Technical Report

UCAM-CL-TR-293
ISSN 1476-2986

Number 293

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Objects and transactions for modelling distributed applications: concurrency control and commitment

Jean Bacon, Ken Moody

April 1993

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Objects and Transactions for Modelling Distributed Applications: Concurrency Control and Commitment

Jean Bacon and Ken Moody

University of Cambridge Computer Laboratory

Abstract

The concepts of *object* and *transaction* form an ideal basis for reasoning about the behaviour of distributed applications. An object model allows the semantics of an application to be used to specify the required concurrency behaviour of each object. A transaction model covers multi-component computations where the components are distributed and therefore subject to concurrent execution and partial failure.

This tutorial establishes an object model for a distributed system in which transactions are used. It focusses on the alternative methods of concurrency control that might be employed and shows how each method might be appropriate for certain application characteristics and system behaviour. The background for this discussion is established in [Bacon 1993].

1. Introduction and overview

Distributed systems are now commonplace and application developers must work in a distributed context. A conceptual basis is needed for reasoning about the behaviour of distributed computations. This must take into account the characteristic properties of distributed systems :

(a) **Concurrent execution.** The components of a distributed computation may execute concurrently.

(b) **Independent failure modes.** The components of a distributed system and the networks connecting them may fail independently. Some parts of a computation may fail, or be unable to communicate, while others continue to run.

(c) **There is no global time.** The components of a distributed system each have a local clock. We cannot assume a consistent value of time throughout a distributed system.

(d) **Inconsistent state;** it takes time for the effects of an event at one point in a distributed system to propagate throughout the system. There will not be a consistent view of system state at every point in the system unless we use algorithms and protocols to ensure it.

The concept of transaction was developed, initially in the context of (centralised) database systems, to handle multi-component computations in the presence of concurrency and failure. An object model is widely accepted as a basis for modelling both centralised and distributed software and the concepts of object and transaction have recently come together in the context of object oriented databases. These concepts give us a basis for modelling distributed computations.

In Section 2 we set up an informal object model which we use throughout to study transactions (Section 3) and distributed transactions (Sections 6 and 8). We assume a specification of the semantics of the operations of an object. The application-specified concurrency behaviour of an object is assumed to be based on the commutativity of its operations (Section 4).

We focus on concurrency control and commitment for object-based transaction systems and the distinction between optimistic and pessimistic methods is emphasised. In Section 7 we describe, in general terms, pessimistic methods based on locking and timestamp ordering then optimistic concurrency control. In Section 9 we discuss how each of these methods might be implemented in a distributed system and in Section 10 how transactions are committed in a distributed system. Section 11 gives a summarises the main points and concludes the tutorial.

2. An object model for a transaction system

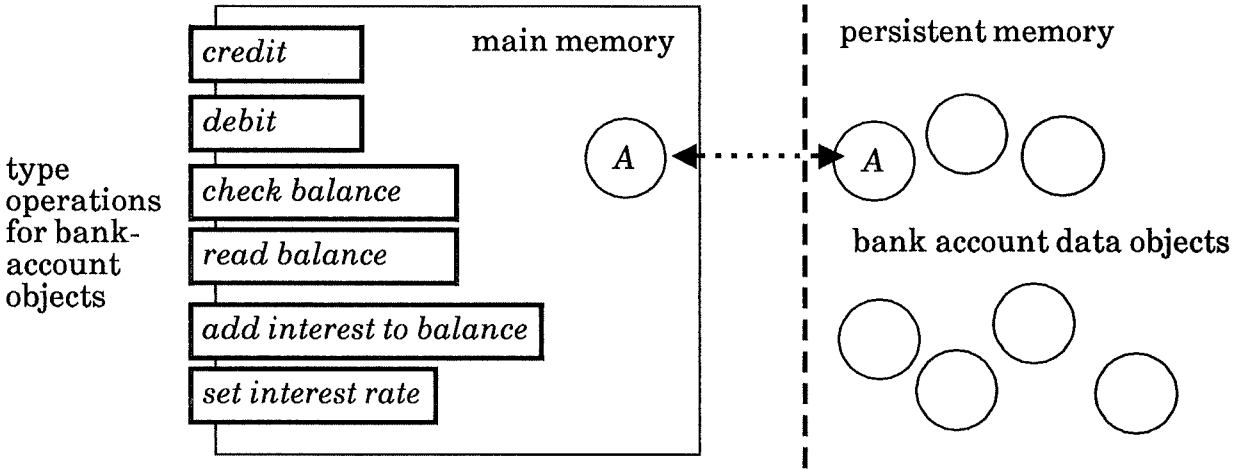


Figure1 An example of a persistent object: a bank account object.

Figure 1 illustrates by means of an example the object model we shall use. We assume that a bank account is an instance of an abstract data type with type operations as defined below.

The **state** of this application comprises the bank account objects in persistent memory which may be distributed. In order to carry out a request of a client of the application, operations are applied to object versions in main memory. A process is the active agent which invokes objects on behalf of a client.

We shall discuss later how an application developer may specify the semantics of the type operations of the application objects. One specification of an account object is as follows. Others are possible, depending on the requirements of the application.

read-balance takes an account name as argument and returns the balance of the account.

check-balance takes an account name and a value as arguments and returns true if the balance of the account is greater than or equal to the argument value, else it returns false.

credit takes an account name and a value as arguments and adds the argument value to the balance. Note that the value of the balance is not output to the client.

debit takes an account name and a value as arguments and subtracts the argument value from the balance. Note that the value of the balance is not output to the client. It is assumed here that the client is responsible for checking the balance before doing a debit, for example, the *transfer* transaction would contain:

if *check-balance* (*account-A*, £1000) **then** *debit* (*account-A*, £1000) ....

set-interest-rate (*r%*) is used to set the daily interest rate to a given percentage.

add-interest-to-balance is run daily by the system administration (probably at 3am when, although cashpoints are available, not many people will be around doing transactions). This operation computes the interest accrued to the account, based on its current value, and adds the interest to the balance.

2.1 Execution of a single operation by concurrent processes

We shall assume that a single operation invocation can be made indivisible or **atomic** in the presence of concurrency and crashes; that is:

- When it terminates normally all its externally visible effects are made permanent (we shall call this the property of **durability**), else it has no effect at all.

If a crash occurs during the execution of an atomic operation, the system can be **rolled back** to the state it was in before the atomic operation was invoked and the operation can be restarted. We do not consider in detail the mechanisms, such as logging, for achieving this.

- Its invocation does not **interfere** or **conflict** with other operation invocations on the same data object by concurrent processes.

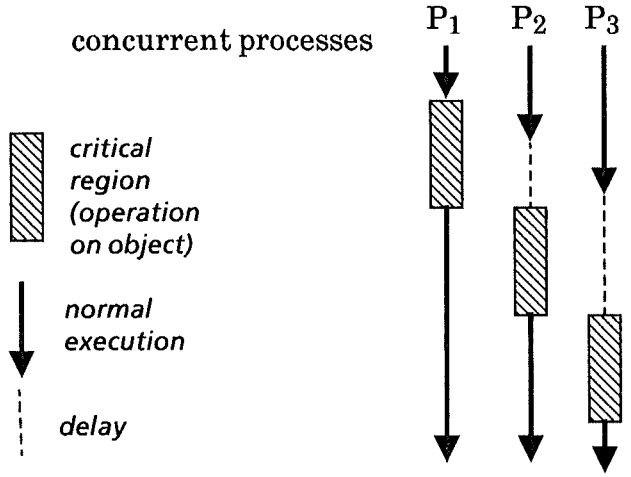


Figure 2 Serialisation of conflicting operations on one object.

Figure 2 shows the serialisation of potentially interfering operations on a single object. We understand how to make such operations atomic in **main memory** by concurrency control mechanisms such as critical regions or guarded commands. These in turn are based on hardware-enforced exclusion or by software protocols. We

do not consider this low-level (or language level) concurrency control problem further, see [Bacon 93].

In our object model we do not assume that every operation on an object must execute under exclusion, as in a monitor. Rather, we assume that it is possible to specify, taking into account the semantics of a given object, which operations may be executed concurrently and which must be executed under exclusion, see Section 4.

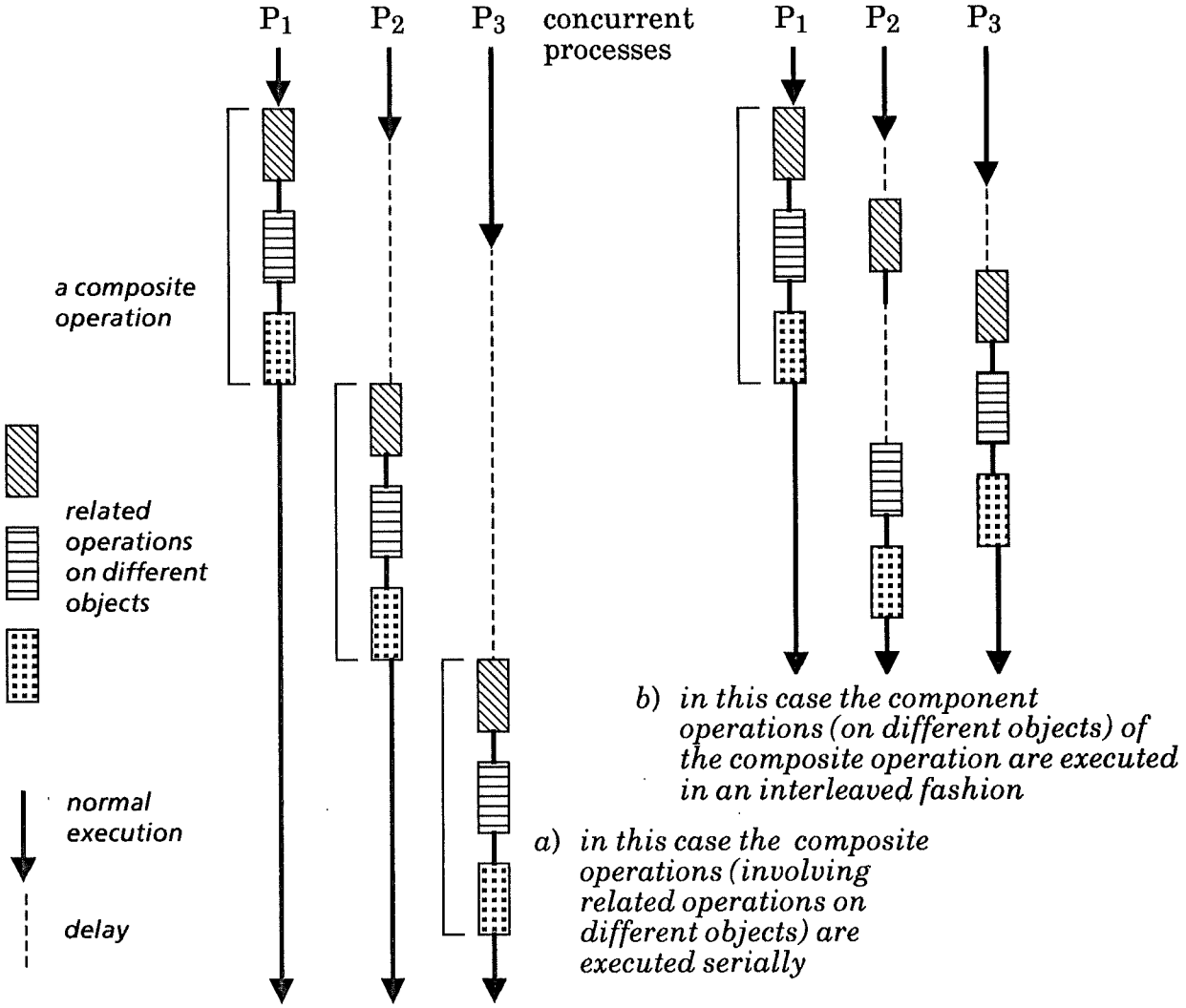


Figure 3 Execution of a composite operation by concurrent processes.

2.2 Execution of composite operations by processes

In practice operations which are meaningful to an application often require a number of related (sub)operations to be carried out on different objects. An example is an operation to *transfer* money from one bank account to another. Figure 3 shows a composite operation, comprising related operations on three different objects, executed by three processes. In 3a the composite operation is serialised as a single operation. Concurrent execution of composite operations can be achieved by interleaving the executions of their suboperations, as shown in Figure 3b. Notice that, in this example, the operations on a given object are shown as serialised and take place in the same order within the composite operation for each process. We shall study how concurrent execution of composite operations may be achieved without sacrificing correctness.

3 Transactions

It is useful to extend the concept of atomic operation to include composite operations. We assume that it is the composite operation that has meaning to whatever application invoked it; the suboperations are related and all or none of them must be carried out. We shall use the term **transaction** to indicate a meaningful atomic operation, such as *transfer*, that may or may not be composite.

In general, a meaningful composite operation may reside at any level in an operation hierarchy. Also, a transaction at a given level in a hierarchy may form part of a transaction at a higher level. This paper establishes a basis for studying how transactions may be implemented with concurrent execution and in the presence of crashes. We shall not study nested transactions explicitly; for further reading see [Weikum 91].

3.1 Commit and abort

Successful termination of a transaction is called **commitment** and a successful transaction is assumed to terminate with a *commit* operation. After a successful *commit* operation, the changes that the transaction has made to the system state are guaranteed to persist. This is the **durability** property of atomic transactions. So that we can specify the duration of a transaction precisely we shall assume that a transaction starts with a specific *start* operation.

We shall discuss in Section 5 whether commitment is also the point at which those changes are allowed to become visible to other transactions. If this is the case then the transaction is said to have the property of **isolation** and the **execution schedule** of the operations of this and concurrent transactions is said to be **strict**. It might be desirable in an implementation to make uncommitted state changes visible, thus achieving greater concurrency, but the effect on long term system state must be as though the property of isolation was enforced.

A transaction management system must be **crash resilient** in order to enforce the property of **atomicity** of transactions, so that either all or none of the operations of a transaction are carried out. If a transaction has not been committed it cannot be assumed that all its operations are complete. When the system restarts after a crash it must be able to roll back (undo) the effects of any transactions that were uncommitted at the time of the crash. This is called **aborting** a transaction. A transaction is defined to end with a *commit* or an *abort* operation.

