2), there is a significant difference, as regards the number of logging and communication operations, between the failure and no-failure cases. This observation suggest a potential way of improving the performance of 2PC in a situation in which the abort rate of transactions can be estimated, by adapting the protocol to the most likely behavior. The definition of the 2PC Presumed Commit (2PC-PC) and 2PC Presumed Abort (2PC-PA) protocols [Mohan et al. 1986] is an attempt towards this goal. The main idea is to reduce the number of FORCE log writes and to eliminate superfluous acknowledge messages.

2PC-PA is optimized for the case of abort, and is identical to standard 2PC in the case of commit. An analysis of the abort case shows that it is safe for the coordinator to log the *abort* record in the NO FORCE mode, and to "forget" about the transaction immediately after the decision to abort. Consequently, a participant which receives the *abort* message does not have to acknowledge it, and also logs it in the NO FORCE mode.

2PC-PC is optimized for the case of commit, and is identical to standard 2PC in the case of abort. In this protocol, the absence of information is interpreted as a *commit* decision. However, the 2PC-PC protocol is not exactly symmetrical to 2PC-PA, in order to avoid inconsistency in the following situation: the coordinator crashes after having broadcast the *prepare* message, but before having made a decision; upon recovery, it aborts the transaction and forgets about it, without informing anyone, since it does not know the participants. However, if a prepared participant times out and inquires the coordinator, it will get (by default) the *commit* answer, which is inconsistent.

To avoid this problem, the coordinator records (by a FORCE log) the names of the participants, before sending the prepare message. It will then know whom to inform of the abort decision after recovery from a crash. If the coordinator decides to commit, it FORCE logs this decision and sends it to the participants. Upon receipt, a participant FORCE logs the decision, without sending an acknowledge message. If the decision is to abort, the coordinator informs the participants that voted YES and waits for the acknowledgments. It then logs the decision in the NO FORCE mode.

Commit protocol	Messages		Forced log writes	
	Commit	Abort	Commit	Abort
2PC	$4p$		$1 + 2p$	
2PC-PA	$4p$	$3p$	$1 + 2p$	p
2PC-PC	$3p$	$4p$	$2 + p$	$1 + 2p$

Table 9.1. Compared costs of 2PC protocols (from [Serrano-Alvarado et al. 2005])

The compared costs of the protocols are summarized on Table 9.1, which shows the gains of 2PC-PA and 2PC-PC with respect to standard 2PC, in the case of abort and commit, respectively. In this table, p denotes the number of participants.

To exploit this optimization, [Serrano-Alvarado et al. 2005] propose to dynamically adapt the commit protocol to the behavior of the application, by selecting 2PC-PA or 2PC-PC according to the observed abort rate. As a result, the average completion time of a transaction is reduced with respect to a non-adaptive commit scheme. The commit protocols are implemented in terms of Fractal components within the GoTM framework (see 9.7): each protocol contains the same basic components, with a different configuration.

9.5 Advanced Transaction Models

The transaction model described in sections 9.2 and 9.3 was developed for centralized databases used by short transactions with a moderate amount of concurrency. This model (called "traditional", or "flat") was found overly constraining for the complex, long-lived applications that came to existence with the rise of large scale distributed systems. Such applications are typically developed by composing several local applications, each of which may need to preserve some degree of autonomy. Relevant application domains include computer aided design and manufacturing (CAD/CAM), software development, medical information systems, and workflow management.

The coupling between the local applications may be strong (e.g., synchronous client-server) or loose (e.g., message-based coordination). The running time of the composite application may be long (hours or days, instead of seconds for traditional applications). The probability of failures increases accordingly. Blocking resources for the entire duration of the application would be overly expensive.

As a consequence, the application may be subject to one or more of the following requirements.

- A local application may be allowed to fail without causing the failure of the global application.
- After a failure, the global application may be allowed to roll back to an intermediate checkpoint, instead of undoing the whole work performed since its beginning.
- Some partial results may need to be made visible before the global application concludes.

Thus the atomicity and isolation requirements of the traditional transaction model may need to be relaxed for a global application, even if they remain valid for the composing sub-applications.

Presentations of advanced transaction models often use the canonical example of a travel agency, which prepares a travel plan by reserving flight tickets, hotel rooms, car rentals, show tickets, etc. For efficiency, reservations for the various kinds of resources can be done in parallel, each in a separate transaction. On the other hand, these transactions are not independent: if a hotel room cannot be found at a certain place, one may try a different location, which in turn would need changing the flight or car rental reservation, even if the corresponding transaction has already committed.

A number of models that relax the ACID properties in some way have been proposed under the general heading of "advanced transaction models". [Elmagarmid 1992] and [Jajodia and Kerschberg 1997] are collections of such proposals, few of which have been tested on actual applications. In the following subsections, we present two models that have been influential for further developments. Both of them rely on a decomposition of a global transaction into sub-transactions. Nested transactions (9.5.1) use a hierarchical decomposition, while sagas (9.5.2) use a sequential one.

9.5.1 Nested Transactions

The nested transactions model has been introduced in [Moss 1985]. Its motivation is to allow internal parallelism within a transaction, while defining atomicity at a finer grain than that of the whole transaction.

The model has the following properties.

- A transaction (the parent) may spawn sub-transactions (the children), thus defining a tree of transactions.
- The children of a transaction may execute in parallel.
- A transaction T_j that is a sub-transaction of T_i starts after T_i and terminates before T_i.
- If a transaction aborts, all of its sub-transactions (and, by recursion, all its descendants) must abort; if some of these transactions have committed, the changes they made must be undone.
- A nested transaction (one that is not the root of the tree) may only (durably) commit if its parent transaction commits. By virtue of recursion, a nested transaction may only commit if all of its ancestors commit.

Thus the model relaxes the ACID properties, since durability is not unconditionally guaranteed for a committed nested transaction. The top-level (the root) transaction has the ACID properties, while the nested transactions (the descendants) only have the AI properties.

Several variants of this model have been proposed. In the simplest form, only leaf transactions (those at the lowest level) can actually access data; transactions at the higher level only serve as controlling entities. Other forms allow all transactions to access data, which raises the problem of concurrency control between parent and child transactions.

