

Algorithm 11.4: Basic Timestamp Ordering (BTO-TM) Algorithm**Input:** *msg* : a message**begin** **repeat** wait for a *msg* ; **switch** *msg type* **do** **case** *transaction operation* {operation from application program } let *op* be the operation ; **switch** *op.Type* **do** **case** *BT* $S \leftarrow \emptyset$; {*S* is the set of sites where transaction executes } assign a timestamp to transaction – call it $ts(T)$; DP(*op*) {call DP with operation} **case** *R, W* find site that stores the requested data item (say S_i) ; BTO-SC $_{S_i}(op, ts(T))$; {send *op* and *ts* to SC at H_i } $S \leftarrow S \cup S_i$ {build list of sites where transaction runs} **case** *A, C* {send *op* to DPs at all sites where transaction runs} DP $_S(op)$ **case** *SC response* {operation must have been rejected by one SC} $op.Type \leftarrow A$; {prepare an abort message} BTO-SC $_S(op, -)$; {ask other SCs where transaction runs to abort}

restart transaction with a new timestamp

case *DP response* {operation completed message} **switch** *transaction operation type* **do** let *op* be the operation ; **case** *R* return *op.val* to the application ; **case** *W* inform application of completion of the write ; **case** *C* **if** *commit msg has been received from all participants* **then**

inform application of successful completion of transaction

else {wait until commit messages come from all}

record the arrival of the commit message

case *A*

inform application of completion of the abort ;

 BTO-SC(*op*) {need to reset read and write timestamps} **until** *forever* ;**end**

Algorithm 11.5: Basic Timestamp Ordering Scheduler (BTO-SC) Algorithm**Input:** $op : Op; ts(T) : \text{Timestamp}$ **begin**

```

    retrieve  $rts(op.arg)$  and  $wts(arg)$  ;
    save  $rts(op.arg)$  and  $wts(arg)$  ;           {might be needed if aborted }
    switch  $op.arg$  do
        case  $R$ 
            if  $ts(T) > wts(op.arg)$  then
                 $DP(op)$  ;           {operation can be executed; send it to the data processor}
                 $rts(op.arg) \leftarrow ts(T)$ 
            else
                send "Reject transaction" message to coordinating TM
        case  $W$ 
            if  $ts(T) > rts(op.arg)$  and  $ts(T) > wts(op.arg)$  then
                 $DP(op)$  ;           {operation can be executed; send it to the data processor}
                 $rts(op.arg) \leftarrow ts(T)$  ;
                 $wts(op.arg) \leftarrow ts(T)$ 
            else
                send "Reject transaction" message to coordinating TM
        case  $A$ 
            forall the  $op.arg$  that has been accessed by transaction do
                reset  $rts(op.arg)$  and  $wts(op.arg)$  to their initial values
    end

```

11.4.2 Conservative TO Algorithm

We indicated in the preceding section that the basic TO algorithm never causes operations to wait, but instead, restarts them. We also pointed out that even though this is an advantage due to deadlock freedom, it is also a disadvantage, because numerous restarts would have adverse performance implications. The conservative TO algorithms attempt to lower this system overhead by reducing the number of transaction restarts.

Let us first present a technique that is commonly used to reduce the probability of restarts. Remember that a TO scheduler restarts a transaction if a younger conflicting transaction is already scheduled or has been executed. Note that such occurrences increase significantly if, for example, one site is comparatively inactive relative to the others and does not issue transactions for an extended period. In this case its timestamp counter indicates a value that is considerably smaller than the counters of other sites. If the TM at this site then receives a transaction, the operations that are

sent to the histories at the other sites will almost certainly be rejected, causing the transaction to restart. Furthermore, the same transaction will restart repeatedly until the timestamp counter value at its originating site reaches a level of parity with the counters of other sites.

The foregoing scenario indicates that it is useful to keep the counters at each site synchronized. However, total synchronization is not only costly—since it requires exchange of messages every time a counter changes—but also unnecessary. Instead, each transaction manager can send its remote operations, rather than histories, to the transaction managers at the other sites. The receiving transaction managers can then compare their own counter values with that of the incoming operation. Any manager whose counter value is smaller than the incoming one adjusts its own counter to one more than the incoming one. This ensures that none of the counters in the system run away or lag behind significantly. Of course, if system clocks are used instead of counters, this approximate synchronization may be achieved automatically as long as the clocks are of comparable speeds.

We can now return to our discussion of conservative TO algorithms. The “conservative” nature of these algorithms relates to the way they execute each operation. The basic TO algorithm tries to execute an operation as soon as it is accepted; it is therefore “aggressive” or “progressive.” Conservative algorithms, on the other hand, delay each operation until there is an assurance that no operation with a smaller timestamp can arrive at that scheduler. If this condition can be guaranteed, the scheduler will never reject an operation. However, this delay introduces the possibility of deadlocks.

The basic technique that is used in conservative TO algorithms is based on the following idea: the operations of each transaction are buffered until an ordering can be established so that rejections are not possible, and they are executed in that order. We will consider one possible implementation of the conservative TO algorithm due to [Herman and Verjus \[1979\]](#).

Assume that each scheduler maintains one queue for each transaction manager in the system. The scheduler at site i stores all the operations that it receives from the transaction manager at site j in queue Q_{ij} . Scheduler i has one such queue for each j . When an operation is received from a transaction manager, it is placed in its appropriate queue in increasing timestamp order. The histories at each site execute the operations from these queues in increasing timestamp order.

This scheme will reduce the number of restarts, but it will not guarantee that they will be eliminated completely. Consider the case where at site i the queue for site j (Q_{ij}) is empty. The scheduler at site i will choose an operation [say, $R(x)$] with the smallest timestamp and pass it on to the data processor. However, site j may have sent to i an operation [say, $W(x)$] with a smaller timestamp which may still be in transit in the network. When this operation reaches site i , it will be rejected since it violates the TO rule: it wants to access a data item that is currently being accessed (in an incompatible mode) by another operation with a higher timestamp.

It is possible to design an extremely conservative TO algorithm by insisting that the scheduler choose an operation to be sent to the data processor only if there is at least one operation in each queue. This guarantees that every operation that the scheduler receives in the future will have timestamps greater than or equal to

those currently in the queues. Of course, if a transaction manager does not have a transaction to process, it needs to send dummy messages periodically to every scheduler in the system, informing them that the operations that it will send in the future will have timestamps greater than that of the dummy message.

The careful reader will realize that the extremely conservative timestamp ordering scheduler actually executes transactions serially at each site. This is very restrictive. One method that has been employed to overcome this restriction is to group transactions into classes. Transaction classes are defined with respect to their read sets and write sets. It is therefore sufficient to determine the class that a transaction belongs to by comparing the transaction's read set and write set, respectively, with the read set and write set of each class. Thus the conservative TO algorithm can be modified so that instead of requiring the existence, at each site, of one queue for each transaction manager, it is only necessary to have one queue for each transaction class. Alternatively, one might mark each queue with the class to which it belongs. With either of these modifications, the conditions for sending an operation to the data processor are changed. It is no longer necessary to wait until there is at least one operation in each queue; it is sufficient to wait until there is at least one operation in each class to which the transaction belongs. This and other weaker conditions that reduce the waiting delay can be defined and are sufficient. A variant of this method is used in the SDD-1 prototype system [Bernstein et al., 1980b].

11.4.3 Multiversion TO Algorithm

Multiversion TO is another attempt at eliminating the restart overhead cost of transactions. Most of the work on multiversion TO has concentrated on centralized databases, so we present only a brief overview. However, we should indicate that multiversion TO algorithm would be a suitable concurrency control mechanism for DBMSs that are designed to support applications that inherently have a notion of versions of database objects (e.g., engineering databases and document databases).

In multiversion TO, the updates do not modify the database; each write operation creates a new version of that data item. Each version is marked by the timestamp of the transaction that creates it. Thus the multiversion TO algorithm trades storage space for time. In doing so, it processes each transaction on a state of the database that it would have seen if the transactions were executed serially in timestamp order.

The existence of versions is transparent to users who issue transactions simply by referring to data items, not to any specific version. The transaction manager assigns a timestamp to each transaction, which is also used to keep track of the timestamps of each version. The operations are processed by the histories as follows:

1. A $R_i(x)$ is translated into a read on one version of x . This is done by finding a version of x (say, x_v) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$. $R_i(x_v)$ is then sent to the data processor to read x_v . This case is depicted in

Figure 11.11a, which shows that R_i can read the version (x_v) that it would have read had it arrived in timestamp order.

2. A $W_i(x)$ is translated into $W_i(x_w)$ so that $ts(x_w) = ts(T_i)$ and sent to the data processor if and only if no other transaction with a timestamp greater than $ts(T_i)$ has read the value of a version of x (say, x_r) such that $ts(x_r) > ts(x_w)$. In other words, if the scheduler has already processed a $R_j(x_r)$ such that

$$ts(T_i) < ts(x_r) < ts(T_j)$$

then $W_i(x)$ is rejected. This case is depicted in Figure 11.11b, which shows that if W_i is accepted, it would create a version (x_w) that R_j should have read, but did not since the version was not available when R_j was executed – it, instead, read version x_k , which results in the wrong history.

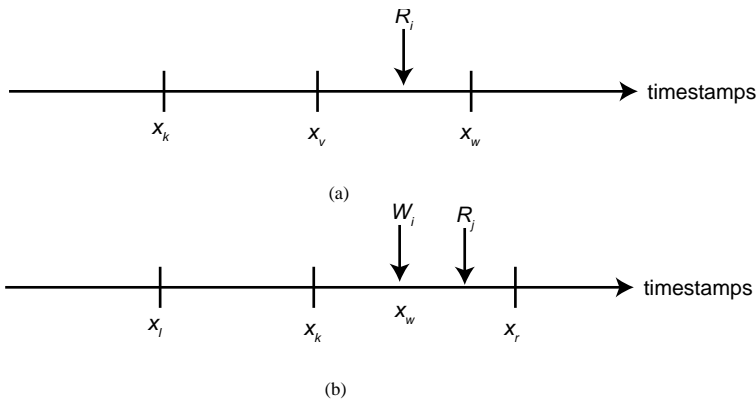


Fig. 11.11 Multiversion TO Cases

A scheduler that processes the read and the write requests of transactions according to the rules noted above is guaranteed to generate serializable histories. To save space, the versions of the database may be purged from time to time. This should be done when the distributed DBMS is certain that it will no longer receive a transaction that needs to access the purged versions.

11.5 Optimistic Concurrency Control Algorithms

The concurrency control algorithms discussed in Sections 11.3 and 11.4 are pessimistic in nature. In other words, they assume that the conflicts between transactions are quite frequent and do not permit a transaction to access a data item if there is a conflicting transaction that accesses that data item. Thus the execution of any

operation of a transaction follows the sequence of phases: validation (V), read (R), computation (C), write (W) (Figure 11.12).² Generally, this sequence is valid for an update transaction as well as for each of its operations.

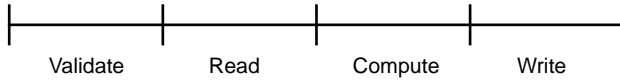


Fig. 11.12 Phases of Pessimistic Transaction Execution

Optimistic algorithms, on the other hand, delay the validation phase until just before the write phase (Figure 11.13). Thus an operation submitted to an optimistic scheduler is never delayed. The read, compute, and write operations of each transaction are processed freely without updating the actual database. Each transaction initially makes its updates on local copies of data items. The validation phase consists of checking if these updates would maintain the consistency of the database. If the answer is affirmative, the changes are made global (i.e., written into the actual database). Otherwise, the transaction is aborted and has to restart.

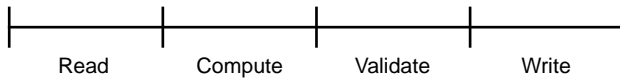


Fig. 11.13 Phases of Optimistic Transaction Execution

It is possible to design locking-based optimistic concurrency control algorithms (see [Bernstein et al., 1987]). However, the original optimistic proposals [Thomas, 1979; Kung and Robinson, 1981] are based on timestamp ordering. Therefore, we describe only the optimistic approach using timestamps.

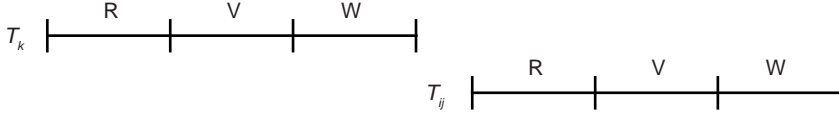
The algorithm that we discuss was proposed by Kung and Robinson [1981] and was later extended for distributed DBMS by Ceri and Owicki [1982]. This is not the only extension of the model to distributed databases, however (see, for example, [Sinha et al., 1985]). It differs from pessimistic TO-based algorithms not only by being optimistic but also in its assignment of timestamps. Timestamps are associated only with transactions, not with data items (i.e., there are no read or write timestamps). Furthermore, timestamps are not assigned to transactions at their initiation but at the beginning of their validation step. This is because the timestamps are needed only during the validation phase, and as we will see shortly, their early assignment may cause unnecessary transaction rejections.

Each transaction T_i is subdivided (by the transaction manager at the originating site) into a number of subtransactions, each of which can execute at many sites. Notationally, let us denote by T_{ij} a subtransaction of T_i that executes at site j . Until

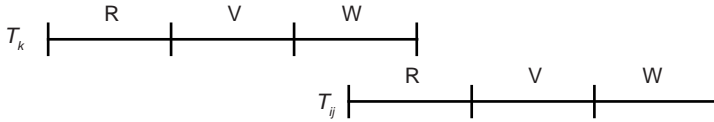
² We consider only the update transactions in this discussion because they are the ones that cause consistency problems. Read-only transactions do not have the computation and write phases. Furthermore, we assume that the write phase includes the commit action.

the validation phase, each local execution follows the sequence depicted in Figure 11.13. At that point a timestamp is assigned to the transaction which is copied to all its subtransactions. The local validation of T_{ij} is performed according to the following rules, which are mutually exclusive.

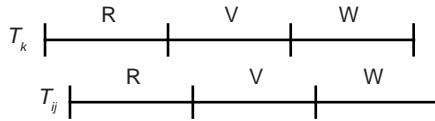
Rule 1. If all transactions T_k where $ts(T_k) < ts(T_{ij})$ have completed their write phase before T_{ij} has started its read phase (Figure 11.14a),³ validation succeeds, because transaction executions are in serial order.



(a)



(b)



(c)

Fig. 11.14 Possible Execution Scenarios

Rule 2. If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$, and which completes its write phase while T_{ij} is in its read phase (Figure 11.14b), the validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \emptyset$.

Rule 3. If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$, and which completes its read phase before T_{ij} completes its read phase (Figure 11.14c), the validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \emptyset$, and $WS(T_k) \cap WS(T_{ij}) = \emptyset$.

Rule 1 is obvious; it indicates that the transactions are actually executed serially in their timestamp order. Rule 2 ensures that none of the data items updated by T_k

³ Following the convention we have adopted, we omit the computation step in this figure and in the subsequent discussion. Thus timestamps are assigned at the end of the read phase.

are read by T_{ij} and that T_k finishes writing its updates into the database before T_{ij} starts writing. Thus the updates of T_{ij} will not be overwritten by the updates of T_k . Rule 3 is similar to Rule 2, but does not require that T_k finish writing before T_{ij} starts writing. It simply requires that the updates of T_k not affect the read phase or the write phase of T_{ij} .

Once a transaction is locally validated to ensure that the local database consistency is maintained, it also needs to be globally validated to ensure that the mutual consistency rule is obeyed. Unfortunately, there is no known optimistic method of doing this. A transaction is globally validated if all the transactions that precede it in the serialization order (at that site) terminate (either by committing or aborting). This is a pessimistic method since it performs global validation early and delays a transaction. However, it guarantees that transactions execute in the same order at each site.

An advantage of optimistic concurrency control algorithms is their potential to allow a higher level of concurrency. It has been shown that when transaction conflicts are very rare, the optimistic mechanism performs better than locking [Kung and Robinson, 1981]. A major problem with optimistic algorithms is the higher storage cost. To validate a transaction, the optimistic mechanism has to store the read and the write sets of several other terminated transactions. Specifically, the read and write sets of terminated transactions that were in progress when transaction T_{ij} arrived at site j need to be stored in order to validate T_{ij} . Obviously, this increases the storage cost.

Another problem is starvation. Consider a situation in which the validation phase of a long transaction fails. In subsequent trials it is still possible that the validation will fail repeatedly. Of course, it is possible to solve this problem by permitting the transaction exclusive access to the database after a specified number of trials. However, this reduces the level of concurrency to a single transaction. The exact “mix” of transactions that would cause an intolerable level of restarts is an issue that remains to be studied.

11.6 Deadlock Management

As we indicated before, any locking-based concurrency control algorithm may result in deadlocks, since there is mutual exclusion of access to shared resources (data) and transactions may wait on locks. Furthermore, we have seen that some TO-based algorithms that require the waiting of transactions (e.g., strict TO) may also cause deadlocks. Therefore, the distributed DBMS requires special procedures to handle them.

A deadlock can occur because transactions wait for one another. Informally, a deadlock situation is a set of requests that can never be granted by the concurrency control mechanism.

Example 11.9. Consider two transactions T_i and T_j that hold write locks on two entities x and y [i.e., $wl_i(x)$ and $wl_j(y)$]. Suppose that T_i now issues a $rl_i(y)$ or a $wl_i(y)$. Since y is currently locked by transaction T_j , T_i will have to wait until T_j

releases its write lock on y . However, if during this waiting period, T_j now requests a lock (read or write) on x , there will be a deadlock. This is because, T_i will be blocked waiting for T_j to release its lock on y while T_j will be waiting for T_i to release its lock on x . In this case, the two transactions T_i and T_j will wait indefinitely for each other to release their respective locks. ♦

A deadlock is a permanent phenomenon. If one exists in a system, it will not go away unless outside intervention takes place. This outside interference may come from the user, the system operator, or the software system (the operating system or the distributed DBMS).

A useful tool in analyzing deadlocks is a *wait-for graph* (WFG). A WFG is a directed graph that represents the wait-for relationship among transactions. The nodes of this graph represent the concurrent transactions in the system. An edge $T_i \rightarrow T_j$ exists in the WFG if transaction T_i is waiting for T_j to release a lock on some entity. Figure 11.15 depicts the WFG for Example 11.9.

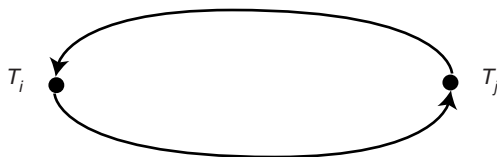


Fig. 11.15 A WFG Example

Using the WFG, it is easier to indicate the condition for the occurrence of a deadlock. A deadlock occurs when the WFG contains a cycle. We should indicate that the formation of the WFG is more complicated in distributed systems, since two transactions that participate in a deadlock condition may be running at different sites. We call this situation a *global deadlock*. In distributed systems, then, it is not sufficient that each local distributed DBMS form a *local wait-for graph* (LWFG) at each site; it is also necessary to form a *global wait-for graph* (GWFG), which is the union of all the LWFGs.

Example 11.10. Consider four transactions T_1, T_2, T_3 , and T_4 with the following wait-for relationship among them: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$. If T_1 and T_2 run at site 1 while T_3 and T_4 run at site 2, the LWFGs for the two sites are shown in Figure 11.16a. Notice that it is not possible to detect a deadlock simply by examining the two LWFGs, because the deadlock is global. The deadlock can easily be detected, however, by examining the GWFG where intersite waiting is shown by dashed lines (Figure 11.16b). ♦

There are three known methods for handling deadlocks: prevention, avoidance, and detection and resolution. In the remainder of this section we discuss each approach in more detail.

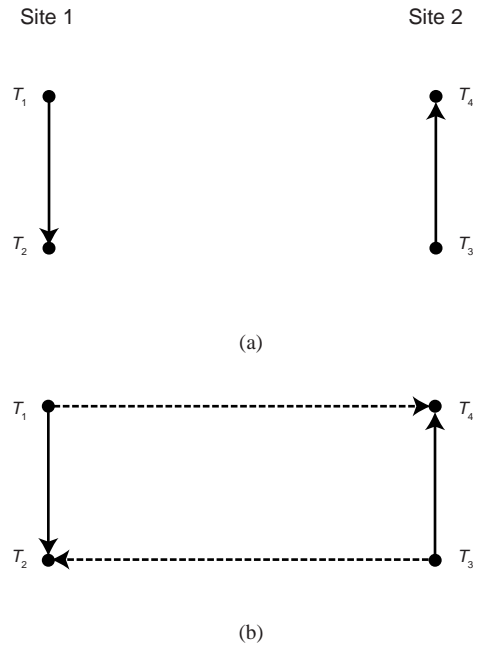


Fig. 11.16 Difference between LWFG and GWFG

11.6.1 Deadlock Prevention

Deadlock prevention methods guarantee that deadlocks cannot occur in the first place. Thus the transaction manager checks a transaction when it is first initiated and does not permit it to proceed if it may cause a deadlock. To perform this check, it is required that all of the data items that will be accessed by a transaction be predeclared. The transaction manager then permits a transaction to proceed if all the data items that it will access are available. Otherwise, the transaction is not permitted to proceed. The transaction manager reserves all the data items that are predeclared by a transaction that it allows to proceed.

Unfortunately, such systems are not very suitable for database environments. The fundamental problem is that it is usually difficult to know precisely which data items will be accessed by a transaction. Access to certain data items may depend on conditions that may not be resolved until run time. For example, in the reservation transaction that we developed in Example 10.3, access to CID and CNAME is conditional upon the availability of free seats. To be safe, the system would thus need to consider the maximum set of data items, even if they end up not being accessed. This would certainly reduce concurrency. Furthermore, there is additional overhead in evaluating whether a transaction can proceed safely. On the other hand, such systems require no run-time support, which reduces the overhead. It has the additional advantage that it is not necessary to abort and restart a transaction due to

deadlocks. This not only reduces the overhead but also makes such methods suitable for systems that have no provisions for undoing processes.⁴

11.6.2 Deadlock Avoidance

Deadlock avoidance schemes either employ concurrency control techniques that will never result in deadlocks or require that potential deadlock situations are detected in advance and steps are taken such that they will not occur. We consider both of these cases.

The simplest means of avoiding deadlocks is to order the resources and insist that each process request access to these resources in that order. This solution was long ago proposed for operating systems. A revised version has been proposed for database systems as well [Garcia-Molina, 1979]. Accordingly, the lock units in the distributed database are ordered and transactions always request locks in that order. This ordering of lock units may be done either globally or locally at each site. In the latter case, it is also necessary to order the sites and require that transactions which access data items at multiple sites request their locks by visiting the sites in the predefined order.

Another alternative is to make use of transaction timestamps to prioritize transactions and resolve deadlocks by aborting transactions with higher (or lower) priorities. To implement this type of prevention method, the lock manager is modified as follows. If a lock request of a transaction T_i is denied, the lock manager does not automatically force T_i to wait. Instead, it applies a prevention test to the requesting transaction and the transaction that currently holds the lock (say T_j). If the test is passed, T_i is permitted to wait for T_j ; otherwise, one transaction or the other is aborted.

Examples of this approach is the WAIT-DIE and WOUND-WAIT algorithms [Rosenkrantz et al., 1978], also used in the MADMAN DBMS [GE, 1976]. These algorithms are based on the assignment of timestamps to transactions. WAIT-DIE is a non-preemptive algorithm in that if the lock request of T_i is denied because the lock is held by T_j , it never preempts T_j , following the rule:

WAIT-DIE Rule. If T_i requests a lock on a data item that is already locked by T_j , T_i is permitted to wait if and only if T_i is older than T_j . If T_i is younger than T_j , then T_i is aborted and restarted with the same timestamp.

A preemptive version of the same idea is the WOUND-WAIT algorithm, which follows the rule:

⁴ This is not a significant advantage since most systems have to be able to undo transactions for reliability purposes, as we will see in Chapter 12.

WOUND-WAIT Rule. If T_i requests a lock on a data item that is already locked by T_j , then T_i is permitted to wait if only if it is younger than T_j ; otherwise, T_j is aborted and the lock is granted to T_i .

The rules are specified from the viewpoint of T_i : T_i waits, T_i dies, and T_i wounds T_j . In fact, the result of wounding and dying are the same: the affected transaction is aborted and restarted. With this perspective, the two rules can be specified as follows:

if $ts(T_i) < ts(T_j)$ then T_i waits else T_i dies (WAIT-DIE)
 if $ts(T_i) < ts(T_j)$ then T_j is wounded else T_i waits (WOUND-WAIT)

Notice that in both algorithms the younger transaction is aborted. The difference between the two algorithms is whether or not they preempt active transactions. Also note that the WAIT-DIE algorithm prefers younger transactions and kills older ones. Thus an older transaction tends to wait longer and longer as it gets older. By contrast, the WOUND-WAIT rule prefers the older transaction since it never waits for a younger one. One of these methods, or a combination, may be selected in implementing a deadlock prevention algorithm.

Deadlock avoidance methods are more suitable than prevention schemes for database environments. Their fundamental drawback is that they require run-time support for deadlock management, which adds to the run-time overhead of transaction execution.

11.6.3 Deadlock Detection and Resolution

Deadlock detection and resolution is the most popular and best-studied method. Detection is done by studying the GWFG for the formation of cycles. We will discuss means of doing this in considerable detail. Resolution of deadlocks is typically done by the selection of one or more *victim* transaction(s) that will be preempted and aborted in order to break the cycles in the GWFG. Under the assumption that the cost of preempting each member of a set of deadlocked transactions is known, the problem of selecting the minimum total-cost set for breaking the deadlock cycle has been shown to be a difficult (NP-complete) problem [Leung and Lai, 1979]. However, there are some factors that affect this choice [Bernstein et al., 1987]:

1. The amount of effort that has already been invested in the transaction. This effort will be lost if the transaction is aborted.
2. The cost of aborting the transaction. This cost generally depends on the number of updates that the transaction has already performed.
3. The amount of effort it will take to finish executing the transaction. The scheduler wants to avoid aborting a transaction that is almost finished. To do this, it must be able to predict the future behavior of active transactions (e.g., based on the transaction's type).

4. The number of cycles that contain the transaction. Since aborting a transaction breaks all cycles that contain it, it is best to abort transactions that are part of more than one cycle (if such transactions exist).

Now we can return to deadlock detection. There are three fundamental methods of detecting distributed deadlocks, referred as *centralized*, *distributed*, and *hierarchical deadlock detection*.

11.6.3.1 Centralized Deadlock Detection

In the centralized deadlock detection approach, one site is designated as the deadlock detector for the entire system. Periodically, each lock manager transmits its LWFG to the deadlock detector, which then forms the GWFG and looks for cycles in it. Actually, the lock managers need only send changes in their graphs (i.e., the newly created or deleted edges) to the deadlock detector. The length of intervals for transmitting this information is a system design decision: the smaller the interval, the smaller the delays due to undetected deadlocks, but the larger the communication cost.