If such a procedure is available for achieving crash resilience it may also be used for **concurrency control** by the management system. Once the possibility of undoing the effects of operations exists in a system we can attempt to achieve greater concurrency than is strictly safe and solve any problems that arise by undoing the effects of the operations that have turned out to be wrong and restarting the transaction that invoked them.

The *abort* operation can also be made available to the application level. A transaction may then be coded to read values from persistent store and, depending on the values, proceed to further processing or abort, requiring any previous operations to be undone. An example is that a *check-balance* operation might find that there is insufficient money in an account to proceed with a *debit* operation.

3.2 A notation for transactions

It is convenient to use a concise notation for transactions and their operations. We assume that a transaction is given a unique identifying number i when it starts and that this number is associated with all its operations. We refer to the transaction as a whole as T_i, its start as S_i, a *commit* operation as C_i and an *abort* operation as A_i. The operations within the transaction will be named appropriately, such as $debit_i$ (*account-A*, £1000).

For example, a *transfer* transaction may be specified in some application level programming language as follows:

begin transaction;
transfer (*account-A*, *account-B*, £1000);
end transaction;

This implies that *transfer* is defined, at a higher level of abstraction, at the programming language level. At a lower level, within a library or the transaction management system, *transfer* could be expanded in terms of operations on bank account objects as follows:

$T_i = S_i$; **if** $check\text{-}balance_i$ (*account-A*, £1000)
then $debit_i$ (*account-A*, £1000); $credit_i$ (*account-B*, £1000); C_i
else $print_i$ ("not enough in account"); A_i
fi

For more complex transactions the application level may require to interact with the transaction manager and it is returned the transaction identifier for this purpose.

3.3 Serialisability and consistency

We defined a transaction as a (possibly composite) atomic operation that is meaningful to the application level. A transaction causes the system to move from one consistent state to another. If the possibility of crashes is ignored, in the first instance, a **consistent system state** is maintained by executing transactions **serially.**

If one process's transaction is executed to completion before any other can start there is no possibility of interference between them. We have made the transaction a single indivisible operation (Figure 3a). Such a procedure (single threading of all, even unrelated, transactions) could be bad for system performance and serial execution of all transactions could not be contemplated for a multiprocessor or distributed system. We must therefore consider concurrent execution of transactions, with interleaving of suboperation executions (Figure 3b).

The idea that consistent system state is maintained by serial execution of transactions is fundamental. If a specific interleaving of the suboperations of concurrent transactions can be shown to be **equivalent in some sense to some serial execution of those transactions**, then we know that the system state will be consistent, given that particular concurrent execution of the transactions. Further discussion can be found in (Korth et al., 1990).

An example illustrates the point. Consider the transaction *Transfer* (*T*) executed concurrently with a transaction *Sum* (*S*) which outputs the total amount of money in the accounts. The *start* and *commit* operations are not shown here. A **serial schedule** of the operations of the transactions may be achieved in two ways:

T before *S*	or	*S* before *T*
T: *debit* (*account-A*, £1000);		*S*: *read-balance* (*account-A*);
T:*credit* (*account-B*, £1000);		*S*: *read-balance* (*account-B*);
S: *read-balance* (*account-A*);		*S*: *print* (*account-A* + *account-B*);
S: *read-balance* (*account-B*);		*T*: *debit* (*account-A*, £1000);
S: *print* (*account-A* + *account-B*);		*T*: *credit* (*account-B*, £1000);

An interleaving which leads to a result in which £1000 is lost from the sum:

T: *debit* (*account-A*, £1000);
S: *read-balance* (*account-A*);
S: *read-balance* (*account-B*);
S: *print* (*account-A* + *account-B*);
T:*credit* (*account-B*, £1000);

is not equivalent to either serial schedule of operations. The problem arises because transaction *S* is seeing an inconsistent system state. We shall study how to achieve concurrent execution of transactions whilst ensuring that no transaction sees an inconsistent system state.

3.4 The ACID properties of transactions

Putting together the points made in the discussion above, the execution of a transaction may be defined as having the following properties:

Atomicity	Either all or none of the transaction's operations are performed.
Consistency	A transaction transforms the system from one consistent state to another.
Isolation	An incomplete transaction cannot reveal its result to other transactions before it is committed.
Durability	Once a transaction is committed the system must guarantee that the results of its operations will persist, even if there are subsequent system failures.

These properties relate to the definition of transactions and do not imply particular methods of implementation. The effect on system state of running transactions is as defined by these properties.

Note that the A and D properties are the concern of crash resilience and the C and I properties are associated with concurrency control.

4. How to specify the concurrency behaviour of an object

It is over-restrictive to insist that every operation on an object should be executed under exclusion. We now define a general method of specifying the concurrency behaviour of objects. As Herlihy (1990) points out, it may be possible to take a more relaxed view of conflict.

4.1 Non-commutative (conflicting) pairs of operations

It is possible to specify which pairs of the type operations of an object do not commute. Operations X and Y are **commutative** if, from any initial state, executing X then Y results in the same object state and external output values as executing Y then X; the order of execution does not matter. We shall use the term **conflicting** as equivalent to **non-commutative**, relating to a pair of operations.

For a given object it must therefore be specified which pairs of operations conflict. Note that it is necessary to include in the pairs each operation with itself. An example of an operation which does not commute with itself is *write*, for example:

the order: *write* (x, 100); *write* (x, 200) results in the final value 200 for x,

the order: *write* (x, 200); *write* (x, 100) results in the final value 100 for x.

In the case of the bank account object:

credit and *credit* are commutative
(the final value of the account is the same whatever the order of execution and there is no external output).

debit and *debit* and *credit* and *debit* may be specified as commutative
(This is the case if account objects may take negative values or if we assume that check balance is always used before debit. Alternatively, the application developer may wish a debit transaction to be aborted if the account has insufficient funds. In this case the order of *credit* and *debit* may be significant).

read-balance and *credit* are not commutative
because the value read and output for the balance is different depending on whether it is executed before or after the credit operation, the final value of the account is the same whatever the order of execution.

read-balance and *debit* are not commutative

read-balance and *read-balance* are commutative, as are *check-balance* and *check-balance*

check-balance and *credit* are not commutative

check-balance and *debit* are not commutative

set-interest-rate and *set-interest-rate* are not commutative,
because the final value of the interest rate depends on the order of execution.

add-interest-to-balance conflicts with *credit* and *debit*
because the value computed for the interest is different depending on whether the *credit* or *debit* was done before or after *add-interest-to-balance*. It conflicts with *read-balance* and *check-balance* with respect to the value of the account output.

4.2 Condition for serialisability

We shall use commutativity as the basis for specifying conflicting operations and will also make the following assumptions:

- objects are identified uniquely in a system;

- the operations are executed without interference; that is, the operations we are considering here are at the finest granularity of decomposition visible to the client;
- there is a single clock associated with an object which is used to indicate the time at which operations take place and therefore their order;
- the object records the time at which each operation invocation takes place with the transaction identifier of the transaction that executed the operation.

It is therefore possible, for any pair of transactions, to determine the order of execution of their operations (in particular the conflicting pairs of operations) on a given object which they both invoke. This leads to the following definition of serialisability of a pair of transactions (Weihl, 1984, 1989):

> *For* **serialisability** *of two transactions it is necessary and sufficient for the order of their invocations of all conflicting pairs of operations to be the same for all the objects which are invoked by both transactions.*

We shall use this definition as the basis for our study of concurrent execution of transactions. In the next section we generalise from pairwise serialisability to serialisability of a number of transactions. Note that the definition holds for a distributed system where there can be no assumption of global time. All that is needed is time local to each object.

4.3 Example

Consider again the example of Section 3.3, of two transactions *Sum* (*S*) and *Transfer* (*T*).

The pairs of conflicting operations on objects invoked by the two transactions are:

> *T*: *debit* (*account-A*, £1000) and *S*: *read-balance* (*account-A*)
>
> *T*: *credit* (*account-B*, £1000) and *S*: *read-balance* (*account-B*).

In both of the serial schedules these pairs of conflicting operations are carried out in the same order by the transactions.

In the non-serial schedule, which leads to an incorrect result, the pairs of conflicting operations on objects invoked by the two transactions are:

> *T*: *debit* (*account-A*, £1000) and *S*: *read-balance* (*account-A*), (*account-A*: *T* before *S)*
>
> *S*: *read-balance* (*account-B*) and *T*: *credit* (*account-B*, £1000), (*account-B*: *S* before *T)*.

The transactions therefore do not meet the condition for serialisability.

4.4 Serialisability illustrated by directed graphs of transactions

In this section we develop a graphical representation for schedules of operations of transactions. Any necessary ordering of the operations within a transaction is indicated in its graph. Figure 4 shows the transactions *Sum* and *Transfer* used above. Taking the *Sum* transaction as an example, *start* comes first, the

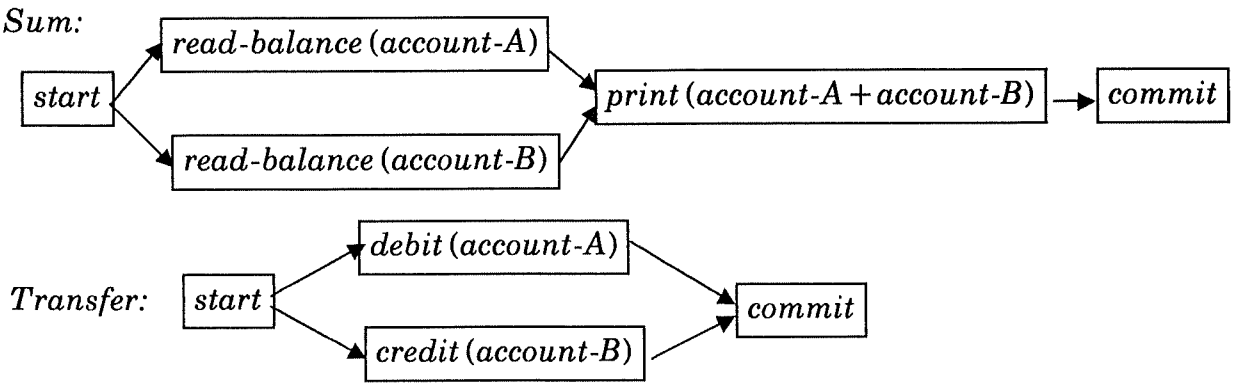


Figure 4 Graphical representation of the *Sum* and *Transfer* transactions.

read- balance operations may take place in either order or in parallel (on a multiprocessor) but must precede the *print* operation, after which comes the *commit* operation.

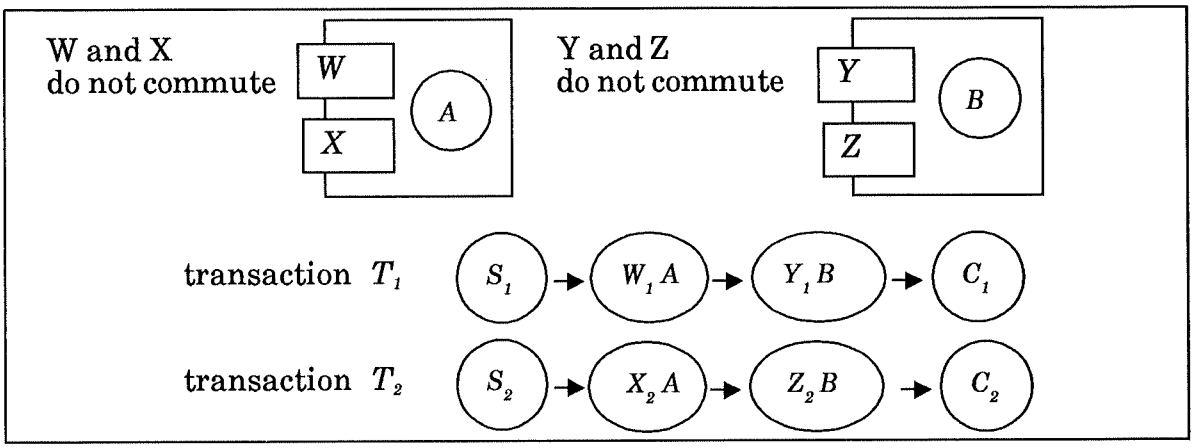


Figure 5 A specification of two transactions.

In the next examples we use a more concise notation for the purposes of discussion; A and B for objects and W, X, Y, Z for operations. Figure 5 specifies two transactions, both of which access objects A and B. The operations W and X on object A are conflicting, as are operations Y and Z on object B. In practice, an object is likely to have many operations but we consider a minimal example in order to highlight the issues. We focus on pairs of conflicting operations in order to explore the definition of serialisability given in Section 4.2. The graphs show the operations within each transaction in serial order for simplicity. Our concern is to explore how concurrent transactions may be represented.

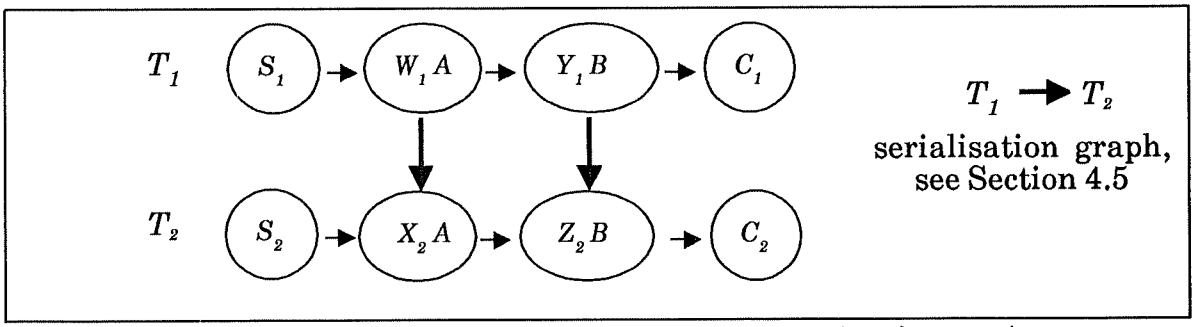


Figure 6 A serialisable schedule of the transactions' operations.