Concurrency control may be ensured by a locking protocol that extends 2PL with lock transmission rules between a transaction and its parent and children. A sub-transaction "inherits" the locks of its ancestors, which allows it to access data as if these locks were its own. When a sub-transaction commits, its locks are not released, but "inherited" (upward transmission) by its parent. When a sub-transaction aborts, only the locks it directly acquired are released; inherited locks are preserved. As a consequence, if two sub-transactions are in conflict for access to a shared object, one of them is blocked until their lowest common ancestor has committed.

The benefits of nested transactions are the possibility of (controlled) concurrent execution of sub-transactions, the ability to manage fine-grain recovery within a global transaction, and the ability to compose a complex transaction out of elementary ones (modularity).

9.5.2 Sagas

The traditional transaction model is not well suited for long-lived transactions, for two main reasons: (1) to maintain atomicity, some objects need to be locked for a long time, preventing other transactions from using them; and (2) the probability of a deadlock increases with the length of the transaction, thus leading to a potentially high abort rate. The Sagas model [García-Molina and Salem 1987] was proposed to alleviate these drawbacks. In this model, a global transaction (called a *saga*) is composed of a sequence of sub-transactions, each of which follows the traditional model. Thus a saga S may be defined as follows:

$$S = (T_1; T_2; \dots T_n)$$

where the T_is are the sub-transactions of S.

A saga only commits if all of its sub-transactions commit; thus a saga is atomic. However, the execution of a saga can be interleaved with the execution of other sagas. As a consequence, sagas do not have the isolation property, since a saga may "see" the results of another, partially completed, saga.

If a sub-transaction of a saga aborts, the entire saga aborts, which means that the effect of the sub-transactions executed up to the current point needs to be canceled. However, since these transactions have committed, a special mechanism must be introduced: for each sub-transaction T_i, one defines a *compensating transaction* C_i whose purpose is to cancel the effect of the execution of T_i.

Thus the effect of the execution of a saga $S = (T_1; \dots T_n)$ is equivalent to either that of

$$T_1; T_2; \dots T_n,$$

in which case the saga commits and achieves its intended purpose, or that of

$$T_1; T_2; \dots T_k; C_k; C_{k-1} \dots C_1 \qquad (k < n),$$

in which case the sub-transaction T_k has aborted, and its effect, as well as that of the preceding sub-transactions, has been compensated for[12].

Recall that isolation is not maintained for sagas. Two consequences follow:

- Compensation of a saga S does not necessarily bring the system back to its state before the execution of S, since sub-transactions of other sagas may have taken place in the meantime.
- Other sagas may have read the results of committed sub-transactions of a saga S, before they were compensated for. Applications are responsible for dealing with this situation.

[12]Note that there is no need for a compensating transaction for the last sub-transaction T_n.

The main practical problem with building applications based on sagas is to define compensating transactions. This may be difficult, or even impossible if a sub-transactions has executed "real" actions.

The main current application of the "advanced transactions" models is in the area of web services transactions (see 9.6.2).

9.6 Transactional Middleware

As seen above, the notion of a transaction has evolved from the traditional model to a variety of "advanced" models, in order to comply with the requirements of complex, composite applications. Transaction support software has evolved accordingly. In centralized databases, transactions were supported by a set of modules closely integrated within database management systems. As systems became distributed, and as transactions were applied to non-database services such as messaging, transaction management was isolated within specialized middleware components. A transaction processing system is typically organized into a transaction manager (TM) and a set of resource managers (RM). Each resource manager is in charge of a local resource, such as a database or a message queuing broker, and is responsible for ensuring transactional properties for access to the local resource. The transaction manager ensures global coordination between the resource managers, using the 2PC protocol.

Several standards have been developed over time to provide a formal support to this organization. The Open Group Distributed Transaction Processing (DTP) standard [X-OPEN/DTP] defines the interfaces between the application and the TM (TX interface), and between the TM and the RMs (XA interface), as shown on Figure 9.4.

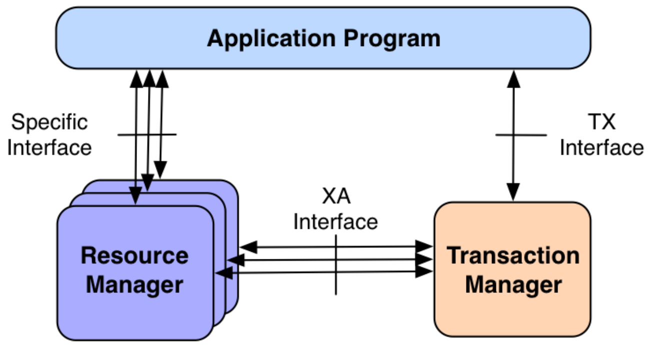


Figure 9.4. Transactional interfaces in X-Open DTP

In the rest of this section, we examine two frameworks that have been developed in recent years for transaction management. These frameworks are representative of two families of middleware systems. In 9.6.1, we describe the principles of transaction management in JEE, a platform for component-based programming. In 9.6.2, we present WS-TX, a standard proposed for web services transactions. We conclude in 9.6.3 with a brief outline of current advances in transactional middleware.

9.6.1 Transaction Management in JEE

JEE [JEE] is a platform for developing distributed, component-based applications in Java. It uses containers (7.5.1) to manage application-built components called *enterprise beans*[13]. Beans contain the code of the application and call each other using (possibly remote) method invocation.

Each bean is is managed by a container, which intercepts all method calls and acts as a broker for system services. There are two kinds of transactions, which differ by the method used for transaction demarcation.

- Bean-managed transactions, in which transaction demarcation instructions (*begin*, *commit*, *rollback*) are explicitly written by the developer of the bean, using the `javax.transaction.UserTransaction` interface.
- Container-managed transactions, in which transaction demarcation is done by the container, using declarative statements provided by the developer, as annotations attached to the bean.

In container-managed transactions, the main issue is to determine whether a method of a bean should be executed within a transaction, and if yes, whether a new transaction should be started. This is specified by a transaction attribute, attached to the method (a transaction attribute may also be attached to a bean class, in which case it applies to all the application methods of the class). The effect of the attribute is described in Table 9.2.

Transaction attribute	Calling activity	Method's transaction
`Required`	None	$T2$
	$T1$	$T1$
`RequiresNew`	None	$T2$
	$T1$	$T2$
`Mandatory`	None	Error
	$T1$	$T1$
`NotSupported`	None	None
	$T1$	None
`Supports`	None	None
	$T1$	$T1$
`Never`	None	None
	$T1$	Error

Table 9.2. Transaction attributes in container-managed transactions

The column "Calling status" indicates whether the call is done within a transaction (denoted as $T1$) or outside a transaction. The column "Method's transaction" indicates whether the method should be run within a transaction (which may be the calling transaction, $T1$, or a new transaction, $T2$).