Centralized deadlock detection has been proposed for distributed INGRES. This method is simple and would be a very natural choice if the concurrency control algorithm were centralized 2PL. However, the issues of vulnerability to failure, and high communication overhead, must also be considered.

11.6.3.2 Hierarchical Deadlock Detection

An alternative to centralized deadlock detection is the building of a hierarchy of deadlock detectors [Menasce and Muntz, 1979] (see Figure 11.17). Deadlocks that are local to a single site would be detected at that site using the LWFG. Each site also sends its LWFG to the deadlock detector at the next level. Thus, distributed deadlocks involving two or more sites would be detected by a deadlock detector in the next lowest level that has control over these sites. For example, a deadlock at site 1 would be detected by the local deadlock detector (*DD*) at site 1 (denoted DD_{21} , 2 for level 2, 1 for site 1). If, however, the deadlock involves sites 1 and 2, then DD_{11} detects it. Finally, if the deadlock involves sites 1 and 4, DD_{0x} detects it, where x is either one of 1, 2, 3, or 4.

The hierarchical deadlock detection method reduces the dependence on the central site, thus reducing the communication cost. It is, however, considerably more complicated to implement and would involve non-trivial modifications to the lock and transaction manager algorithms.

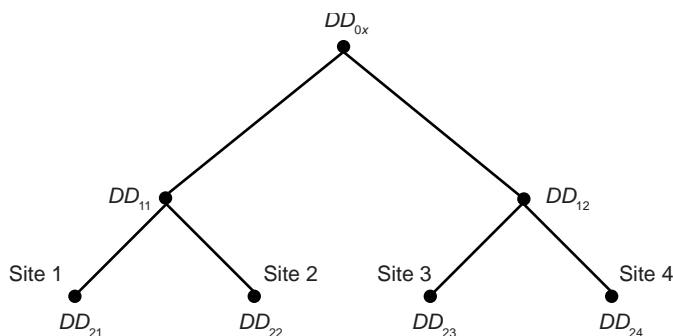


Fig. 11.17 Hierarchical Deadlock Detection

11.6.3.3 Distributed Deadlock Detection

Distributed deadlock detection algorithms delegate the responsibility of detecting deadlocks to individual sites. Thus, as in the hierarchical deadlock detection, there are local deadlock detectors at each site that communicate their LWFGs with one another (in fact, only the potential deadlock cycles are transmitted). Among the various distributed deadlock detection algorithms, the one implemented in System R* [Obermarck, 1982; Mohan et al., 1986] seems to be the more widely known and referenced. We therefore briefly outline that method, basing the discussion on [Obermarck, 1982].

The LWFG at each site is formed and is modified as follows:

1. Since each site receives the potential deadlock cycles from other sites, these edges are added to the LWFGs.
2. The edges in the LWFG which show that local transactions are waiting for transactions at other sites are joined with edges in the LWFGs which show that remote transactions are waiting for local ones.

Example 11.11. Consider the example depicted in Figure 11.16. The local WFG for the two sites are modified as shown in Figure 11.18. ♦

Local deadlock detectors look for two things. If there is a cycle that does not include the external edges, there is a local deadlock that can be handled locally. If, on the other hand, there is a cycle involving these external edges, there is a potential distributed deadlock and this cycle information has to be communicated to other deadlock detectors. In the case of Example 11.11, the possibility of such a distributed deadlock is detected by both sites.

A question that needs to be answered at this point is to whom to transmit the information. Obviously, it can be transmitted to all deadlock detectors in the system. In the absence of any more information, this is the only alternative, but it incurs a high overhead. If, however, one knows whether the transaction is ahead or behind in the deadlock cycle, the information can be transmitted forward or backward along

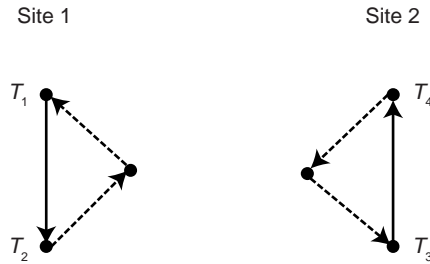


Fig. 11.18 Modified LWFGs

the sites in this cycle. The receiving site then modifies its LWFG as discussed above, and checks for deadlocks. Obviously, there is no need to transmit along the deadlock cycle in both the forward and backward directions. In the case of Example 11.11, site 1 would send it to site 2 in both forward and backward transmission along the deadlock cycle.

The distributed deadlock detection algorithms require uniform modification to the lock managers at each site. This uniformity makes them easier to implement. However, there is the potential for excessive message transmission. This happens, for example, in the case of Example 11.11: site 1 sends its potential deadlock information to site 2, and site 2 sends its information to site 1. In this case the deadlock detectors at both sites will detect the deadlock. Besides causing unnecessary message transmission, there is the additional problem that each site may choose a different victim to abort. Obermack's algorithm solves the problem by using transaction timestamps as well as the following rule. Let the path that has the potential of causing a distributed deadlock in the local WFG of a site be $T_i \rightarrow \dots \rightarrow T_j$. A local deadlock detector forwards the cycle information only if $ts(T_i) < ts(T_j)$. This reduces the average number of message transmissions by one-half. In the case of Example 11.11, site 1 has a path $T_1 \rightarrow T_2 \rightarrow T_3$, whereas site 2 has a path $T_3 \rightarrow T_4 \rightarrow T_1$. Therefore, assuming that the subscripts of each transaction denote their timestamp, only site 1 will send information to site 2.

11.7 “Relaxed” Concurrency Control

For most of this chapter, we focused only on distributed concurrency control algorithms that are designed for flat transactions and enforce serializability as the correctness criterion. This is the baseline case. There have been studies that (a) relax serializability in arguing for correctness of concurrent execution, and (b) consider other transaction models, primarily nested ones. We will briefly review these in this section.

11.7.1 Non-Serializable Histories

Serializability is a fairly simple and elegant concept which can be enforced with acceptable overhead. However, it is considered to be too “strict” for certain applications since it does not consider as correct certain histories that might be argued as reasonable. We have shown one case when we discussed the ordered shared lock concept. In addition, consider the Reservation transaction of Example 10.10. One can argue that the history generated by two concurrent executions of this transaction can be non-serializable, but correct – one may do the Airline reservation first and then do the Hotel reservation while the other one reverses the order – as long as both executions successfully terminate. The question, however, is how one can generalize these intuitive observations. The solution is to observe and exploit the “semantics” of these transactions.

There have been a number of proposals for exploiting transaction semantics. Of particular interest for distributed DBMS is one class that depends on identifying transaction *steps*, which may consist of a single operation or a set of operations, and establishing how transactions can interleave with each other between steps. Garcia-Molina [1983] classifies transactions into classes such that transactions in the same class are *compatible* and can interleave arbitrarily while transactions in different classes are incompatible and have to be synchronized. The synchronization is based on semantic notions, allowing more concurrency than serializability. The use of the concept of transaction classes can be traced back to SDD-1 [Bernstein et al., 1980b].

The concept of compatibility is refined by Lynch [1983b] and several levels of compatibility among transactions are defined. These levels are structured hierarchically so that interleavings at higher levels include those at lower levels. Furthermore, Lynch [1983b] introduces the concept of *breakpoints* within transactions, which represent points at which other transactions can interleave. This is an alternative to the use of compatibility sets.

Another work along these lines uses breakpoints to indicate the interleaving points, but does not require that the interleavings be hierarchical [Farrag and Özsu, 1989]. A transaction is modeled as consisting of a number of steps. Each step consists of a sequence of atomic operations and a breakpoint at the end of these operations. For each breakpoint in a transaction the set of transaction types that are allowed to interleave at that breakpoint is specified. A correctness criterion called *relative consistency* is defined based on the correct interleavings among transactions. Intuitively, a relatively consistent history is equivalent to a history that is stepwise serial (i.e., the operations and breakpoint of each step appear without interleaving), and in which a step (T_{ik}) of transaction T_i interleaves two consecutive steps (T_{jm} and $T_{j(m+1)}$) of transaction T_j only if transactions of T_i 's type are allowed to interleave T_{jm} at its breakpoint. It can be shown that some of the relatively consistent histories are not serializable, but are still “correct” [Farrag and Özsu, 1989].

A unifying framework that combines the approaches of Lynch [1983b] and Farrag and Özsu [1989] has been proposed by Agrawal et al. [1994]. A correctness criterion called *semantic relative atomicity* is introduced which provides finer interleavings and more concurrency.

The above mentioned relaxed correctness criteria have formal underpinnings similar to serializability, allowing their formal analysis. However, these have not been extended to distributed DBMS even though this possibility exists.

11.7.2 Nested Distributed Transactions

We introduced the nested transaction model in the previous chapter. The concurrent execution of nested transactions is interesting, especially since they are good candidates for distributed execution.

Let us first consider closed nested transactions [Moss, 1985]. The concurrency control of nested transactions have generally followed a locking-based approach. The following rules govern the management of the locks and the completion of transaction execution in the case of closed nested transactions:

1. Each subtransaction executes as a transaction and upon completion transfers its lock to its parent transaction.
2. A parent inherits both the locks and the updates of its committed subtransactions.
3. The inherited state will be visible only to descendants of the inheriting parent transaction. However, to access the state, a descendant must acquire appropriate locks. Lock conflicts are determined as for flat transactions, except that one ignores inherited locks retained by ancestor's of the requesting subtransaction.
4. If a subtransaction aborts, then all locks and updates that the subtransaction and its descendants are discarded. The parent of an aborted subtransaction need not, but may, choose to abort.

From the perspective of ACID properties, closed nested transactions relax durability since the effects of successfully completed subtransactions can be erased if an ancestor transaction aborts. They also relax the isolation property in a limited way since they share their state with other subtransactions within the same nested transaction.

The distributed execution potential of nested transactions is obvious. After all, nested transactions are meant to improve intra-transaction concurrency and one can view each subtransaction as a potential unit of distribution if data are also appropriately distributed.

However, from the perspective of lock management, some care has to be observed. When subtransactions release their locks to their parents, these lock releases cannot be reflected in the lock tables automatically. The subtransaction commit commands do not have the same semantics as flat transactions.

Open nested transactions are even more relaxed than their closed nested counterparts. They have been called “anarchic” forms of nested transactions [Gray and Reuter, 1993]. The open nested transaction model is best exemplified in the saga

model [Garcia-Molina and Salem, 1987; Garcia-Molina et al., 1990] which was discussed in Section 10.3.2.

From the perspective of lock management, open nested transactions are easy to deal with. The locks held by a subtransaction are released as soon as it commits or aborts and this is reflected in the lock tables.

A variant of open nested transactions with precise and formal semantics is the *multilevel transaction* model [Weikum, 1986; Weikum and Schek, 1984; Beeri et al., 1988; Weikum, 1991]. Multilevel transactions “are a variant of open nested transactions in which the subtransactions correspond to operations at different levels of a layered system architecture” [Weikum and Hasse, 1993]. We introduce the concept with an example taken from [Weikum, 1991]. We consider a transaction specification language which allows users to write transactions involving abstract operations so as to be able to exploit application semantics.

Consider two transactions that transfer funds from one bank account to another:

$$\begin{array}{ll} T_1: & \text{Withdraw}(o, x) \\ & \text{Deposit}(p, x) \\ T_2: & \text{Withdraw}(o, y) \\ & \text{Deposit}(p, y) \end{array}$$

The notation here is that each T_i withdraws x (y) amount from account o and deposits that amount to account p . The semantics of Withdraw is test-and-withdraw to ensure that the account balance is sufficient to meet the withdrawal request. In relational systems, each of these abstract operations will be translated to tuple operations Select (*Sel*), and Update (*Upd*) which will, in turn, be translated into page-level Read and Write operations (assuming o is on page r and p is on page w). This results in a layered abstraction of transaction execution as depicted in Figure 11.19.

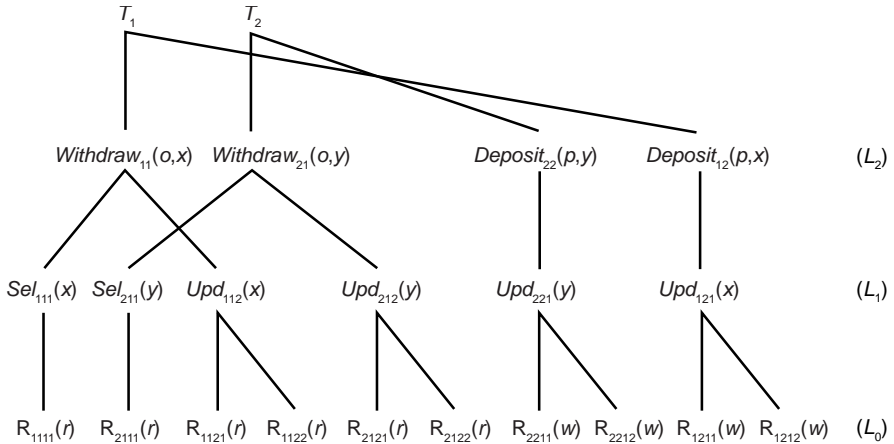


Fig. 11.19 Multilevel Transaction Example (Based on [Weikum, 1991])

The traditional method of dealing with these types of histories is to develop a scheduler that enforces serializability at the lowest level (L_0). This, however, reduces

the level of concurrency since it does not take into account application semantics and the granularity of synchronization is too coarse. Abstracting from the lower-level details can provide higher concurrency. For example, the page-level history (L_0) in Figure 11.19 is not serializable with respect to transactions T_1 and T_2 , but the tuple-level history L_1 is serializable ($T_2 \rightarrow T_1$). When one goes up to level L_2 , it is possible to make use of the semantics of the abstract operations (i.e., their commutativity) to provide even more concurrency. Therefore, *multilevel serializability* is defined to reason about the serializability of multilevel histories and *multilevel histories* are proposed to enforce it [Weikum, 1991].

11.8 Conclusion

In this chapter we discussed distributed currency control algorithms that provide the isolation and consistency properties of transactions. The distributed concurrency control mechanism of a distributed DBMS ensures that the consistency of the distributed database is maintained and is therefore one of the fundamental components of a distributed DBMS. This is evidenced by the significant amount of research that has been conducted in this area.

Our discussion in this chapter assumed that both the hardware and the software components of the computer systems were totally reliable. Even though this assumption is completely unrealistic, it has served a didactic purpose. It has permitted us to focus only on the concurrency control aspects, leaving to another chapter the features that need to be added to a distributed DBMS to make it reliable in an unreliable environment. We have also assumed a non-replicated distributed database, leaving replication issues to Chapter 13.

There are a few issues that we have omitted from this chapter. We mention them here for the benefit of the interested reader.

1. *Performance evaluation of concurrency control algorithms.* We have not explicitly included performance analysis results or methodologies. This may be somewhat surprising given the significant number of papers that have appeared in the literature. However, the reasons for this omission are numerous. First, there is no comprehensive and definitive performance study of concurrency control algorithms. The performance studies have developed rather haphazardly and for specific purposes. Therefore, each has made different assumptions and has measured different parameters. Although these have identified a number of important performance tradeoffs, it is quite difficult, if not impossible, to make meaningful generalizations that extend beyond the obvious. Second, the analytical methods for conducting these performance analysis studies have not been developed sufficiently.

The relative performance characteristics of distributed concurrency methods is less understood than their centralized counterparts [Thomasian, 1996]. The main reason for this is the complexity of these algorithms. This complexity

has resulted in a number of simplifying assumptions such as a fully replicated database, fully interconnected network, network delays represented by simplistic queueing models (M/M/1), etc. [Thomasian, 1996].

2. *Other concurrency control methods.* There is another class of concurrency control algorithms, called “serializability graph testing methods,” which we have not mentioned in this chapter. Such mechanisms work by explicitly building a *dependency* (or *serializability*) *graph* and checking it for cycles. The dependency (serializability) graph of a history H , denoted $DG(H)$, is a directed graph representing the conflict relations among the transactions in H . The nodes of this graph are the set of transactions in H [i.e., each transaction T_i in H is represented by a node in $DG(H)$]. An edge (T_i, T_j) exists in $DG(H)$ if and only if there is an operation in T_i that conflicts with and precedes another operation in T_j .

Schedulers update their dependency graphs whenever one of the following conditions is fulfilled: (1) a new transaction starts in the system, (2) a read or a write operation is received by the scheduler, (3) a transaction terminates, or (4) a transaction aborts.

It is now possible to talk about “correct” concurrency control algorithms based on the dependency graph. Given a history H , if its dependency graph $DG(H)$ is acyclic, then H is serializable. In the distributed case we may use a global dependency graph, which can be formed by taking the union of the local dependency graphs and further annotating each transaction by the identifier of the site where it is executed. It is then necessary to show that the global dependency graph is acyclic.

Example 11.12. The dependency graph of history H_1 discussed in Example 11.6 is given in Figure 11.20. Since the graph is acyclic, H_1 is serializable. ♦

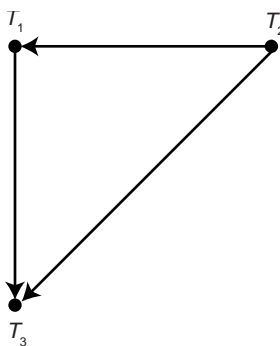


Fig. 11.20 Dependency Graph

3. *Assumptions about transactions.* In our discussions, we did not make any distinction between read-only transactions and update transactions. It is pos-

sible to improve significantly the performance of transactions that only read data items, or of systems with a high ratio of read-only transactions to update transactions. These issues are beyond the scope of this book.

We have also treated read and write locks in an identical fashion. It is possible to differentiate between them and develop concurrency control algorithms that permit “lock conversion,” whereby transactions can obtain locks in one mode and then modify their lock modes as they change their requirements. Typically, the conversion is from read locks to write locks.

4. *More “general” algorithms.* There are some indications which suggest that it should be possible to study the two fundamental concurrency control primitives (i.e., locking and timestamp ordering) using a unifying framework. Three major indications are worth mentioning. First, it is possible to develop both pessimistic and optimistic algorithms based on either one of the primitives. Second, a strict TO algorithm performs similarly to a locking algorithm, since it delays the acceptance of a transaction until all older ones are terminated. This does not mean that all histories which can be generated by a strict TO scheduler would be permitted by a 2PL scheduler. However, this similarity is interesting. Finally, it is possible to develop hybrid algorithms that use both timestamp ordering and locking. Furthermore, it is possible to state precisely rules for their interaction.

One study [Farrag and Özsu, 1985, 1987] has resulted in the development of a theoretical framework for the uniform treatment of both of these primitives. Based on this theoretical foundation, it was shown that 2PL and TO algorithms are two endpoints of a range of algorithms that can be generated by a more general concurrency control algorithm. This study, which is only for centralized database systems, is significant not only because it indicates that locking and timestamp ordering are related, but also because it would be interesting to study the nature and characteristics of the algorithms that lie between these two endpoints. In addition, such a uniform framework may be helpful in conducting comprehensive and internally consistent performance studies.

5. *Transaction execution models.* The algorithms that we have described all assume a computational model where the transaction manager at the originating site of a transaction coordinates the execution of each database operation of that transaction. This is called *centralized execution* [Carey and Livny, 1988]. It is also possible to consider a *distributed execution* model where a transaction is decomposed into a set of subtransactions each of which is allocated to one site where the transaction manager coordinates its execution. This is intuitively more attractive because it may permit load balancing across the multiple sites of a distributed database. However, the performance studies indicate that distributed computation performs better only under light load.

11.9 Bibliographic Notes

As indicated earlier, distributed concurrency control has been a very popular area of study. [Bernstein and Goodman, 1981] is a comprehensive study of the fundamental primitives which also lays down the rules for building hybrid algorithms. The issues that are addressed in this chapter are discussed in much more detail in [Cellary et al., 1988; Bernstein et al., 1987; Papadimitriou, 1986] and [Gray and Reuter, 1993].

Nested transaction models and their specific concurrency control algorithms have been the subjects of some study. Specific results can be found in [Moss, 1985; Lynch, 1983a; Lynch and Merritt, 1986; Fekete et al., 1987a,b; Goldman, 1987; Beeri et al., 1989; Fekete et al., 1989] and more recently in [Lynch et al., 1993].

The work on transaction management with semantic knowledge is presented in [Lynch, 1983b; Garcia-Molina, 1983], and [Farrag and Özsu, 1989]. The processing of read-only transactions is discussed in [Garcia-Molina and Wiederhold, 1982]. Transaction groups [Skarra et al., 1986; Skarra, 1989] also exploit a correctness criterion called *semantic patterns* that is more relaxed than serializability. Furthermore, work on the ARIES system [Haderle et al., 1992] is also within this class of algorithms. In particular, [Rothermel and Mohan, 1989] discusses ARIES within the context of nested transactions. Epsilon serializability [Ramamritham and Pu, 1995; Wu et al., 1997] and NT/PV model [Kshemkalyani and Singhal, 1994] are other “relaxed” correctness criteria. An algorithm based on ordering transactions using *serialization numbers* is discussed in [Halici and Dogac, 1989].

There are a number of papers that discuss the results of performance evaluation studies on distributed concurrency control algorithms. These include [Gelenbe and Sevcik, 1978; Garcia-Molina, 1979; Potier and LeBlanc, 1980; Menasce and Nakanishi, 1982a,b; Lin, 1981; Lin and Nolte, 1982, 1983; Goodman et al., 1983; Sevcik, 1983; Carey and Stonebraker, 1984; Merrett and Rallis, 1985; Özsu, 1985b,a; Koon and Özsu, 1986; Tsuchiya et al., 1986; Li, 1987; Agrawal et al., 1987; Bhide, 1988; Carey and Livny, 1988], and [Carey and Livny, 1991]. [Liang and Tripathi, 1996] studies the performance of sagas and Thomasian has conducted a series of performance studies that focus on various aspects of transaction processing in centralized and distributed DBMSs [Thomasian, 1993, 1998; Yu et al., 1989]. [Kumar, 1996] focuses on the performance of centralized DBMSs; the performance of distributed concurrency control methods are discussed in [Thomasian, 1996] and [Cellary et al., 1988]. An early but comprehensive review of deadlock management is [Isloor and Marsland, 1980]. Most of the work on distributed deadlock management has been on detection and resolution (see, e.g., [Obermarck, 1982; Elmagarmid et al., 1988]). Two surveys of the important algorithms are included in [Elmagarmid, 1986] and [Knapp, 1987]. A more recent survey is [Singhal, 1989]. There are two annotated bibliographies on the deadlock problem which do not emphasize the database issues but consider the problem in general: [Newton, 1979; Zobel, 1983]. The research activity on this topic has slowed down in the last years. Some of the recent relevant papers are [Yeung and Hung, 1995; Hofri, 1994; Lee and Kim, 1995; Kshemkalyani and Singhal, 1994; Chen et al., 1996; Park et al., 1995] and [Makki and Pissinou, 1995].

Exercises

Problem 11.1. Which of the following histories are conflict equivalent?

$$H_1 = \{W_2(x), W_1(x), R_3(x), R_1(x), W_2(y), R_3(y), R_3(z), R_2(x)\}$$

$$H_2 = \{R_3(z), R_3(y), W_2(y), R_2(z), W_1(x), R_3(x), W_2(x), R_1(x)\}$$

$$H_3 = \{R_3(z), W_2(x), W_2(y), R_1(x), R_3(x), R_2(z), R_3(y), W_1(x)\}$$

$$H_4 = \{R_2(z), W_2(x), W_2(y), W_1(x), R_1(x), R_3(x), R_3(z), R_3(y)\}$$

Problem 11.2. Which of the above histories $H_1 - H_4$ are serializable?

Problem 11.3. Give a history of two complete transactions which is not allowed by a strict 2PL scheduler but is accepted by the basic 2PL scheduler.

Problem 11.4 (*). One says that history H is *recoverable* if, whenever transaction T_i reads (some item x) from transaction T_j ($i \neq j$) in H and C_i occurs in H , then $C_j \prec_S C_i$. T_i “reads x from” T_j in H if

1. $W_j(x) \prec_H R_i(x)$, and
2. $A_j \text{not} \prec_H R_i(x)$, and
3. if there is some $W_k(x)$ such that $W_j(x) \prec_H W_k(x) \prec_H R_i(x)$, then $A_k \prec_H R_i(x)$.

Which of the following histories are recoverable?

$$H_1 = \{W_2(x), W_1(x), R_3(x), R_1(x), C_1, W_2(y), R_3(y), R_3(z), C_3, R_2(x), C_2\}$$

$$H_2 = \{R_3(z), R_3(y), W_2(y), R_2(z), W_1(x), R_3(x), W_2(x), R_1(x), C_1, C_2, C_3\}$$

$$H_3 = \{R_3(z), W_2(x), W_2(y), R_1(x), R_3(x), R_2(z), R_3(y), C_3, W_1(x), C_2, C_1\}$$

$$H_4 = \{R_2(z), W_2(x), W_2(y), C_2, W_1(x), R_1(x), A_1, R_3(x), R_3(z), R_3(y), C_3\}$$

Problem 11.5 (*). Give the algorithms for the transaction managers and the lock managers for the distributed two-phase locking approach.

Problem 11.6 ().** Modify the centralized 2PL algorithm to handle phantoms. (See Chapter 10 for a definition of phantoms.)

Problem 11.7. Timestamp ordering-based concurrency control algorithms depend on either an accurate clock at each site or a global clock that all sites can access (the clock can be a counter). Assume that each site has its own clock which “ticks” every 0.1 second. If all local clocks are resynchronized every 24 hours, what is the maximum drift in seconds per 24 hours permissible at any local site to ensure that a timestamp-based mechanism will successfully synchronize transactions?

Problem 11.8 ().** Incorporate the distributed deadlock strategy described in this chapter into the distributed 2PL algorithms that you designed in Problem 11.5.

Problem 11.9. Explain the relationship between transaction manager storage requirement and transaction size (number of operations per transaction) for a transaction manager using an optimistic timestamp ordering for concurrency control.

Problem 11.10 (*). Give the scheduler and transaction manager algorithms for the distributed optimistic concurrency controller described in this chapter.

Problem 11.11. Recall from the discussion in Section 11.7 that the computational model that is used in our descriptions in this chapter is a centralized one. How would the distributed 2PL transaction manager and lock manager algorithms change if a distributed execution model were to be used?

Problem 11.12. It is sometimes claimed that serializability is quite a restrictive correctness criterion. Can you give examples of distributed histories that are correct (i.e., maintain the consistency of the local databases as well as their mutual consistency) but are not serializable?

Chapter 12

Distributed DBMS Reliability

We have referred to “reliability” and “availability” of the database a number of times so far without defining these terms precisely. Specifically, we mentioned these terms in conjunction with data replication, because the principle method of building a reliable system is to provide redundancy in system components. We also claimed in Chapter 1 that the distribution of data enhances system reliability. However, the distribution of the database or the replication of data items is not sufficient to make the distributed DBMS reliable. A number of protocols need to be implemented within the DBMS to exploit this distribution and replication in order to make operations more reliable.