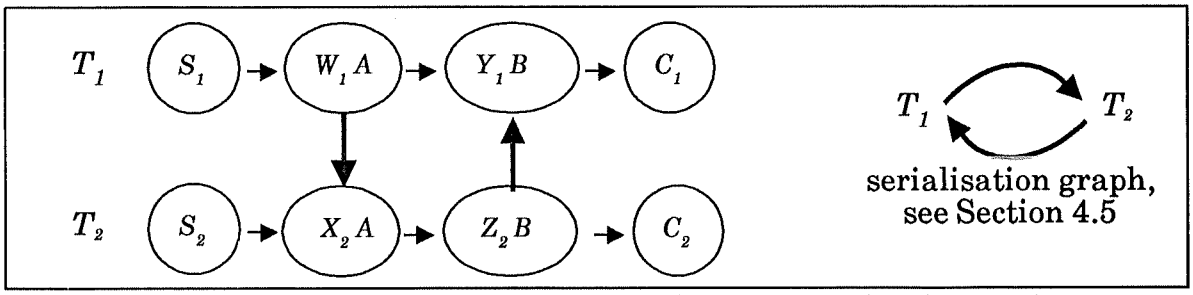


Figure 7 A non-serialisable schedule of the transactions' operations.

Figure 6 shows a serialisable execution of the operations of the two transactions:

T_1 invokes W on A before T_2 invokes X on A, (object A: T_1 before T_2)
T_1 invokes Y on B before T_2 invokes Z on B. (object B: T_1 before T_2)

That is, the order of pairs of conflicting operations is the same for all the objects which are invoked by both transactions.

Figure 7 shows a non-serialisable execution of the operations of the two transactions:

T_1 invokes W on A before T_2 invokes X on A, (object A: T_1 before T_2)
T_2 invokes Z on B before T_1 invokes Y on B. (object B: T_2 before T_1)

In this case, the pair of conflicting operations on A (W, X) is invoked in the order T_1 then T_2. The pair of conflicting operations on B (Z, Y) is invoked in the order T_2 then T_1. There is no ordering of the transactions that is consistent with the order of operations at both objects.

4.5 Histories and serialisation graphs

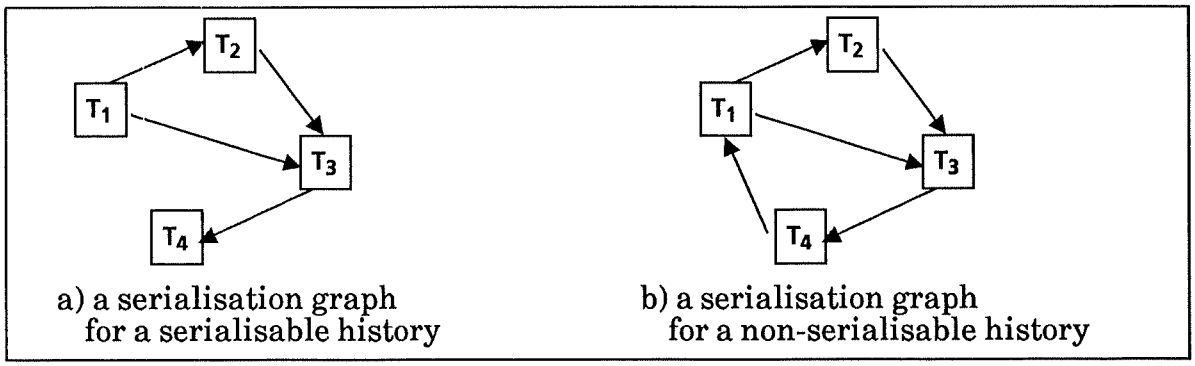
A **history** is a data structure which represents a concurrent execution of a set of transactions. The directed graphs of Figures 6 and 7 are simple examples; they show the operations within the transactions, and the order of invocation of conflicting pairs of operations by different transactions. Note that the order of invocation of all conflicting pairs of operations on all objects must be shown in the history.

A **history** is **serialisable** if it represents a serialisable execution of the transactions. That is, there is a serial ordering of the transactions in which all conflicting pairs of operations at each object are invoked in the same order as in the given history.

An object is a witness to an order dependency between two transactions if they have invoked a conflicting pair of operations at that object. A **serialisation graph** is a directed graph that shows only transaction identifiers and dependencies between transactions; the vertices of the graph are the transactions T_1, and there is an edge $T_1 -> T_2$ if and only if some object is a witness to that order dependency. For example $T_1 -> T_2$ is the transaction graph for the history in Figure 6.

Figure 8 gives examples of possible serialisation graphs for four transactions. In both 8a and b every pair of transactions have conflicting operations executed in the same order (there is at most one edge between each pair of transactions). In Figure 8b the serialisation graph has a cycle and the history represented by the serialisation graph is not serialisable.

In general we must ascertain whether a given schedule of the operations within a set of transactions is serialisable. We require a total ordering of the set of transactions that is consistent with the schedule.



a) a serialisation graph for a serialisable history

b) a serialisation graph for a non-serialisable history

Figure 8 Examples of serialisation graphs.

Each object knows which pairs of its operations conflict.

Each object knows which transactions have invoked a pair of conflicting operations: it is a witness to an order dependency between them.

Provided that the order dependencies are consistent both at each given object and between objects then an ordering is determined for each pair of transactions involved. IF NOT, then there is a cycle in the serialisation graph, say $T_1 -> T_2 -> T_1$ as in Figure 7, and the transaction history cannot be serialisable.

This information can be assembled for all the objects invoked by the set of transactions, giving rise to the serialisation graph.

To find a total ordering of the set of transactions that is consistent with the pairwise order dependencies requires a topological sort of the serialisation graph. This can be done if and only if the graph is acyclic (Aho et al., 1983).

Suppose that a TP system maintains a serialisation graph of the transactions in progress. A new transaction is submitted and the system attempts to execute it concurrently with the ongoing transactions. Any proposed schedule of the operations of the new transaction can be tested by creating a serialisation graph which is the original one extended with the operations of the new transaction. A schedule can be rejected if the serialisation graph thus extended has a cycle.

5 Dealing with aborts: more about the property of isolation

The theory outlined above does not take into account that the operations of a transaction might be undone due to an *abort* termination. It must be possible to return the system to a consistent state as though the transaction had not taken place. The following problems could arise through concurrent execution of transactions, even if a serialisable schedule of suboperations had been devised. It is demonstrated that serialisability is necessary but not sufficient for correct concurrent operation.

5.1 Cascading aborts

Figure 9 shows a serialisable schedule of the transactions T_1 and T_2 used above in Section 4. This time, T_1 happens to abort.

Suppose that the transaction scheduler, having noted the order of operations for a serialisable transaction, had in order to achieve maximum concurrency allowed T_2 to execute operation X on object A as soon as T_1 had completed operation W and

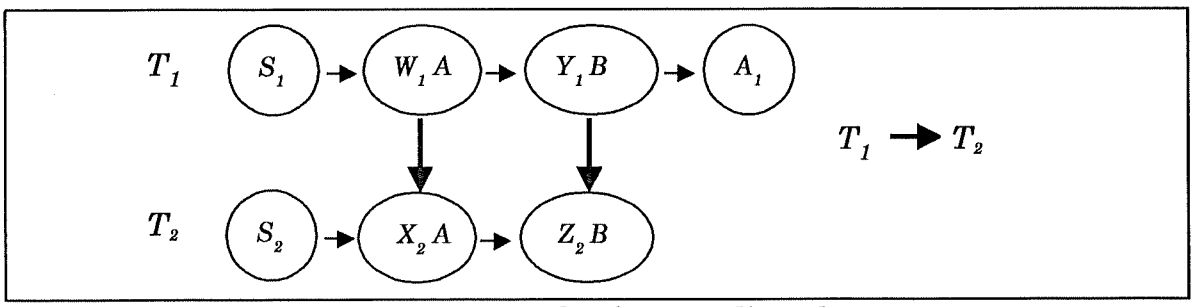


Figure 9 Example of a cascading abort.

similarly for object B. T_2 may have seen state or performed output that is now incorrect because T_1 has aborted and T_2 must also be aborted.

In general, aborting one transaction could lead to the need to abort a number of related transactions, called **cascading aborts**. This behaviour might degrade system performance so badly that it could be advisable to ensure that any state seen by a transaction has been written by a committed transaction. In other words, the effects of the suboperations of a transaction are not made visible to other transactions until the transaction commits: thus enforcing the property of **isolation** in the implementation. A schedule of operation invocations which enforces this property is called a **strict** schedule.

This approach can cause difficulties in systems where transactions may be long and contention is likely. Strictness may be deemed unnecessary if aborts are unlikely to happen. Note, however, that if non-strict operations are allowed to go ahead, transactions which invoke them will be delayed when they request to commit if they have seen uncommitted state.

5.2 The ability to recover state

The discussion here is in general terms. In Section 7 specific methods for achieving correct (serialisable) executions of concurrent transactions are described. Some of the scenarios given below as examples might not in practice be allowed to arise, or some of the theoretically possible system actions might be deemed too expensive to implement.

If *abort* is supported it must be possible to return the system to a consistent state, as though the aborted transaction had not taken place. Consider the following interleaving of operations within a concurrent execution of transactions T_1 and T_2. The operations involved, several credit operations on bank accounts, are commutative so there is no problem with serialisability.

		suppose	A=£5000	B=£8000
$start_1$				
$credit_1$	(*account-A*, £1000)		A=£6000	
$credit_1$	(*account-B*, £500)			B=£8500
$start_2$				
$credit_2$	(*account-A*, £200)		A=£6200	
$abort_1$			(A=£5200	B=£8000 should be achieved)
$credit_2$	(*account-B*, £600)			B=£8600
$abort_2$			(A=£5000	B=£8000 should be achieved)

This example schedule is not strict, that is, it violates the property of isolation. If this is the case, greater care must be taken on abort or on crash recovery than merely restoring each object's state to that prior to the aborted operation. When T_1 aborts, T_2 has already done another *credit* operation on *account-A*. The value of *account-A* cannot simply be put back to that prior to $credit_1$ (*account-A*, £1000). Neither can we take no action at all; T_2 goes on to abort. We cannot then put the value of *account-A* back to what it was prior to $credit_2$(*account-A*, £200) (this was the value after the credit by T_1 which has already aborted). If we had discarded the value prior to T_1's invocation the original state would be irrecoverable. For this reason we assume that a record of invocations is kept with the object and we have higher level semantics than merely a record of state changes.
We shall assume that every operation has an **inverse** or **undo operation.** When a transaction aborts, each of its invocations must be undone. For a given object, if there have been no conflicting invocations since the one that is to be undone then we simply apply the *undo* operation to the current state (the order of invocation of commutative operations is irrelevant). In this example, the inverse of *credit* is *debit*. When T_1 aborts we can simply *debit* (*account-A*, £1000) and remove the record of the original invocation from the object.

If there has been a conflicting invocation, such as an *add-interest-to-balance* invocation, since the invocation we require to abort then we must undo all the invocations back to the conflicting operation. After we have undone that, we can perform the undo to achieve the abort we require, then we must do the subsequent operations again.

This is a complex procedure and is the penalty to be paid for relaxing the property of isolation in an implementation and allowing non-strict conflicting operations to be invoked. In general, we shall assume a strict execution of operations to avoid this complexity, although as stated above, strictness may not be realistic in some systems. A strict execution can be enforced if each object delays any request to invoke an operation which conflicts with an uncommitted operation. An object must then be told when a transaction that has invoked it commits; this is assumed to be through a commit operation in the object's interface. Commit would cause all state changes resulting from invocations on the object by the transaction to be made permanent and visible to other transactions.

6 Distributed, object oriented transaction processing

Figure 10 shows two instances of a transaction processing system (TPS) such as would occur at two nodes in a distributed TPS. We assume:

- A client submits a transaction at one node only, we shall call it the **coordinating node.**
- A given object resides at one and only one node; that is, we assume there is no object replication. An object invocation takes place at this **home node.**
- There are mechanisms for locating an object, given its unique identifier.

A transaction manager is responsible for validating the clients' submissions and for passing the component operations of the transactions to the local scheduler, if the objects to be invoked are in the local database, or to a combination of the local and remote schedulers according to the location of the objects involved.

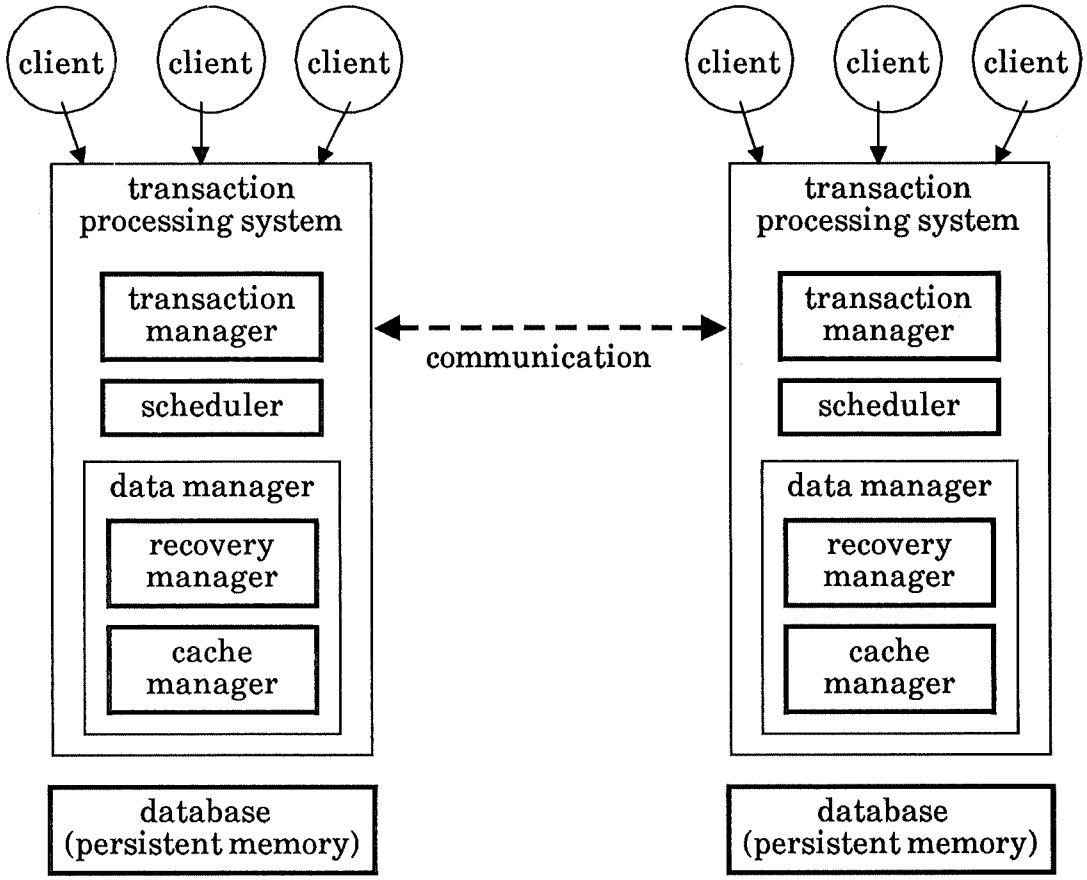


Figure 10 A distributed transaction processing system.