For instance, suppose the called method has the `Required` attribute. Then, if it is called from outside a transaction, a new transaction must be started. If the method is called

[13] There are several kinds of beans (differing by persistence properties). We ignore these distinctions, which are not relevant in this presentation.

from within a transaction, it must be executed under that transaction. Some situations cause an error, in which case an exception is raised; if a transaction was running, it will automatically be rolled back.

In order to explicitly abort a container-managed transaction (for example, if the bean throws an application exception) the application should invoke the `setRollbackOnly` method of the `EJBContext` interface.

9.6.2 Web Services Transactions

Web services [Alonso et al. 2004] provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. In a more specific sense (see [W3C-WSA 2004]), Web services define a set of standards allowing applications to be integrated and executed over the Internet, following a service oriented architecture such as described in 3.3.4.

Web services (which would be more appropriately called Internet services) typically use a loosely-coupled mode of communication, in which applications execute over long periods and may negotiate the terms and conditions of their interaction. The conventional transaction model based on ACID properties proves too constraining for Web services. Therefore, new standards are being developed to support extended transaction models adapted to the Web services environment.

The emerging standard for Web services transactions is the set of Web Services Transaction specifications (collectively known as WS-TX) produced by OASIS (Organization for the Advancement of Structured Information Standards [OASIS]), a consortium that develops "open standards for the global information society".

The WS-TX collection is composed of three standards.

- WS-Coordination (WS-C) [OASIS WS-TX TC 2007c] This specification describes an extensible framework used to coordinate a number of parties participating in a common activity. The framework has two parts: (a) a generic part, which allows participants to share a common context and to register for a common activity, possibly controlled by a coordinator; and (b) a specific part, which defines a particular protocol in the form of a "coordination type". WS-Coordination is the base on which the transaction services described below are built, as specific coordination types. WS-Coordination is presented in 6.7.

- WS-AtomicTransaction (WS-AT) [OASIS WS-TX TC 2007a]. This specification defines an atomic transaction protocol, to be used for short duration transactions between participants with a high degree of mutual trust. It is based on 2PC (9.4.2), with two variants: Volatile 2PC, for participants managing volatile resources (e.g., caches), and Durable 2PC, for participants managing durable resources (e.g., databases).

- WS-BusinessActivity (WS-BA) [OASIS WS-TX TC 2007b]. This specification defines a protocol (in the form of a set of coordination types) to build applications involving long-running, loosely coupled distributed activities, for which the conventional ACID transaction model is inadequate. WS-BusinessActivity is based on an extended transaction model (9.5) in which the results of an activity may be visible before its completion, and the effect of a completed activity may be undone by means of compensation, such as defined in the saga model (9.5.2).

Separating WS-Coordination from the specific transaction protocols has two main benefits:

- Separation of concerns. The functions related to coordination are isolated and may serve for other purposes than transaction management.

- Extensibility. New transaction protocols, in addition to WS-AT and WS-BA, may be added in the future as coordination types in WS-C.

While WS-AT is essentially a conventional 2PC protocol for ACID transactions, its main advantage is to allow interoperability between applications that participate in a common transactional activity, by wrapping them into Web services. Thus independently developed applications, using different platforms and standards, may be made to interoperate in a closely coupled environment.

Since WS-BA addresses applications that integrate loosely coupled, independent participants running in different domains of trust, flexibility is an important requirement. It is achieved through the main following features:

- Nested scopes. A long running activity may be structured as a set of tasks, each of which is a short duration unit of work, which may typically follow a conventional transaction model. A task may itself be organized in a hierarchy of (child) tasks. Each task, which uses a collection of Web services, defines a *scope*. Scopes can be arbitrarily nested. This model is close to that of nested transactions (9.5.1), but is more flexible, since a parent may catch an error in a child task, and decide on whether to compensate, to abort, or to produce a non-atomic outcome.

- Flexible coordination model. WS-BA supports two coordination types, which define the behavior of the coordinator of a transaction. In the AtomicOutcome type, the coordinator must direct all participants either to close (i.e., to complete successfully) or to compensate. In the MixedOutcome type (optional), the coordinator may direct each individual participant to either close or compensate.

- Custom termination protocol. Two protocols are defined, each of which can use one of the two above coordination types. The difference between these protocols is about which entity decides whether a participant's activity is terminated. In the BusinessAgreementWithParticipantCompletion (BAwPC) protocol, the decision is made by the participant, which notifies the coordinator. In the BusinessAgreementWithCoordinatorCompletion (BAwCC) protocol, the decision is made by the coordinator, which has information about the tasks requested from the participant. In both cases, according to its decision about the outcome of the global activity, the coordinator requests the participant either to close (successfully complete) or to compensate the task.

The life-cycle of a transaction, as seen by the coordinator or by a participant, is represented as a state diagram. Transitions between states are triggered by messages sent either by a participant or by the coordinator.

In the BAwPC protocol (Figure 9.5), when an active participant decides that it has completed its work, it notifies the coordinator by a *Completed* message and goes to the Completed state. The coordinator then decides to either accept the participant's work (message *Close*) or to request that the work be compensated for[14] (message *Compensate*). In both cases (in no error occurs during compensation) , the participant eventually goes to the *Ended* state, in which it forgets about the transaction. Other states shown on the figure are introduced to deal with abnormal events at various stages of the work.