A reliable distributed database management system is one that can continue to process user requests even when the underlying system is unreliable. In other words, even when components of the distributed computing environment fail, a reliable distributed DBMS should be able to continue executing user requests without violating database consistency.

The purpose of this chapter is to discuss the reliability features of a distributed DBMS. From Chapter 10 the reader will recall that the reliability of a distributed DBMS refers to the atomicity and durability properties of transactions. Two specific aspects of reliability protocols that need to be discussed in relation to these properties are the commit and the recovery protocols. In that sense, in this chapter we relax one of the major assumptions of Chapter 11: that the underlying distributed system is fully reliable and does not experience any hardware or software failures. Furthermore, the commit protocols discussed in this chapter constitute the support provided by the distributed DBMS for the execution of commit commands in transactions.

The organization of this chapter is as follows. We start with a definition of the fundamental reliability concepts and reliability measures in Section 12. In Section 12.2 we discuss the reasons for failures in distributed systems and focus on the types of failures in distributed DBMSs. Section 12.3 focuses on the functions of the local recovery manager and provides an overview of reliability measures in centralized DBMS. This discussion forms the foundation for the distributed commit and recovery protocols, which are introduced in Section 12.4. In Sections 12.5 and 12.6 we present detailed protocols for dealing with site failures and network partitioning, respectively.

Implementation of these protocols within our architectural model is the topic of Section 12.7.

12.1 Reliability Concepts and Measures

Too often, the terms *reliability* and *availability* are used loosely in literature. Even among the researchers in the area of reliable computer systems, the definitions of these terms sometimes vary. In this section, we give precise definitions of a number of concepts that are fundamental to an understanding and study of reliable systems. Our definitions follow those of Anderson and Lee [1985] and Randell et al. [1978]. Nevertheless, we indicate where these definitions might differ from other usage of the terms.

12.1.1 System, State, and Failure

Reliability refers to a *system* that consists of a set of *components*. The system has a *state*, which changes as the system operates. The behavior of the system in providing response to all the possible external stimuli is laid out in an authoritative *specification* of its behavior. The specification indicates the valid behavior of each system state.

Any deviation of a system from the behavior described in the specification is considered a *failure*. For example, in a distributed transaction manager the specification may state that only serializable schedules for the execution of concurrent transactions should be generated. If the transaction manager generates a non-serializable schedule, we say that it has failed.

Each failure obviously needs to be traced back to its cause. Failures in a system can be attributed to deficiencies either in the components that make it up, or in the design, that is, how these components are put together. Each state that a reliable system goes through is valid in the sense that the state fully meets its specification. However, in an unreliable system, it is possible that the system may get to an internal state that may not obey its specification. Further transitions from this state would eventually cause a system failure. Such internal states are called *erroneous states*; the part of the state that is incorrect is called an *error* in the system. Any error in the internal states of the components of a system or in the design of a system is called a *fault* in the system. Thus, a fault causes an error that results in a system failure (Figure 12.1).

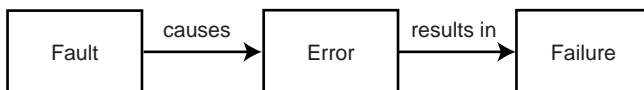


Fig. 12.1 Chain of Events Leading to System Failure

We differentiate between errors (or faults and failures) that are permanent and those that are not permanent. Permanence can apply to a failure, a fault, or an error, although we typically use the term with respect to faults. A *permanent fault*, also commonly called a *hard fault*, is one that reflects an irreversible change in the behavior of the system. Permanent faults cause permanent errors that result in permanent failures. The characteristics of these failures is that recovery from them requires intervention to “repair” the fault. Systems also experience *intermittent* and *transient faults*. In the literature, these two are typically not differentiated; they are jointly called *soft faults*. The dividing line in this differentiation is the reparability of the system that has experienced the fault [Siewiorek and Swarz, 1982]. An intermittent fault refers to a fault that demonstrates itself occasionally due to unstable hardware or varying hardware or software states. A typical example is the faults that systems may demonstrate when the load becomes too heavy. On the other hand, a transient fault describes a fault that results from temporary environmental conditions. A transient fault might occur, for example, due to a sudden increase in the room temperature. The transient fault is therefore the result of environmental conditions that may be impossible to repair. An intermittent fault, on the other hand, can be repaired since the fault can be traced to a component of the system.

Remember that we have also indicated that system failures can be due to design faults. Design faults together with unstable hardware cause intermittent errors that result in system failure. A final source of system failure that may not be attributable to a component fault or a design fault is operator mistakes. These are the sources of a significant number of errors as the statistics included further in this section demonstrate. The relationship between various types of faults and failures is depicted in Figure 12.2.

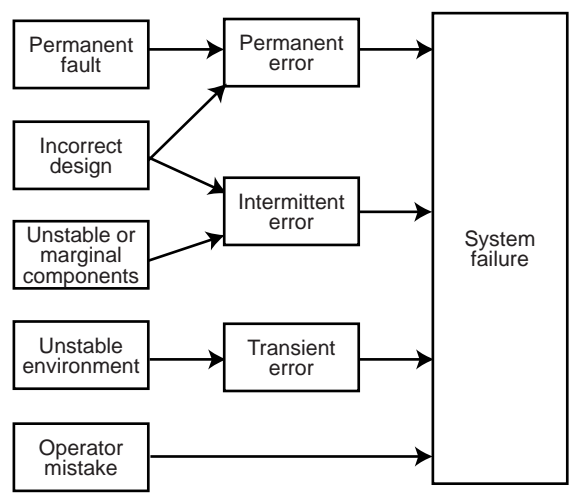


Fig. 12.2 Sources of System Failure (Based on [Siewiorek and Swarz, 1982])

12.1.2 Reliability and Availability

Reliability refers to the probability that the system under consideration does not experience any failures in a given time interval. It is typically used to describe systems that cannot be repaired (as in space-based computers), or where the operation of the system is so critical that no downtime for repair can be tolerated.

Formally, the reliability of a system, $R(t)$, is defined as the following conditional probability:

$$R(t) = \Pr\{0 \text{ failures in time } [0, t] \mid \text{no failures at } t = 0\}$$

If we assume that failures follow a Poisson distribution (which is usually the case for hardware), this formula reduces to

$$R(t) = \Pr\{0 \text{ failures in time } [0, t]\}$$

Under the same assumptions, it is possible to derive that

$$\Pr\{k \text{ failures in time } [0, t]\} = \frac{e^{-m(t)} [m(t)]^k}{k!}$$

where $m(t) = \int_0^t z(x) dx$. Here $z(t)$ is known as the *hazard function*, which gives the time-dependent failure rate of the specific hardware component under consideration. The probability distribution for $z(t)$ may be different for different electronic components.

The expected (mean) number of failures in time $[0, t]$ can then be computed as

$$E[k] = \sum_{k=0}^{\infty} k \frac{e^{-m(t)} [m(t)]^k}{k!} = m(t)$$

and the variance as

$$\text{Var}[k] = E[k^2] - (E[k])^2 = m(t)$$

Given these values, $R(t)$ can be written as

$$R(t) = e^{-m(t)}$$

Note that the reliability equation above is written for one component of the system. For a system that consists of n non-redundant components (i.e., they all have to function properly for the system to work) whose failures are independent, the overall system reliability can be written as

$$R_{\text{sys}}(t) = \prod_{i=1}^n R_i(t)$$

Availability, $A(t)$, refers to the probability that the system is operational according to its specification at a given point in time t . A number of failures may have occurred

prior to time t , but if they have all been repaired, the system is available at time t . Obviously, availability refers to systems that can be repaired.

If one looks at the limit of availability as time goes to infinity, it refers to the expected percentage of time that the system under consideration is available to perform useful computations. Availability can be used as some measure of “goodness” for those systems that can be repaired and which can be out of service for short periods of time during repair. Reliability and availability of a system are considered to be contradictory objectives [Siewiorek and Swarz, 1982]. It is usually accepted that it is easier to develop highly available systems as opposed to highly reliable systems.

If we assume that failures follow a Poisson distribution with a failure rate λ , and that repair time is exponential with a mean repair time of $1/\mu$, the steady-state availability of a system can be written as

$$A = \frac{\mu}{\lambda + \mu}$$

12.1.3 Mean Time between Failures/Mean Time to Repair

Two single-parameter measures have become more popular than the reliability and availability functions given above to model the behavior of systems. These two measures used are *mean time between failures* (MTBF) and *mean time to repair* (MTTR). MTBF is the expected time between subsequent failures in a system with repair.¹ MTBF can be calculated either from empirical data or from the reliability function as

$$\text{MTBF} = \int_0^{\infty} R(t) dt$$

Since $R(t)$ is related to the system failure rate, there is a direct relationship between MTBF and the failure rate of a system. MTTR is the expected time to repair a failed system. It is related to the repair rate as MTBF is related to the failure rate. Using these two metrics, the steady-state availability of a system with exponential failure and repair rates can be specified as

$$A = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

System failures may be *latent*, in that a failure is typically detected some time after its occurrence. This period is called *error latency*, and the average error latency time over a number of identical systems is called *mean time to detect* (MTTD).

¹ A distinction is sometimes made between MTBF and MTTF (mean time to fail). MTTF is defined as the expected time of the first system failure given a successful startup at time 0. MTBF is then defined only for systems that can be repaired. An approximation for MTBF is given as $\text{MTBF} = \text{MTTF} + \text{MTTR}$ [McConnel and Siewiorek, 1982]. We do not make this distinction in this book.

Figure 12.3 depicts the relationship of various reliability measures with the actual occurrences of faults.

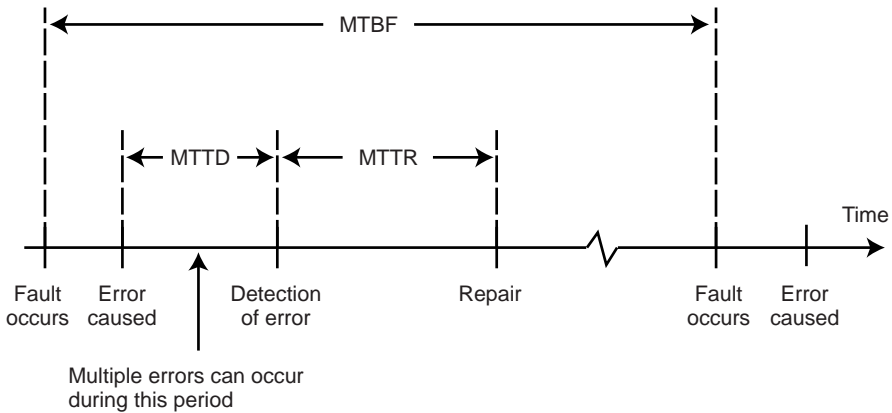


Fig. 12.3 Occurrence of Events over Time

12.2 Failures in Distributed DBMS

Designing a reliable system that can recover from failures requires identifying the types of failures with which the system has to deal. In a distributed database system, we need to deal with four types of failures: transaction failures (aborts), site (system) failures, media (disk) failures, and communication line failures. Some of these are due to hardware and others are due to software. The ratio of hardware failures vary from study to study and range from 18% to over 50%. Soft failures make up more than 90% of all hardware system failures. It is interesting to note that this percentage has not changed significantly since the early days of computing. A 1967 study of the U.S. Air Force indicates that 80% of electronic failures in computers are intermittent [Roth et al., 1967]. A study performed by IBM during the same year concludes that over 90% of all failures are intermittent [Ball and Hardie, 1967]. More recent studies indicate that the occurrence of soft failures is significantly higher than that of hard failures ([Longbottom, 1980; Gray, 1987]). Gray [1987] also mentions that most of the software failures are transient—and therefore soft—suggesting that a dump and restart may be sufficient to recover without any need to “repair” the software.

Software failures are typically caused by “bugs” in the code. The estimates for the number of bugs in software vary considerably. Figures such as 0.25 bug per 1000 instructions to 10 bugs per 1000 instructions have been reported. As stated before, most of the software failures are soft failures. The statistics for software failures are comparable to those we have previously reported on hardware failures. The fundamental reason for the dominance of soft failures in software is the significant

amount of design review and code inspection that a typical software project goes through before it gets to the testing stage. Furthermore, most commercial software goes through extensive alpha and beta testing before being released for field use.

12.2.1 Transaction Failures

Transactions can fail for a number of reasons. Failure can be due to an error in the transaction caused by incorrect input data (e.g., Example 10.3) as well as the detection of a present or potential deadlock. Furthermore, some concurrency control algorithms do not permit a transaction to proceed or even to wait if the data that they attempt to access are currently being accessed by another transaction. This might also be considered a failure. The usual approach to take in cases of transaction failure is to *abort* the transaction, thus resetting the database to its state prior to the start of this transaction.²

The frequency of transaction failures is not easy to measure. An early study reported that in System R, 3% of the transactions aborted abnormally [Gray et al., 1981]. In general, it can be stated that (1) within a single application, the ratio of transactions that abort themselves is rather constant, being a function of the incorrect data, the available semantic data control features, and so on; and (2) the number of transaction aborts by the DBMS due to concurrency control considerations (mainly deadlocks) is dependent on the level of concurrency (i.e., number of concurrent transactions), the interference of the concurrent applications, the granularity of locks, and so on [Härder and Reuter, 1983].

12.2.2 Site (System) Failures

The reasons for system failure can be traced back to a hardware or to a software failure. The important point from the perspective of this discussion is that a system failure is always assumed to result in the loss of main memory contents. Therefore, any part of the database that was in main memory buffers is lost as a result of a system failure. However, the database that is stored in secondary storage is assumed to be safe and correct. In distributed database terminology, system failures are typically referred to as *site failures*, since they result in the failed site being unreachable from other sites in the distributed system.

We typically differentiate between partial and total failures in a distributed system. *Total failure* refers to the simultaneous failure of all sites in the distributed system; *partial failure* indicates the failure of only some sites while the others remain operational. As indicated in Chapter 1, it is this aspect of distributed systems that makes them more available.

² Recall that all transaction aborts are not due to failures; in some cases, application logic requires transaction aborts as in Example 10.3.

12.2.3 Media Failures

Media failure refers to the failures of the secondary storage devices that store the database. Such failures may be due to operating system errors, as well as to hardware faults such as head crashes or controller failures. The important point from the perspective of DBMS reliability is that all or part of the database that is on the secondary storage is considered to be destroyed and inaccessible. Duplexing of disk storage and maintaining archival copies of the database are common techniques that deal with this sort of catastrophic problem.

Media failures are frequently treated as problems local to one site and therefore not specifically addressed in the reliability mechanisms of distributed DBMSs. We consider techniques for dealing with them in Section 12.3.5 under local recovery management. We then turn our attention to site failures when we consider distributed recovery functions.

12.2.4 Communication Failures

The three types of failures described above are common to both centralized and distributed DBMSs. Communication failures, however, are unique to the distributed case. There are a number of types of communication failures. The most common ones are the errors in the messages, improperly ordered messages, lost (or undeliverable) messages, and communication line failures. As discussed in Chapter 2, the first two errors are the responsibility of the computer network; we will not consider them further. Therefore, in our discussions of distributed DBMS reliability, we expect the underlying computer network hardware and software to ensure that two messages sent from a process at some originating site to another process at some destination site are delivered without error and in the order in which they were sent.

Lost or undeliverable messages are typically the consequence of communication line failures or (destination) site failures. If a communication line fails, in addition to losing the message(s) in transit, it may also divide the network into two or more disjoint groups. This is called *network partitioning*. If the network is partitioned, the sites in each partition may continue to operate. In this case, executing transactions that access data stored in multiple partitions becomes a major issue.

Network partitions point to a unique aspect of failures in distributed computer systems. In centralized systems the system state can be characterized as all-or-nothing: either the system is operational or it is not. Thus the failures are complete: when one occurs, the entire system becomes non-operational. Obviously, this is not true in distributed systems. As we indicated a number of times before, this is their potential strength. However, it also makes the transaction management algorithms more difficult to design.

If messages cannot be delivered, we will assume that the network does nothing about it. It will not buffer it for delivery to the destination when the service is reestablished and will not inform the sender process that the message cannot be

delivered. In short, the message will simply be lost. We make this assumption because it represents the least expectation from the network and places the responsibility of dealing with these failures to the distributed DBMS.

As a consequence, the distributed DBMS is responsible for detecting that a message is undeliverable is left to the application program (in this case the distributed DBMS). The detection will be facilitated by the use of timers and a timeout mechanism that keeps track of how long it has been since the sender site has not received a confirmation from the destination site about the receipt of a message. This timeout interval needs to be set to a value greater than that of the maximum round-trip propagation delay of a message in the network. The term for the failure of the communication network to deliver messages and the confirmations within this period is *performance failure*. It needs to be handled within the reliability protocols for distributed DBMSs.

12.3 Local Reliability Protocols

In this section we discuss the functions performed by the local recovery manager (LRM) that exists at each site. These functions maintain the atomicity and durability properties of local transactions. They relate to the execution of the commands that are passed to the LRM, which are **begin_transaction**, **read**, **write**, **commit**, and **abort**. Later in this section we introduce a new command into the LRM's repertoire that initiates recovery actions after a failure. Note that in this section we discuss the execution of these commands in a centralized environment. The complications introduced in distributed databases are addressed in the upcoming sections.

12.3.1 Architectural Considerations

It is again time to use our architectural model and discuss the specific interface between the LRM and the database buffer manager (BM). First note that the LRM is implemented within the data processor introduced in Chapter 11. The simple DP implementation that was given earlier will be enhanced with the reliability protocols discussed in this section. Also remember that all accesses to the database are via the database buffer manager. The detailed discussion of the algorithms that the buffer manager implements is beyond the scope of this book; we provide a summary later in this subsection. Even without these details, we can still specify the interface and its function, as depicted in Figure 12.4.³

In this discussion we assume that the database is stored permanently on secondary storage, which in this context is called the *stable storage* [Lampson and Sturgis, 1976]. The stability of this storage medium is due to its robustness to failures. A

³ This architectural model is similar to that used by Härder and Reuter [1983] and Bernstein et al. [1987].

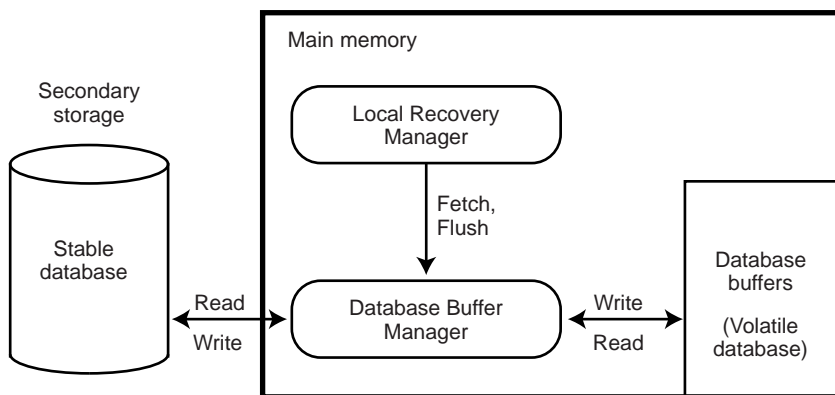


Fig. 12.4 Interface Between the Local Recovery Manager and the Buffer Manager

stable storage device would experience considerably less-frequent failures than would a non-stable storage device. In today's technology, stable storage is typically implemented by means of duplexed magnetic disks which store duplicate copies of data that are always kept mutually consistent (i.e., the copies are identical). We call the version of the database that is kept on stable storage the *stable database*. The unit of storage and access of the stable database is typically a *page*.

The database buffer manager keeps some of the recently accessed data in main memory buffers. This is done to enhance access performance. Typically, the buffer is divided into pages that are of the same size as the stable database pages. The part of the database that is in the database buffer is called the *volatile database*. It is important to note that the LRM executes the operations on behalf of a transaction only on the volatile database, which, at a later time, is written back to the stable database.

When the LRM wants to read a page of data⁴ on behalf of a transaction—strictly speaking, on behalf of some operation of a transaction—it issues a **fetch** command, indicating the page that it wants to read. The buffer manager checks to see if that page is already in the buffer (due to a previous fetch command from another transaction) and if so, makes it available for that transaction; if not, it reads the page from the stable database into an empty database buffer. If no empty buffers exist, it selects one of the buffer pages to write back to stable storage and reads the requested stable database page into that buffer. There are a number of different algorithms by which the buffer manager may choose the buffer page to be replaced; these are discussed in standard database textbooks.

The buffer manager also provides the interface by which the LRM can actually force it to write back some of the buffer pages. This can be accomplished by means of the **flush** command, which specifies the buffer pages that the LRM wants to be

⁴ The LRM's unit of access may be in blocks which have sizes different from a page. However, for simplicity, we assume that the unit of access is the same.

written back. We should indicate that different LRM implementations may or may not use this forced writing. This issue is discussed further in subsequent sections.

As its interface suggests, the buffer manager acts as a conduit for all access to the database via the buffers that it manages. It provides this function by fulfilling three tasks:

1. *Searching* the buffer pool for a given page;
2. If it is not found in the buffer, *allocating* a free buffer page and *loading* the buffer page with a data page that is brought in from secondary storage;
3. If no free buffer pages are available, choosing a buffer page for *replacement*.

Searching is quite straightforward. Typically, the buffer pages are shared among the transactions that execute against the database, so search is global.

Allocation of buffer pages is typically done dynamically. This means that the allocation of buffer pages to processes is performed as processes execute. The buffer manager tries to calculate the number of buffer pages needed to run the process efficiently and attempts to allocate that number of pages. The best known dynamic allocation method is the *working-set algorithm* [Denning, 1968, 1980].

A second aspect of allocation is fetching data pages. The most common technique is *demand paging*, where data pages are brought into the buffer as they are referenced. However, a number of operating systems prefetch a group of data pages that are in close physical proximity to the data page referenced. Buffer managers choose this route if they detect sequential access to a file.

In replacing buffer pages, the best known technique is the least recently used (LRU) algorithm that attempts to determine the *logical reference strings* [Effelsberg and Härder, 1984] of processes to buffer pages and to replace the page that has not been referenced for an extended period. The anticipation here is that if a buffer page has not been referenced for a long time, it probably will not be referenced in the near future.

The techniques discussed above are the most common. Other alternatives are discussed in [Effelsberg and Härder, 1984].

Clearly, these functions are similar to those performed by operating system (OS) buffer managers. However, quite frequently, DBMSs bypass OS buffer managers and manage disks and main memory buffers themselves due to a number of problems (see, e.g., [Stonebraker, 1981]) that are beyond the scope of this book. Basically, the requirements of DBMSs are usually incompatible with the services that OSs provide. The consequence is that DBMS kernels duplicate OS services with an implementation that is more suitable for their needs.

12.3.2 Recovery Information

In this section we assume that only system failures occur. We defer the discussion of techniques for recovering from media failures until later. Since we are dealing with centralized database recovery, communication failures are not applicable.

When a system failure occurs, the volatile database is lost. Therefore, the DBMS has to maintain some information about its state at the time of the failure in order to be able to bring the database to the state that it was in when the failure occurred. We call this information the *recovery information*.

The recovery information that the system maintains is dependent on the method of executing updates. Two possibilities are in-place updating and out-of-place updating. *In-place updating* physically changes the value of the data item in the stable database. As a result, the previous values are lost. *Out-of-place updating*, on the other hand, does not change the value of the data item in the stable database but maintains the new value separately. Of course, periodically, these updated values have to be integrated into the stable database. We should note that the reliability issues are somewhat simpler if in-place updating is not used. However, most DBMSs use it due to its improved performance.

12.3.2.1 In-Place Update Recovery Information

Since in-place updates cause previous values of the affected data items to be lost, it is necessary to keep enough information about the database state changes to facilitate the recovery of the database to a consistent state following a failure. This information is typically maintained in a *database log*. Thus each update transaction not only changes the database but the change is also recorded in the database log (Figure 12.5). The log contains information necessary to recover the database state following a failure.

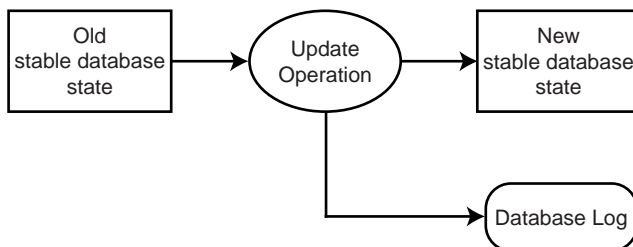


Fig. 12.5 Update Operation Execution

For the following discussion assume that the LRM and buffer manager algorithms are such that the buffer pages are written back to the stable database only when the buffer manager needs new buffer space. In other words, the **flush** command is not used by the LRM and the decision to write back the pages into the stable database is taken at the discretion of the buffer manager. Now consider that a transaction T_1 had completed (i.e., committed) before the failure occurred. The durability property of transactions would require that the effects of T_1 be reflected in the database. However, it is possible that the volatile database pages that have been updated by T_1 may not have been written back to the stable database at the time of the failure. Therefore, upon recovery, it is important to be able to *redo* the operations of T_1 . This requires some information to be stored in the database log about the effects of T_1 . Given this information, it is possible to recover the database from its “old” state to the “new” state that reflects the effects of T_1 (Figure 12.6).

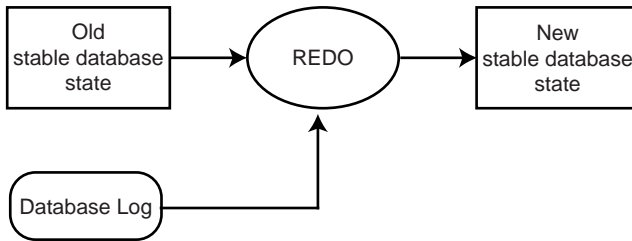


Fig. 12.6 REDO Action

Now consider another transaction, T_2 , that was still running when the failure occurred. The atomicity property would dictate that the stable database not contain any effects of T_2 . It is possible that the buffer manager may have had to write into the stable database some of the volatile database pages that have been updated by T_2 . Upon recovery from failures it is necessary to *undo* the operations of T_2 .⁵ Thus the recovery information should include sufficient data to permit the undo by taking the “new” database state that reflects partial effects of T_2 and recovers the “old” state that existed at the start of T_2 (Figure 12.7).

We should indicate that the undo and redo actions are assumed to be idempotent. In other words, their repeated application to a transaction would be equivalent to performing them once. Furthermore, the undo/redo actions form the basis of different methods of executing the commit commands. We discuss this further in Section 12.3.3.

The contents of the log may differ according to the implementation. However, the following minimal information for each transaction is contained in almost all

⁵ One might think that it could be possible to continue with the operation of T_2 following restart instead of undoing its operations. However, in general it may not be possible for the LRM to determine the point at which the transaction needs to be restarted. Furthermore, the failure may not be a system failure but a transaction failure (i.e., T_2 may actually abort itself) after some of its actions have been reflected in the stable database. Therefore, the possibility of undoing is necessary.