A scheduler will use some strategy to achieve a serialisable schedule of the operations of the transactions in progress. Section 7 introduces the general principles of concurrency control which apply both to a centralised and a distributed implementation of a TPS. Section 8 discusses the issues which are specific to a distributed TPS.

The data objects in persistent memory will be transferred into main memory for operation invocation and new values will be written back. This is the concern of the cache manager. The recovery manager is responsible for ensuring that sufficient of the object histories is recorded in persistent memory to allow the TPS to support the properties of atomicity and durability of transactions in the presence of crashes. For example, the results of operation invocations must be recorded in persistent memory before a transaction is acknowledged to the application as committed. Crash recovery procedures are not discussed in this tutorial.

For further reading on implementations of TPS see [Gray and Reuter 93].

6.1 An object model for a distributed TPS

Figure 11 shows an object with type operations and some management operations such as *commit* and *abort* which may be needed for practical implementation of a TPS. We assume (Section 4.2) that each object holds information on the operations that have been invoked on it. This information includes the transaction identifier of the invoker and the time of the invocation. It should be emphasised that this is a theoretical starting point: all relevant information is assumed to be held. It would

not be possible in a practical implementation to hold an indefinite history from the initial *create* operation with every object and optimisations would have to be made.

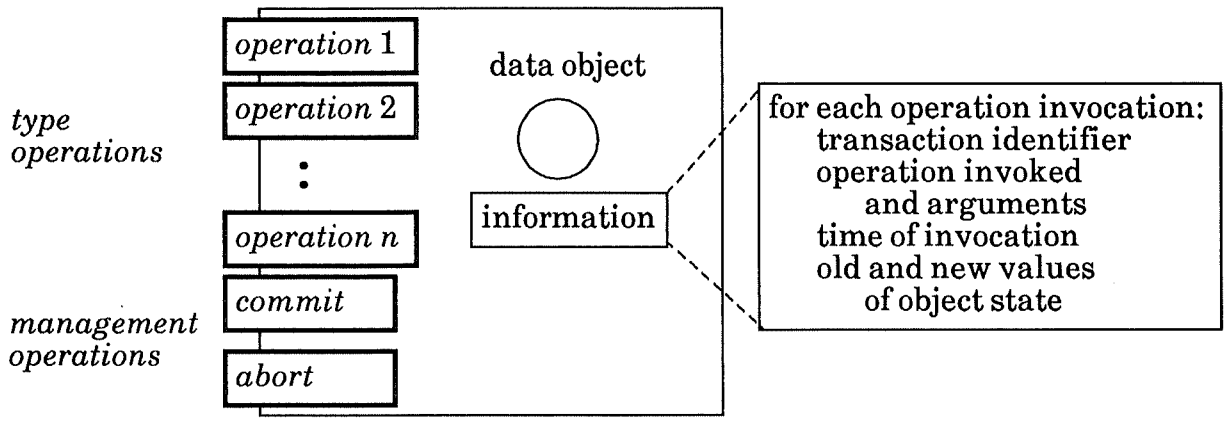


Figure 11 An object used in a transaction system.

7 Concurrency control

7.1 Concurrency control through locking

An object may be assumed to have *lock* and *unlock* operations as well as those previously discussed. We assume initially that an object can be locked and only the holder of the lock can invoke an operation on the object; that is, we are enforcing exclusive access to the object and are not, at present, considering the use of operation semantics to achieve a greater degree of concurrency. We are also assuming that the granularity of locking is the whole object. In order to carry out a composite operation comprising related operations on distinct objects, a number of locks are needed.

Let us assume that the transaction scheduler will issue *lock* and *unlock* operation invocations as well as those discussed previously. A possible strategy is to lock all the objects required by a transaction at its start and to release them on *commit* or *abort*. Can we achieve better concurrency behaviour than this?

7.1.1 Two-phase locking

In two-phase locking, locks can be acquired for a transaction as they are needed. The constraint which defines two-phase locking is that no lock can be released until all locks have been acquired. A transaction therefore has a phase during which it builds up the number of locks it holds until it reaches its total requirement.

In the general form of two-phase locking a transaction can release locks piecemeal as it finishes with the associated objects. If atomic transactions are to be supported with the property of isolation (that the effects of a transaction are not visible to other transactions before commit), a safe procedure is to release all locks on commit. This is called **strict two-phase locking**. Allowing visibility earlier allows more concurrency at the risk of cascading aborts and state which is difficult to recover, as discussed in Section 5.2.

Two-phase locking guarantees that all conflicting pairs of operations of two transactions are scheduled in the same order and thus enforces a serialisable schedule of transactions. This is reasonably intuitive but we will discuss it further after looking at an example.

It is possible for a lock request to fail because the object is locked already. In this case the transaction may be blocked for a time in the hope that the transaction holding the lock will complete and release the lock. It is possible for deadlock to occur as shown below.

7.1.2 An example of two-phase locking

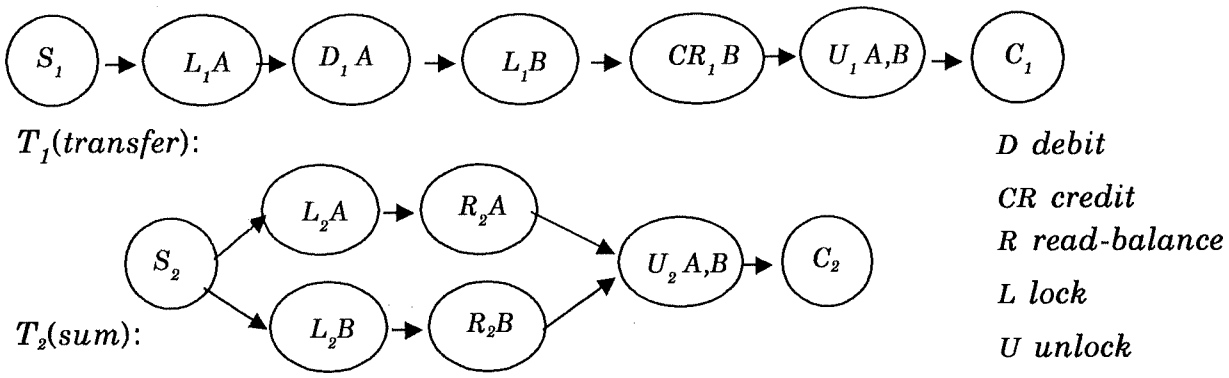


Figure 12 Two transactions including *lock* and *unlock* operations.

In Figure 12 T_1 is the *transfer* transaction which first *debits* (*D*) *A* then *credits* (*CR*) *B*. For conciseness we shall not show a balance check here. T_2 is a transaction which sums the values of *A* and *B* using *read-balance* (*R*). *Lock* (*L*) and *unlock* (*U*) operations have been inserted. When T_1 and T_2 are run concurrently, any of the following can happen:

1. T_2 locks *A* before T_1 locks *A*. T_2 proceeds to lock *B* and calculates and outputs *A*+*B*. T_1 is delayed when it attempts to lock *A*. A serialisable schedule $T_2 \rightarrow T_1$ is achieved.
2. T_1 locks *A* before T_2 locks *A*. T_1 proceeds to lock *B*. T_2 is delayed when it attempts to lock *A* or *B*. A serialisable schedule $T_1 \rightarrow T_2$ is achieved.
3. T_1 locks *A* before T_2 locks *A*. T_2 locks *B*. Deadlock is inevitable.

Two phase locking ensures that a non-serialisable schedule of the operations of transactions cannot occur. The method is subject to deadlock but the occurrence of deadlock means that a non-serialisable schedule has been attempted and prevented.

7.1.3 Deadlock in two-phase locking

Allowing the objects required by a transaction to be locked separately rather than all together and allowing processes to hold their current objects while requesting further locks (the definition of two-phase locking) can lead to deadlock. That is, the rules of two-phase locking set up the conditions which make deadlock possible: 1) exclusive allocation (in the sense that a request for an object invocation can be refused) 2) resource hold while waiting and 3) no preemption, see [Bacon 93]. Concurrency control based on two-phase locking must therefore have provision for dealing with deadlock.

The ability to *abort* a transaction is likely to be in place for crash resilience and application requirements. Deadlock detection followed by abortion of the deadlocked transactions is likely to be a better design option than deadlock avoidance, which involves a greater overhead. A simple alternative to maintaining complex data

structures and running an algorithm on them for deadlock detection is to **time-out** requests for locks and to abort transactions with timed-out lock requests.

A general point is that if the ability to abort is essential in a system design for reasons other than recovery from deadlock (for crash resilience or because the applications require it) then deadlock becomes a problem that is relatively easy to deal with without introducing excessive overhead. The overhead of supporting abort was already there!

7.1.4 Serialisability of two-phase locking

Suppose two transactions have a pair of conflicting operations on object A, and another pair on object B. A particular ordering of the conflicting operations is determined as soon as one of the transactions locks one of the objects. It cannot release the object until it has locked the other object (the two-phase locking rule) which it may or may not succeed in doing. If it succeeds, it has acquired locks on both objects over which there is conflict. If it fails because the other transaction has locked the other object, deadlock is inevitable. This argument generalises to any number of objects. It is not quite so obvious that it generalises to any number of transactions.

We have established that two-phase locking enforces that the conflicting operations of every pair of transactions are scheduled in the same order. It remains to argue that a cycle involving a number of transactions is not possible. The intuition here is that if T_1 is "before" T_2 (in the sense of Section 4: the operations in T_1 of all conflicting pairs are scheduled before the conflicting operations in T_2) and T_2 is before T_3, then T_1 must be before T_3: the **before** relation is transitive. Note that this argument generalises to a distributed implementation because a **before** relation comprises decisions on the order of invocations at individual objects. There is no need for a single value of system time.

7.1.5 Semantic locking

The above discussion has assumed a crude locking policy; that an entire object is locked for exclusive use before an operation is invoked on it. For some operations, such as *read-balance*, any number of invocations could take place concurrently without interference. We could at least refine the system's locking policy to allow for shared (read) locks and exclusive (write) locks to be taken out.

In this case **lock conversion** might be required in some circumstances. A transaction might read a large number of object values and, on that basis, decide which object to update. The shared lock on the object to be updated would be converted to an exclusive lock and the shared locks on all the other objects could be released, at the time allowed by the two-phase rule. Deadlock could arise if two transactions holding a given shared lock both required to convert it to an exclusive lock.

By regarding each object as a separate entity there is scope for indicating which operations can be executed concurrently and which can't (recall Section 4.1). Locking could be associated with each operation on an object and not provided as a separate operation. When an invocation is requested a check of any degree of sophistication could be computed to determine whether to go ahead or consider the object locked

against this invoker at this time and with this current object state. In general, **semantic locking** can be based on conflict specification, based on commutativity (Section 4.1). The computational overhead of two-phase semantic locking would be large and the approach has not been used in practice.

An alternative [Wu, 93], [Wu et al. 93] is to lock only the component of an object required for a given invocation instead of the entire object. Invocations which access different parts of an object can then proceed in parallel.

7.2 Timestamp ordering

We are aiming to run transactions concurrently and to produce a serialisable execution of their operations. An alternative approach to locking for achieving this is to associate a timestamp with each transaction. One serialisable order is then imposed on the operations: that of the timestamps of the transactions they comprise. Assume initially that the timestamp is the time of the start of the transaction and is recorded at the invoked object with every operation that transaction invokes.

Suppose a transaction invokes an operation. Suppose a second transaction attempts to invoke an operation that conflicts with it. If the timestamp of the second transaction is later than (>) that of the first transaction then the operation can go ahead. If the timestamp of the second transaction is earlier than (<) that of the first it is deemed TOO LATE and is rejected (the requesting transaction is aborted and restarted with a new, later, timestamp). If this is enforced for all conflicting pairs of operations at every object then we have a serialisable schedule of the operations of the concurrent transactions.

This approach enforces one particular serialisable order on the operations of the concurrent transactions: that of the transactions' timestamps. This sacrifice of flexibility can be justified on the following grounds:

- the implementation is simple and efficient, thus improving system performance for all transactions,
- the information recorded for concurrency control is associated only with each object and is not held or processed centrally,
- objects are not "locked" for longer than the duration of a single operation, unlike two-phase locking, thus giving more potential for concurrent access to objects (but see below for a discussion of strictness in timestamp ordering).

Let us consider implementation through the simple example used above and illustrated here in Figure 13. Assume the timestamps indicate $T_1 < T_2$.

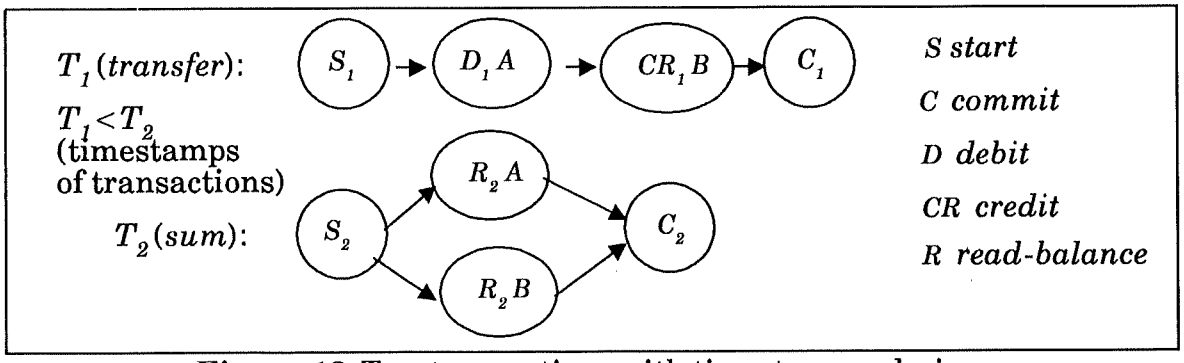


Figure 13 Two transactions with timestamp ordering.