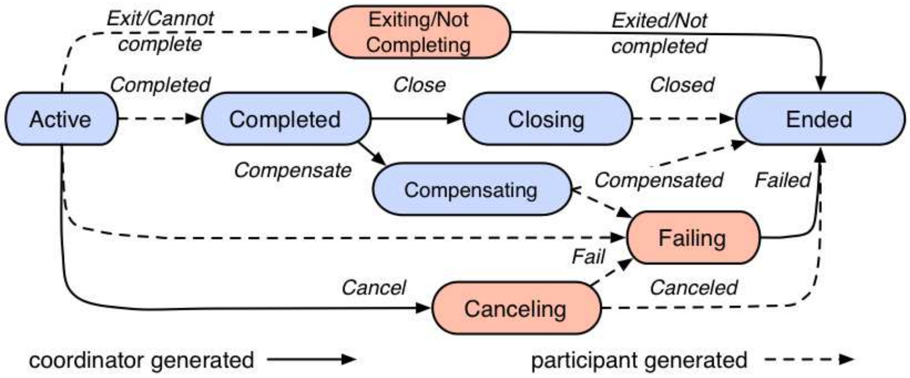


Figure 9.5. BusinessAgreementWithParticipantCompletion: state diagram (adapted from [OASIS WS-TX TC 2007b])

In the BAwCC protocol (Figure 9.6), the decision about termination is made by the coordinator, which sends a *Complete* message to the participant, which then goes to a Completing stage. In the absence of errors, the participant answers with a *Completed* message and goes to the completed state. The rest of the protocol is as described above. Various errors may occur during the Completing phase, leading to the additional transitions shown on the figure.

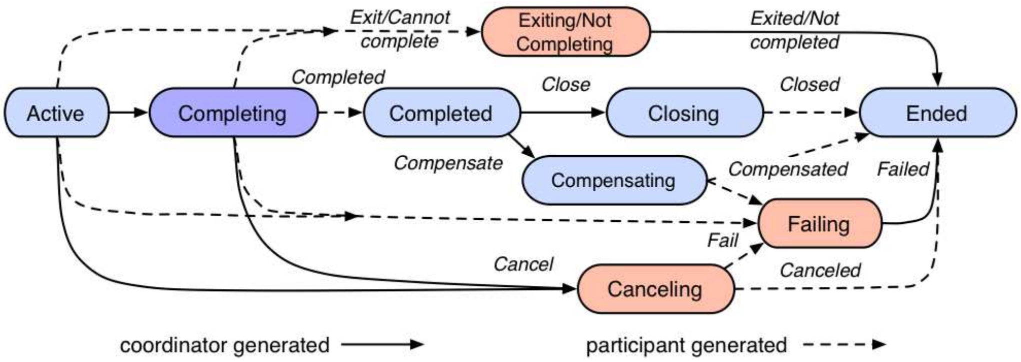


Figure 9.6. BusinessAgreementWithCoordinatorCompletion: state diagram (adapted from [OASIS WS-TX TC 2007b])

An open source implementation of the WS-TX protocols is underway at

[14]The WS-BA standard does not define the compensation operations, which are the responsibility of the application's logic.

[Kandula 2007]. It includes an extension to the WS-BA protocol, described in [Erven et al. 2007], which introduces a new role called the initiator to facilitate the development of applications using WS-BA.

9.6.3 Advances in Transactional Middleware

The emergence of new applications areas has motivated a variety of requirements regarding transactions. Several standards have been proposed to satisfy these requirements, which cover a wide spectrum. As a consequence, it has become clear that no single transaction model is able to cover all possible cases. Transactional middleware is therefore required to be adaptable and extensible. More precisely, it should satisfy the following requirements.

- Allowing various transactional services to be built on a common infrastructure, by composing a set of parts.
- Allowing the coexistence of several "personalities" of transaction support systems, conforming to various transaction standards and providing different guarantees to the applications.
- Allowing run-time extension and adaptation to react to changes in the environment.

To comply with these requirements, advances have been done in two complementary directions.

The first direction is to define new abstractions in order to identify commonalities in the transaction models and tools, and to better specify these models and tools. Separating coordination from transaction management in the WS-TX protocols (9.6.2) is a step in this direction. Another example of this approach is the abstraction of transaction demarcation (i.e., the policies that specify whether and how an action is to be executed under an active transaction), in [Rouvoy and Merle 2003].

The second direction is to design middleware infrastructures to support adaptable and extensible transaction systems, specially through the use of components. Two recent examples are the Argos framework [Arntsen et al. 2008] and the GoTM framework [Rouvoy and Merle 2007]. GoTM is presented in the next section.

9.7 Case Study: GoTM, a Framework for Transaction Services

GoTM (GoTM is an open Transaction Monitor) is a component-based framework for building highly adaptable and extensible transaction services. To achieve this goal, GoTM relies on an abstract architecture for transaction services, in the form of a set of common design patterns, which are reified as assemblies of fine-grained components based on the Fractal component model (7.6). Thus GoTM provides an extensible component library, which can be used to develop the main functions required from a transaction service.

The main aspects of a transaction service that may be subject to variation are the following:

- Transaction standard. Various standards have been defined for different environments. These standards specify the user interface of the transaction service. Examples include Object Transaction Service (OTS) for Corba, Java Transaction Service for JEE, and Web Services Atomic Transaction for Web Services. Some of these standards rely on common notions (e.g., the flat transaction model) or common mechanisms (e.g., transaction creation, context management).

- Transaction model. Various transaction models have been developed (see 9.5) to fit different application requirements. One major aspect of variation is the degree of isolation. Again, the goal is to allow different transaction models to coexist in a transaction service.

- Commitment protocol. As noted in 9.4.4, the two-phase commit protocol has a number of variants optimized for various execution environments. The abstract notion of a 2PC protocol captures the core notions that are common to these variants.

GoTM allows building transaction services that can be adapted according to the above criteria. This is achieved by the systematic use of reification of different notions in the form of fine-grain components. This reification process is applied at two levels:

- For the design patterns used to specify the architecture of the transaction service. Reifying the patterns in the form of components allows using component description and composition tools to adapt the structure of the transaction service.

- For the functions and the execution states of a transaction service. Since a fine-grain component represents an elementary aspect, it may be easily shared among different transaction services.

Thus the adaptation mechanisms of GoTM essentially apply to the architecture of the transaction services, rather than to the contents of the elements.

The next two subsections describe the architecture of the GoTM framework (9.7.1) and the main features of its implementation (9.7.2). The last subsection (9.7.3) shows how GoTM may be used for transaction service adaptation. This presentation only describes the main elements of GoTM. Refer to [Rouvoy 2006] and [Rouvoy and Merle 2007] for details.

9.7.1 Architecture of GoTM

GoTM is organized in two levels: a static part, the *Transaction Service*, whose function is to create transactions; and a dynamic part, the *Transactions* created by the transaction service. Both parts rely on design patterns, as shown on Figure 9.7.