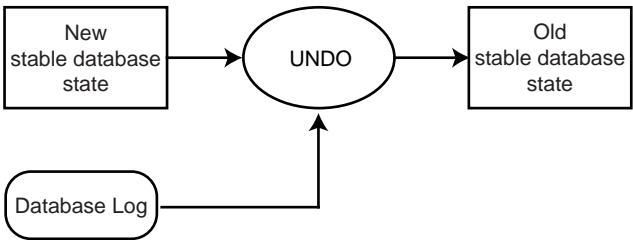


Fig. 12.7 UNDO Action

database logs: a begin_transaction record, the value of the data item before the update (called the *before image*), the updated value of the data item (called the *after image*), and a termination record indicating the transaction termination condition (commit, abort). The granularity of the before and after images may be different, as it is possible to log entire pages or some smaller unit. As an alternative to this form of *state logging*, *operational logging*, as in ARIES [Haderle et al., 1992], may be supported where the operations that cause changes to the database are logged rather than the before and after images.

The log is also maintained in main memory buffers (called *log buffers*) and written back to stable storage (called *stable log*) similar to the database buffer pages (Figure 12.8). The log pages can be written to stable storage in one of two ways. They can be written *synchronously* (more commonly known as *forcing a log*) where the addition of each log record requires that the log be moved from main memory to stable storage. They can also be written *asynchronously*, where the log is moved to stable storage either at periodic intervals or when the buffer fills up. When the log is written synchronously, the execution of the transaction is suspended until the write is complete. This adds some delay to the response-time performance of the transaction. On the other hand, if a failure occurs immediately after a forced write, it is relatively easy to recover to a consistent database state.

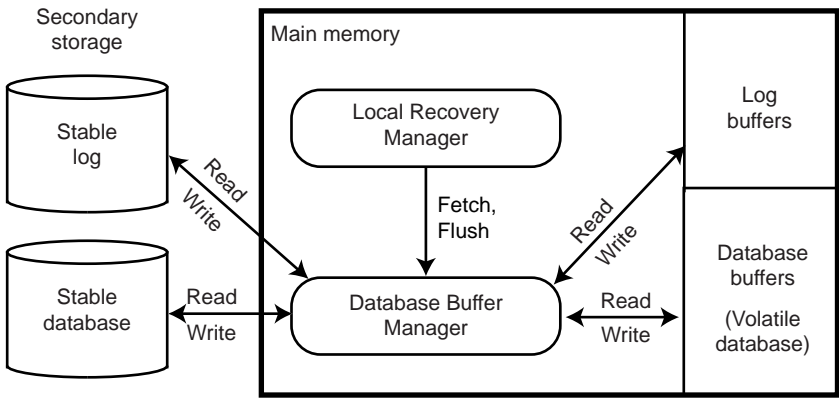


Fig. 12.8 Logging Interface

Whether the log is written synchronously or asynchronously, one very important protocol has to be observed in maintaining logs. Consider a case where the updates to the database are written into the stable storage before the log is modified in stable storage to reflect the update. If a failure occurs before the log is written, the database will remain in updated form, but the log will not indicate the update that makes it impossible to recover the database to a consistent and up-to-date state. Therefore, the stable log is always updated prior to the updating of the stable database. This is known as the *write-ahead logging* (WAL) protocol [Gray, 1979] and can be precisely specified as follows:

1. Before a stable database is updated (perhaps due to actions of a yet uncommitted transaction), the before images should be stored in the stable log. This facilitates undo.
2. When a transaction commits, the after images have to be stored in the stable log prior to the updating of the stable database. This facilitates redo.

12.3.2.2 Out-of-Place Update Recovery Information

As we mentioned above, the most common update technique is in-place updating. Therefore, we provide only a brief overview of the other updating techniques and their recovery information. Details can be found in [Verhofstadt, 1978] and the other references given earlier.

Typical techniques for out-of-place updating are *shadowing* ([Astrahan et al., 1976; Gray, 1979]) and *differential files* [Severence and Lohman, 1976]. Shadowing uses duplicate stable storage pages in executing updates. Thus every time an update is made, the old stable storage page, called the *shadow page*, is left intact and a new page with the updated data item values is written into the stable database. The access path data structures are updated to point to the new page, which contains the current data so that subsequent accesses are to this page. The old stable storage page is retained for recovery purposes (to perform undo).

Recovery based on shadow paging is implemented in System R's recovery manager [Gray et al., 1981]. This implementation uses shadowing together with logging.

The differential file approach was discussed in Chapter 5 within the context of integrity enforcement. In general, the method maintains each stable database file as a read-only file. In addition, it maintains a corresponding read-write differential file that stores the changes to that file. Given a logical database file F , let us denote its read-only part as FR and its corresponding differential file as DF . DF consists of two parts: an insertions part, which stores the insertions to F , denoted DF^+ , and a corresponding deletions part, denoted DF^- . All updates are treated as the deletion of the old value and the insertion of a new one. Thus each logical file F is considered to be a view defined as $F = (FR \cup DF^+) - DF^-$. Periodically, the differential file needs to be merged with the read-only base file.

Recovery schemes based on this method simply use private differential files for each transaction, which are then merged with the differential files of each file at

commit time. Thus recovery from failures can simply be achieved by discarding the private differential files of non-committed transactions.

There are studies that indicate that the shadowing and differential files approaches may be advantageous in certain environments. One study by [Agrawal and DeWitt \[1985\]](#) investigates the performance of recovery mechanisms based on logging, differential files, and shadow paging, integrated with locking and optimistic (using timestamps) concurrency control algorithms. The results indicate that shadowing, together with locking, can be a feasible alternative to the more common log-based recovery integrated with locking if there are only large (in terms of the base-set size) transactions with sequential access patterns. Similarly, differential files integrated with locking can be a feasible alternative if there are medium-sized and large transactions.

12.3.3 Execution of LRM Commands

Recall that there are five commands that form the interface to the LRM. These are the **begin.transaction**, **read**, **write**, **commit**, and **abort** commands. As we indicated in Chapter 10, some DBMSs do not have an explicit commit command. In this case the end (of transaction) indicator serves as the commit command. For simplicity, we specify commit explicitly.

In this section we introduce a sixth interface command to the LRM: **recover**. The **recover** command is the interface that the operating system has to the LRM. It is used during recovery from system failures when the operating system asks the DBMS to recover the database to the state that existed when the failure occurred.

The execution of some of these commands (specifically, **abort**, **commit**, and **recover**) is quite dependent on the specific LRM algorithms that are used as well as on the interaction of the LRM with the buffer manager. Others (i.e., **begin.transaction**, **read**, and **write**) are quite independent of these considerations.

The fundamental design decision in the implementation of the local recovery manager, the buffer manager, and the interaction between the two components is whether or not the buffer manager obeys the local recovery manager's instructions as to when to write the database buffer pages to stable storage. Specifically, two decisions are involved. The first one is whether the buffer manager may write the buffer pages updated by a transaction into stable storage during the execution of that transaction, or it waits for the LRM to instruct it to write them back. We call this the *fix/no-fix* decision. The reasons for the choice of this terminology will become apparent shortly. Note that it is also called the steal/no-steal decision by [Härder and Reuter \[1983\]](#). The second decision is whether the buffer manager will be forced to flush the buffer pages updated by a transaction into the stable storage at the end of that transaction (i.e., the commit point), or the buffer manager flushes them out whenever it needs to according to its buffer management algorithm. We call this the *flush/no-flush* decision. It is called the force/no-force decision by [Härder and Reuter \[1983\]](#).

Accordingly, four alternatives can be identified: (1) no-fix/no-flush, (2) no-fix/flush, (3) fix/no-flush, and (4) fix/flush. We will consider each of these in more detail. However, first we present the execution methods of the **begin_transaction**, **read**, and **write** commands, which are quite independent of these considerations. Where modifications are required in these methods due to different LRM and buffer manager implementation strategies, we will indicate them.

12.3.3.1 **Begin_transaction, Read, and Write Commands**

Begin_transaction.

This command causes various components of the DBMS to carry out some bookkeeping functions. We will also assume that it causes the LRM to write a **begin_transaction** record into the log. This is an assumption made for convenience of discussion; in reality, writing of the **begin_transaction** record may be delayed until the first **write** to improve performance by reducing I/O.

Read.

The **read** command specifies a data item. The LRM tries to read the specified data item from the buffer pages that belong to the transaction. If the data item is not in one of these pages, it issues a **fetch** command to the buffer manager in order to make the data available. Upon reading the data, the LRM returns it to the scheduler.

Write.

The **write** command specifies the data item and the new value. As with a read command, if the data item is available in the buffers of the transaction, its value is modified in the database buffers (i.e., the volatile database). If it is not in the private buffer pages, a **fetch** command is issued to the buffer manager, and the data is made available and updated. The before image of the data page, as well as its after image, are recorded in the log. The local recovery manager then informs the scheduler that the operation has been completed successfully.

12.3.3.2 **No-fix/No-flush**

This type of LRM algorithm is called a redo/undo algorithm by [Bernstein et al. \[1987\]](#) since it requires, as we will see, performing both the redo and undo operations upon recovery. It is called steal/no-force by [Härder and Reuter \[1983\]](#).

Abort.

As we indicated before, abort is an indication of transaction failure. Since the buffer manager may have written the updated pages into the stable database, abort will have to undo the actions of the transaction. Therefore, the LRM reads the log records for that specific transaction and replaces the values of the updated data items in the volatile database with their before images. The scheduler is then informed about the successful completion of the abort action. This process is called the *transaction undo* or *partial undo*.

An alternative implementation is the use of an *abort list*, which stores the identifiers of all the transactions that have been aborted. If such a list is used, the abort action is considered to be complete as soon as the transaction's identifier is included in the abort list.

Note that even though the values of the updated data items in the stable database are not restored to their before images, the transaction is considered to be aborted at this point. The buffer manager will write the "corrected" volatile database pages into the stable database at a future time, thereby restoring it to its state prior to that transaction.

Commit.

The **commit** command causes an *end_of.transaction* record to be written into the log by the LRM. Under this scenario, no other action is taken in executing a commit command other than informing the scheduler about the successful completion of the commit action.

An alternative to writing an *end_of.transaction* record into the log is to add the transaction's identifier to a *commit list*, which is a list of the identifiers of transactions that have committed. In this case the commit action is accepted as complete as soon as the transaction identifier is stored in this list.

Recover.

The LRM starts the recovery action by going to the beginning of the log and redoing the operations of each transaction for which both a *begin_transaction* and an *end_of.transaction* record is found. This is called *partial redo*. Similarly, it undoes the operations of each transaction for which a *begin_transaction* record is found in the log without a corresponding *end_of.transaction* record. This action is called *global undo*, as opposed to the transaction undo discussed above. The difference is that the effects of all incomplete transactions need to be rolled back, not one.

If commit list and abort list implementations are used, the recovery action consists of redoing the operations of all the transactions in the commit list and undoing the operations of all the transactions in the abort list. In the remainder of this chapter

we will not make this distinction, but rather will refer to both of these recovery implementations as global undo.

12.3.3.3 No-fix/Flush

The LRM algorithms that use this strategy are called undo/no-redo in [Bernstein et al. \[1987\]](#) and steal/force by [Härder and Reuter \[1983\]](#).

Abort.

The execution of **abort** is identical to the previous case. Upon transaction failure, the LRM initiates a partial undo for that particular transaction.

Commit.

The LRM issues a **flush** command to the buffer manager, forcing it to write back all the updated volatile database pages into the stable database. The commit command is then executed either by placing a record in the log or by insertion of the transaction identifier into the commit list as specified for the previous case. When all of this is complete, the LRM informs the scheduler that the commit has been carried out successfully.

Recover.

Since all the updated pages are written into the stable database at the commit point, there is no need to perform redo; all the effects of successful transactions will have been reflected in the stable database. Therefore, the recovery action initiated by the LRM consists of a global undo.

12.3.3.4 Fix/No-flush

In this case the LRM controls the writing of the volatile database pages into stable storage. The key here is not to permit the buffer manager to write any updated volatile database page into the stable database until at least the transaction commit point. This is accomplished by the **fix** command, which is a modified version of the **fetch** command whereby the specified page is fixed in the database buffer and cannot be written back to the stable database by the buffer manager. Thus any **fetch** command to the buffer manager for a write operation is replaced by a **fix** command.⁶ Note

⁶ Of course, any page that was previously fetched for read but is now being updated also needs to be fixed.

that this precludes the need for a global undo operation and is therefore called a redo/no-undo algorithm by Bernstein et al. [1987] and a no-force/no-steal algorithm by Härder and Reuter [1983].

Abort.

Since the volatile database pages have not been written to the stable database, no special action is necessary. To release the buffer pages that have been fixed by the transaction, however, it is necessary for the LRM to send an **unfix** command to the buffer manager for all such pages. It is then sufficient to carry out the abort action either by writing an abort record in the log or by including the transaction in the abort list, informing the scheduler and then forgetting about the transaction.

Commit.

The LRM sends an **unfix** command to the buffer manager for every volatile database page that was previously fixed by that transaction. Note that these pages may now be written back to the stable database at the discretion of the buffer manager. The commit command is then executed either by placing an `end_of_transaction` record in the log or by inserting the transaction identifier into the commit list as specified for the preceding case. When all of this is complete, the LRM informs the scheduler that the commit has been successfully carried out.

Recover.

As we mentioned above, since the volatile database pages that have been updated by ongoing transactions are not yet written into the stable database, there is no necessity for global undo. The LRM, therefore, initiates a partial redo action to recover those transactions that may have already committed, but whose volatile database pages may not have yet been written into the stable database.

12.3.3.5 Fix/Flush

This is the case where the LRM forces the buffer manager to write the updated volatile database pages into the stable database at precisely the commit point—not before and not after. This strategy is called no-undo/no-redo by Bernstein et al. [1987] and no-steal/force by Härder and Reuter [1983].

Abort.

The execution of **abort** is identical to that of the fix/no-flush case.

Commit.

The LRM sends an **unfix** command to the buffer manager for every volatile database page that was previously fixed by that transaction. It then issues a **flush** command to the buffer manager, forcing it to write back all the unfixed volatile database pages into the stable database.⁷ Finally, the **commit** command is processed by either writing an end_of_transaction record into the log or by including the transaction in the commit list. The important point to note here is that all three of these operations have to be executed as an atomic action. One step that can be taken to achieve this atomicity is to issue only a **flush** command, which serves to unfix the pages as well. This eliminates the need to send two messages from the LRM to the buffer manager, but does not eliminate the requirement for the atomic execution of the flush operation and the writing of the database log. The LRM then informs the scheduler that the **commit** has been carried out successfully. Methods for ensuring this atomicity are beyond the scope of our discussion (see [Bernstein et al., 1987]).

Recover.

The **recover** command does not need to do anything in this case. This is true since the stable database reflects the effects of all the successful transactions and none of the effects of the uncommitted transactions.

12.3.4 Checkpointing

In most of the LRM implementation strategies, the execution of the recovery action requires searching the entire log. This is a significant overhead because the LRM is trying to find all the transactions that need to be undone and redone. The overhead can be reduced if it is possible to build a wall which signifies that the database at that point is up-to-date and consistent. In that case, the redo has to start from that point on and the undo only has to go back to that point. This process of building the wall is called *checkpointing*.

Checkpointing is achieved in three steps [Gray, 1979]:

⁷ Our discussion here gives the impression that two commands (*unfix* and *flush*) need to be sent to the BM by the LRM for each commit action. We have chosen to explain the action in this way only for presentation simplicity. In reality, it is, of course, possible and preferable to implement one command that instructs the BM to both unfix and flush, thereby reducing the message overhead between DBMS components.

1. Write a `begin_checkpoint` record into the log.
2. Collect the checkpoint data into the stable storage.
3. Write an `end_checkpoint` record into the log.

The first and the third steps enforce the atomicity of the checkpointing operation. If a system failure occurs during checkpointing, the recovery process will not find an `end_checkpoint` record and will consider checkpointing not completed.

There are a number of different alternatives for the data that is collected in Step 2, how it is collected, and where it is stored. We will consider one example here, called *transaction-consistent checkpointing* ([Gray, 1979; Gray et al., 1981]). The checkpointing starts by writing the `begin_checkpoint` record in the log and stopping the acceptance of any new transactions by the LRM. Once the active transactions are all completed, all the updated volatile database pages are flushed to the stable database followed by the insertion of an `end_checkpoint` record into the log. In this case, the redo action only needs to start from the `end_checkpoint` record in the log. The undo action can go the reverse direction, starting from the end of the log and stopping at the `end_checkpoint` record.

Transaction-consistent checkpointing is not the most efficient algorithm, since a significant delay is experienced by all the transactions. There are alternative checkpointing schemes such as action-consistent checkpoints, fuzzy checkpoints, and others ([Gray, 1979; Lindsay, 1979]).

12.3.5 Handling Media Failures

As we mentioned before, the previous discussion on centralized recovery considered non-media failures, where the database as well as the log stored in the stable storage survive the failure. Media failures may either be quite catastrophic, causing the loss of the stable database or of the stable log, or they can simply result in partial loss of the database or the log (e.g., loss of a track or two).

The methods that have been devised for dealing with this situation are again based on duplexing. To cope with catastrophic media failures, an *archive* copy of both the database and the log is maintained on a different (tertiary) storage medium, which is typically the magnetic tape or CD-ROM. Thus the DBMS deals with three levels of memory hierarchy: the main memory, random access disk storage, and magnetic tape (Figure 12.9). To deal with less catastrophic failures, having duplicate copies of the database and log may be sufficient.

When a media failure occurs, the database is recovered from the archive copy by redoing and undoing the transactions as stored in the archive log. The real question is how the archive database is stored. If we consider the large sizes of current databases, the overhead of writing the entire database to tertiary storage is significant. Two methods that have been proposed for dealing with this are to perform the archiving activity concurrent with normal processing and to archive the database incrementally

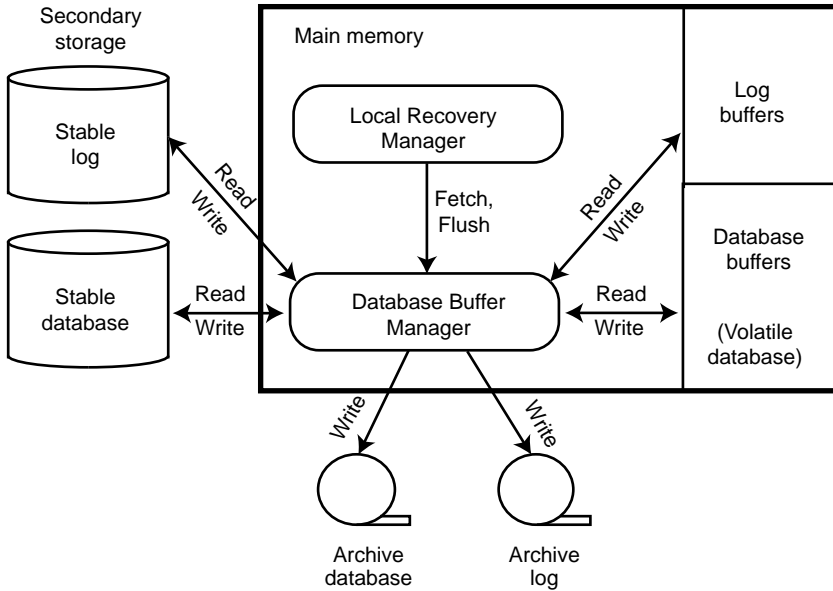


Fig. 12.9 Full Memory Hierarchy Managed by LRM and BM

as changes occur so that each archive version contains only the changes that have occurred since the previous archiving.

12.4 Distributed Reliability Protocols

As with local reliability protocols, the distributed versions aim to maintain the atomicity and durability of distributed transactions that execute over a number of databases. The protocols address the distributed execution of the **begin_transaction**, **read**, **write**, **abort**, **commit**, and **recover** commands.

At the outset we should indicate that the execution of the **begin_transaction**, **read**, and **write** commands does not cause any significant problems. **Begin_transaction** is executed in exactly the same manner as in the centralized case by the transaction manager at the originating site of the transaction. The **read** and **write** commands are executed as discussed in Chapter 11. At each site, the commands are executed in the manner described in Section 12.3.3. Similarly, **abort** is executed by undoing its effects.

The implementation of distributed reliability protocols within the architectural model we have adopted in this book raises a number of interesting and difficult issues. We discuss these in Section 12.7 after we introduce the protocols. For the time being, we adopt a common abstraction: we assume that at the originating site of a transaction there is a *coordinator* process and at each site where the transaction

executes there are *participant* processes. Thus, the distributed reliability protocols are implemented between the coordinator and the participants.

12.4.1 Components of Distributed Reliability Protocols

The reliability techniques in distributed database systems consist of commit, termination, and recovery protocols. Recall from the preceding section that the commit and recovery protocols specify how the commit and the recover commands are executed. Both of these commands need to be executed differently in a distributed DBMS than in a centralized DBMS. Termination protocols are unique to distributed systems. Assume that during the execution of a distributed transaction, one of the sites involved in the execution fails; we would like the other sites to terminate the transaction somehow. The techniques for dealing with this situation are called *termination protocols*. Termination and recovery protocols are two opposite faces of the recovery problem: given a site failure, termination protocols address how the operational sites deal with the failure, whereas recovery protocols deal with the procedure that the process (coordinator or participant) at the failed site has to go through to recover its state once the site is restarted. In the case of network partitioning, the termination protocols take the necessary measures to terminate the active transactions that execute at different partitions, while the recovery protocols address the establishment of mutual consistency of replicated databases following reconnection of the partitions of the network.

The primary requirement of commit protocols is that they maintain the atomicity of distributed transactions. This means that even though the execution of the distributed transaction involves multiple sites, some of which might fail while executing, the effects of the transaction on the distributed database is all-or-nothing. This is called *atomic commitment*. We would prefer the termination protocols to be *non-blocking*. A protocol is non-blocking if it permits a transaction to terminate at the operational sites without waiting for recovery of the failed site. This would significantly improve the response-time performance of transactions. We would also like the distributed recovery protocols to be *independent*. Independent recovery protocols determine how to terminate a transaction that was executing at the time of a failure without having to consult any other site. Existence of such protocols would reduce the number of messages that need to be exchanged during recovery. Note that the existence of independent recovery protocols would imply the existence of non-blocking termination protocols, but the reverse is not true.

12.4.2 Two-Phase Commit Protocol

Two-phase commit (2PC) is a very simple and elegant protocol that ensures the atomic commitment of distributed transactions. It extends the effects of local atomic

commit actions to distributed transactions by insisting that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent. There are a number of reasons why such synchronization among sites is necessary. First, depending on the type of concurrency control algorithm that is used, some schedulers may not be ready to terminate a transaction. For example, if a transaction has read a value of a data item that is updated by another transaction that has not yet committed, the associated scheduler may not want to commit the former. Of course, strict concurrency control algorithms that avoid cascading aborts would not permit the updated value of a data item to be read by any other transaction until the updating transaction terminates. This is sometimes called the *recoverability condition* ([Hadzilacos, 1988; Bernstein et al., 1987]).

Another possible reason why a participant may not agree to commit is due to deadlocks that require a participant to abort the transaction. Note that, in this case, the participant should be permitted to abort the transaction without being told to do so. This capability is quite important and is called *unilateral abort*.

A brief description of the 2PC protocol that does not consider failures is as follows. Initially, the coordinator writes a `begin_commit` record in its log, sends a “prepare” message to all participant sites, and enters the WAIT state. When a participant receives a “prepare” message, it checks if it could commit the transaction. If so, the participant writes a `ready` record in the log, sends a “vote-commit” message to the coordinator, and enters READY state; otherwise, the participant writes an `abort` record and sends a “vote-abort” message to the coordinator. If the decision of the site is to abort, it can forget about that transaction, since an abort decision serves as a veto (i.e., unilateral abort). After the coordinator has received a reply from every participant, it decides whether to commit or to abort the transaction. If even one participant has registered a negative vote, the coordinator has to abort the transaction globally. So it writes an `abort` record, sends a “global-abort” message to all participant sites, and enters the ABORT state; otherwise, it writes a `commit` record, sends a “global-commit” message to all participants, and enters the COMMIT state. The participants either commit or abort the transaction according to the coordinator’s instructions and send back an acknowledgment, at which point the coordinator terminates the transaction by writing an `end_of_transaction` record in the log.

Note the manner in which the coordinator reaches a global termination decision regarding a transaction. Two rules govern this decision, which, together, are called the *global commit rule*:

1. If even one participant votes to abort the transaction, the coordinator has to reach a global abort decision.
2. If all the participants vote to commit the transaction, the coordinator has to reach a global commit decision.

The operation of the 2PC protocol between a coordinator and one participant in the absence of failures is depicted in Figure 12.10, where the circles indicate the states and the dashed lines indicate messages between the coordinator and the participants. The labels on the dashed lines specify the nature of the message.

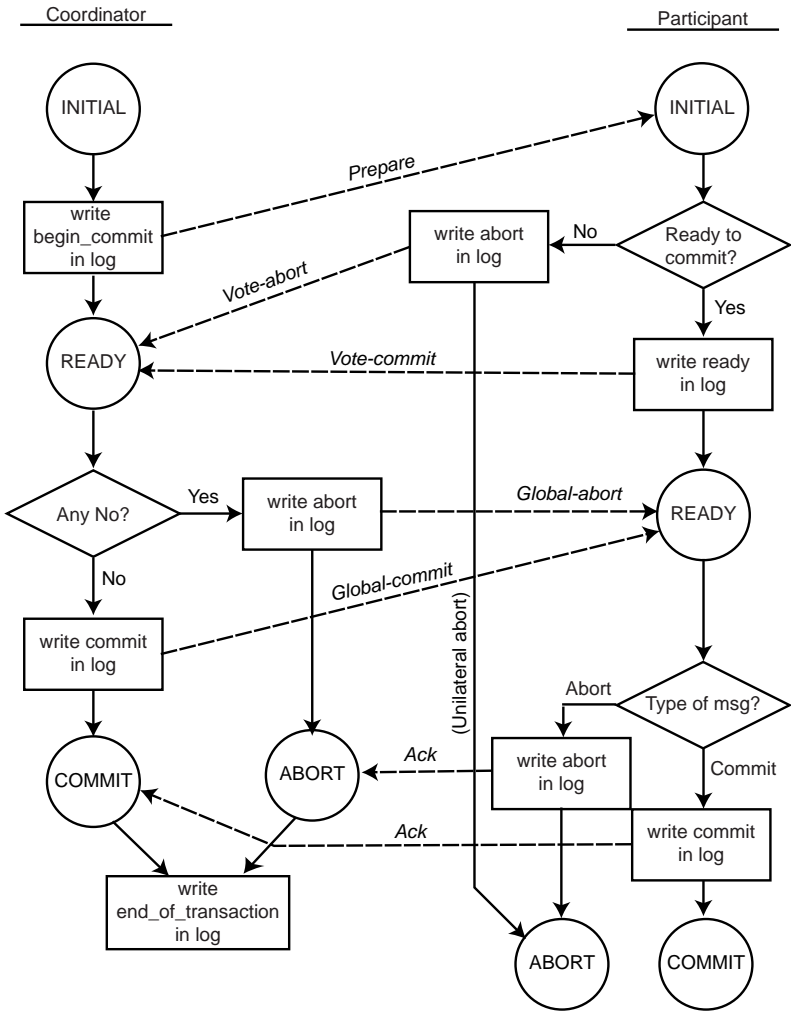


Fig. 12.10 2PC Protocol Actions

A few important points about the 2PC protocol that can be observed from Figure 12.10 are as follows. First, 2PC permits a participant to unilaterally abort a transaction until it has decided to register an affirmative vote. Second, once a participant votes to commit or abort a transaction, it cannot change its vote. Third, while a participant is in the READY state, it can move either to abort the transaction or to commit it, depending on the nature of the message from the coordinator. Fourth, the global termination decision is taken by the coordinator according to the global commit rule. Finally, note that the coordinator and participant processes enter certain states where they have to wait for messages from one another. To guarantee that they can exit from these states and terminate, timers are used. Each process sets its timer when

it enters a state, and if the expected message is not received before the timer runs out, the process times out and invokes its timeout protocol (which will be discussed later).