Assume that objects A and B record the timestamps of the transactions which carried out potentially conflicting pairs of operations (*debit* (D) and *read-balance* (R) are non-commutative, as are *credit* (CR) and *read-balance*). Consider the following examples of orderings of the operations of T_1 and T_2 and the corresponding actions taken by objects A and B:

1. D_1A, R_2A, R_2B, CR_1B FAILS because it conflicts with R_2B which has a higher recorded timestamp (of T_2), T_1 is aborted.
2. R_2A, R_2B, D_1A FAILS because it conflicts with R_2A which has a higher recorded timestamp, T_1 is aborted, even though the order $T_2 < T_1$ is serialisable.

An early transaction fails when it attempts to invoke an operation on an object on which a later transaction has already carried out a conflicting operation. Transaction abort could therefore be a common occurrence if contention was likely. If contention is unlikely, the method incurs little overhead.

The following examples illustrate that the definition of conflicting behaviour must be considered carefully. Suppose a transaction to read a large number of items, process them and write a value depending on all the values read (the account to be credited is that with the lowest balance) was run concurrently with transactions each of which updates one of the values read. Figure 14 illustrates the point with a small number of objects.

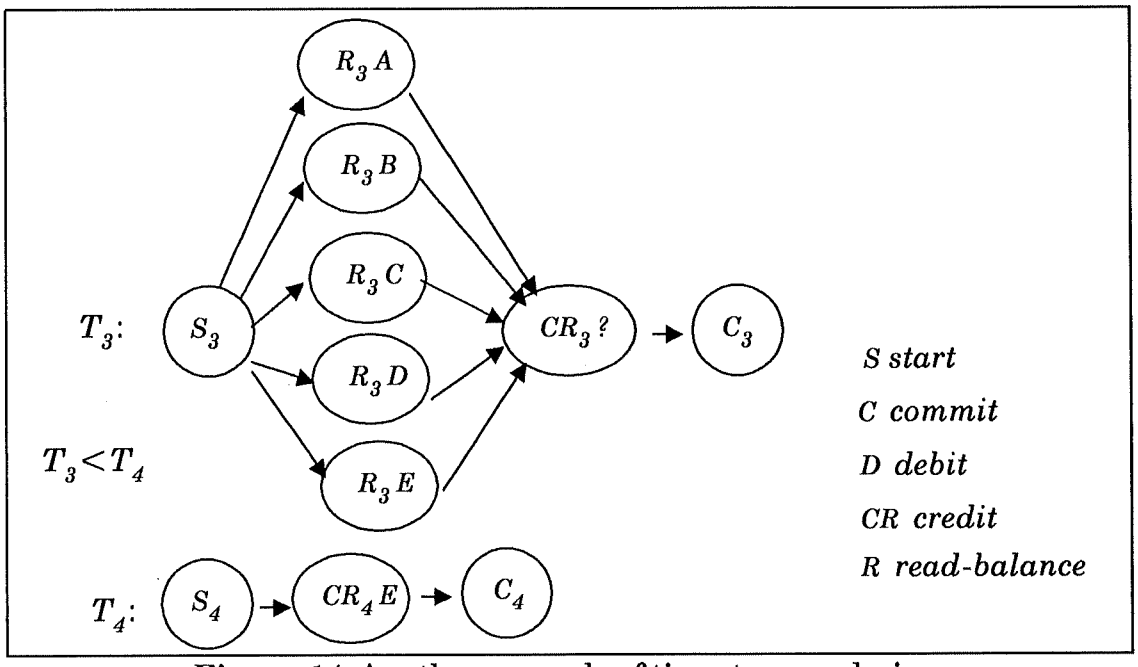


Figure 14 Another example of timestamp ordering.

R_3A, R_3B, R_3C, R_3D, CR_4E, R_3E – this fails because R_3E conflicts with CR_4E which has a higher recorded timestamp. The semantics of the transaction indicate that it need not fail, since although the value read by T_3 relates to the state after T_4 has run, rather than before, it is not incorrect, in fact it is more relevant. Forcing serialisation in the order $T_3 -> T_4$ happens to be unnecessary in this case but we can only say this because we know the intention of T_3 is to invoke a credit on whichever of objects A, B, C, D and E has the lowest value.

Two-phase (exclusive) locking was shown to limit concurrency more than might be strictly necessary but delivered correct results. Timestamp ordering may achieve a higher degree of concurrency because an object is available unless a potentially conflicting operation is being invoked on it (but see the next section). We have seen that a large number of aborts could be made necessary by the particular serialisation enforced and that no simple general definition of conflicting operations would allow us to be more flexible than this. The simple description given here has used the time of the start of a transaction as its timestamp; it might be more appropriate to use the time of its first invocation of an operation which belongs to a conflicting pair. Refinements of the basic scheme are discussed further in Bernstein et al. (1987).

Timestamp ordering can be a simple and effective scheme when conflicts are unlikely. The fact that the decision on whether an operation on an object can go ahead is made on the basis only of information recorded with the object itself makes it a suitable method for a distributed system.

7.2.1 Strict timestamp ordering

Timestamp ordering, as described above, does not enforce the property of isolation and is therefore subject to cascading aborts and complex recovery of object state as discussed in Section 5. Recall that it is necessary to be able to *undo* and *redo* operations.

If isolation (strict execution) is to be enforced in the implementation, additional mechanism to that described above is needed. A transaction scheduler together with the individual object managers could achieve this. An object could ensure that a *commit* operation had been invoked for a given transaction before allowing any operation of any conflicting pair of operations to go ahead for another transaction with a later timestamp.

Note that this does not introduce the possibility of deadlock. Circular wait is prevented by the timestamp ordering of invocations; that is, a cycle of transactions cannot occur such that each has invoked an operation and is waiting for another transaction to commit before invoking another operation.

Strict timestamp ordering introduces the requirement for **atomic commitment.** Assume that a given transaction has invoked, on a number of objects, operations which belong to conflicting pairs. All the objects must agree whether the transaction is to commit or abort. That is, all or none of the objects invoked by the transaction must commit the state changes effected by the transaction. This is not difficult to achieve in a centralised system in the absence of failures. In practice, crashes must be anticipated and distributed implementations may be required, see Section 10.

7.3 Optimistic concurrency control (OCC)

Optimistic schemes for concurrency control are based on the premise that conflict is unlikely and crashes and transaction aborts are rare. We should therefore be careful to avoid high computational overhead due to concurrency control mechanisms, but we must still ensure a serialisable execution. OCC achieves high availability of objects so delay is minimised at transaction start. OCC is appropriate for certain application areas where these conditions and requirements hold; that is, for applications which need a transaction system, but where it is unusual for different

transactions to touch the same objects. Applications which need real-time response and therefore cannot tolerate delay on accessing objects also benefit from an optimistic approach. These issues are discussed further in Section 11.

The strategy of OCC is to apply no changes to persistent memory during the execution of transactions. When a transaction requests *commit* its history is validated to determine whether it can be serialised along with transactions that have already been accepted. (This is sometimes called backward validation. Forward validation would also take into account current transactions). Once a serial order of validated transactions is established, updates are applied in that order to objects in persistent memory. The update of persistent memory must be such that in any state read from an object either all or none of the changes at that object associated with a given transaction are visible.

During transaction execution invocations are made on workspace copies, **shadow copies**, of objects. Each such shadow copy has a well defined **version**, which indicates the transaction whose updates have most recently been applied to the object in persistent memory. Let us also assume that a **timestamp** is recorded with the transaction identifier as part of the version information and that the timestamp is the time when the transaction is validated and its updates are guaranteed.

Each transaction undergoes three phases:

1. **Execution (read)**: the transaction executes to completion (*commit* or *abort*) using shadow copies of data objects.
2. **Validation**: following *commit* the execution schedule is checked to ensure serialisability.
3. **Update (write)**: update invocations are applied to objects in persistent memory in serial order, transaction by transaction. It is the responsibility of the update manager to ensure that all updates succeed. The update manager will know at any time those transactions for which updates have succeeded. It can therefore be asserted, at a given time, that the updates up to those of some transaction have succeeded.

For valid execution each transaction must interact with a consistent set of shadow copies. One (heavyweight) way of achieving this is to ensure that updates are applied atomically across all objects participating in a transaction, using an atomic commitment protocol such as 2-phase commit in a distributed system (see Section 10). Validation can take place at each object as part of the first phase of the protocol, with update taking place only if all objects can accept the transaction. Recall that a transaction is defined to take the system from one consistent state to another. We can then make sure that a set of shadows taken at the start of a transaction is consistent; that is, we must also ensure that taking a set of shadows is made atomic.

There are objections to this approach:

- The enforcement of update atomicity using a protocol such as 2-phase commit reduces concurrency and is bad for performance in general. That is, there is overhead in using such an algorithm which penalises all clients of the system. Also, specific transactions will not experience high availability of objects if they are held for the atomic commitment of some other transaction.

- At the start of transaction execution we may not know what shadows are required. Even if we enforce atomic commitment this does not help unless all shadows are taken "at the same time".
- More importantly, there is a mismatch of philosophy. OCC is postulated on the assumption that interference between transactions is unlikely. It is not worth going to a lot of trouble to ensure that it does not occur. The approach we are objecting to is pessimistic rather than optimistic!

We should therefore abandon the requirement that we take a consistent set of shadows at transaction start. We can then delay making a shadow copy of an object until an operation is invoked on it, noting the object's version so that it can be checked by the validator. There is then the risk that execution will proceed using inconsistent data, but this risk applies also to other schemes that aim for high concurrency, such as allowing non-strict execution in a 2-phase locking approach. We can achieve high concurrency only if we are prepared to risk *abort*. Figure 15 illustrates the problem.

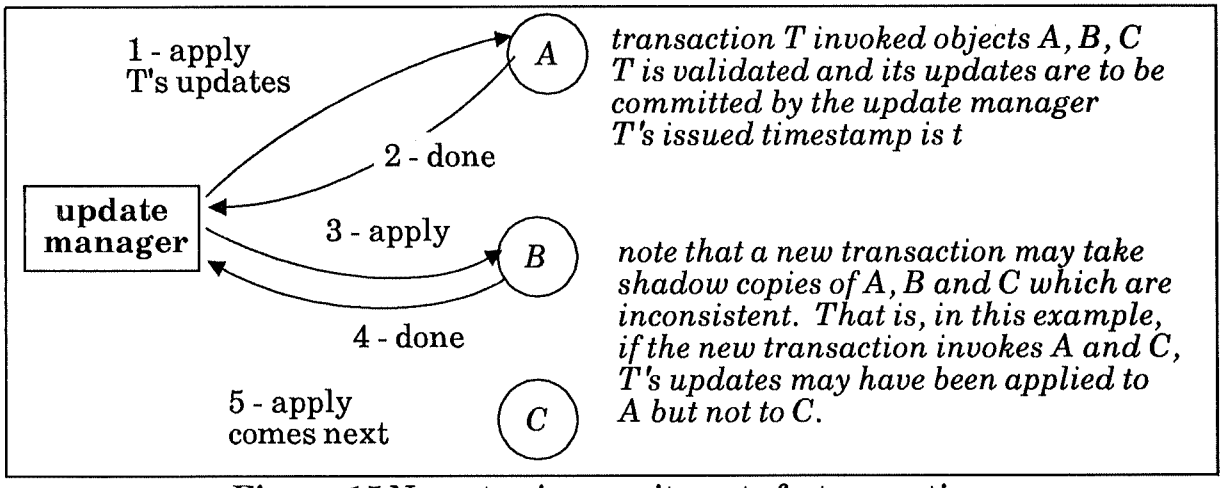


Figure 15 Non-atomic commitment of a transaction.

The execution of the transaction continues, invoking shadow objects until:

either	*abort*:	the shadow objects are simply discarded;
or	*commit*:	the validator is called.

The validator has knowledge of all transactions whose validation or update phases overlap execution of the transaction that is to be checked. When a transaction invokes an operation on a shadow object there may be transactions with outstanding updates guaranteed for that object. The validator must ensure that there has been no conflict. The information used by the validator might be extended to take into account transactions that have started to execute since this one, but we shall not consider this possibility further.

Two conditions need to be checked by the validator. If either cannot be met the transaction must be aborted.

1. The execution must be based on a consistent system state.

   The requirement is that the versions of the shadow objects were all current at some particular transaction timestamp. Figure 16 shows a possible scenario. Suppose that at the start S_T of transaction T the earliest unacknowledged timestamp is u; that is, a transaction with timestamp u is in the process of being

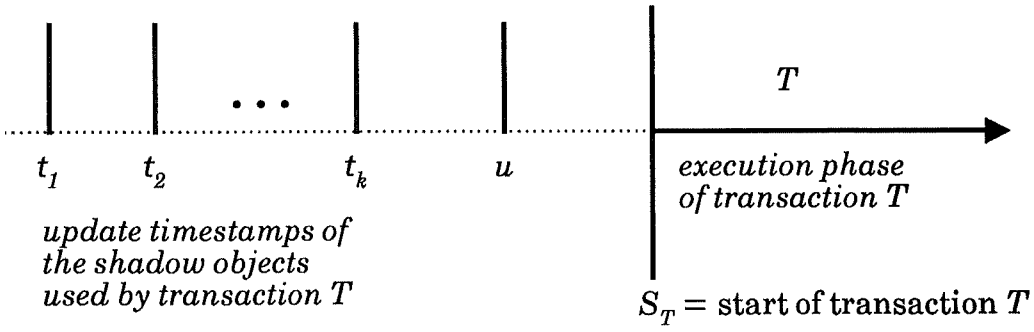


Figure 16 checking for a consistent state.

committed and its updates are not acknowledged by all the objects it has invoked. Suppose that during T's execution phase shadow copies are made whose version timestamps are (in some order) $t_1, t_2, \ldots t_k$. If all timestamps $t_1, t_2, \ldots t_k$ are earlier than u, then certainly all versions were current at the latest of the timestamps recorded, say t_k.

If interference is low it is likely that all updates to the objects used by T will have been acknowledged before the transaction starts, and also that no further updates occur during the execution phase. Even if this precise scenario is not followed it is still possible for the set of shadows to be consistent.

If validation of the shadow objects used by T succeeds then the execution of T is based on a timestamp at which all the shadow versions were consistent, let us call this the **base timestamp** of T.