The front-end components in both parts of GoTM are based on the Façade design pattern [Gamma et al. 1994]. The role of this pattern is to provide a simple, unified interface to a complex system that may itself involve a number of interfaces[15]. GoTM uses Façade to implement various transactions standards (e.g., JTS, OTS, WS-AT) in

[15] Façade is thus related to the Adapter pattern (2.3.3). While Adapter gives access to an existing interface by way of another existing interface, Façade builds a new interface to integrate a set of existing interfaces.

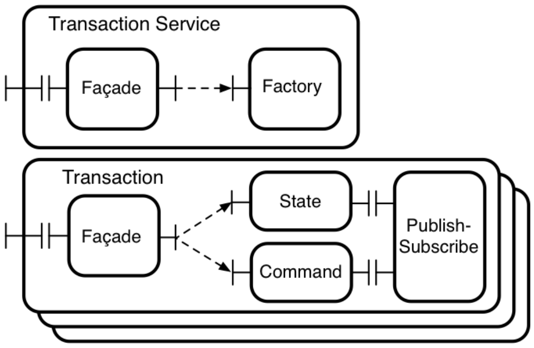


Figure 9.7. Patterns in the architecture of GoTM (adapted from [Rouvoy and Merle 2007])

terms of the interfaces exported by its component library. A specific FAÇADE component is automatically generated for each standard (e.g., *JTS-Façade*, *OTS-Façade*, etc.).

The core of the transaction service (Figure 9.8) is a *Transaction Factory* component, based on the FACTORY design pattern (2.3.2). This component creates transaction instances conforming to a specified transaction model (also represented as a component). The transaction model acts as a set of templates, which may be cloned to create components of transaction instances. For better performance, the factory is enhanced with caching and pooling facilities. Note that this construction makes use of the ability of the Fractal model to share components (the component *Factory* is shared between the three transaction services of the different standards).

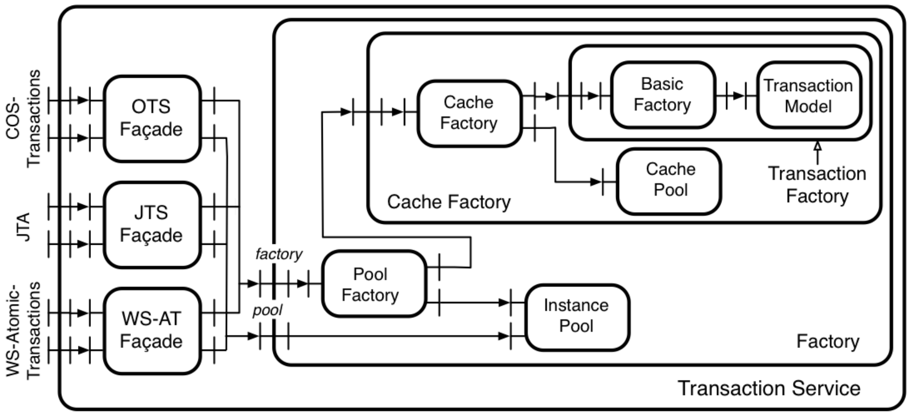


Figure 9.8. The GoTM transaction service (adapted from [Rouvoy and Merle 2007])

As an example, Figure 9.9 shows the transaction model component for JTS transactions. This model acts as a prototype to create instances of transactions using the JTS standard and its JTA interface. Note the conventions for the figures: a star denotes an interface of type *Collection*, and components shared with other models are enclosed by dotted lines.

In addition to FAÇADE, transaction instances use three main design patterns: STATE, COMMAND, and PUBLISH-SUBSCRIBE.

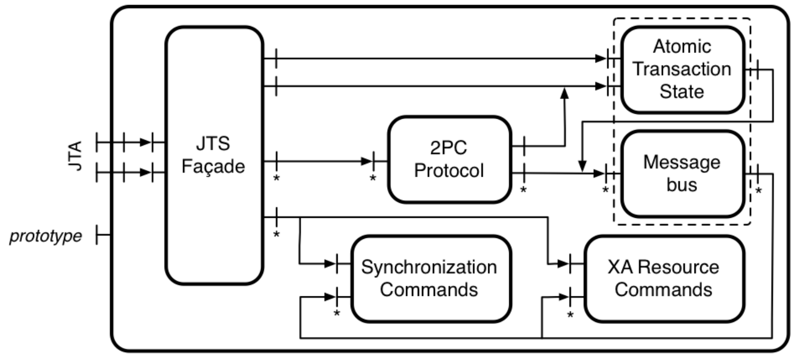


Figure 9.9. A prototype for JTS transactions in GoTM (adapted from [Rouvoy 2006])

The function of STATE is to represent and to control the different states of a transaction. Using this pattern (embodied in the component *AtomicTransactionState*), GoTM implements the state automaton defined by a transaction model (Figure 9.10). Each state is reified by a component, while the transitions between states are represented by bindings between these components (further comments on this technique in 9.7.2).

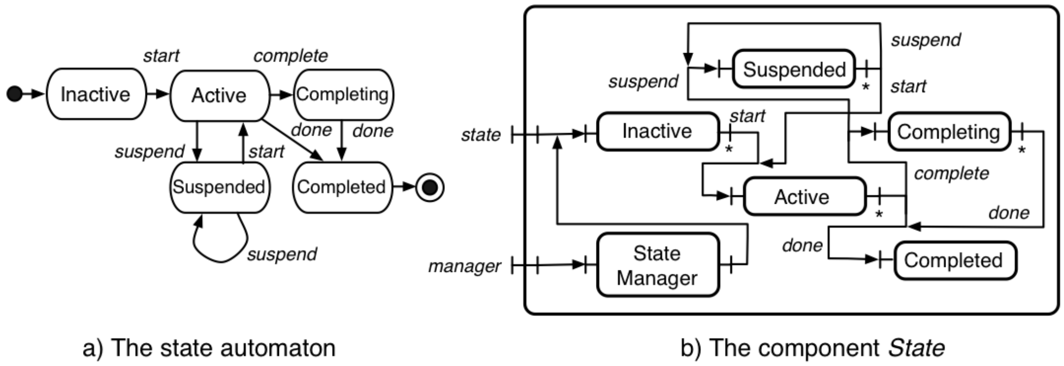


Figure 9.10. Controlling state in GoTM (adapted from [Rouvoy and Merle 2007])

The function of COMMAND is to encapsulate a command (and its parameters) into an object. This provides an abstraction to build generic components, and facilitates the building of undoable commands (e.g. by storing the command objects into a stack).