There are a number of different communication paradigms that can be employed in implementing a 2PC protocol. The one discussed above and depicted in Figure 12.10 is called a *centralized 2PC* since the communication is only between the coordinator and the participants; the participants do not communicate among themselves. This communication structure, which is the basis of our subsequent discussions in this chapter, is depicted more clearly in Figure 12.11.

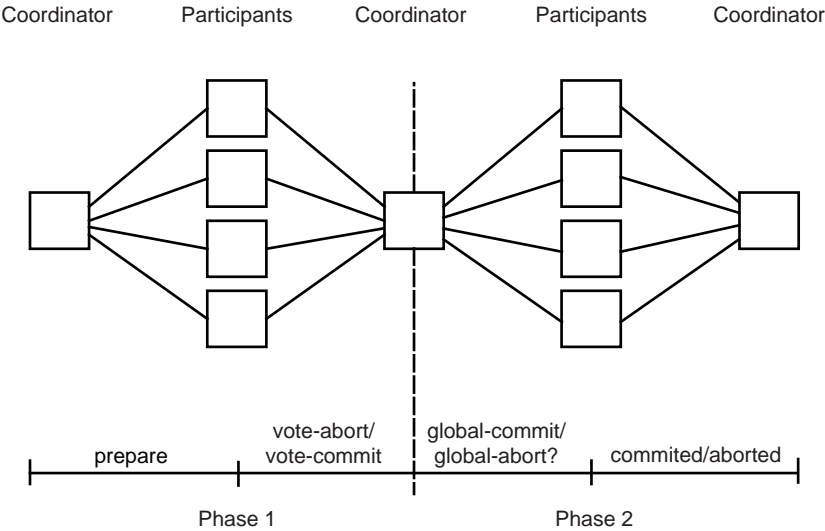


Fig. 12.11 Centralized 2PC Communication Structure

Another alternative is *linear 2PC* (also called *nested 2PC* [Gray, 1979]) where participants can communicate with one another. There is an ordering between the sites in the system for the purposes of communication. Let us assume that the ordering among the sites that participate in the execution of a transaction are 1, ..., N , where the coordinator is the first one in the order. The 2PC protocol is implemented by a forward communication from the coordinator (number 1) to N , during which the first phase is completed, and by a backward communication from N to the coordinator, during which the second phase is completed. Thus linear 2PC operates in the following manner.

The coordinator sends the “prepare” message to participant 2. If participant 2 is not ready to commit the transaction, it sends a “vote-abort” message (VA) to participant 3 and the transaction is aborted at this point (unilateral abort by 2). If, on the other hand, participant 2 agrees to commit the transaction, it sends a “vote-commit” message (VC) to participant 3 and enters the READY state. This process continues until a “vote-commit” vote reaches participant N . This is the end of the

first phase. If N decides to commit, it sends back to $N - 1$ “global-commit” (GC); otherwise, it sends a “global-abort” message (GA). Accordingly, the participants enter the appropriate state (COMMIT or ABORT) and propagate the message back to the coordinator.

Linear 2PC, whose communication structure is depicted in Figure 12.12, incurs fewer messages but does not provide any parallelism. Therefore, it suffers from low response-time performance.

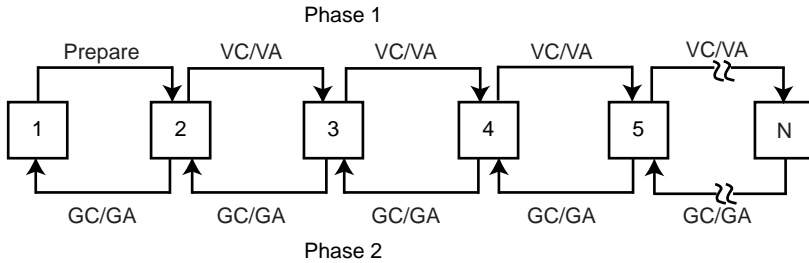


Fig. 12.12 Linear 2PC Communication Structure. VC, vote.commit; VA, vote.abort; GC, global.commit; GA, global.abort.)

Another popular communication structure for implementation of the 2PC protocol involves communication among all the participants during the first phase of the protocol so that they all independently reach their termination decisions with respect to the specific transaction. This version, called *distributed 2PC*, eliminates the need for the second phase of the protocol since the participants can reach a decision on their own. It operates as follows. The coordinator sends the prepare message to all participants. Each participant then sends its decision to all the other participants (and to the coordinator) by means of either a “vote-commit” or a “vote-abort” message. Each participant waits for messages from all the other participants and makes its termination decision according to the global commit rule. Obviously, there is no need for the second phase of the protocol (someone sending the global abort or global commit decision to the others), since each participant has independently reached that decision at the end of the first phase. The communication structure of distributed commit is depicted in Figure 12.13.

One point that needs to be addressed with respect to the last two versions of 2PC implementation is the following. A participant has to know the identity of either the next participant in the linear ordering (in case of linear 2PC) or of all the participants (in case of distributed 2PC). This problem can be solved by attaching the list of participants to the prepare message that is sent by the coordinator. Such an issue does not arise in the case of centralized 2PC since the coordinator clearly knows who the participants are.

The algorithm for the centralized execution of the 2PC protocol by the coordinator is given in Algorithm 12.1, and the algorithm for participants is given in Algorithm 12.2.

Algorithm 12.1: 2PC Coordinator Algorithm (2PC-C)

```

begin
  repeat
    wait for an event ;
    switch event do
      case Msg Arrival
        Let the arrived message be msg ;
        switch msg do
          case Commit {commit command from scheduler}
            write begin.commit record in the log ;
            send “Prepared” message to all the involved
            participants ;
            set timer
          case Vote-abort {one participant has voted to abort;
            unilateral abort}
            write abort record in the log ;
            send “Global-abort” message to the other involved
            participants ;
            set timer
          case Vote-commit
            update the list of participants who have answered ;
            if all the participants have answered then {all must
            have voted to commit}
              write commit record in the log ;
              send “Global-commit” to all the involved
              participants ;
              set timer
            case Ack
              update the list of participants who have acknowledged ;
              if all the participants have acknowledged then
                write end_of_transaction record in the log
              else
                send global decision to the unanswering participants
          case Timeout
            execute the termination protocol
        until forever ;
  end

```

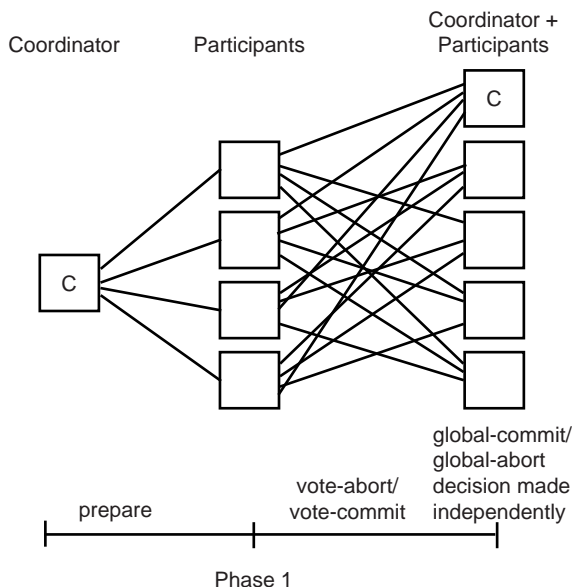


Fig. 12.13 Distributed 2PC Communication Structure

12.4.3 Variations of 2PC

Two variations of 2PC have been proposed to improve its performance. This is accomplished by reducing (1) the number of messages that are transmitted between the coordinator and the participants, and (2) the number of times logs are written. These protocols are called *presumed abort* and *presumed commit* [Mohan and Lindsay, 1983; Mohan et al., 1986]. Presumed abort is a protocol that is optimized to handle read-only transactions as well as those update transactions, some of whose processes do not perform any updates to the database (called partially read-only). The presumed commit protocol is optimized to handle the general update transactions. We will discuss briefly both of these variations.

12.4.3.1 Presumed Abort 2PC Protocol

In the presumed abort 2PC protocol the following assumption is made. Whenever a prepared participant polls the coordinator about a transaction's outcome and there is no information in virtual storage about it, the response to the inquiry is to abort the transaction. This works since, in the case of a commit, the coordinator does not forget about a transaction until all participants acknowledge, guaranteeing that they will no longer inquire about this transaction.

When this convention is used, it can be seen that the coordinator can forget about a transaction immediately after it decides to abort it. It can write an abort record and

Algorithm 12.2: 2PC Participant Algorithm (2PC-P)

```

begin
  repeat
    wait for an event ;
    switch ev do
      case Msg Arrival
        Let the arrived message be msg ;
        switch msg do
          case Prepare      {Prepare command from the coordinator}
            if ready to commit then
              write ready record in the log ;
              send "Vote-commit" message to the coordinator ;
              set timer
            else                                     {unilateral abort}
              write abort record in the log ;
              send "Vote-abort" message to the coordinator ;
              abort the transaction
          case Global-abort
            write abort record in the log ;
            abort the transaction
          case Global-commit
            write commit record in the log ;
            commit the transaction
        case Timeout
          execute the termination protocol
      until forever ;
  end

```

not expect the participants to acknowledge the abort command. The coordinator does not need to write an `end_of_transaction` record after an abort record.

The abort record does not need to be forced, because if a site fails before receiving the decision and then recovers, the recovery routine will check the log to determine the fate of the transaction. Since the abort record is not forced, the recovery routine may not find any information about the transaction, in which case it will ask the coordinator and will be told to abort it. For the same reason, the abort records do not need to be forced by the participants either.

Since it saves some message transmission between the coordinator and the participants in case of aborted transactions, presumed abort 2PC is expected to be more efficient.

12.4.3.2 Presumed Commit 2PC Protocol

The presumed abort 2PC protocol, as discussed above, improves performance by forgetting about transactions once a decision is reached to abort them. Since most transactions are expected to commit, it is reasonable to expect that it may be similarly possible to improve performance for commits. Hence the presumed commit 2PC protocol.

Presumed commit 2PC is based on the premise that if no information about the transaction exists, it should be considered committed. However, it is not an exact dual of presumed abort 2PC, since an exact dual would require that the coordinator forget about a transaction immediately after it decides to commit it, that commit records (also the ready records of the participants) not be forced, and that commit commands need not be acknowledged. Consider, however, the following scenario. The coordinator sends prepared messages and starts collecting information, but fails before being able to collect all of them and reach a decision. In this case, the participants will wait until they timeout, and then turn the transaction over to their recovery routines. Since there is no information about the transaction, the recovery routines of each participant will commit the transaction. The coordinator, on the other hand, will abort the transaction when it recovers, thus causing inconsistency.

A simple variation of this protocol, however, solves the problem and that variant is called the *presumed commit 2PC*. The coordinator, prior to sending the prepare message, force-writes a collecting record, which contains the names of all the participants involved in executing that transaction. The participant then enters the COLLECTING state, following which it sends the prepare message and enters the WAIT state. The participants, when they receive the prepare message, decide what they want to do with the transaction, write an abort record, or write a ready record and respond with either a “vote-abort” or a “vote-commit” message. When the coordinator receives decisions from all the participants, it decides to abort or commit the transaction. If the decision is to abort, the coordinator writes an abort record, enters the ABORT state, and sends a “global-abort” message. If it decides to commit the transaction, it writes a commit record, sends a “global-commit” command, and forgets the transaction. When the participants receive a “global-commit” message, they write a commit record and update the database. If they receive a “global-abort” message, they write an abort record and acknowledge. The participant, upon receiving the abort acknowledgment, writes an `end_of_transaction` record and forgets about the transaction.

12.5 Dealing with Site Failures

In this section we consider the failure of sites in the network. Our aim is to develop non-blocking termination and independent recovery protocols. As we indicated before, the existence of independent recovery protocols would imply the existence of non-blocking recovery protocols. However, our discussion addresses both aspects

separately. Also note that in the following discussion we consider only the standard 2PC protocol, not its two variants presented above.

Let us first set the boundaries for the existence of non-blocking termination and independent recovery protocols in the presence of site failures. It can formally be proven that such protocols exist when a single site fails. In the case of multiple site failures, however, the prospects are not as promising. A negative result indicates that it is not possible to design independent recovery protocols (and, therefore, non-blocking termination protocols) when multiple sites fail [Skeen and Stonebraker, 1983]. We first develop termination and recovery protocols for the 2PC algorithm and show that 2PC is inherently blocking. We then proceed to the development of atomic commit protocols which are non-blocking in the case of single site failures.

12.5.1 Termination and Recovery Protocols for 2PC

12.5.1.1 Termination Protocols

The termination protocols serve the timeouts for both the coordinator and the participant processes. A timeout occurs at a destination site when it cannot get an expected message from a source site within the expected time period. In this section we consider that this is due to the failure of the source site.

The method for handling timeouts depends on the timing of failures as well as on the types of failures. We therefore need to consider failures at various points of 2PC execution. This discussion is facilitated by means of the state transition diagram of the 2PC protocol given in Figure 12.14. Note that the state transition diagram is a simplification of Figure 12.10. The states are denoted by circles and the edges represent the state transitions. The terminal states are depicted by concentric circles. The interpretation of the labels on the edges is as follows: the reason for the state transition, which is a received message, is given at the top, and the message that is sent as a result of state transition is given at the bottom.

Coordinator Timeouts.

There are three states in which the coordinator can timeout: WAIT, COMMIT, and ABORT. Timeouts during the last two are handled in the same manner. So we need to consider only two cases:

1. *Timeout in the WAIT state.* In the WAIT state, the coordinator is waiting for the local decisions of the participants. The coordinator cannot unilaterally commit the transaction since the global commit rule has not been satisfied. However, it can decide to globally abort the transaction, in which case it writes an abort record in the log and sends a “global-abort” message to all the participants.

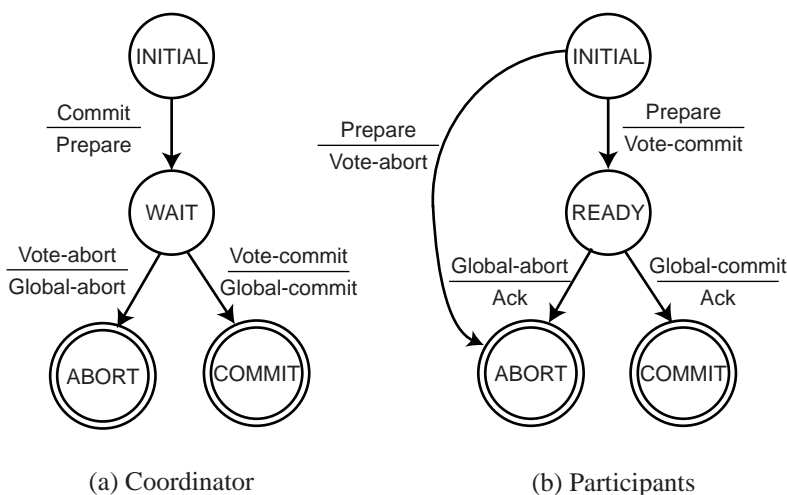


Fig. 12.14 State Transitions in 2PC Protocol

2. *Timeout in the COMMIT or ABORT states.* In this case the coordinator is not certain that the commit or abort procedures have been completed by the local recovery managers at all of the participant sites. Thus the coordinator repeatedly sends the “global-commit” or “global-abort” commands to the sites that have not yet responded, and waits for their acknowledgement.

Participant Timeouts.

A participant can time out⁸ in two states: INITIAL and READY. Let us examine both of these cases.

1. *Timeout in the INITIAL state.* In this state the participant is waiting for a “prepare” message. The coordinator must have failed in the INITIAL state. The participant can unilaterally abort the transaction following a timeout. If the “prepare” message arrives at this participant at a later time, this can be handled in one of two possible ways. Either the participant would check its log, find the abort record, and respond with a “vote-abort,” or it can simply ignore the “prepare” message. In the latter case the coordinator would time out in the WAIT state and follow the course we have discussed above.
2. *Timeout in the READY state.* In this state the participant has voted to commit the transaction but does not know the global decision of the coordinator. The participant cannot unilaterally make a decision. Since it is in the READY state,

⁸ In some discussions of the 2PC protocol, it is assumed that the participants do not use timers and do not time out. However, implementing timeout protocols for the participants solves some nasty problems and may speed up the commit process. Therefore, we consider this more general case.

it must have voted to commit the transaction. Therefore, it cannot now change its vote and unilaterally abort it. On the other hand, it cannot unilaterally decide to commit it since it is possible that another participant may have voted to abort it. In this case the participant will remain blocked until it can learn from someone (either the coordinator or some other participant) the ultimate fate of the transaction.

Let us consider a centralized communication structure where the participants cannot communicate with one another. In this case the participant that is trying to terminate a transaction has to ask the coordinator for its decision and wait until it receives a response. If the coordinator has failed, the participant will remain blocked. This is undesirable.

If the participants can communicate with each other, a more distributed termination protocol may be developed. The participant that times out can simply ask all the other participants to help it make a decision. Assuming that participant P_i is the one that times out, each of the other participants (P_j) responds in the following manner:

1. P_j is in the INITIAL state. This means that P_j has not yet voted and may not even have received the “prepare” message. It can therefore unilaterally abort the transaction and reply to P_i with a “vote-abort” message.
2. P_j is in the READY state. In this state P_j has voted to commit the transaction but has not received any word about the global decision. Therefore, it cannot help P_i to terminate the transaction.
3. P_j is in the ABORT or COMMIT states. In these states, either P_j has unilaterally decided to abort the transaction, or it has received the coordinator’s decision regarding global termination. It can, therefore, send P_i either a “vote-commit” or a “vote-abort” message.

Consider how the participant that times out (P_i) can interpret these responses. The following cases are possible:

1. P_i receives “vote-abort” messages from all P_j . This means that none of the other participants had yet voted, but they have chosen to abort the transaction unilaterally. Under these conditions, P_i can proceed to abort the transaction.
2. P_i receives “vote-abort” messages from some P_j , but some other participants indicate that they are in the READY state. In this case P_i can still go ahead and abort the transaction, since according to the global commit rule, the transaction cannot be committed and will eventually be aborted.
3. P_i receives notification from all P_j that they are in the READY state. In this case none of the participants knows enough about the fate of the transaction to terminate it properly.
4. P_i receives “global-abort” or “global-commit” messages from all P_j . In this case all the other participants have received the coordinator’s decision. Therefore, P_i can go ahead and terminate the transaction according to the messages

it receives from the other participants. Incidentally, note that it is not possible for some of the P_j to respond with a “global-abort” while others respond with “global-commit” since this cannot be the result of a legitimate execution of the 2PC protocol.

5. P_i receives “global-abort” or “global-commit” from some P_j , whereas others indicate that they are in the READY state. This indicates that some sites have received the coordinator’s decision while others are still waiting for it. In this case P_i can proceed as in case 4 above.

These five cases cover all the alternatives that a termination protocol needs to handle. It is not necessary to consider cases where, for example, one participant sends a “vote-abort” message while another one sends “global-commit.” This cannot happen in 2PC. During the execution of the 2PC protocol, no process (participant or coordinator) is more than one state transition apart from any other process. For example, if a participant is in the INITIAL state, all other participants are in either the INITIAL or the READY state. Similarly, the coordinator is either in the INITIAL or the WAIT state. Thus, all the processes in a 2PC protocol are said to be *synchronous within one state transition* [Skeen, 1981].

Note that in case 3 the participant processes stay blocked, as they cannot terminate a transaction. Under certain circumstances there may be a way to overcome this blocking. If during termination all the participants realize that only the coordinator site has failed, they can elect a new coordinator, which can restart the commit process. There are different ways of electing the coordinator. It is possible either to define a total ordering among all sites and elect the next one in order [Hammer and Shipman, 1980], or to establish a voting procedure among the participants [Garcia-Molina, 1982]. This will not work, however, if both a participant site and the coordinator site fail. In this case it is possible for the participant at the failed site to have received the coordinator’s decision and have terminated the transaction accordingly. This decision is unknown to the other participants; thus if they elect a new coordinator and proceed, there is the danger that they may decide to terminate the transaction differently from the participant at the failed site. It is clear that it is not possible to design termination protocols for 2PC that can guarantee non-blocking termination. The 2PC protocol is, therefore, a blocking protocol.

Since we had assumed a centralized communication structure in developing the 2PC algorithms in Algorithms 12.1 and 12.2, we will continue with the same assumption in developing the termination protocols. The portion of code that should be included in the timeout section of the coordinator and the participant 2PC algorithms is given in Algorithms 12.3 and 12.4, respectively.

12.5.1.2 Recovery Protocols

In the preceding section, we discussed how the 2PC protocol deals with failures from the perspective of the operational sites. In this section, we take the opposite viewpoint: we are interested in investigating protocols that a coordinator or participant can use

Algorithm 12.3: 2PC Coordinator Terminate

```

begin
  if in WAIT state then                                {coordinator is in ABORT state}
    | write abort record in the log ;
    | send “Global-abort” message to all the participants
  else                                                    {coordinator is in COMMIT state}
    | check for the last log record ;
    | if last log record = abort then
    |   | send “Global-abort” to all participants that have not responded
    | else
    |   | send “Global-commit” to all the participants that have not
    |   | responded
    | set timer ;
end

```

Algorithm 12.4: 2PC-Participant Terminate

```

begin
  if in INITIAL state then
    | write abort record in the log
  else
    | send “Vote-commit” message to the coordinator ;
    | reset timer
end

```

to recover their states when their sites fail and then restart. Remember that we would like these protocols to be independent. However, in general, it is not possible to design protocols that can guarantee independent recovery while maintaining the atomicity of distributed transactions. This is not surprising given the fact that the termination protocols for 2PC are inherently blocking.

In the following discussion, we again use the state transition diagram of Figure 12.14. Additionally, we make two interpretive assumptions: (1) the combined action of writing a record in the log and sending a message is assumed to be atomic, and (2) the state transition occurs after the transmission of the response message. For example, if the coordinator is in the WAIT state, this means that it has successfully written the *begin_commit* record in its log and has successfully transmitted the “prepare” command. This does not say anything, however, about successful completion of the message transmission. Therefore, the “prepare” message may never get to the participants, due to communication failures, which we discuss separately. The first assumption related to atomicity is, of course, unrealistic. However, it simplifies our discussion of fundamental failure cases. At the end of this section we show that the other cases that arise from the relaxation of this assumption can be handled by a combination of the fundamental failure cases.

Coordinator Site Failures.

The following cases are possible:

1. *The coordinator fails while in the INITIAL state.* This is before the coordinator has initiated the commit procedure. Therefore, it will start the commit process upon recovery.
2. *The coordinator fails while in the WAIT state.* In this case, the coordinator has sent the “prepare” command. Upon recovery, the coordinator will restart the commit process for this transaction from the beginning by sending the “prepare” message one more time.
3. *The coordinator fails while in the COMMIT or ABORT states.* In this case, the coordinator will have informed the participants of its decision and terminated the transaction. Thus, upon recovery, it does not need to do anything if all the acknowledgments have been received. Otherwise, the termination protocol is involved.

Participant Site Failures.

There are three alternatives to consider:

1. *A participant fails in the INITIAL state.* Upon recovery, the participant should abort the transaction unilaterally. Let us see why this is acceptable. Note that the coordinator will be in the INITIAL or WAIT state with respect to this transaction. If it is in the INITIAL state, it will send a “prepare” message and then move to the WAIT state. Because of the participant site’s failure, it will not receive the participant’s decision and will time out in that state. We have already discussed how the coordinator would handle timeouts in the WAIT state by globally aborting the transaction.
2. *A participant fails while in the READY state.* In this case the coordinator has been informed of the failed site’s affirmative decision about the transaction before the failure. Upon recovery, the participant at the failed site can treat this as a timeout in the READY state and hand the incomplete transaction over to its termination protocol.
3. *A participant fails while in the ABORT or COMMIT state.* These states represent the termination conditions, so, upon recovery, the participant does not need to take any special action.

Additional Cases.

Let us now consider the cases that may arise when we relax the assumption related to the atomicity of the logging and message sending actions. In particular, we assume

that a site failure may occur after the coordinator or a participant has written a log record but before it can send a message. For this discussion, the reader may wish to refer to Figure 12.10.

1. *The coordinator fails after the begin_commit record is written in the log but before the “prepare” command is sent.* The coordinator would react to this as a failure in the WAIT state (case 2 of the coordinator failures discussed above) and send the “prepare” command upon recovery.
2. *A participant site fails after writing the ready record in the log but before sending the “vote-commit” message.* The failed participant sees this as case 2 of the participant failures discussed before.
3. *A participant site fails after writing the abort record in the log but before sending the “vote-abort” message.* This is the only situation that is not covered by the fundamental cases discussed before. However, the participant does not need to do anything upon recovery in this case. The coordinator is in the WAIT state and will time out. The coordinator termination protocol for this state globally aborts the transaction.
4. *The coordinator fails after logging its final decision record (abort or commit), but before sending its “global-abort” or “global-commit” message to the participants.* The coordinator treats this as its case 3, while the participants treat it as a timeout in the READY state.
5. *A participant fails after it logs an abort or a commit record but before it sends the acknowledgment message to the coordinator.* The participant can treat this as its case 3. The coordinator will handle this by timeout in the COMMIT or ABORT state.

12.5.2 Three-Phase Commit Protocol

The three-phase commit protocol (3PC) [Skeen, 1981] is designed as a non-blocking protocol. We will see in this section that it is indeed non-blocking when failures are restricted to site failures.

Let us first consider the necessary and sufficient conditions for designing non-blocking atomic commitment protocols. A commit protocol that is synchronous within one state transition is non-blocking if and only if its state transition diagram contains neither of the following:

1. No state that is “adjacent” to both a commit and an abort state.
2. No non-committable state that is “adjacent” to a commit state ([Skeen, 1981; Skeen and Stonebraker, 1983]).