2. The transactions must be serialisable.

   The transactions with which the given transaction T must be reconciled are those validated for update with a timestamp later than the base timestamp of T, whether or not their updates have been applied. Recall that once a transaction is validated its updates are guaranteed, and that updates are applied at each object in the order of the timestamps issued by the validator.

   The requirement is that an ordering of these transactions can be found in which serial update is meaningful: that is, that the final system state reflected after the (serial) update of the set of transactions must be consistent with all of their execution phases, performed concurrently. Although the definitions of conflict needed can be based on non-commutativity this can be unnecessarily restrictive, as we shall see in an example.

Provided that both conditions are met, the transaction can be accepted, recorded as validated and issued a timestamp. This establishes its position in the queue of validated transactions that are waiting to update. In simple cases object update is just a matter of copying a shadow object back into persistent memory. In other cases it may be necessary to reapply the operations of a transaction to a version more recent than the original shadow.

OCC has very different properties from timestamp ordering. In the latter transactions are scheduled in a predetermined order, usually that of transaction start. In OCC the order is determined at validation time, and in theory the validator is free to insert the current transaction at any position in the queue for update. The validation algorithm could therefore become quite elaborate, but it is probably not worth going to great lengths in an attempt to optimise. OCC is suitable only if there is little interference between transactions, and the hope is that simple validation will normally succeed.

7.3.1 Examples

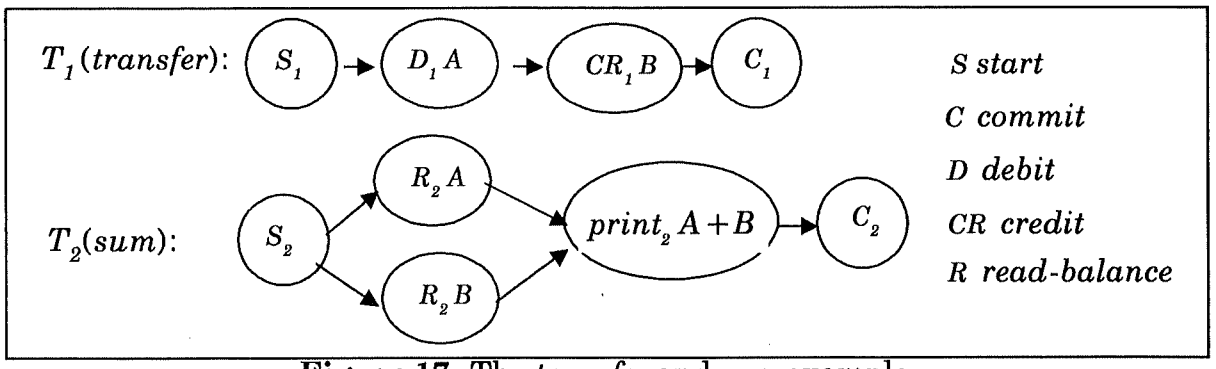


Figure 17 The *transfer* and *sum* example.

Figure 17 shows concurrent *transfer* and *sum* transactions. Let us assume that both transactions are using shadows of the same values of A and B. Note that *sum* is a read-only transaction. It might be argued on the basis of this example that there is no point in taking shadows for a read-only transaction. The counter argument is that a number of reads might be required from an object and taking a shadow ensures that the reads are performed on the same version of the object. We shall make the assumption that shadows are taken when an object is first invoked, either for reading or writing.

The *sum* transaction uses its shadow values of A and B quite independently of what the *transfer* transaction is doing to its shadow values of A and B. In Section 4.1 we considered *read-balance* and *credit* (or *debit*) to be non-commutative. If consistent shadow copies are taken by both transactions this is no longer relevant. In fact, if the shadows used by a read-only transaction represent a consistent system state then that transaction cannot fail. If we allow the possibility that the shadows used by a transaction do not represent a consistent state then it can be rejected when it attempts to *commit*. A problem here is that the transaction may have performed output based on inconsistent object values. This problem is not exclusive to systems which use OCC. Whatever concurrency control scheme is used in a system there must be a policy on how to deal with aborted transactions that have performed output.

If we assume each transaction is working on a consistent set of shadows we need only be concerned with operations that are non-commutative with respect to state changes at the object. Values output by the transaction relate to a consistent version of the system state and cause no problems. The *sum* transaction will therefore be validated as correct at *commit*, whenever this is requested. It has not changed the value of A or B.

The values output by a number of transactions that are working on the same version of system state are not the same as those that would be output by a serial execution of those transactions on the persistent state; they execute in parallel on the same version. The transactions are however forced to *commit* in some serial order. It should, if possible, be arranged that the output is not misleading to the application, for example "credit of £100 accepted" rather than "new balance = £1000". In some cases it may be necessary to abort a transaction because of the output it has performed.

When the *transfer* transaction requests *commit*, the operations it has done on A and B are **validated**. The information recorded at the persistent objects and their

shadows is sufficient for the *commit* to be validated as correct or rejected. Suppose *transfer* has invoked an operation on *A* or *B* which belongs to a conflicting pair. If some other transaction has committed (since the shadow was taken for *transfer*) the result of an invocation of the conflicting operation of that pair, then *transfer* must be aborted. The validation phase checks this for all the operations that the transaction requesting *commit* has invoked on all objects.

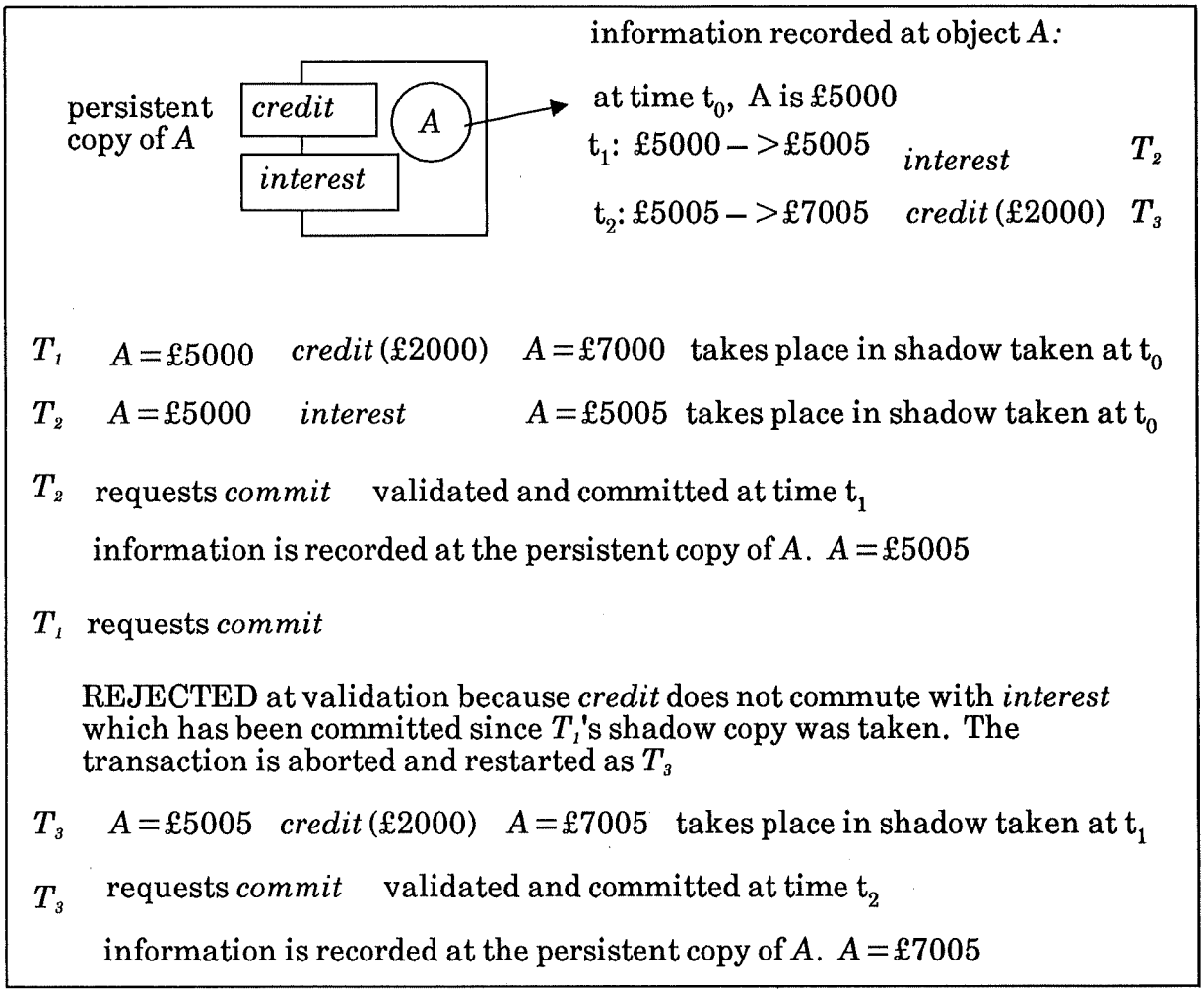


Figure 18 Example showing abort and restart.

Figure 18 shows transaction *abort* and restart when non-commutative *credit* and *add-interest-to-balance* operations are invoked on the same value of a bank account. The validation phase indicates whether *commit* is possible and the *commit* phase must ensure the correct persistent values, taking into account changes that have been committed since the shadows were taken.

In the example, T_1 invokes *credit* (£2000) on a shadow copy of *account-A*, changing its value from £5000 to £7000. A shadow copy taken from the same persistent object value has *add-interest-to-balance* invoked by T_2, changing its value from £5000 to £5005. This latter transaction T_2 is first to *commit* and the persistent value of *account-A* is updated to £5005. T_1 now requests *commit*. Because *credit* and *add-interest-to-balance* are defined to be non-commutative, therefore conflicting, the *commit* is rejected. The transaction is restarted as T_3 with a shadow of *account-A* with value £5005. The *credit* (£2000) is performed at the shadow giving £7005 for the value and this value is then committed at the persistent copy of the object.

Notice that when transaction T_1 requests *commit* and is rejected, applying the *credit* operation at that stage at the persistent copy of the object would yield £7005! The requirement for commutativity is too strong for a case such as this. Commutativity enforces that the same result is obtained whatever the order of execution of a pair of transactions. As soon as one transaction has committed, the serialisation order is defined. If the rejected transaction is aborted and restarted it is from the system state committed by the first. They are no longer running in parallel. Ideally, this should be taken into account when T_1 requests *commit*.

If invocations are carried out on shadow copies which do not conflict with subsequent updates at the persistent object, the invocations can be reapplied to the object on *commit*. For example, suppose a shadow copy of *account-A* was credited by £1000, changing its value from £4000 to £5000. Suppose at *commit*, other transactions have caused the balance to reach the value £8000 by invoking operations that commute with *credit*. The credit operation is redone at the object giving a balance of £9000.

7.3.2 Discussion of the examples

Generalising from these examples, when a transaction requests *commit* and after a consistent starting point has been ascertained (condition 1):

- the validation phase uses the information recorded with each persistent object involved and its shadow, to check whether any non-commutative pairs of operations have been invoked on the object by this transaction and any other that has committed since this transaction took its shadows. As noted above, this definition of conflict may be too restrictive. See Herlihy (1990) for further reading.
- If the validation phase is successful, the transaction is committed. This may involve redoing (at the persistent copy of the objects) the operations that have changed the values of shadow objects. This can be done because the validation phase has rejected the *commit* if any of the invocations of the transaction do not commute with committed invocations.
- If the validation phase is not successful the transaction is aborted.

Optimistic concurrency control allows every operation invocation to go ahead without the overhead of locking or timestamp checking; it achieves high object availability. The fact that shadows are taken and work proceeds without delay makes this method suitable for applications in which timing guarantees are required and in which conflict is rare. The overhead occurs when *commit* is requested. The validation phase uses the information stored locally at each object. If all the objects invoked by a transaction indicate that *commit* is possible then the updates can go ahead.

Optimistic concurrency control operates on a first come (to *commit*) first served basis. If there are several shadows of an object, the state of the first to *commit* becomes the new object state, without regard to fairness or priority of the transaction. If contention is rare the method works well. If the application is such that transactions might invoke heavily-used objects (data "hot-spots") they are likely to be aborted and restarted, thus wasting system resources. The method should probably not be used if this is likely to occur.

8 Distributed concurrency control and commitment

In Figure 10 the TPS instances comprising the distributed TPS must cooperate. A client submits a transaction to one TPS. The transaction manager identifies and locates the objects invoked by the transaction. Local object invocations are passed to the local scheduler, remote object invocations are passed to the (scheduler of the) appropriate remote TPS. A TPS must therefore handle both transaction requests from local clients and requests from remote TPSs to invoke operations on its local objects. For this latter type of request we assume initially that the TPS does not have a specification of the whole transaction. The scheduler at each node is passed operations which come from both local and remote transaction submissions. As before, the scheduler is at liberty to invoke the operations in any order, subject to the concurrency control algorithm it implements.

We must consider:

- **Concurrency control**: how a serialisable schedule is achieved in a distributed TPS. The methods introduced in Section 7: locking, timestamping and optimistic concurrency control will be reconsidered for a distributed system.
- **Commitment**: The transaction manager at a single node receives a client request for a transaction and initiates local and remote operation invocations. It must be notified of the results of attempted invocations; whether an invocation was accepted and done, or rejected, or perhaps that a *lock* request has been outstanding for some specified timeout period, depending on the method used to achieve concurrency control. Assuming that all the transaction's invocations (at all the nodes) have been notified as "done" to the initiating transaction manager the transaction must then be committed. We shall study how this can be achieved in a distributed system in the presence of partial failures.

Communication

The above discussion assumes communication between TPS instances. In some cases specific applications protocols are needed, for example, an atomic commitment protocol. Application protocols are implemented above general communications protocols such as remote procedure call or some form of message passing.

Communications protocols are designed to allow for the possibilities of congestion and failure of the network and the communicating nodes. The mechanism used is the timeout. If a timeout expires the protocol may immediately inform the higher level which invoked it or may retry a few times to allow for congestion. We shall assume the latter here for simplicity. The higher level may therefore receive a "success" notification or an exception, indicating a failure. The application protocol must be designed on this basis, as we shall see for atomic commitment in the presence of failures.