In GoTM, COMMAND is used to notify the participants to a transaction of significant events, defined as state changes. This applies both to actual participants (those involved in the transaction) and to "synchronization participants", which are only notified of the beginning and the end of the transaction.

The use of the PUBLISH-SUBSCRIBE pattern is described in the next section as an illustration of the implementation techniques of GoTM.

9.7.2 Implementation of GoTM

The implementation techniques used in GoTM are intended to make evolution easy. Using a component-based architecture is a first step towards that goal. In addition, a special

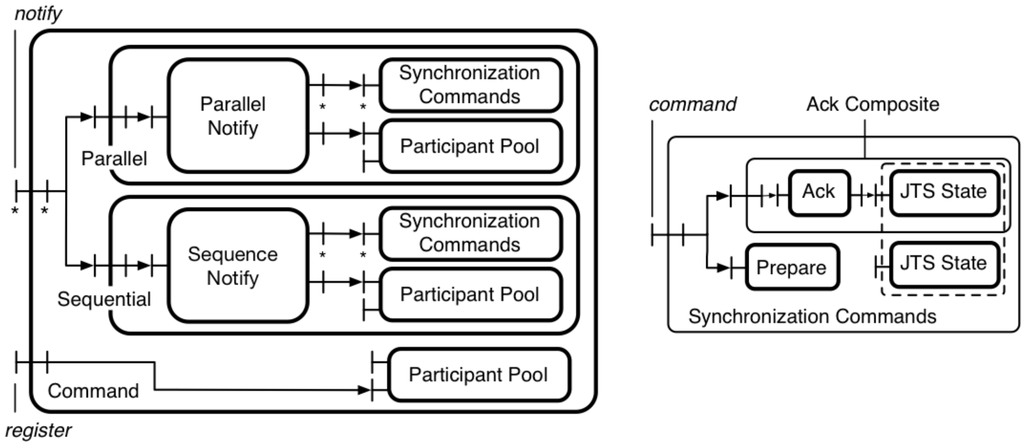


Figure 9.11. Using the COMMAND pattern in GoTM (adapted from [Rouvoy 2006])

effort has been made towards a very fine grain definition of the components and interfaces, guided by the principle of separation of concerns. Each function of a transaction service is reified in the form of a component. Within a transaction model, reusable functions are identified, and implemented as fine-grained shared components. The states that are defined by a specific transaction model are likewise reified. Thus a transaction service is described as an assembly of "micro-components".

This approach has the following benefits:

- The interfaces of the micro-components are simple (in practice, no more than four operations per interface).
- Component sharing and reuse is favored (and made technically easy by the component sharing facility of the Fractal model).
- Configuration attributes are reified into bindings, thus allowing the use of an ADL as an evolution tool.

This is illustrated by the example of the *Publish-Subscribe* component of transactions (Figure 9.12). *Publish-Subscribe* is used to synchronize the transaction participants during the execution of the Two-phase commit protocol (9.4.2).

In the implementation shown, the *publish* operation may be synchronous or asynchronous. Selecting one of these options could have been done through a parameter in the *publish* interface. Instead, the options are provided through two micro-components, *Synchronous* and *Asynchronous*, which have a common interface *publish*, of type *Collection*. The client binds to both components, through client interfaces (*pub-sync* and *pub-async*, respectively), and selects the component to use according to its needs (for instance, emphasizing safety or performance). Note that bindings may be modified at run time, e.g., to introduce a new option. The role of the *State Checker* components is to ensure that the execution of *publish* conforms to the state automaton that specifies the semantics of this operation. Components *State* and *Subscriber Pool* are shared (this is denoted by dotted lines on the figure).

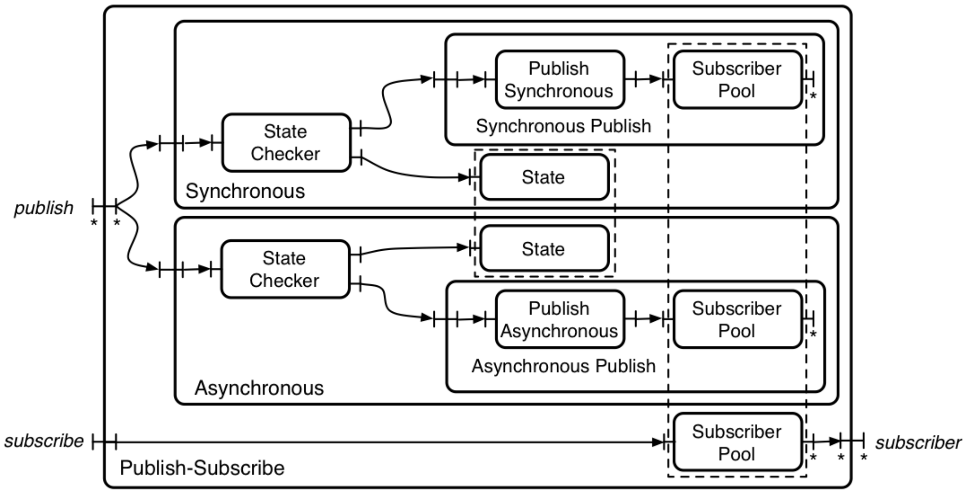


Figure 9.12. Publish-Subscribe in GoTM (from [Rouvoy 2006])

9.7.3 Using GoTM

We illustrate the adaptation capabilities of GoTM with three simple examples.

- Adapting the commitment protocol of a transaction. The algorithm of the commitment protocol is isolated through the STRATEGY design pattern. Thus, to modify this algorithm (e.g., to use 2PC-PA instead of 2PC-PC), one only needs to update the description of the component which implements the protocol. This is done (statically) by modifying one line in the ADL description of the framework. This may also be done at execution time (see [Rouvoy 2006] for details).

- Adapting the management of the participants to a transaction. This may be done at two levels. One may first add new commands in the *Synchronization Commands* components (each command is itself represented as a component, according to the COMMAND pattern). One may also replace the whole component *Synchronization Commands*, in order to modify the type of the participants of a transaction (i.e. the available commands). Both extensions are again performed by a simple, local, modification of the ADL description.

- Adapting the transaction standard. As shown in 9.7.1, this is done by providing a model for the new standard, together with a front end implemented as a new *Façade* component.