The term *adjacent* here means that it is possible to go from one state to the other with a single state transition.

Consider the COMMIT state in the 2PC protocol (see Figure 12.14). If any process is in this state, we know that all the sites have voted to commit the transaction. Such states are called *committable*. There are other states in the 2PC protocol that are *non-committable*. The one we are interested in is the READY state, which is non-committable since the existence of a process in this state does not imply that all the processes have voted to commit the transaction.

It is obvious that the WAIT state in the coordinator and the READY state in the participant 2PC protocol violate the non-blocking conditions we have stated above. Therefore, one might be able to make the following modification to the 2PC protocol to satisfy the conditions and turn it into a non-blocking protocol.

We can add another state between the WAIT (and READY) and COMMIT states which serves as a buffer state where the process is ready to commit (if that is the final decision) but has not yet committed. The state transition diagrams for the coordinator and the participant in this protocol are depicted in Figure 12.15. This is called the three-phase commit protocol (3PC) because there are three state transitions from the INITIAL state to a COMMIT state. The execution of the protocol between the coordinator and one participant is depicted in Figure 12.16. Note that this is identical to Figure 12.10 except for the addition of the PRECOMMIT state. Observe that 3PC is also a protocol where all the states are synchronous within one state transition. Therefore, the foregoing conditions for non-blocking 2PC apply to 3PC.

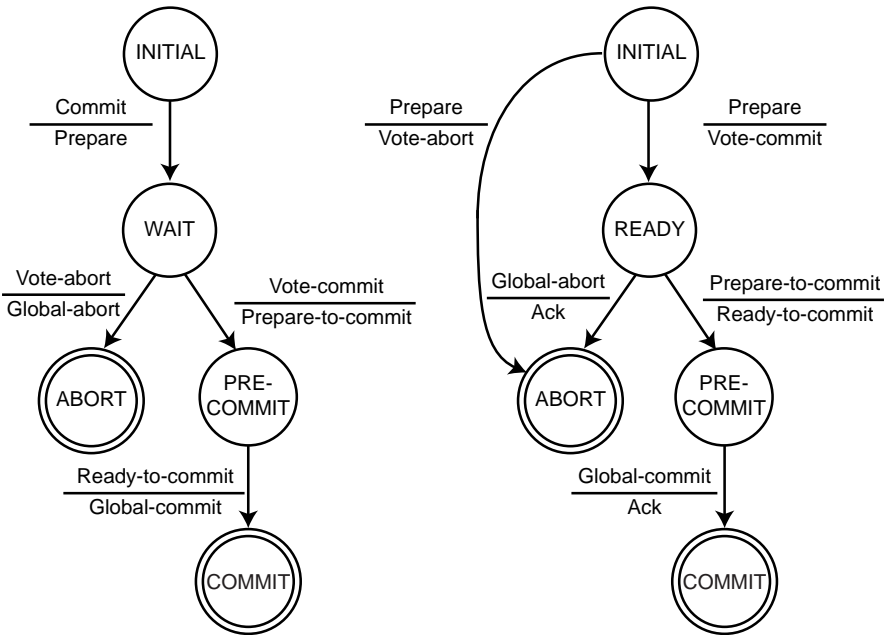


Fig. 12.15 State Transitions in 3PC Protocol

It is possible to design different 3PC algorithms depending on the communication topology. The one given in Figure 12.16 is centralized. It is also straightforward to design a distributed 3PC protocol. A linear 3PC protocol is somewhat more involved, so we leave it as an exercise.

12.5.2.1 Termination Protocol

As we did in discussing the termination protocols for handling timeouts in the 2PC protocol, let us investigate timeouts at each state of the 3PC protocol.

Coordinator Timeouts.

In 3PC, there are four states in which the coordinator can time out: WAIT, PRECOMMIT, COMMIT, or ABORT.

1. *Timeout in the WAIT state.* This is identical to the coordinator timeout in the WAIT state for the 2PC protocol. The coordinator unilaterally decides to abort the transaction. It therefore writes an abort record in the log and sends a “global-abort” message to all the participants that have voted to commit the transaction.
2. *Timeout in the PRECOMMIT state.* The coordinator does not know if the non-responding participants have already moved to the PRECOMMIT state. However, it knows that they are at least in the READY state, which means that they must have voted to commit the transaction. The coordinator can therefore move all participants to PRECOMMIT state by sending a “prepare-to-commit” message go ahead and globally commit the transaction by writing a commit record in the log and sending a “global-commit” message to all the operational participants.
3. *Timeout in the COMMIT (or ABORT) state.* The coordinator does not know whether the participants have actually performed the commit (abort) command. However, they are at least in the PRECOMMIT (READY) state (since the protocol is synchronous within one state transition) and can follow the termination protocol as described in case 2 or case 3 below. Thus the coordinator does not need to take any special action.

Participant Timeouts.

A participant can time out in three states: INITIAL, READY, and PRECOMMIT. Let us examine all of these cases.

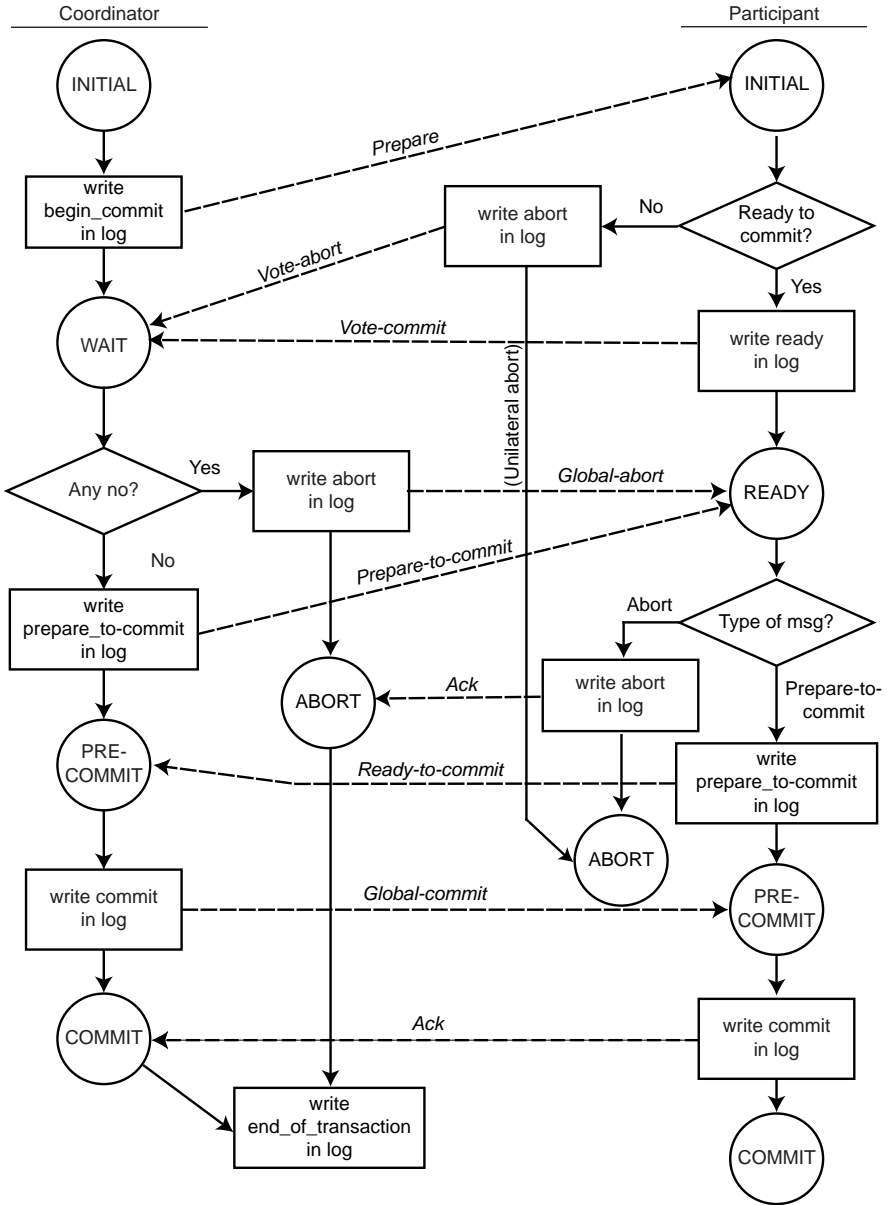


Fig. 12.16 3PC Protocol Actions

1. *Timeout in the INITIAL state.* This can be handled identically to the termination protocol of 2PC.
2. *Timeout in the READY state.* In this state the participant has voted to commit the transaction but does not know the global decision of the coordinator. Since communication with the coordinator is lost, the termination protocol proceeds by electing a new coordinator, as discussed earlier. The new coordinator then terminates the transaction according to a termination protocol that we discuss below.
3. *Timeout in the PRECOMMIT state.* In this state the participant has received the “prepare-to-commit” message and is awaiting the final “global-commit” message from the coordinator. This case is handled identically to case 2 above.

Let us now consider the possible termination protocols that can be adopted in the last two cases. There are various alternatives; let us consider a centralized one [Skeen, 1981]. We know that the new coordinator can be in one of three states: WAIT, PRECOMMIT, COMMIT or ABORT. It sends its own state to all the operational participants, asking them to assume that state. Any participant who has proceeded ahead of the new coordinator (which is possible since it may have already received and processed a message from the old coordinator) simply ignores the new coordinator’s message; others make their state transitions and send back the appropriate message. Once the new coordinator gets messages from the participants, it guides the participants toward termination as follows:

1. If the new coordinator is in the WAIT state, it will globally abort the transaction. The participants can be in the INITIAL, READY, ABORT, or PRECOMMIT states. In the first three cases, there is no problem. However, the participants in the PRECOMMIT state are expecting a “global-commit” message, but they get a “global-abort” instead. Their state transition diagram does not indicate any transition from the PRECOMMIT to the ABORT state. This transition is necessary for the termination protocol, so it should be added to the set of legal transitions that can occur during execution of the termination protocol.
2. If the new coordinator is in the PRECOMMIT state, the participants can be in the READY, PRECOMMIT or COMMIT states. No participant can be in ABORT state. The coordinator will therefore globally commit the transaction and send a “global-commit” message.
3. If the new coordinator is in the ABORT state, at the end of the first message all the participants will have moved into the ABORT state as well.

The new coordinator is not keeping track of participant failures during this process. It simply guides the operational sites toward termination. If some participants fail in the meantime, they will have to terminate the transaction upon recovery according to the methods discussed in the next section. Also, the new coordinator

may fail during the process; the termination protocol therefore needs to be reentrant in implementation.

This termination protocol is obviously non-blocking. The operational sites can properly terminate all the ongoing transactions and continue their operations. The proof of correctness of the algorithm is given in [Skeen, 1982b].

12.5.2.2 Recovery Protocols

There are some minor differences between the recovery protocols of 3PC and those of 2PC. We only indicate those differences.

1. *The coordinator fails while in the WAIT state.* This is the case we discussed at length in the earlier section on termination protocols. The participants have already terminated the transaction. Therefore, upon recovery, the coordinator has to ask around to determine the fate of the transaction.
2. *The coordinator fails while in the PRECOMMIT state.* Again, the termination protocol has guided the operational participants toward termination. Since it is now possible to move from the PRECOMMIT state to the ABORT state during this process, the coordinator has to ask around to determine the fate of the transaction.
3. *A participant fails while in the PRECOMMIT state.* It has to ask around to determine how the other participants have terminated the transaction.

One property of the 3PC protocol becomes obvious from this discussion. When using the 3PC protocol, we are able to terminate transactions without blocking. However, we pay the price that fewer cases of independent recovery are possible. This also results in more messages being exchanged during recovery.

12.6 Network Partitioning

In this section we consider how the network partitions can be handled by the atomic commit protocols that we discussed in the preceding section. Network partitions are due to communication line failures and may cause the loss of messages, depending on the implementation of the communication subnet. A partitioning is called a *simple partitioning* if the network is divided into only two components; otherwise, it is called *multiple partitioning*.

The termination protocols for network partitioning address the termination of the transactions that were active in each partition at the time of partitioning. If one can develop non-blocking protocols to terminate these transactions, it is possible for the sites in each partition to reach a termination decision (for a given transaction) which

is consistent with the sites in the other partitions. This would imply that the sites in each partition can continue executing transactions despite the partitioning.

Unfortunately, it is not in general possible to find non-blocking termination protocols in the presence of network partitions. Remember that our expectations regarding the reliability of the communication subnet are minimal. If a message cannot be delivered, it is simply lost. In this case it can be proven that no non-blocking atomic commitment protocol exists that is resilient to network partitioning [Skeen and Stonebraker, 1983]. This is quite a negative result since it also means that if network partitioning occurs, we cannot continue normal operations in all partitions, which limits the availability of the entire distributed database system. A positive counter result, however, indicates that it is possible to design non-blocking atomic commit protocols that are resilient to simple partitions. Unfortunately, if multiple partitions occur, it is again not possible to design such protocols [Skeen and Stonebraker, 1983].

In the remainder of this section we discuss a number of protocols that address network partitioning in non-replicated databases. The problem is quite different in the case of replicated databases, which we discuss in the next chapter.

In the presence of network partitioning of non-replicated databases, the major concern is with the termination of transactions that were active at the time of partitioning. Any new transaction that accesses a data item that is stored in another partition is simply blocked and has to await the repair of the network. Concurrent accesses to the data items within one partition can be handled by the concurrency control algorithm. The significant problem, therefore, is to ensure that the transaction terminates properly. In short, the network partitioning problem is handled by the commit protocol, and more specifically, by the termination and recovery protocols.

The absence of non-blocking protocols that would guarantee atomic commitment of distributed transactions points to an important design decision. We can either permit all the partitions to continue their normal operations and accept the fact that database consistency may be compromised, or we guarantee the consistency of the database by employing strategies that would permit operation in one of the partitions while the sites in the others remain blocked. This decision problem is the premise of a classification of partition handling strategies. We can classify the strategies as *pessimistic* or *optimistic* [Davidson et al., 1985]. Pessimistic strategies emphasize the consistency of the database, and would therefore not permit transactions to execute in a partition if there is no guarantee that the consistency of the database can be maintained. Optimistic approaches, on the other hand, emphasize the availability of the database even if this would cause inconsistencies.

The second dimension is related to the correctness criterion. If serializability is used as the fundamental correctness criterion, such strategies are called *syntactic* since the serializability theory uses only syntactic information. However, if we use a more abstract correctness criterion that is dependent on the semantics of the transactions or the database, the strategies are said to be *semantic*.

Consistent with the correctness criterion that we have adopted in this book (serializability), we consider only syntactic approaches in this section. The following two sections outline various syntactic strategies for non-replicated databases.

All the known termination protocols that deal with network partitioning in the case of non-replicated databases are pessimistic. Since the pessimistic approaches emphasize the maintenance of database consistency, the fundamental issue that we need to address is which of the partitions can continue normal operations. We consider two approaches.

12.6.1 *Centralized Protocols*

Centralized termination protocols are based on the centralized concurrency control algorithms discussed in Chapter 11. In this case, it makes sense to permit the operation of the partition that contains the central site, since it manages the lock tables.

Primary site techniques are centralized with respect to each data item. In this case, more than one partition may be operational for different queries. For any given query, only the partition that contains the primary site of the data items that are in the write set of that transaction can execute that transaction.

Both of these are simple approaches that would work well, but they are dependent on the concurrency control mechanism employed by the distributed database manager. Furthermore, they expect each site to be able to differentiate network partitioning from site failures properly. This is necessary since the participants in the execution of the commit protocol react differently to the different types of failures.

12.6.2 *Voting-based Protocols*

Voting as a technique for managing concurrent data accesses has been proposed by a number of researchers. A straightforward voting with majority was first proposed in [Thomas, 1979] as a concurrency control method for fully replicated databases. The fundamental idea is that a transaction is executed if a majority of the sites vote to execute it.

The idea of majority voting has been generalized to voting with *quorums*. Quorum-based voting can be used as a replica control method (as we discuss in the next chapter), as well as a commit method to ensure transaction atomicity in the presence of network partitioning. In the case of non-replicated databases, this involves the integration of the voting principle with commit protocols. We present a specific proposal along this line [Skeen, 1982b].

Every site in the system is assigned a vote V_i . Let us assume that the total number of votes in the system is V , and the abort and commit quorums are V_a and V_c , respectively. Then the following rules must be obeyed in the implementation of the commit protocol:

1. $V_a + V_c > V$, where $0 \leq V_a, V_c \leq V$.
2. Before a transaction commits, it must obtain a commit quorum V_c .

3. Before a transaction aborts, it must obtain an abort quorum V_a .

The first rule ensures that a transaction cannot be committed and aborted at the same time. The next two rules indicate the votes that a transaction has to obtain before it can terminate one way or the other.

The integration of these rules into the 3PC protocol requires a minor modification of the third phase. For the coordinator to move from the PRECOMMIT state to the COMMIT state, and to send the “global-commit” command, it is necessary for it to have obtained a commit quorum from the participants. This would satisfy rule 2. Note that we do not need to implement rule 3 explicitly. This is due to the fact that a transaction which is in the WAIT or READY state is willing to abort the transaction. Therefore, an abort quorum already exists.

Let us now consider the termination of transactions in the presence of failures. When a network partitioning occurs, the sites in each partition elect a new coordinator, similar to the 3PC termination protocol in the case of site failures. There is a fundamental difference, however. It is not possible to make the transition from the WAIT or READY state to the ABORT state in one state transition, for a number of reasons. First, more than one coordinator is trying to terminate the transaction. We do not want them to terminate differently or the transaction execution will not be atomic. Therefore, we want the coordinators to obtain an abort quorum explicitly. Second, if the newly elected coordinator fails, it is not known whether a commit or abort quorum was reached. Thus it is necessary that participants make an explicit decision to join either the commit or the abort quorum and not change their votes afterward. Unfortunately, the READY (or WAIT) state does not satisfy these requirements. Thus we introduce another state, PREABORT, between the READY and ABORT states. The transition from the PREABORT state to the ABORT state requires an abort quorum. The state transition diagram is given in Figure 12.17.

With this modification, the termination protocol works as follows. Once a new coordinator is elected, it requests all participants to report their local states. Depending on the responses, it terminates the transaction as follows:

1. If at least one participant is in the COMMIT state, the coordinator decides to commit the transaction and sends a “global-commit” message to all the participants.
2. If at least one participant is in the ABORT state, the coordinator decides to abort the transaction and sends a “global-abort” message to all the participants.
3. If a commit quorum is reached by the votes of participants in the PRECOMMIT state, the coordinator decides to commit the transaction and sends a “global-commit” message to all the participants.
4. If an abort quorum is reached by the votes of participants in the PREABORT state, the coordinator decides to abort the transaction and sends a “global-abort” message to all the participants.
5. If case 3 does not hold but the sum of the votes of the participants in the PRECOMMIT and READY states are enough to form a commit quorum, the

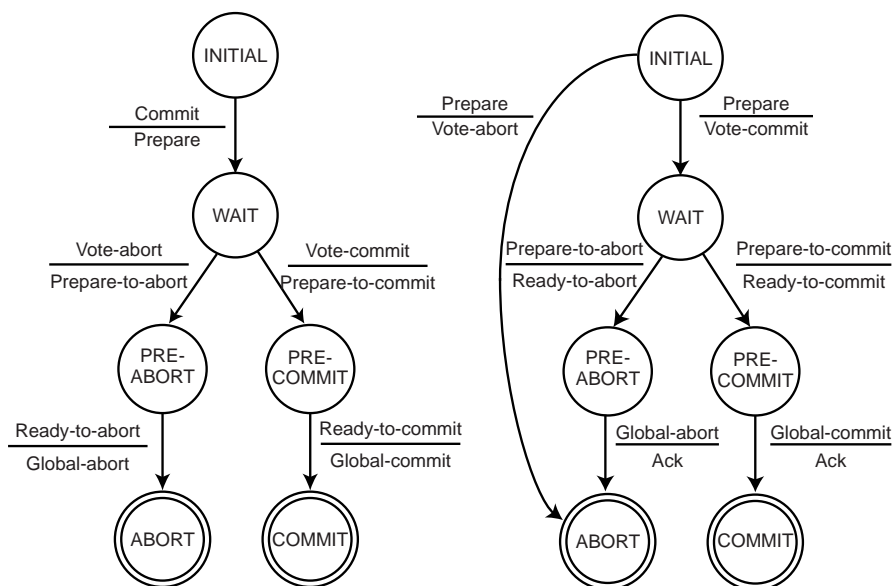


Fig. 12.17 State Transitions in Quorum 3PC Protocol

coordinator moves the participants to the PRECOMMIT state by sending a “prepare-to-commit” message. The coordinator then waits for case 3 to hold.

6. Similarly, if case 4 does not hold but the sum of the votes of the participants in the PREABORT and READY states are enough to form an abort quorum, the coordinator moves the participants to the PREABORT state by sending a “prepare-to-abort” message. The coordinator then waits for case 4 to hold.

Two points are important about this quorum-based commit algorithm. First, it is blocking; the coordinator in a partition may not be able to form either an abort or a commit quorum if messages get lost or multiple partitionings occur. This is hardly surprising given the theoretical bounds that we discussed previously. The second point is that the algorithm is general enough to handle site failures as well as network partitioning. Therefore, this modified version of 3PC can provide more resiliency to failures.

The recovery protocol that can be used in conjunction with the above-discussed termination protocol is very simple. When two or more partitions merge, the sites that are part of the new larger partition simply execute the termination protocol. That is, a coordinator is elected to collect votes from all the participants and try to terminate the transaction.

12.7 Architectural Considerations

In previous sections we have discussed the atomic commit protocols at an abstract level. Let us now look at how these protocols can be implemented within the framework of our architectural model. This discussion involves specification of the interface between the concurrency control algorithms and the reliability protocols. In that sense, the discussions of this chapter relate to the execution of **commit**, **abort**, and **recover** commands.

Unfortunately, it is quite difficult to specify precisely the execution of these commands. The difficulty is twofold. First, a significantly more detailed model of the architecture than the one we have presented needs to be considered for correct implementation of these commands. Second, the overall scheme of implementation is quite dependent on the recovery procedures that the local recovery manager implements. For example, implementation of the 2PC protocol on top of a LRM that employs a no-fix/no-flush recovery scheme is quite different from its implementation on top of a LRM that employs a fix/flush recovery scheme. The alternatives are simply too numerous. We therefore confine our architectural discussion to three areas: implementation of the coordinator and participant concepts for the commit and replica control protocols within the framework of the transaction manager-scheduler-local recovery manager architecture, the coordinator's access to the database log, and the changes that need to be made in the local recovery manager operations.

One possible implementation of the commit protocols within our architectural model is to perform both the coordinator and participant algorithms within the transaction managers at each site. This provides some uniformity in executing the distributed commit operations. However, it entails unnecessary communication between the participant transaction manager and its scheduler; this is because the scheduler has to decide whether a transaction can be committed or aborted. Therefore, it may be preferable to implement the coordinator as part of the transaction manager and the participant as part of the scheduler. Of course, the replica control protocol is implemented as part of the transaction manager as well. If the scheduler implements a strict concurrency control algorithm (i.e., does not allow cascading aborts), it will be ready automatically to commit the transaction when the prepare message arrives. Proof of this claim is left as an exercise. However, even this alternative of implementing the coordinator and the participant outside the data processor has problems. The first issue is database log management. Recall from Section 12.3 that the database log is maintained by the LRM and the buffer manager. However, implementation of the commit protocol as described here requires the transaction manager and the scheduler to access the log as well. One possible solution to this problem is to maintain a commit log (which could be called the *distributed transaction log* [Bernstein et al., 1987; Lampson and Sturgis, 1976]) that is accessed by the transaction manager and is separate from the database log that the LRM and buffer manager maintain. The other alternative is to write the commit protocol records into the same database log. This second alternative has a number of advantages. First, only one log is maintained; this simplifies the algorithms that have to be implemented in order to save log records on stable storage. More important, the recovery from failures in a distributed database

requires the cooperation of the local recovery manager and the scheduler (i.e., the participant). A single database log can serve as a central repository of recovery information for both these components.

A second problem associated with implementing the coordinator within the transaction manager and the participant as part of the scheduler has to be with integration with the concurrency control protocols. This implementation is based on the schedulers determining whether a transaction can be committed. This is fine for distributed concurrency control algorithms where each site is equipped with a scheduler. However, in centralized protocols such as the centralized 2PL, there is only one scheduler in the system. In this case, the participants may be implemented as part of the data processors (more precisely, as part of local recovery managers), requiring modification to both the algorithms implemented by the LRM and, possibly, to the execution of the 2PC protocol. We leave the details to exercises.

Storing the commit protocol records in the database log maintained by the LRM and the buffer manager requires some changes to the LRM algorithms. This is the third architectural issue we address. Unfortunately, these changes are dependent on the type of algorithm that the LRM uses. In general, however, the LRM algorithms have to be modified to handle separately the prepare command and global commit (or global abort) decisions. Furthermore, upon recovery, the LRM should be modified to read the database log and to inform the scheduler as to the state of each transaction, in order that the recovery procedures discussed before can be followed. Let us take a more detailed look at this function of the LRM.

The LRM first has to determine whether the failed site is the host of the coordinator or of a participant. This information can be stored together with the `begin_transaction` record. The LRM then has to search for the last record written in the log record during execution of the commit protocol. If it cannot even find a `begin_commit` record (at the coordinator site) or an abort or commit record (at the participant sites), the transaction has not started to commit. In this case, the LRM can continue with its recovery procedure as we discussed in Section 12.3.3. However, if the commit process has started, the recovery has to be handed over to the coordinator. Therefore, the LRM sends the last log record to the scheduler.

12.8 Conclusion

In this chapter we discussed the reliability aspects of distributed transaction management. The studied algorithms (2PC and 3PC) guarantee the atomicity and durability of distributed transactions even when failures occur. One of these algorithms (3PC) can be made non-blocking, which would permit each site to continue its operation without waiting for recovery of the failed site. An unfortunate result that we presented relates to network partitioning. It is not possible to design protocols that guarantee the atomicity of distributed transactions and permit each partition of the distributed system to continue its operation under the assumptions made in this chapter with respect to the functionality of the communication subnet. The performance of the

distributed commit protocols with respect to the overhead they add to the concurrency control algorithms is an interesting issue. Some studies have addressed this issue [Dwork and Skeen, 1983; Wolfson, 1987].