9 Concurrency control in a distributed transaction system

9.1 Two-phase locking (2PL)

In Section 7.1 two-phase locking was shown to enforce a serialisable order on the object invocations of transactions. We noted that semantic locking would increase concurrency compared with the exclusive locking approach which is usually employed. We should consider how the two phases, of acquiring and releasing locks, can be implemented in a distributed system. In a centralised system the transaction

manager knows when locks on all the objects of a transaction have been acquired and the operations done. The *unlock* operation can then be invoked on all the objects.

In a distributed system, all the schedulers involved in a transaction must inform the transaction manager at the coordinating node that the requested locking and invocation of objects is done. Only then can the unlock operations be sent back to the schedulers concerned. Notice that use of a protocol of this kind prevents timing problems. The phases are defined at one node: the coordinating node of the transaction. For a strict execution, that enforces the property of isolation in the implementation, the locks are not released until the transaction is committed.

The method is subject to deadlock and we assumed in Section 7.1 that deadlock detection and recovery would be carried out by a component of a (centralised) TPS. This component, let us call it the lock manager, maintains information on the objects that have been locked by transactions and the outstanding lock requests. The implicit assumption was that all the objects concerned were local to the TPS so that complete information on all transactions was available. A deadlock detection algorithm could be run and action taken such as aborting some or all of the deadlocked transactions.

In a distributed TPS the lock manager at any node can maintain the same information as described above for invocations by local transactions. It can be told about requests for remote invocations by local transactions. It can also know about the requests for local invocations by remote transactions. What it does not know is the remote locks held by these remote transactions and their outstanding requests, and so on until the transitive closure of locks and requests is computed. This information is needed for deadlock (cycle) detection.

The overhead of two-phase locking is large, particularly when extended for use in a distributed system. Each node must maintain a great deal of information to detect and recover from deadlock and the method scales badly. In practice a simpler approach based on timeout might be adopted. If a transaction fails to acquire a lock in a given time it is aborted and all the locks it holds are therefore freed.

9.2 Timestamp ordering (TSO)

The major advantage of this method for a distributed implementation of concurrency control is that only information held at each object is used to achieve serialisation. Contrast this with the overhead described above for the distributed deadlock detection associated with distributed two-phase locking.

At first sight it seems that there might be a problem associated with time in using the method in a distributed TPS. In a centralised system the timestamps have a serial order because they are generated from a single clock. In a distributed system a system-wide ordering of timestamps is needed for correct serialisation of transactions. This is quite easy to achieve. The essential requirement for correctness is that every object takes the same decision about the relative order of two timestamps. First, suppose that we use the local time of the coordinating node of the transaction for the timestamp. Except for the case of identically equal times, these values could be used to achieve a correct serialisable execution. To deal with the case of equal times we just need a system-wide policy to achieve the arbitration. The node-identifiers could be used, for example.

Although this method of generating and using timestamps achieves correctness it favours nodes with fast-running clocks when arbitration between equal times is needed.

9.3 Optimistic concurrency control

In Section 7.3 we argued that it would be pessimistic, rather than optimistic, to ensure at transaction start that the shadow copies of objects used by the transaction during the **execution phase** represent a consistent system state. To achieve this consistency we should have to sacrifice guaranteed high availability of objects since an object might be held during commit of some transaction when required by another. It would be necessary to enforce atomic commitment over all the objects invoked by a transaction and to take shadow copies of all the objects needed by a transaction atomically. Section 10 shows how atomic commitment can be carried out in a distributed system. This is too heavyweight when we optimistically assume that conflict is unlikely. Also it is not always possible to know at transaction start all the objects that will be needed by a transaction.

As in timestamp ordering, the decision on whether a transaction may *commit* is based on information recorded at each object. The decision is made during the **validation phase**, after a transaction requests *commit*. Objects vote independently to *accept* or *reject* the transaction, and this aspect of OCC is therefore appropriate for a distributed system. There is a need to ensure that the local contexts for validation at the objects participating in a transaction are consistent.

The discussion of Section 7.3 was equally applicable to a centralised and a distributed implementation. An essential requirement in a distributed system is that transactions are validated for update in a well-defined serial order. Decisions on validation must be communicated to the participating objects atomically, and we shall sketch a protocol to achieve this in Section 10.2.

We required that, in the **update phase** of a transaction, update invocations are applied to objects in persistent memory in serial order, transaction by transaction. It is the responsibility of the update manager to ensure that all updates succeed. The update manager will know at any time those transactions for which updates have succeeded. It can therefore be asserted that the updates up to those of some transaction have succeeded.

This places a requirement on the underlying communications system used for making remote object invocations. There are issues specific to a distributed implementation that are associated with the independent failure modes of its components. It is necessary to assume that the invocations are made at the object in the order they are sent by the update manager and that these invocations are acknowledged to the update manager. We require that messages are not lost without notification and are not received in a different order from that in which they are sent. This can be achieved by selecting an appropriate communications protocol.

10 Commit and abort in distributed systems

Let us assume in the case of 2PL and TSO that the transaction manager at the coordinating node has received a request to *commit* a transaction. We have to ensure:

Atomicity: either all nodes commit the changes or none do, and any other transaction perceives the changes made at every node or those at none.

Isolation: that the effects of the transaction are not made visible until all nodes have made an irrevocable decision to commit or abort.

We have set up the conditions that no scheduler will refuse to *commit* the transaction on correctness grounds. In 2PL and TSO we have avoided this possibility by only allowing serialisable executions to take place. For these pessimistic methods there are two remaining issues to consider:

- Nodes or network connections might fail during *commit.*
- Other nodes may be attempting to carry out distributed *commit* at the same time and this might involve an intersecting set of objects.

Atomic commitment protocols address these issues. The two-phase commit protocol is discussed in Section 10.1.

In 2PL and TSO we can ensure isolation by holding locks until after *commit*, thus guaranteeing *strictness*. If strictness is enforced we can assume that all the objects that were invoked by the transaction to be committed are available to the *commit* procedure for it. To achieve this we have introduced a possible additional delay when an object is invoked in 2PL and TSO. Once again we are restricting concurrency (object availability) in order to ensure that transactions see a consistent system state.

In the case of OCC we have made no attempt to ensure the correctness of an executing transaction, preventing harmful consequences by invoking operations on shadow objects. After an executing transaction has issued *commit* we have to ensure during validation:

Consistency: the execution has been based on shadow objects derived from a consistent system state, and there has been no interference at any object from transactions executing concurrently.

We have argued against the atomic commitment of updates for OCC, but its use has definite advantages. Herlihy (1990) proves the correctness of OCC algorithms that are based on a two-phase protocol for update in which validation is performed at each object during the first phase. In this paper he also shows that optimistic and pessimistic methods can be mixed on a per-object basis. Should we wish to enforce strictness (execution based on consistent system state only) for OCC it would be necessary not only to commit updates atomically but also to take the shadow copies needed by a transaction atomically. The drawback of this in a distributed system is that it can greatly reduce object availability, which was one of the goals when OCC was introduced.

On the other hand we have to ensure a serialisable execution, which means that a consistent serial order of committed transactions must be established system wide. Global consistency is enforced during validation, and locks held during this phase relate only to the process of validation, not to the objects themselves. Once a transaction has its updates guaranteed these can be applied asynchronously at the participating objects, and executing transactions merely read whatever version the object has reached when creating a shadow object. Objects are therefore available except at the moment of version change. The drawback is loss of strictness, with the

result that a transaction may be rejected simply because its shadow objects were inconsistent. Since transactions always execute to completion there can be a considerable waste of system resources. A protocol for atomic validation is described in Section 10.2.

We shall now look at a widely used atomic commitment protocol: two-phase commit (2PC). Other such protocols have been defined which vary with respect to the failures they can tolerate and the number of communications that are needed. Further reading on the topic may be found in Bernstein et al. (1987), Ceri and Pelagatti (1985), Bell and Grimson (1992).

10.1 The two-phase commit protocol

We assume a number of participating nodes and a *commit* manager at the coordinating node of the transaction, see Figure 19a. Each participating node "votes" for *commit* or *abort* of the transaction. Ultimately, all the nodes must make the same decision and the purpose of the protocol is to ensure this. The two phases involved are, broadly:

phase 1: the *commit* manager requests and assembles the "votes" for *commit* or *abort* of the transaction from each participating node;

phase 2: the *commit* manager decides to *commit* or *abort*, on the basis of the votes, and propagates the decision to the participating nodes.

Showing more detail of the steps involved:

1. The *commit* manager sends a request to each participating node for its vote.
2. Each node either votes *commit* and awaits further instructions; or votes *abort* and stops (exits from the algorithm). Note that a *commit* vote indicates that both the new value of the data object and the old value are stored safely in stable storage so that the node has the ability to *commit* or *abort*.
3. The *commit* manager receives the votes and adds its own vote. If all the votes are to *commit* it decides *commit* and sends *commit* to every participating node. If any vote is *abort* it decides *abort* and sends *abort* to all the nodes that voted *commit* (the others have already stopped). The *commit* manager stops.
4. The participating nodes that voted *commit* are awaiting notification of the decision. They receive this notification, decide accordingly and stop.

We assume that the decision indicated in the above description is permanent, guaranteed to persist; there is a point of decision in the algorithm at the *commit* manager.

We must consider how the protocol might handle congestion and failures in the nodes and connections involved. The two-phase commit protocol is an application protocol which is implemented above lower level protocols. Each communication involved in two-phase commit will have a success indication or an exception returned from the level below. We shall assume that the lower levels have made allowance for congestion (by retrying after timeouts) and that an exception indicates a failure of some kind. The protocol must be designed on this basis. Bear in mind that a decision cannot be reversed; once a decision is made, failure recovery procedures must ensure it is implemented.

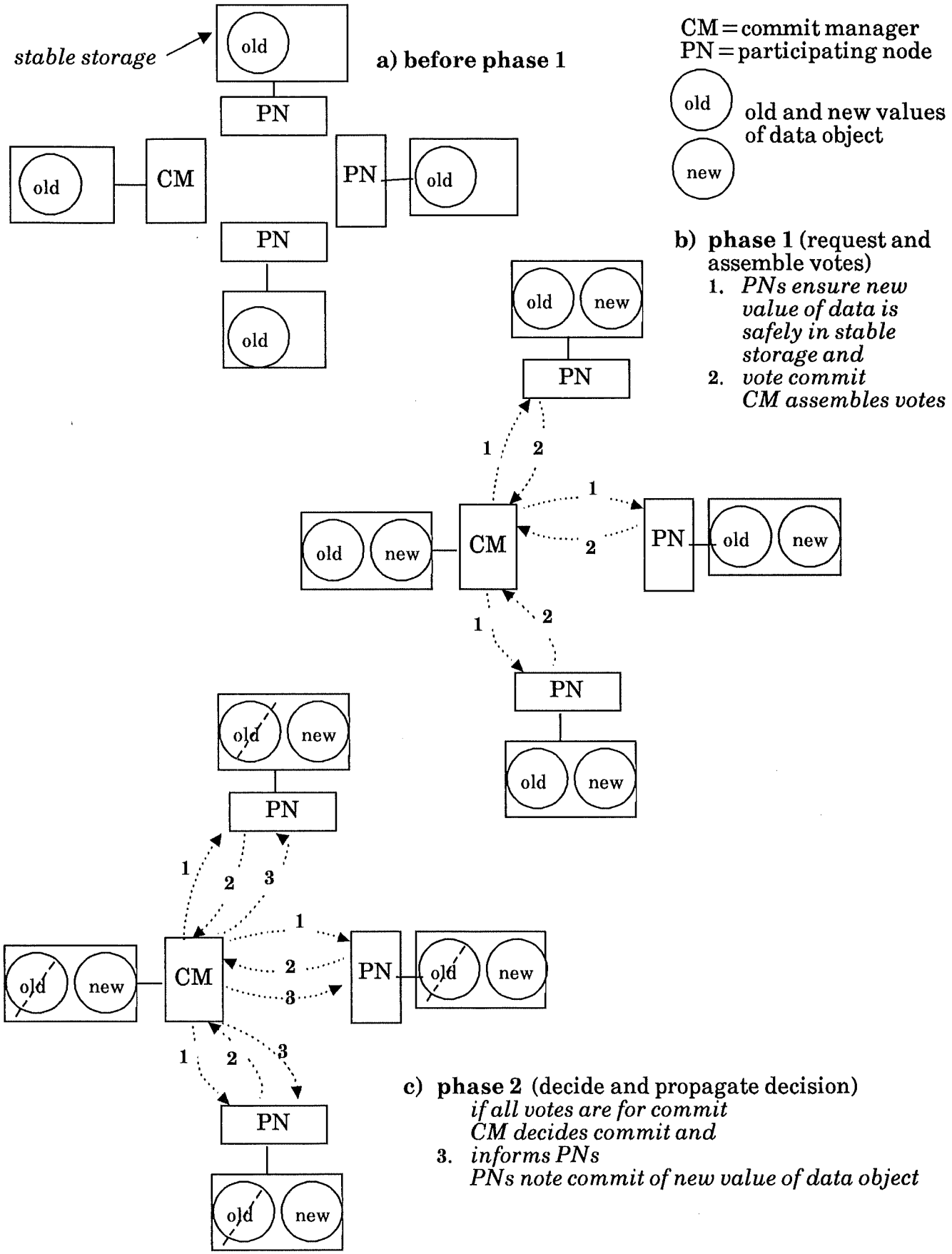


Figure 19 The two-phase commit protocol when all nodes vote commit.

Suppose that an RPC from the coordinating node to each participating node is used to implement steps 1 (request for vote) and 2 (reply with vote). A failure of any one of these RPCs is assumed to indicate a failure of that participating node. The vote from that node might have been *abort*; an *abort* vote is the only safe assumption, so the transaction is aborted.

Suppose that step 3 (send the decision to nodes that voted *commit*) is also implemented by RPC, the reply indicating just an acknowledgement of receipt. A failure of any one of these RPCs indicates that the decision to *commit* or *abort* may still need to be effected at that node. The decision cannot be changed; it has been made and put into effect at the management node and at the nodes which have received the decision from the manager. Recovery from failure at any node must therefore involve terminating correctly any two-phase commit that was in progress when the node failed and sufficient information must be stored in persistent storage to make this possible. On restarting, the node could ask the manager for the decision. The manager knows that the node failed and can expect the request.