In conclusion, the separation of concerns which guided the design of GoTM, together with the systematic application of design patterns and the implementation techniques using micro-components achieve the goal of flexibility set up by the designers of GoTM. Experience reported in [Rouvoy et al. 2006a, Rouvoy et al. 2006b] shows that these benefits do not entail a significant performance overhead.

GoTM is described in [Rouvoy 2006, Rouvoy and Merle 2007]. It is available under an open source LGPL license at [GoTM].

9.8 Historical Note

The notion of a transaction as a well-identified processing unit acting on a set of data was empirically used in commercial exchanges, even before computers. With the advent of databases in the late 1960s and early 1970s, this notion was embodied in the first transaction processing monitors, such as CICS, developed by IBM. However, although atomicity was identified as an important issue, no formal model existed, and the implementations relied on empirical rules.

The first attempt at establishing a formal support for concurrency control is the landmark paper [Eswaran et al. 1976]. It lays the foundations of the theory of serializability, which is still the base of today's approach to concurrency control, and introduces two-phase locking. Fault tolerance issues were also beginning to be investigated at that time (see 11.10), and found their way into the database world. Thus [Gray 1978] introduces the DO-UNDO-REDO and WAL (write-ahead logging) protocols. The ACID properties are identified in the early 1980s (e.g., [Gray 1981]). An early survey on recovery techniques is [Haerder and Reuter 1983]. All these notions were applied to the design of large scale database management systems, among which System-R [Gray et al. 1981], which has been extensively documented.

Distributed commitment, the key problem of distributed transaction processing, is examined in [Gray 1978], which introduces two-phase commit (2PC). This stimulated research on non-blocking protocols, first leading to three-phase commit (3PC) [Skeen 1981], and later to other developments described further on.

The first distributed databases were developed in the early 1980s. R* [Mohan et al. 1986] was a distributed extension of System-R. Other influential distributed transactional systems include Camelot, later known as Encina [Eppinger et al. 1991] and UNITS (Unix Transaction System), later known as Tuxedo [Andrade et al. 1996] Both Encina and Tuxedo have evolved into products that are still in current use.

Transaction models relaxing the ACID rules (known as "advanced" models) were investigated in the 1980s. The most influential ones (9.5) are nested transactions [Moss 1985] and sagas [García-Molina and Salem 1987].

In the early 1990s, considerable progress was made in the understanding of the fundamental aspects of distributed transactions. Non-blocking atomic commitment was analyzed in detail [Babaoğlu and Toueg 1993, Guerraoui and Schiper 1995] and its relationship with consensus was clarified [Guerraoui 1995]. For recent advances in this area, see [Gray and Lamport 2006].

The development of middleware platforms for distributed applications, which started in the mid-1990s, called for advances in transactional middleware. After experience with technical solutions has been gathered, the main effort was devoted to the elaboration of standards. This work is still ongoing. "Advanced" transaction models, which are well adapted to the constraints of long transactions, are finding their way into web services.

The mainstream research in the field of transactions has now shifted to transactional memory, a mechanism using transactions to manage concurrent memory access, a potential bottleneck for applications using multicore processors. Transactional memory may be implemented by hardware, by software, or by a combination of both. See [Larus and Rajwar 2007] for a survey of this area.

Historical sources for various aspects of transactions may be found in the historical notes of [Gray and Reuter 1993]. A recent detailed historical account of transaction management is [Wang et al. 2008].

References

[Alonso et al. 2004] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2004). *Web Services*. Springer. 354 pp.

[Andrade et al. 1996] Andrade, J. M., Carges, M., Dwyer, T., and Felts, S. D. (1996). *The Tuxedo System: Software for Constructing and Managing Distributed Business Applications*. Addison-Wesley. 444 pp.

[Arntsen et al. 2008] Arntsen, A.-B., Mortensen, M., Karlsen, R., Andersen, A., and Munch-Ellinsen, A. (2008). Flexible transaction processing in the Argos middleware. In *Proceedings of the 2008 EDBT Workshop on Software Engineering for Tailor-made Data Management, Nantes, France*, pages 12–17.

[Babaoğlu and Toueg 1993] Babaoğlu, Ö. and Toueg, S. (1993). Non-Blocking Atomic Commitment. In Mullender, S., editor, *Distributed Systems*, pages 147–168. Addison-Wesley.

[Berenson et al. 1995] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A critique of ANSI SQL isolation levels. In *SIGMOD'95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, New York, NY, USA. ACM.

[Bernstein and Goodman 1981] Bernstein, P. A. and Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221.

[Bernstein et al. 1987] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley. 370 pp.

[Besancenot et al. 1997] Besancenot, J., Cart, M., Ferrié, J., Guerraoui, R., Pucheral, Ph., and Traverson, B. (1997). *Les systèmes transactionnels : concepts, normes et produits*. Hermès. 416 pp.

[Courtois et al. 1971] Courtois, P. J., Heymans, F., and Parnas, D. L. (1971). Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668.

[Elmagarmid 1992] Elmagarmid, A. K., editor (1992). *Database transaction models for advanced applications*. Morgan Kaufmann. 611 p.

[Eppinger et al. 1991] Eppinger, J. L., Mummert, L. B., and Spector, A. Z. (1991). *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann.

[Erven et al. 2007] Erven, H., Hicker, G., Huemer, C., and Zapletal, M. (2007). The Web Services-BusinessActivity-Initiator (WS-BA-I) Protocol: an Extension to the Web Services-BusinessActivity Specification. In *Proceedings of the 2007 IEEE International Conference on Web Services (ICWS 2007)*, pages 216–224. IEEE Computer Society.

[Eswaran et al. 1976] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. (1976). The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633.

[Fekete et al. 2005] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. (2005). Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528.

[Franklin 2004] Franklin, M. J. (2004). Concurrency control and recovery. In Tucker, A. J., editor, *Computer Science Handbook*, chapter 56. Chapman and Hall/CRC. 2nd ed., 2752 pp.

[Gamma et al. 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 416 pp.

[García-Molina and Salem 1987] García-Molina, H. and Salem, K. (1987). Sagas. In Dayal, U. and Traiger, I. L., editors, *SIGMOD'87: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259. ACM Press.

[GoTM] GoTM. The GoTM Project. http://gotm.objectweb.org/.

[Gray 1978] Gray, J. (1978). Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK. Springer-Verlag.