A final point that should be stressed is the following. We have considered only failures that are attributable to errors. In other words, we assumed that every effort was made to design and implement the systems (hardware and software), but that because of various faults in the components, the design, or the operating environment, they failed to perform properly. Such failures are called *failures of omission*. There is another class of failures, called *failures of commission*, where the systems may not have been designed and implemented so that they would work properly. The difference is that in the execution of the 2PC protocol, for example, if a participant receives a message from the coordinator, it treats this message as correct: the coordinator is operational and is sending the participant a correct message to go ahead and process. The only failure that the participant has to worry about is if the coordinator fails or if its messages get lost. These are failures of omission. If, on the other hand, the messages that a participant receives cannot be trusted, the participant also has to deal with failures of commission. For example, a participant site may pretend to be the coordinator and may send a malicious message. We have not discussed reliability measures that are necessary to cope with these types of failures. The techniques that address failures of commission are typically called *byzantine agreement*.

12.9 Bibliographic Notes

There are numerous books on the reliability of computer systems. These include [Anderson and Lee, 1981; Anderson and Randell, 1979; Avizienis et al., 1987; Longbottom, 1980; Gibbons, 1976; Pradhan, 1986; Siewiorek and Swarz, 1982], and [Shrivastava, 1985]. In addition, the survey paper [Randell et al., 1978] addresses the same issues. Myers [1976] specifically addresses software reliability. An important software fault tolerance technique that we have not discussed in this chapter is exception handling. This issue is treated in [Cristian, 1982, 1985], and [Cristian, 1987]. Jr and Malek [1988] surveys the existing software tools for reliability measurement.

The fundamental principles employed in fault-tolerant systems are *redundancy* in system components and *modularization* of the design. These two concepts are utilized in typical systems by means of *fail-stop modules* (also called *fail-fast* [Gray, 1985]) and *process pairs*. A fail-stop module constantly monitors itself, and when it detects a fault, shuts itself down automatically [Schlichting and Schneider, 1983]. Process pairs provide fault tolerance by duplicating software modules. The idea is to eliminate single points of failure by implementing each system service as two processes that communicate and cooperate in providing the service. One of these processes is called the *primary* and the other the *backup*. Both the primary and the backup are typically implemented as fail-stop modules that cooperate in providing a service. There are a number of different ways of implementing process pairs, depending on the mode of communication between the primary and the backup.

The five common types are *lock-step*, *automatic checkpointing*, *state checkpointing*, *delta checkpointing*, and *persistent* process pairs. With respect to our discussion of process pairs, the lock-step process pair approach is implemented in the Stratus/32 systems ([Computers, 1982; Kim, 1984]) for hardware processes. An automatic checkpointing process pairs approach is used in the Auras (TM) operating system for Aurogen computers ([Borg et al., 1983; Gastonian, 1983]). State checkpointing has been used in earlier versions of the Tandem operating systems [Bartlett, 1978, 1981], which have later utilized the delta checkpointing approach [Borr, 1984]. A review of different implementations appears in [Gray, 1985].

More detailed material on the functions of the local recovery manager discussed in Section 12.3 can be found in [Verhofstadt, 1978; Härder and Reuter, 1983]. Implementation of the local recovery functions in System R is described in [Gray et al., 1981].

Kohler [1981] presents a general discussion of the reliability issues in distributed database systems. Hadzilacos [1988] is a formalization of the reliability concept. The reliability aspects of System R* are given in [Traiger et al., 1982], whereas Hammer and Shipman [1980] describe the same for the SDD-1 system.

The two-phase commit protocol is first described in [Gray, 1979]. Modifications to it are presented in [Mohan and Lindsay, 1983]. The definition of three-phase commit is due to Skeen [1981, 1982a]. Formal results on the existence of non-blocking termination protocols is due to Skeen and Stonebraker [1983].

Replication and replica control protocols have been the subject of significant research in recent years. This work is summarized very well in [Helal et al., 1997]. Replica control protocols that deal with network partitioning are surveyed in [Davidson et al., 1985]. Besides the algorithms we have described here, some notable others are given in [Davidson, 1984; Eager and Sevcik, 1983; Herlihy, 1987; Minoura and Wiederhold, 1982; Skeen and Wright, 1984; Wright, 1983]. These algorithms are generally called *static* since the vote assignments and read/write quorums are fixed a priori. An analysis of one such protocol (such analyses are rare) is given in [Kumar and Segev, 1993]. Examples of *dynamic replication protocols* are in [Jajodia and Mutchler, 1987; Barbara et al., 1986, 1989] among others. It is also possible to change the way data are replicated. Such protocols are called *adaptive* and one example is described in [Wolfson, 1987]. An interesting replication algorithm based on economic models is described in [Sidell et al., 1996].

Our discussion of checkpointing has been rather short. Further treatment of the issue can be found in [Bhargava and Lian, 1988; Dadam and Schlageter, 1980; Schlageter and Dadam, 1980; Kuss, 1982; Ng, 1988; Ramanathan and Shin, 1988]. Byzantine agreement is surveyed in [Strong and Dolev, 1983] and is discussed in [Babaoglu, 1987; Pease et al., 1980].

Exercises

Problem 12.1. Briefly describe the various implementations of the process pairs concept. Comment on how process pairs may be useful in implementing a fault-tolerant distributed DBMS.

Problem 12.2 (*). Discuss the site failure termination protocol for 2PC using a distributed communication topology.

Problem 12.3 (*).

Design a 3PC protocol using the linear communication topology.

Problem 12.4 (*). In our presentation of the centralized 3PC termination protocol, the first step involves sending the coordinator's state to all participants. The participants move to new states according to the coordinator's state. It is possible to design the termination protocol such that the coordinator, instead of sending its own state information to the participants, asks the participants to send their state information to the coordinator. Modify the termination protocol to function in this manner.

Problem 12.5 ().** In Section 12.7 we claimed that a scheduler which implements a strict concurrency control algorithm will always be ready to commit a transaction when it receives the coordinator's "prepare" message. Prove this claim.

Problem 12.6 ().** Assuming that the coordinator is implemented as part of the transaction manager and the participant as part of the scheduler, give the transaction manager, scheduler, and the local recovery manager algorithms for a non-replicated distributed DBMS under the following assumptions.

- (a) The scheduler implements a distributed (strict) two-phase locking concurrency control algorithm.
- (b) The commit protocol log records are written to a central database log by the LRM when it is called by the scheduler.
- (c) The LRM may implement any of the protocols that have been discussed in Section 12.3.3. However, it is modified to support the distributed recovery procedures as we discussed in Section 12.7.

Problem 12.7 (*). Write the detailed algorithms for the no-fix/no-flush local recovery manager.

Problem 12.8 ().** Assume that

- (a) The scheduler implements a centralized two-phase locking concurrency control,
- (b) The LRM implements no-fix/no-flush protocol.

Give detailed algorithms for the transaction manager, scheduler, and local recovery managers.

Chapter 13

Data Replication

As we discussed in previous chapters, distributed databases are typically replicated. The purposes of replication are multiple:

1. **System availability.** As discussed in Chapter 1, distributed DBMSs may remove single points of failure by replicating data, so that data items are accessible from multiple sites. Consequently, even when some sites are down, data may be accessible from other sites.
2. **Performance.** As we have seen previously, one of the major contributors to response time is the communication overhead. Replication enables us to locate the data closer to their access points, thereby localizing most of the access that contributes to a reduction in response time.
3. **Scalability.** As systems grow geographically and in terms of the number of sites (consequently, in terms of the number of access requests), replication allows for a way to support this growth with acceptable response times.
4. **Application requirements.** Finally, replication may be dictated by the applications, which may wish to maintain multiple data copies as part of their operational specifications.

Although data replication has clear benefits, it poses the considerable challenge of keeping different copies synchronized. We will discuss this shortly, but let us first consider the execution model in replicated databases. Each replicated data item x has a number of copies x_1, x_2, \dots, x_n . We will refer to x as the *logical data item* and to its copies (or *replicas*)¹ as *physical data items*. If replication transparency is to be provided, user transactions will issue read and write operations on the logical data item x . The replica control protocol is responsible for mapping these operations to reads and writes on the physical data items x_1, \dots, x_n . Thus, the system behaves as if there is a single copy of each data item – referred to as *single system image* or *one-copy equivalence*. The specific implementation of the Read and Write interfaces

¹ In this chapter, we use the terms “replica”, “copy”, and “physical data item” interchangeably.

of the transaction monitor differ according to the specific replication protocol, and we will discuss these differences in the appropriate sections.

There are a number of decisions and factors that impact the design of replication protocols. Some of these were discussed in previous chapters, while others will be discussed here.

- **Database design.** As discussed in Chapter 3, a distributed database may be fully or partially replicated. In the case of a partially replicated database, the number of physical data items for each logical data item may vary, and some data items may even be non-replicated. In this case, transactions that access only non-replicated data items are *local transactions* (since they can be executed locally at one site) and their execution typically does not concern us here. Transactions that access replicated data items have to be executed at multiple sites and they are *global transactions*.
- **Database consistency.** When global transactions update copies of a data item at different sites, the values of these copies may be different at a given point in time. A replicated database is said to be in a *mutually consistent* state if all the replicas of each of its data items have identical values. What differentiates different mutual consistency criteria is how tightly synchronized replicas have to be. Some ensure that replicas are mutually consistent when an update transaction commits, thus, they are usually called *strong consistency* criteria. Others take a more relaxed approach, and are referred to as *weak consistency* criteria.
- **Where updates are performed.** A fundamental design decision in designing a replication protocol is where the database updates are first performed [Gray et al., 1996]. The techniques can be characterized as *centralized* if they perform updates first on a *master* copy, versus *distributed* if they allow updates over any replica. Centralized techniques can be further identified as *single master* when there is only one master database copy in the system, or *primary copy* where the master copy of each data item may be different².
- **Update propagation.** Once updates are performed on a replica (master or otherwise), the next decision is how updates are propagated to the others. The alternatives are identified as *eager* versus *lazy* [Gray et al., 1996]. Eager techniques perform all of the updates within the context of the global transaction that has initiated the write operations. Thus, when the transaction commits, its updates will have been applied to all of the copies. Lazy techniques, on the other hand, propagate the updates sometime after the initiating transaction has committed. Eager techniques are further identified according to when they push each write to the other replicas – some push each write operation individually, others batch the writes and propagate them at the commit point.

² Centralized techniques are referred to, in the literature, as *single master*, while distributed ones are referred to as *multi-master* or *update anywhere*. These terms, in particular “single master”, are confusing, since they refer to alternative architectures for implementing centralized protocols (more on this in Section 13.2.3). Thus, we prefer the more descriptive terms “centralized” and “distributed”.

- **Degree of replication transparency.** Certain replication protocols require each user application to know the master site where the transaction operations are to be submitted. These protocols provide only *limited replication transparency* to user applications. Other protocols provide *full replication transparency* by involving the Transaction Manager (TM) at each site. In this case, user applications submit transactions to their local TMs rather than the master site.

We discuss consistency issues in replicated databases in Section 13.1, and analyze centralized versus distributed update application as well as update propagation alternatives in Section 13.2. This will lead us to a discussion of the specific protocols in Section 13.3. In Section 13.4, we discuss the use of group communication primitives in reducing the messaging overhead of replication protocols. In these sections, we will assume that no failures occur so that we can focus on the replication protocols. We will then introduce failures and investigate how protocols are revised to handle failures (Section 13.5). Finally, in Section 13.6, we discuss how replication services can be provided in multidatabase systems (i.e., outside the component DBMSs).

13.1 Consistency of Replicated Databases

There are two issues related to consistency of a replicated database. One is mutual consistency, as discussed above, that deals with the convergence of the values of physical data items corresponding to one logical data item. The second is transaction consistency as we discussed in Chapter 11. Serializability, which we introduced as the transaction consistency criterion needs to be recast in the case of replicated databases. In addition, there are relationships between mutual consistency and transaction consistency. In this section we first discuss mutual consistency approaches and then focus on the redefinition of transaction consistency and its relationship to mutual consistency.

13.1.1 Mutual Consistency

As indicated earlier, mutual consistency criteria for replicated databases can either be strong or weak. Each is suitable for different classes of applications with different consistency requirements.

Strong mutual consistency criteria require that all copies of a data item have the same value at the end of the execution of an update transaction. This is achieved by a variety of means, but the execution of 2PC at the commit point of an update transaction is a common way to achieve strong mutual consistency.

Weak mutual consistency criteria do not require the values of replicas of a data item to be identical when an update transaction terminates. What is required is that, if the update activity ceases for some time, the values *eventually* become identical. This is commonly referred to as *eventual consistency*, which refers to the fact that

replica values may diverge over time, but will eventually converge. It is hard to define this concept formally or precisely, although the following definition is probably as precise as one can hope to get [Saito and Shapiro, 2005]:

“A replicated [data item] is *eventually consistent* when it meets the following conditions, assuming that all replicas start from the same initial state.

- At any moment, for each replica, there is a prefix of the [history] that is equivalent to a prefix of the [history] of every other replica. We call this a *committed prefix* for the replica.
- The committed prefix of each replica grows monotonically over time.
- All non-aborted operations in the committed prefix satisfy their preconditions.
- For every submitted operation α , either α or [its abort] will eventually be included in the committed prefix.”

It should be noted that this definition of eventual consistency is rather strong – in particular the requirements that history prefixes are the same at any given moment and that the committed prefix grows monotonically. Many systems that claim to provide eventual consistency would violate these requirements.

Epsilon serializability (ESR) [Pu and Leff, 1991; Ramamritham and Pu, 1995] allows a query to see inconsistent data while replicas are being updated, but requires that the replicas converge to a one-copy serializable state once the updates are propagated to all of the copies. It bounds the error on the read values by an epsilon (ϵ) value (hence the name), which is defined in terms of the number of updates (write operations) that a query “misses”. Given a read-only transaction (query) T_Q , let T_U be all the update transactions that are executing concurrently with T_Q . If $RS(T_Q) \cap WS(T_U) \neq \emptyset$ (T_Q is reading some copy of some data items while T_U is updating (possibly a different) copy of those data items) then there is a read-write conflict and T_Q may be reading inconsistent data. The inconsistency is bounded by the changes performed by T_U . Clearly, ESR does not sacrifice database consistency, but only allows read-only transactions (queries) to read inconsistent data. For this reason, it has been claimed that ESR does not weaken database consistency, but “stretches” it [Wu et al., 1997].

Other looser bounds have also been discussed. It has even been suggested that users should be allowed to specify *freshness constraints* that are suitable for particular applications and the replication protocols should enforce these [Pacitti and Simon, 2000; Röhm et al., 2002b; Bernstein et al., 2006]. The types of freshness constraints that can be specified are the following:

- Time-bound constraints. Users may accept divergence of physical copy values up to a certain time: x_i may reflect the value of an update at time t while x_j may reflect the value at $t - \Delta$ and this may be acceptable.
- Value-bound constraints. It may be acceptable to have values of all physical data items within a certain range of each other. The user may consider the database to be mutually consistent if the values do not diverge more than a certain amount (or percentage).

- Drift constraints on multiple data items. For transactions that read multiple data items, users may be satisfied if the time drift between the update timestamps of two data items is less than a threshold (i.e., they were updated within that threshold) or, in the case of aggregate computation, if the aggregate computed over a data item is within a certain range of the most recent value (i.e., even if the individual physical copy values may be more out of sync than this range, as long as a particular aggregate computation is within range, it may be acceptable).

An important criterion in analyzing protocols that employ criteria that allow replicas to diverge is *degree of freshness*. The degree of freshness of a given replica r_i at time t is defined as the proportion of updates that have been applied at r_i at time t to the total number of updates [Pacitti et al., 1998, 1999].

13.1.2 Mutual Consistency versus Transaction Consistency

Mutual consistency, as we have defined it here, and transactional consistency as we discussed in Chapter 11 are related, but different. Mutual consistency refers to the replicas converging to the same value, while transaction consistency requires that the global execution history be serializable. It is possible for a replicated DBMS to ensure that data items are mutually consistent when a transaction commits, but the execution history may not be globally serializable. This is demonstrated in the following example.

Example 13.1. Consider three sites (A, B, and C) and three data items (x, y, z) that are distributed as follows: Site A hosts x , Site B hosts x, y , Site C hosts x, y, z . We will use site identifiers as subscripts on the data items to refer to a particular replica.

Now consider the following three transactions:

T_1 : $x \leftarrow 20$	T_2 : Read(x)	T_3 : Read(x)
Write(x)	$y \leftarrow x + y$	Read(y)
Commit	Write(y)	$z \leftarrow (x * y) / 100$
	Commit	Write(z)
		Commit

Note that T_1 's Write has to be executed at all three sites (since x is replicated at all three sites), T_2 's Write has to be executed at B and C, and T_3 's Write has to be executed only at C. We are assuming a transaction execution model where transactions can read their local replicas, but have to update all of the replicas.

Assume that the following three local histories are generated at the sites:

$$\begin{aligned}
 H_A &= \{W_1(x_A), C_1\} \\
 H_B &= \{W_1(x_B), C_1, R_2(x_B), W_2(y_B), C_2\} \\
 H_C &= \{W_2(y_C), C_2, R_3(x_C), R_3(y_C), W_3(z_C), C_3, W_1(x_C), C_1\}
 \end{aligned}$$

The serialization order in H_B is $T_1 \rightarrow T_2$ while in H_C it is $T_2 \rightarrow T_3 \rightarrow T_1$. Therefore, the global history is not serializable. However, the database is mutually consistent. Assume, for example, that initially $x_A = x_B = x_C = 10$, $y_B = y_C = 15$, and $z_C = 7$. With the above histories, the final values will be $x_A = x_B = x_C = 20$, $y_B = y_C = 35$, $z_C = 3.5$. All the physical copies (replicas) have indeed converged to the same value. ♦

Of course, it is possible for both the database to be mutually inconsistent, and the execution history to be globally non-serializable, as demonstrated in the following example.

Example 13.2. Consider two sites (A and B), and one data item (x) that is replicated at both sites (x_A and x_B). Further consider the following two transactions:

T_1 : Read(x)	T_2 : Read(x)
$x \leftarrow x + 5$	$x \leftarrow x * 10$
Write(x)	Write(x)
Commit	Commit

Assume that the following two local histories are generated at the two sites (again using the execution model of the previous example):

$$H_A = \{R_1(x_A), W_1(x_A), C_1, R_2(x_A), W_2(x_A), C_2\}$$

$$H_B = \{R_2(x_B), W_2(x_B), C_2, R_1(x_B), W_1(x_B), C_1\}$$

Although both of these histories are serial, they serialize T_1 and T_2 in reverse order; thus the global history is not serializable. Furthermore, the mutual consistency is violated as well. Assume that the value of x prior to the execution of these transactions was 1. At the end of the execution of these schedules, the value of x is 60 at site A while it is 15 at site B. Thus, in this example, the global history is non-serializable, **and** the databases are mutually inconsistent. ♦

Given the above observation, the transaction consistency criterion given in Chapter 11 is extended in replicated databases to define *one-copy serializability*. One-copy serializability (ISR) states that the effects of transactions on replicated data items should be the same as if they had been performed one at-a-time on a single set of data items. In other words, the histories are equivalent to some serial execution over non-replicated data items.

Snapshot isolation that we introduced in Chapter 11 has been extended for replicated databases [Lin et al., 2005] and used as an alternative transactional consistency criterion within the context of replicated databases [Plattner and Alonso, 2004; Daudjee and Salem, 2006]. Similarly, a weaker form of serializability, called *relaxed concurrency (RC-) serializability* has been defined that corresponds to “read committed” isolation level (Section 10.2.3) [Bernstein et al., 2006].

13.2 Update Management Strategies

As discussed earlier, the replication protocols can be classified according to when the updates are propagated to copies (eager versus lazy) and where updates are allowed to occur (centralized versus distributed). These two decisions are generally referred to as *update management* strategies. In this section, we discuss these alternatives before we present protocols in the next section.

13.2.1 Eager Update Propagation

The eager update propagation approaches apply the changes to all the replicas within the context of the update transaction. Consequently, when the update transaction commits, all the copies have the same value. Typically, eager propagation techniques use 2PC at commit point, but, as we will see later, alternatives are possible to achieve agreement. Furthermore, eager propagation may use *synchronous* propagation of each update by applying it on all the replicas at the same time (when the *Write* is issued), or *deferred* propagation whereby the updates are applied to one replica when they are issued, but their application on the other replicas is batched and deferred to the end of the transaction. Deferred propagation can be implemented by including the updates in the “Prepare-to-Commit” message at the start of 2PC execution.

Eager techniques typically enforce strong mutual consistency criteria. Since all the replicas are mutually consistent at the end of an update transaction, a subsequent read can read from any copy (i.e., one can map a $Read(x)$ to $Read(x_i)$ for any x_i). However, a $Write(x)$ has to be applied to all x_i (i.e., $Write(x_i), \forall x_i$). Thus, protocols that follow eager update propagation are known as *read-one/write-all* (ROWA) protocols.

The advantages of eager update propagation are threefold. First, they typically ensure that mutual consistency is enforced using 1SR; therefore, there are no transactional inconsistencies. Second, a transaction can read a local copy of the data item (if a local copy is available) and be certain that an up-to-date value is read. Thus, there is no need to do a remote read. Finally, the changes to replicas are done atomically; thus recovery from failures can be governed by the protocols we have already studied in the previous chapter.

The main disadvantage of eager update propagation is that a transaction has to update all the copies before it can terminate. This has two consequences. First, the response time performance of the update transaction suffers, since it typically has to participate in a 2PC execution, and because the update speed is restricted by the slowest machine. Second, if one of the copies is unavailable, then the transaction cannot terminate since all the copies need to be updated. As discussed in Chapter 12, if it is possible to differentiate between site failures and network failures, then one can terminate the transaction as long as only one replica is unavailable (recall that more than one site unavailability causes 2PC to be blocking), but it is generally not possible to differentiate between these two types of failures.

13.2.2 Lazy Update Propagation

In lazy update propagation the replica updates are not all performed within the context of the update transaction. In other words, the transaction does not wait until its updates are applied to all the copies before it commits – it commits as soon as one replica is updated. The propagation to other copies is done *asynchronously* from the original transaction, by means of *refresh transactions* that are sent to the replica sites some time after the update transaction commits. A refresh transaction carries the sequence of updates of the corresponding update transaction.

Lazy propagation is used in those applications for which strong mutual consistency may be unnecessary and too restrictive. These applications may be able to tolerate some inconsistency among the replicas in return for better performance. Examples of such applications are Domain Name Service (DNS), databases over geographically widely distributed sites, mobile databases, and personal digital assistant databases [Saito and Shapiro, 2005]. In these cases, usually weak mutual consistency is enforced.

The primary advantage of lazy update propagation techniques is that they generally have lower response times for update transactions, since an update transaction can commit as soon as it has updated one copy. The disadvantages are that the replicas are not mutually consistent and some replicas may be out-of-date, and, consequently, a local read may read stale data and does not guarantee to return the up-to-date value. Furthermore, under some scenarios that we will discuss later, transactions may not see their own writes, i.e., $Read_i(x)$ of an update transaction T_i may not see the effects of $Write_i(x)$ that was executed previously. This has been referred to as *transaction inversion*. Strong one-copy serializability (strong 1SR) [Daudjee and Salem, 2004] and strong snapshot isolation (strong SI) [Daudjee and Salem, 2006] prevent all transaction inversions at 1SR and SI isolation levels, respectively, but are expensive to provide. The weaker guarantees of 1SR and global SI, while being much less expensive to provide than their stronger counterparts, do not prevent transaction inversions. Session-level transactional guarantees at the 1SR and SI isolation levels have been proposed that address these shortcomings by preventing transaction inversions within a client session but not necessarily across sessions [Daudjee and Salem, 2004, 2006]. These session-level guarantees are less costly to provide than their strong counterparts while preserving many of the desirable properties of the strong counterparts.

13.2.3 Centralized Techniques

Centralized update propagation techniques require that updates are first applied at a master copy and then propagated to other copies (which are called *slaves*). The site that hosts the master copy is similarly called the *master site*, while the sites that host the slave copies for that data item are called *slave sites*.

In some techniques, there is a single master for all replicated data. We refer to these as *single master* centralized techniques. In other protocols, the master copy for each data item may be different (i.e., for data item x , the master copy may be x_i stored at site S_i , while for data item y , it may be y_j stored at site S_j). These are typically known as *primary copy* centralized techniques.

The advantages of centralized techniques are two-fold. First, application of the updates is easy since they happen at only the master site, and they do not require synchronization among multiple replica sites. Second, there is the assurance that at least one site – the site that holds the master copy – has up-to-date values for a data item. These protocols are generally suitable in data warehouses and other applications where data processing is centralized at one or a few master sites.

The primary disadvantage is that, as in any centralized algorithm, if there is one central site that hosts all of the masters, this site can be overloaded and can become a bottleneck. Distributing the master site responsibility for each data item as in primary copy techniques is one way of reducing this overhead, but it raises consistency issues, in particular with respect to maintaining global serializability in lazy replication techniques since the refresh transactions have to be executed at the replicas in the same serialization order. We discuss these further in relevant sections.

13.2.4 Distributed Techniques

Distributed techniques apply the update on the local copy at the site where the update transaction originates, and then the updates are propagated to the other replica sites. These are called distributed techniques since different transactions can update different copies of the same data item located at different sites. They are appropriate for collaborative applications with distributive decision/operation centers. They can more evenly distribute the load, and may provide the highest system availability if coupled with lazy propagation techniques.

A serious complication that arises in these systems is that different replicas of a data item may be updated at different sites (masters) concurrently. If distributed techniques are coupled by eager propagation methods, then the distributed concurrency control methods can adequately address the concurrent updates problem. However, if lazy propagation methods are used, then transactions may be executed in different orders at different sites causing non-ISR global history. Furthermore, various replicas will get out of sync. To manage these problems, a reconciliation method is applied involving undoing and redoing transactions in such a way that transaction execution is the same at each site. This is not an easy issue since the reconciliation is generally application dependent.

13.3 Replication Protocols

In the previous section, we discussed two dimensions along which update management techniques can be classified. These dimensions are orthogonal; therefore four combinations are possible: eager centralized, eager distributed, lazy centralized, and lazy distributed. We discuss each of these alternatives in this section. For simplicity of exposition, we assume a fully replicated database, which means that all update transactions are global. We further assume that each site implements a 2PL-based concurrency control technique.

13.3.1 Eager Centralized Protocols

In eager centralized replica control, a master site controls the operations on a data item. These protocols are coupled with strong consistency techniques, so that updates to a logical data item are applied to all of its replicas within the context of the update transaction, which is committed using the 2PC protocol (although non-2PC alternatives exist as we discuss shortly). Consequently, once the update transaction completes, all replicas have the same values for the updated data items (i.e., mutually consistent), and the resulting global history is 1SR.

The two design parameters that we discussed earlier determine the specific implementation of eager centralized replica protocols: where updates are performed, and degree of replication transparency. The first parameter, which was discussed in Section 13.2.3, refers to whether there is a single master site for all data items (single master), or different master sites for each, or, more likely, for a group of data items (primary copy). The second parameter indicates whether each application knows the location of the master copy (limited application transparency) or whether it can rely on its local TM for determining the location of the master copy (full replication transparency).