The above discussion has outlined how failure resilience might be approached in two-phase commit if one or more of the participating nodes fail. The manager might also fail:

1. after sending requests for votes but before deciding. All the participating nodes that voted *commit* will time out (at the two-phase commit level) waiting for a decision.
2. after deciding (and recording the decision in persistent store) but before sending the decision to any participating nodes. All the participating nodes that voted *commit* will time out waiting for a decision.
3. after deciding (and recording the decision in persistent store) and after sending the decision to some but not all participating nodes. Some of the participating nodes that voted *commit* will time out waiting for the decision.

Any one participating node which times out cannot distinguish between these three possibilities. So far we have assumed that the participating nodes know about the manager but not about each other. It would be easy to add a list of participating nodes to the request for vote. Any node that timed out could attempt to find out the decision from the other nodes. Bell and Grimson (1992) and Bernstein et al. (1987) give detailed termination protocols for 2PC and also discuss three-phase commit (3PC) protocols.

10.2 Two-phase validation for optimistic concurrency control

We assume a number of participating nodes and a *validation manager* at the coordinating node of each transaction, see Figure 20. In addition there is a single logical agent in the system, the *update manager*, which is responsible for the queue of transactions that have been validated for update. Each transaction involves a number of participating objects, and each object votes independently *accept* or *reject* on whether there has been conflict. In addition, any object that votes to accept a transaction notifies the validation manager of the version timestamp of the shadow object that was created for the execution phase.

In the figure transaction T is being validated by validation manager 2, the participating objects being A, D and X. Another transaction involving objects C and

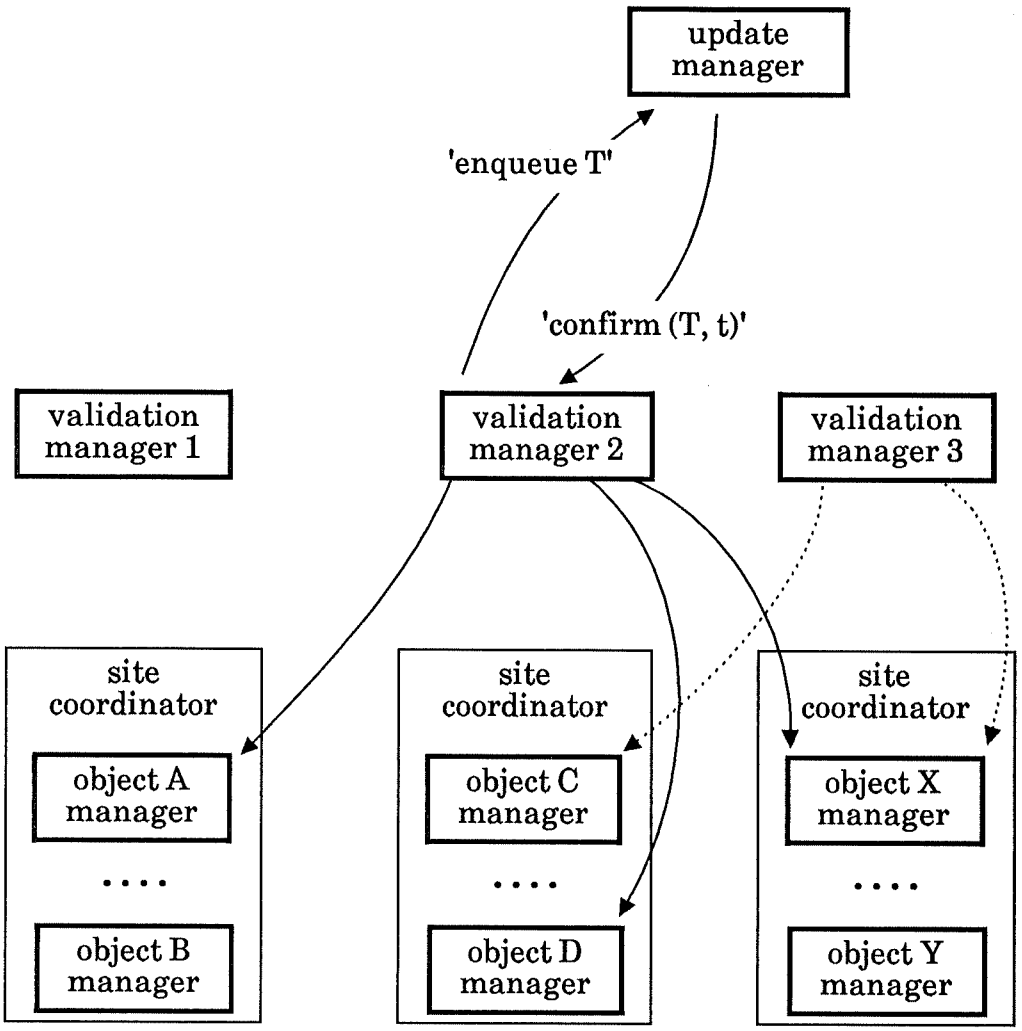


Figure 20 Distributed validation for OCC.

X has just issued *commit*, and validation manager 3 is to validate it. Two-phase validation has much the same general structure as the two-phase commit protocol described in section 10.1, but there are two important differences. First, an object may be involved in several transactions concurrently; one transaction may issue commit when the object is already participating in the validation phase of some other transaction. It would be possible to block the newly requested validation, but such a policy would run the risk of deadlock. A better approach is to ask the validation manager to try again later if the object's vote is still of interest. Secondly, if all participants vote *accept* at the first phase and the transaction is validated successfully, they do not need to apply the updates during the second phase. Instead the validation manager applies to the update manager for a timestamp for the transaction, and at this point the serialisation order is determined. This interaction is atomic. The validation manager must then inform each participating object of the decision, so that any subsequent validation takes place in a consistent context.

Interaction between the validation manager and participating objects follows the general pattern shown in Figure 20, but the details of the protocol must take account of the above differences. The two phases involved are, broadly:

phase 1: the *validation manager* requests and assembles the "votes" for *accept* or *reject* of the transaction from each participating object, except those that say *busy*;

phase 2: the *validation manager* decides to *commit*, *reject* or *retry*, on the basis of the votes, taking account of shadow object consistency. If the decision is *commit* it applies to the *update manager* for a timestamp. The decision is propagated to participating objects.

Considering the steps involved in more detail:

1. The *validation manager* sends a request to each participating object for its vote on the transaction execution.
2. Any object that is performing validation of some other transaction replies *busy* and awaits further information. Other objects either vote *accept* (indicating the shadow object version timestamp) and await further instructions; or vote *reject*, record rejection locally and discard the shadow object. Objects that vote *reject* need not be contacted further.
3. The *validation manager* receives the responses and determines what action to take.

   If any vote is *reject* it decides *reject* and sends *reject* to all the objects that replied *accept*
   or *busy* (the others have already stopped).

   Otherwise if any vote is *busy* it asks all the objects to suspend validation for a subsequent *retry*. Objects which voted *accept* are then free to validate other transactions. The *validation manager* will retry after a suitable interval. Objects which voted *accept* originally may then vote *reject*.

   If all the votes are *accept* the *validation manager* decides whether to *commit* on the basis of shadow object consistency. If the versions were inconsistent, see Section 18.6, it decides *reject* and sends *reject* to all the objects; the *validation manager* stops.
4. If the decision is *commit* the *validation manager* applies to the *update manager* for a timestamp for the transaction. The decision is propagated to all participating objects, together with the timestamp. The *validation manager* stops.

The point of decision in the above algorithm occurs when the update manager issues a timestamp for the transaction. At that point all participating objects have voted *accept* in a two-phase protocol, and they must be prepared to apply updates at some later stage.

It is worth considering the extent to which concurrent execution is sacrificed, and the consequences for object availability. First, the interaction to obtain a timestamp from the update manager is atomic, and requests must be serviced by a single queue manager. Secondly, when a validation manager receives a *busy* reply from an object it abandons the attempt to validate for a while. Objects will service only one request to validate at a time. Both of these restrictions apply to the **validation phase** of a transaction. The *busy* reply may increase the chance that a transaction is rejected, thus wasting system resources. On the other hand there are no bad implications for object availability at the **execution phase**, since updates are applied locally at each object during the **update phase** without a protocol that involves external sites.

Shadow objects can therefore be created except when a request is received during a change of object version (essentially a rename operation).

This discussion has not considered how the protocol might handle congestion and failures in the nodes and connections involved. Two-phase validation, like two-phase commit, is an application protocol which is implemented above lower level protocols. The considerations outlined in Section 10.1 apply equally here.

11 Summary and conclusions

The nature of distributed systems is such that the designers of distributed applications must take account of concurrent execution and failure of their component parts. Transactions were developed in the context of database systems to cope with just these issues.

We use an object model throughout and assume that a single operation on an object can be made atomic; that is, recoverable and free from interference with other operation invocations on the same object. The discussion focusses on transactions which implement composite operations, comprising related operations on one or more objects.

Using an object model for transactions allows concurrency control to be based on application semantics. The concurrency behaviour of each object may be specified by an application developer, thus informing the implementation which operation invocations on an object may proceed in parallel and which conflict. A general criterion for specifying whether a pair of operations conflict is commutativity.

The order of the invocations by transactions of potentially conflicting pairs of operations on objects is the basis on which serialisability of transactions can be established. The individual object is the witness to these orderings. An object model is therefore equally suited to centralised and distributed implementations of transactions. Decision are taken locally at each object and there is no requirement for system-wide time.

Pessimistic methods of concurrency control include two-phase locking (2PL) and timestamp ordering (TSO). Practical implementations of lock-based concurrency control tend to use exclusive locks or perhaps a combination of shared read locks and exclusive write locks with the possibility of converting a shared lock to an exclusive one. We have assumed that the granularity of locking is the whole object. A higher degree of concurrency is achievable if components of objects can be locked.

Lock-based methods are subject to deadlock and a simple way of dealing with the possibility is to use a timeout mechanism instead of a formal deadlock detection procedure. The latter would introduce a great deal of overhead in a distributed implementation.

Timestamp ordering is simple to implement but allows only one possible serial ordering of conflicting pairs of operations, that of the associated transactions' timestamps. A transaction which is serialisable with ongoing transactions may be aborted if its timestamp is deemed too late by any object it invokes.

All methods of concurrency control must ensure that all the objects invoked by a transaction take the same decision on *commit* or *abort*. Pessimistic methods

implement *commit* as a single atomic operation; that is, all the objects invoked by a transaction are unavailable to other transactions during *commit*. We studied one atomic commitment protocol, two-phase commit, as the basis for transaction commit in a distributed system.

A strict execution schedule enforces the property of isolation in an implementation of a pessimistic method of concurrency control; that is, no transaction can see the results of an uncommitted transaction. This prevents cascading aborts and complex procedures for recovering state on transaction abort or on a crash. If strictness is enforced, applications must be able to tolerate delay on access to objects. Even if strictness is relaxed, an atomic commitment procedure may still cause delay on access to objects.

Optimistic concurrency control (OCC) has been proposed for applications where conflict is rare. If OCC is used when conflict occurs frequently, an application may do work on shadow versions of objects which is later rejected and must be repeated. This wastes both system and application resources. The attraction of OCC is that it incurs very little overhead when there is no conflict. In this case, validation of a transaction succeeds and it is committed by recording its results at the persistent objects it has invoked. An application's work on shadow versions of objects may simply be discarded if there is a failure (or if the transaction must abort because of conflict).

A characteristic of OCC is that the objects required by a transaction may be accessed without delay. Objects are not made unavailable by concurrency control or atomic commitment procedures. This may be important for applications which must meet real-time requirements.

The fact that a transaction works with shadow versions of objects in OCC gives a suitable model for continued working in the presence of server or communication failure and for planned detached working. If the primary copies of objects are not accessible because of failures, local versions may be used. The commit procedure for OCC, taking into account object semantics, gives a coherent model for merging the results of a detached transaction with persistent system state. In the case of planned detached working a simpler procedure than that described in Section 7 is appropriate. The application can arrange to work with a set of shadows that are known to be consistent.

A hybrid approach to concurrency control can be used in a system which is based on object semantics, see for example [Herlihy 90]. It is desirable to use OCC for objects which are rarely shared; when it is important to avoid delay on access to objects; when there are failures or when detached working is required. When these conditions do not hold OCC can be expensive and a pessimistic method may be used for some objects in a system.

Acknowledgements

To members of the OPERA project: Noha Adly, Mohammed Afshar, John Bates, Huang Feng, Richard Hayton, Sai Lai Lo, Scarlet Schwiderski, Robert Sultana, ZhixueWu. To Heather Brown, visiting from the University of Kent and to John Wilkes of HP Research Labs., Palo Alto, CA.

References

[Aho et al. 83]
Aho A.V., Hopcroft J.E.. and Ullman J.D.
"Data Structures and Algorithms", Addison Wesley, 1983

[Bacon 93]
Bacon J. M. "Concurrent Systems", Addison Wesley 1993

[Bell and Grimson 92]
Bell D. and Grimson J., "Distributed Database Systems" Addison Wesley 1992

[Bernstein et al. 87]
Bernstein P.A., Hadzilacos V. and Goodman N.
"Concurrency Control and Recovery in Database Systems" Addison Wesley 1987

[Ceri and Pelagatti 85]
Ceri S. and Pelagatti G. "Distributed Databases, Principles and Systems"
McGraw Hill 1985

[Gray and Reuter 93]
Gray J and Reuter R, "Transaction Processing: Concepts and Techniques"
Morgan Kaufmann 1993

[Herlihy 90]
Herlihy M. "Apologizing versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types" ACM Transactions on Database Systems 15(1), March 90

[Korth and Silberschatz 91]
Korth H.F. and Silberschatz A. "Database System Concepts" 2nd edition,
McGraw Hill New York1991

[Weihl 84]
Weihl W. E. "Specification and Implementation of Atomic Data Types" Tech. Rept. MIT/LCS/TR-314, MIT Lab for Computer Science, March 1984, PhD Dissertation

[Weihl 89]
Weihl W. E. "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types" ACM Trans Prog Lang and Sys, 11(2), April 1989

[Weikum 91]
Weikum G, "Principles and Realization Strategies of Multilevel Transactions"
ACM Transactions on Database Systems 16(1): 132-180, March 1991

[Wu 93]
Wu Z, "A New Approach to Implementing Atomic Datatypes"
Computer Laboratory PhD thesis in preparation, 1993

[Yahalom 91]
Yahalom R., "Managing the Order of Transactions in Widely Distributed Data Systems" University of Cambridge PhD thesis and TR 231, Aug 1991