[Gray 1981] Gray, J. (1981). The transaction concepts: Virtues and limitations. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05)*, pages 144–154. Cannes, France, September 9–11 (invited paper).

[Gray and Lamport 2006] Gray, J. and Lamport, L. (2006). Consensus on transaction commit. *ACM Transactions on Database Systems*, 31:133–160. ACM Press.

[Gray et al. 1981] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., and Traiger, I. (1981). The recovery manager of the System-R database manager. *ACM Computing Surveys*, 13(2):223–242.

[Gray and Reuter 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann. 1070 pp.

[Guerraoui 1995] Guerraoui, R. (1995). Revistiting the relationship between non-blocking atomic commitment and consensus. In *WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 87–100, London, UK. Springer-Verlag.

[Guerraoui and Schiper 1995] Guerraoui, R. and Schiper, A. (1995). The decentralized non-blocking atomic commitment protocol. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Systems*, pages 2–9.

[Haerder and Reuter 1983] Haerder, T. and Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317.

[Jajodia and Kerschberg 1997] Jajodia, S. and Kerschberg, L., editors (1997). *Advanced Transaction Models and Architectures*. Kluwer. 400 p.

[JEE] JEE. Java Platform, Enterprise Edition. Sun Microsystems. http://java.sun.com/javaee.

[Kandula 2007] Kandula (2007). The Apacha Kandula project. http://ws.apache.org/kandula/.

[Lampson and Sturgis 1979] Lampson, B. W. and Sturgis, H. E. (1979). Crash recovery in a distributed data storage system. Technical report, Xerox PARC (unpublished), 25 pp. Partially reproduced in: *Distributed Systems – Architecture and Implementation*, ed. Lampson, Paul, and Siegert, Lecture Notes in Computer Science 105, Springer, 1981, pp. 246–265 and pp. 357–370.

[Larus and Rajwar 2007] Larus, J. R. and Rajwar, R. (2007). *Transactional Memory*. Morgan & Claypool. 211 pp.

[Mohan et al. 1992] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. (1992). Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162.

[Mohan et al. 1986] Mohan, C., Lindsay, B., and Obermarck, R. (1986). Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396.

[Moss 1985] Moss, J. E. B. (1985). *Nested transactions: an approach to reliable distributed computing*. Massachusetts Institute of Technology, Cambridge, MA, USA. Based on the author's PhD thesis, 1981. 160 p.

[OASIS] OASIS. Organization for the Advancement of Structured Information Standards. http://www.oasis-open.org/.

[OASIS WS-TX TC 2007a] OASIS WS-TX TC (2007a). Web Services Atomic Transaction Version 1.1. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx.

[OASIS WS-TX TC 2007b] OASIS WS-TX TC (2007b). Web Services Business Activity Version 1.1. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx.

[OASIS WS-TX TC 2007c] OASIS WS-TX TC (2007c). Web Services Coordination Version 1.1. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx.

[Obermarck 1982] Obermarck, R. (1982). Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7(2):187–208.

[Papadimitriou 1979] Papadimitriou, C. H. (1979). The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653.

[Papadimitriou 1986] Papadimitriou, C. H. (1986). *The Theory of Database Concurrency Control*. Computer Science Press.

[Raz 1992] Raz, Y. (1992). The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB'92)*, pages 292–312, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Rosenkrantz et al. 1978] Rosenkrantz, D. J., Stearns, R. E., and Philip M. Lewis, I. (1978). System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198.

[Rouvoy 2006] Rouvoy, R. (2006). *Une démarche à granularité extrêmement fine pour la construction de canevas intergiciels hautement adaptables: application aux services de transactions.* PhD thesis, Université des Sciences et Technologies de Lille. 248 pp.

[Rouvoy and Merle 2003] Rouvoy, R. and Merle, Ph. (2003). Abstraction of transaction demarcation in component-oriented platforms. In *Proceedings of the 4th ACM/IFIP/USENIX Middleware Conference (Middleware 2003)*, pages 305–323. LNCS 2672 (Springer).

[Rouvoy and Merle 2007] Rouvoy, R. and Merle, Ph. (2007). Using microcomponents and design patterns to build evolutionary transaction services. *Electronic Notes in Theoretical Computer Science*, 166:111–125. Proceedings of the ERCIM Working Group on Software Evolution (2006).

[Rouvoy et al. 2006a] Rouvoy, R., Serrano-Alvarado, P., and Merle, Ph. (2006a). A component-based approach to compose transaction standards. In *Software Composition*, pages 114–130. LNCS 4089 (Springer).

[Rouvoy et al. 2006b] Rouvoy, R., Serrano-Alvarado, P., and Merle, Ph. (2006b). Towards context-aware transaction services. In *Proceedings of the 6th IFIP Conference on Distributed Applications and Interoperable Services (DAIS)*, pages 272–288. LNCS 4025 (Springer).

[Serrano-Alvarado et al. 2005] Serrano-Alvarado, P., Rouvoy, R., and Merle, P. (2005). Self-adaptive component-based transaction commit management. In *ARM '05: Proceedings of the 4th Workshop on Reflective and Adaptive Middleware Systems*, New York, NY, USA. ACM.

[Skeen 1981] Skeen, D. (1981). Non-blocking commit protocols. In *SIGMOD'81: Proceedings of the 1981 ACM-SIGMOD International Conference on Management of Data*, pages 133–142, Orlando, FL, USA.

[Stearns et al. 1976] Stearns, R. E., Lewis, P. M., and Rosenkrantz, D. J. (1976). Concurrency control in database systems. In *Proceedings of the 17th Symposium on Foundations of Computer Science (FOCS'76)*, pages 19–32, Houston, Texas, USA.

[Thomas 1979] Thomas, R. H. (1979). A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209.

[W3C-WSA 2004] W3C-WSA (2004). W3C-WSA Group, Web Services Architecture. http://www.w3.org/TR/ws-arch/.

[Wang et al. 2008] Wang, T., Vonk, J., Kratz, B., and Grefen, P. (2008). A survey on the history of transaction management: from flat to grid transactions. *Distributed and Parallel Databases*, 23(3):235–270.

[Weikum and Vossen 2002] Weikum, G. and Vossen, G. (2002). *Transactional Information Systems*. Morgan Kaufmann. 853 pp.

[X-OPEN/DTP] X-OPEN/DTP. Distributed Transaction Processing. The Open Group. http://www.opengroup.org/products/publications/catalog/tp.htm.