13.3.1.1 Single Master with Limited Replication Transparency

The simplest case is to have a single master for the entire database (i.e., for all data items) with limited replication transparency so that user applications know the master site. In this case, global update transactions (i.e., those that contain at least one $Write(x)$ operation where x is a replicated data item) are submitted directly to the master site – more specifically, to the transaction manager (TM) at the master site. At the master, each $Read(x)$ operation is performed on the master copy (i.e., $Read(x)$ is converted to $Read(x_M)$, where M signifies master copy) and executed as follows: a read lock is obtained on x_M , the read is performed, and the result is returned to the user. Similarly, each $Write(x)$ causes an update of the master copy (i.e., executed as $Write(x_M)$) by first obtaining a write lock and then performing the write operation. The master TM then forwards the $Write$ to the slave sites either

synchronously or in a deferred fashion (Figure 13.1). In either case, it is important to propagate updates such that conflicting updates are executed at the slaves in the same order they are executed at the master. This can be achieved by timestamping or by some other ordering scheme.

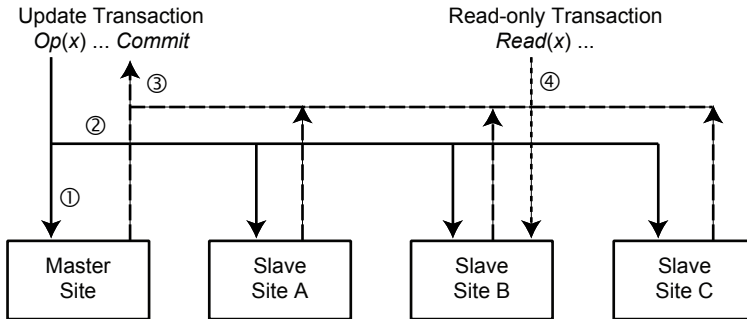


Fig. 13.1 Eager Single Master Replication Protocol Actions. (1) A *Write* is applied on the master copy; (2) *Write* is then propagated to the other replicas; (3) Updates become permanent at commit time; (4) Read-only transaction's *Read* goes to any slave copy.

The user application may submit a read-only transaction (i.e., all operations are *Read*) to any slave site. The execution of read-only transactions at the slaves can follow the process of centralized concurrency control algorithms, such as C2PL (Algorithms 11.1-11.3), where the centralized lock manager resides at the master replica site. Implementations within C2PL require minimal changes to the TM at the non-master sites, primarily to deal with the *Write* operations as described above, and its consequences (e.g., in the processing of Commit command). Thus, when a slave site receives a *Read* operation (from a read-only transaction), it forwards it to the master site to obtain a read lock. The *Read* can then be executed at the master and the result returned to the application, or the master can simply send a “lock granted” message to the originating site, which can then execute the *Read* on the local copy.

It is possible to reduce the load on the master by performing the *Read* on the local copy without obtaining a read lock from the master site. Whether synchronous or deferred propagation is used, the local concurrency control algorithm ensures that the local read-write conflicts are properly serialized, and since the *Write* operations can only be coming from the master as part of update propagation, local write-write conflicts won't occur as the propagation transactions are executed in each slave in the order dictated by the master. However, a *Read* may read data item values at a slave either before an update is installed or after. The fact that a read transaction at one slave site may read the value of one replica before an update while another read transaction reads another replica at another slave after the same update is inconsequential from the perspective of ensuring global 1SR histories. This is demonstrated by the following example.

Example 13.3. Consider a data item x whose master site is at Site A with slaves at sites B and C. Consider the following three transactions:

T_1 : Write(x)	T_2 : Read(x)	T_3 : Read(x)
Commit	Commit	Commit

Assume that T_2 is sent to slave at Site B and T_3 to slave at Site C. Assume that T_2 reads x at B [$Read(x_B)$] before T_1 's update is applied at B, while T_3 reads x at C [$Read(x_C)$] after T_1 's update at C. Then the histories generated at the two slaves will be as follows:

$$H_B = \{R_2(x), C_2, W_1(x), C_1\}$$

$$H_C = \{W_1(x), C_1, R_3(x), C_3\}$$

The serialization order at Site B is $T_2 \rightarrow T_1$, while at Site C it is $T_1 \rightarrow T_3$. The global serialization order, therefore, is $T_2 \rightarrow T_1 \rightarrow T_3$, which is fine. Therefore the history is 1SR. \blacklozenge

Consequently, if this approach is followed, read transactions may read data that are concurrently updated at the master, but the global history will still be 1SR.

In this alternative protocol, when a slave site receives a $Read(x)$, it obtains a local read lock, reads from its local copy (i.e., $Read(x_i)$) and returns the result to the user application; this can only come from a read-only transaction. When it receives a $Write(x)$, if the $Write$ is coming from the master site, then it performs it on the local copy (i.e., $Write(x_i)$). If it receives a $Write$ from a user application, then it rejects it, since this is obviously an error given that update transactions have to be submitted to the master site.

These alternatives of a single master eager centralized protocol are simple to implement. One important issue to address is how one recognizes a transaction as “update” or “read-only” – it may be possible to do this by explicit declaration within the `Begin.Transaction` command.

13.3.1.2 Single Master with Full Replication Transparency

Single master eager centralized protocols require each user application to know the master site, and they put significant load on the master that has to deal with (at least) the *Read* operations within update transactions as well as acting as the coordinator for these transactions during 2PC execution. These issues can be addressed, to some extent, by involving, in the execution of the update transactions, the TM at the site where the application runs. Thus, the update transactions are not submitted to the master, but to the TM at the site where the application runs (since they don't need to know the master). This TM can act as the coordinating TM for both update and read-only transactions. Applications can simply submit their transactions to their local TM, providing full transparency.

There are alternatives to implementing full transparency – the coordinating TM may only act as a “router”, forwarding each operation directly to the master site. The master site can then execute the operations locally (as described above) and return the results to the application. Although this alternative implementation provides full

transparency and has the advantage of being simple to implement, it does not address the overloading problem at the master. An alternative implementation may be as follows.

1. The coordinating TM sends each operation, as it gets it, to the central (master) site. This requires no change to the C2PL-TM algorithm (Algorithm 11.1).
2. If the operation is a $Read(x)$, then the centralized lock manager (C2PL-LM in Algorithm 11.2) can proceed by setting a read lock on its copy of x (call it x_M) on behalf of this transaction and informs the coordinating TM that the read lock is granted. The coordinating TM can then forward the $Read(x)$ to any slave site that holds a replica of x (i.e., converts it to a $Read(x_i)$). The read can then be carried out by the data processor (DP) at that slave.
3. If the operation is a $Write(x)$, then the centralized lock manager (master) proceeds as follows:
 - (a) It first sets a write lock on its copy of x .
 - (b) It then calls its local DP to perform the $Write$ on its own copy of x (i.e., converts the operation to $Write(x_M)$).
 - (c) Finally, it informs the coordinating TM that the write lock is granted.

The coordinating TM, in this case, sends the $Write(x)$ to all the slaves where a copy of x exists; the DPs at these slaves apply the $Write$ to their local copies.

The fundamental difference in this case is that the master site does not deal with *Reads* or with the coordination of the updates across replicas. These are left to the TM at the site where the user application runs.

It is straightforward to see that this algorithm guarantees that the histories are 1SR since the serialization orders are determined at a single master (similar to centralized concurrency control algorithms). It is also clear that the algorithm follows the ROWA protocol, as discussed above – since all the copies are ensured to be up-to-date when an update transaction completes, a *Read* can be performed on any copy.

To demonstrate how eager algorithms combine replica control and concurrency control, we show how the Transaction Management algorithm for the coordinating TM (Algorithm 13.1) and the Lock Management algorithm for the master site (Algorithm 13.2). We show only the revisions to the centralized 2PL algorithms (Algorithms 11.1 and 11.2 in Chapter 11).

Note that in the algorithm fragments that we have given, the LM simply sends back a “Lock granted” message and not the result of the update operation. Consequently, when the update is forwarded to the slaves by the coordinating TM, they need to execute the update operation themselves. This is sometimes referred to as *operation transfer*. The alternative is for the “Lock granted” message to include the result of the update computation, which is then forwarded to the slaves who simply need to apply the result and update their logs. This is referred to as *state transfer*. The distinction may seem trivial if the operations are simply in the form $Write(x)$, but recall that this

Algorithm 13.1: Eager Single Master Modifications to C2PL-TM

```

begin
  :
  if lock request granted then
    if  $op.Type = W$  then
      |  $S \leftarrow$  set of all sites that are slaves for the data item
    else
      |  $S \leftarrow$  any one site which has a copy of data item
      |  $DP_S(op)$  {send operation to all sites in set  $S$ }
    else
      | inform user about the termination of transaction
  :
end

```

Algorithm 13.2: Eager Single Master Modifications to C2PL-LM

```

begin
  :
  switch  $op.Type$  do
    case  $R$  or  $W$  {lock request; see if it can be granted}
      | find the lock unit  $lu$  such that  $op.arg \subseteq lu$  ;
      | if  $lu$  is unlocked or lock mode of  $lu$  is compatible with  $op.Type$ 
      | then
      |   | set lock on  $lu$  in appropriate mode on behalf of transaction
      |   |    $op.tid$  ;
      |   | if  $op.Type = W$  then
      |   |   |  $DP_M(op)$  {call local DP (M for “master”) with operation}
      |   |   | send “Lock granted” to coordinating TM of transaction
      |   | else
      |   |   | put  $op$  on a queue for  $lu$ 
      |   :
    :
end

```

Write operation is an abstraction; each update operation may require the execution of an SQL expression, in which case the distinction is quite important.

The above implementation of the protocol relieves some of the load on the master site and alleviates the need for user applications to know the master. However, its implementation is more complicated than the first alternative we discussed. In particular, now the TM at the site where transactions are submitted has to act as the 2PC coordinator and the master site becomes a participant. This requires some care in revising the algorithms at these sites.

13.3.1.3 Primary Copy with Full Replication Transparency

Let us now relax the requirement that there is one master for all data items; each data item can have a different master. In this case, for each replicated data item, one of the replicas is designated as the *primary copy*. Consequently, there is no single master to determine the global serialization order, so more care is required. In the case of fully replicated databases, any replica can be primary copy for a data item, however for partially replicated databases, limited replication transparency option only makes sense if an update transaction accesses only data items whose primary sites are at the same site. Otherwise, the application program cannot forward the update transactions to one master; it will have to do it operation-by-operation, and, furthermore, it is not clear which primary copy master would serve as the coordinator for 2PC execution. Therefore, the reasonable alternative is the full transparency support, where the TM at the application site acts as the coordinating TM and forwards each operation to the primary site of the data item that it acts on. Figure 13.2 depicts the sequence of operations in this case where we relax our previous assumption of fully replication. Site A is the master for data item *x* and sites B and C hold replicas (i.e., they are slaves); similarly data item *y*'s master is site C with slave sites B and D.

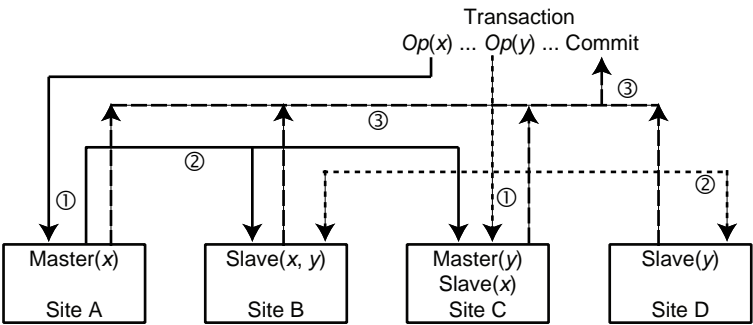


Fig. 13.2 Eager Primary Copy Replication Protocol Actions. (1) Operations (*Read* or *Write*) for each data item are routed to that data item's master and a *Write* is first applied at the master; (2) *Write* is then propagated to the other replicas; (3) Updates become permanent at commit time.

Recall that this version still applies the updates to all the replicas within transactional boundaries, requiring integration with concurrency control techniques. A very early proposal is the *primary copy two-phase locking* (PC2PL) algorithm proposed for the prototype distributed version of INGRES [Stonebraker and Neuhold, 1977]. PC2PL is a straightforward extension of the single master protocol discussed above in an attempt to counter the latter's potential performance problems. Basically, it implements lock managers at a number of sites and makes each lock manager responsible for managing the locks for a given set of lock units for which it is the master site. The transaction managers then send their lock and unlock requests to the lock managers that are responsible for that specific lock unit. Thus the algorithm treats one copy of each data item as its primary copy.

As a combined replica control/concurrency control technique, primary copy approach demands a more sophisticated directory at each site, but it also improves the previously discussed approaches by reducing the load of the master site without causing a large amount of communication among the transaction managers and lock managers.

13.3.2 Eager Distributed Protocols

In eager distributed replica control, the updates can originate anywhere, and they are first applied on the local replica, then the updates are propagated to other replicas. If the update originates at a site where a replica of the data item does not exist, it is forwarded to one of the replica sites, which coordinates its execution. Again, all of these are done within the context of the update transaction, and when the transaction commits, the user is notified and the updates are made permanent. Figure 13.3 depicts the sequence of operations for one logical data item x with copies at sites A, B, C and D, and where two transactions update two different copies (at sites A and D).

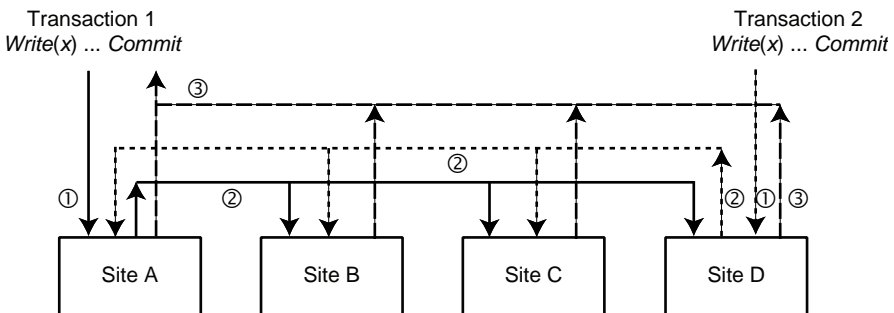


Fig. 13.3 Eager Distributed Replication Protocol Actions. (1) Two *Write* operations are applied on two local replicas of the same data item; (2) The *Write* operations are independently propagated to the other replicas; (3) Updates become permanent at commit time (shown only for Transaction 1).

As can be clearly seen, the critical issue is to ensure that concurrent conflicting *Writes* initiated at different sites are executed in the same order at every site where they execute together (of course, the local executions at each site also have to be serializable). This is achieved by means of the concurrency control techniques that are employed at each site. Consequently, read operations can be performed on any copy, but writes are performed on all copies within transactional boundaries (e.g., ROWA) using a concurrency control protocol.

13.3.3 Lazy Centralized Protocols

Lazy centralized replication algorithms are similar to eager centralized replication ones in that the updates are first applied to a master replica and then propagated to the slaves. The important difference is that the propagation does not take place within the update transaction, but after the transaction commits as a separate refresh transaction. Consequently, if a slave site performs a *Read*(x) operation on its local copy, it may read stale (non-fresh) data, since x may have been updated at the master, but the update may not have yet been propagated to the slaves.

13.3.3.1 Single Master with Limited Transparency

In this case, the update transactions are submitted and executed directly at the master site (as in the eager single master); once the update transaction commits, the refresh transaction is sent to the slaves. The sequence of execution steps are as follows: (1) an update transaction is first applied to the master replica, (2) the transaction is committed at the master, and then (3) the refresh transaction is sent to the slaves (Figure 13.4).

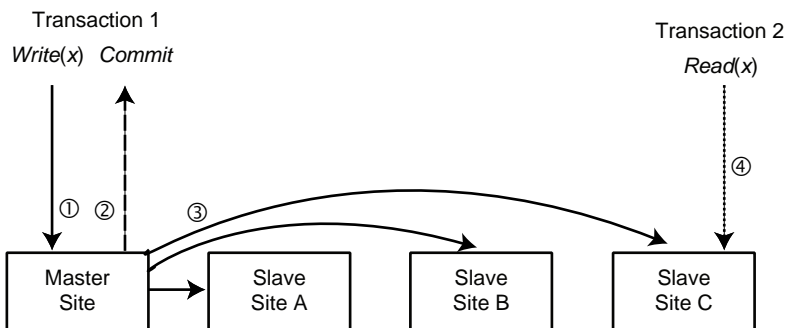


Fig. 13.4 Lazy Single Master Replication Protocol Actions. (1) Update is applied on the local replica; (2) Transaction commit makes the updates permanent at the master; (3) Update is propagated to the other replicas in refresh transactions; (4) Transaction 2 reads from local copy.

When a slave (secondary) site receives a $Read(x)$, it reads from its local copy and returns the result to the user. Notice that, as indicated above, its own copy may not be up-to-date if the master is being updated and the slave has not yet received and executed the corresponding refresh transaction. A $Write(x)$ received by a slave is rejected (and the transaction aborted), as this should have been submitted directly to the master site. When a slave receives a refresh transaction from the master, it applies the updates to its local copy. When it receives a *Commit* or *Abort* (*Abort* can happen for only locally submitted read-only transactions), it locally performs these actions.

The case of primary copy with limited transparency is similar, so we don't discuss it in detail. Instead of going to a single master site, $Write(x)$ is submitted to the primary copy of x ; the rest is straightforward.

How can it be ensured that the refresh transactions can be applied at all of the slaves in the same order? In this architecture, since there is a single master copy for all data items, the ordering can be established by simply using timestamps. The master site would attach a timestamp to each refresh transaction according to the commit order of the actual update transaction, and the slaves would apply the refresh transactions in timestamp order.

A similar approach may be followed in the primary copy, limited transparency case. In this case, a site contains slave copies of a number of data items, causing it to get refresh transactions from multiple masters. The execution of these refresh transactions need to be ordered the same way at all of the involved slaves to ensure that the database states eventually converge. There are a number of alternatives that can be followed.

One alternative is to assign timestamps such that refresh transactions issued from different masters have different timestamps (by appending the site identifier to a monotonic counter at each site). Then the refresh transactions at each site can be executed in their timestamp order. However, those that come out of order cause difficulty. In traditional timestamp-based techniques discussed in Chapter 11, these transactions would be aborted; however in lazy replication, this is not possible since the transaction has already been committed at the primary copy site. The only possibility is to run a compensating transaction (which, effectively, aborts the transaction by rolling back its effects) or to perform update reconciliation that will be discussed shortly. The issue can be addressed by a more careful study of the resulting histories. An approach proposed by Breitbart and Korth [1997] uses a serialization graph approach that builds a *replication graph* whose nodes consist of transactions (T) and sites (S) and an edge $\langle T_i, S_j \rangle$ exists in the graph if and only if T_i performs a *Write* on a (replicated) physical copy that is stored at S_j . When an operation (op_k) is submitted, the appropriate nodes (T_k) and edges are inserted into the replication graph, which is checked for cycles. If there is no cycle, then the execution can proceed. If a cycle is detected and it involves a transaction that has committed at the master, but whose refresh transactions have not yet committed at all of the involved slaves, then the current transaction (T_k) is aborted (to be restarted later) since its execution would cause the history to be non-ISR. Otherwise, T_k can wait until the other transactions in the cycle are completed (i.e., they are committed at their masters and their refresh transactions are committed at all of the slaves). When a transaction

is completed in this manner, the corresponding node and all of its incident edges are removed from the replication graph. This protocol is proven to produce 1SR histories. An important issue is the maintenance of the replication graph. If it is maintained by a single site, then this becomes a centralized algorithm. We leave the distributed construction and maintenance of the replication graph as an exercise.

Another alternative is to rely on the group communication mechanism provided by the underlying communication infrastructure (if it can provide it). We discuss this alternative in Section 13.4.

Recall from Section 13.3.1 that, in the case of partially replicated databases, eager primary copy with limited replication transparency approach makes sense if the update transactions access only data items whose master sites are the same, since the update transactions are run completely at a master. The same problem exists in the case of lazy primary copy, limited replication approach. The issue that arises in both cases is how to design the distributed database so that meaningful transactions can be executed. This problem has been studied within the context of lazy protocols [Chundi et al., 1996] and a primary site selection algorithm was proposed that, given a set of transactions, a set of sites, and a set of data items, finds a primary site assignment to these data items (if one exists) such that the set of transactions can be executed to produce a 1SR global history.

13.3.3.2 Single Master or Primary Copy with Full Replication Transparency

We now turn to alternatives that provide full transparency by allowing (both read and update) transactions to be submitted at any site and forwarding their operations to either the single master or to the appropriate primary master site. This is tricky and involves two problems: the first is that, unless one is careful, 1SR global history may not be guaranteed; the second problem is that a transaction may not see its own updates. The following two examples demonstrate these problems.

Example 13.4. Consider the single master scenario and two sites M and B where M holds the master copies of x and y and B holds their slave copies. Now consider the following two transactions: T_1 submitted at site B, while transaction T_2 submitted at site M:

T_1 : Read(x)	T_2 : Write(x)
Write(y)	Write(y)
Commit	Commit

One way these would be executed under full transparency is as follows. T_2 would be executed at site M since it contains the master copies of both x and y . Sometime after it commits, refresh transactions for its *Writes* are sent to site B to update the slave copies. On the other hand, T_1 would read the local copy of x at site B, but its *Write*(x) would be forwarded to x 's master copy, which is at site M. Some time after *Write*₁(x) is executed at the master site and commits there, a refresh transaction

would be sent back to site B to update the slave copy. The following is a possible sequence of steps of execution (Figure 13.5):

1. $Read_1(x)$ is submitted at site B, where it is performed;
2. $Write_2(x)$ is submitted at site M, and it is executed;
3. $Write_2(y)$ is submitted at site M, and it is executed;
4. T_2 submits its *Commit* at site M and commits there;
5. $Write_1(x)$ is submitted at site B; since the master copy of x is at site M, the *Write* is forwarded to M;
6. $Write_1(x)$ is executed at site M and the confirmation is sent back to site B;
7. T_1 submits *Commit* at site B, which forwards it to site M; it is executed there and B is informed of the commit where T_1 also commits;
8. Site M now sends refresh transaction for T_2 to site B where it is executed and commits;
9. Site M finally sends refresh transaction for T_1 to site B (this is for T_1 's *Write* that was executed at the master), it is executed at B and commits.

The following two histories are now generated at the two sites where the superscript r on operations indicate that they are part of a refresh transaction:

$$H_M = \{W_2(x_M), W_2(y_M), C_2, W_1(y_M), C_1\}$$

$$H_B = \{R_1(x_B), C_1, W_2^r(x_B), W_2^r(y_B), C_2^r, W_1^r(x_B), C_1^r\}$$

The resulting global history over the *logical* data items x and y is non-1SR. ♦

Example 13.5. Again consider a single master scenario, where site M holds the master copy of x and site D holds its slave. Consider the following simple transaction:

T_3 : Write(x)
 Read(x)
 Commit

Following the same execution model as in Example 13.4, the sequence of steps would be as follows:

1. $Write_3(x)$ is submitted at site D, which forwards it to site M for execution;
2. The *Write* is executed at M and the confirmation is sent back to site D;
3. $Read_3(x)$ is submitted at site D and is executed on the local copy;
4. T_3 submits commit at D, which is forwarded to M, executed there and a notification is sent back to site D, which also commits the transaction;
5. Site M sends a refresh transaction to site D for the $W_3(x)$ operation;
6. Site D executes the refresh transaction and commits it.

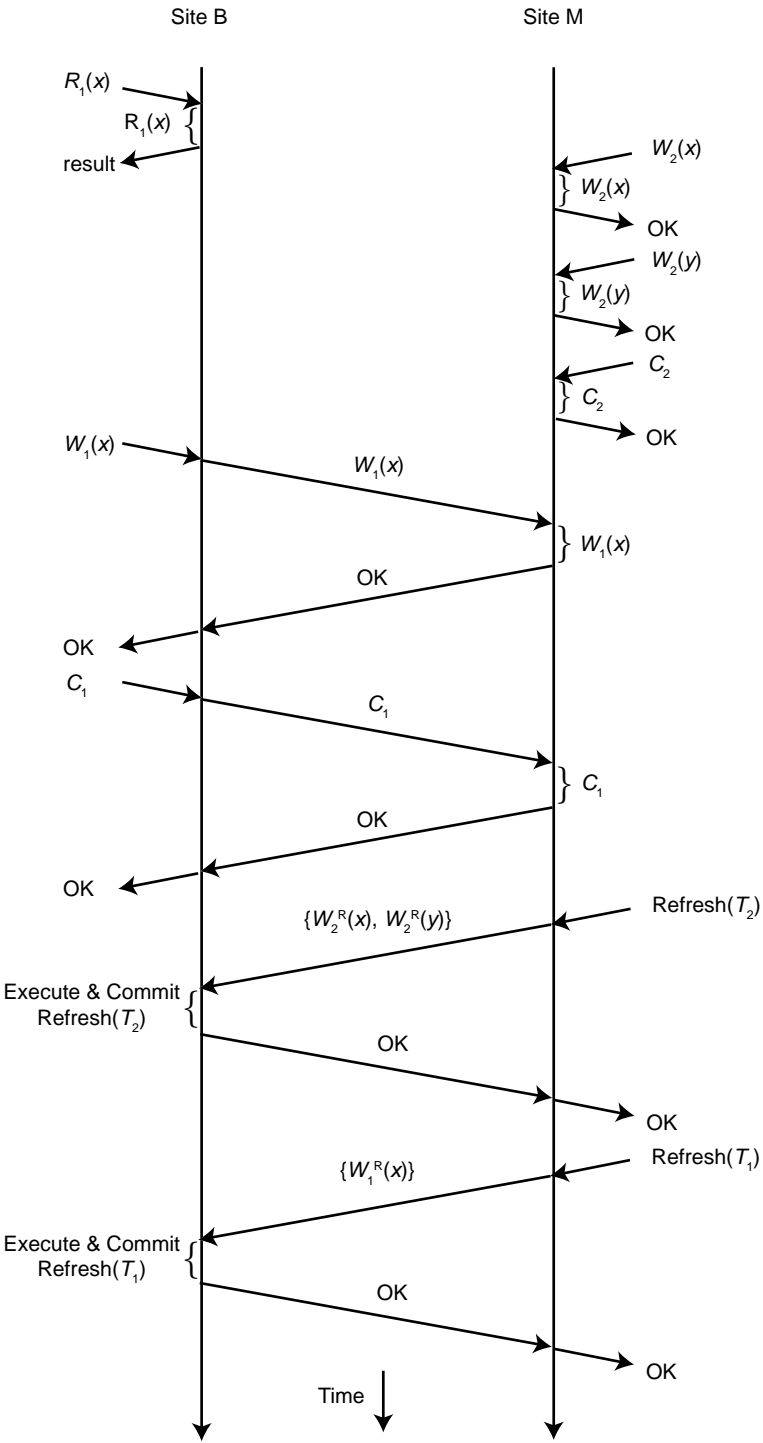


Fig. 13.5 Time sequence of executions of transactions