Chapter 11

DISTRIBUTED CONCURRENCY CONTROL

In this chapter we make one major assumption: the distributed system is fully reliable and does not experience any failures (of hardware or software). Even though this is an entirely unrealistic assumption, there is a reason for making it. It permits us to delineate the issues related to the management of concurrency from those related to the operation of a reliable distributed system. In Chapter 12, we discuss how the algorithms that are presented in this chapter need to be revised to operate in an unreliable environment. We start our discussion of concurrency control with a presentation of serializability theory in Section 11.1. Serializability is the most widely accepted correctness criterion for concurrency control algorithms. In Section 11.2 we present a taxonomy of algorithms that will form the basis for most of the discussion in the remainder of the chapter. Sections 11.3 and 11.4 cover the two major classes of algorithms: locking-based and timestamp ordering-based. Both locking and timestamp ordering classes cover what is called pessimistic algorithms; optimistic concurrency control is discussed in Section 11.5. Any locking-based algorithm may result in deadlocks, requiring special management methods. Various deadlock management techniques are therefore the topic of Section 11.6. In Section 11.7, we discuss "relaxed" concurrency control approaches. These are mechanisms which use weaker correctness criteria than serializability, or relax the isolation property of transactions.

11.1 SERIALIZABILITY THEORY

In Section 10.1.3 we discussed the issue of isolating transactions from one another in terms of their effects on the database. We also pointed out that if the concurrent execution of transactions leaves the database in a state that can be achieved by their serial execution in some order, problems such as lost updates will be resolved. This is exactly the point of the serializability argument. The remainder of this section addresses serializability issues more formally.

A *schedule* S (also called a *history*) is defined over a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ and specifies an interleaved order of execution of these transactions' operations. Based on the definition of a transaction introduced in

Section 10.1, the schedule can be specified as a partial order over T. We need a few preliminaries, though, before we present the formal definition.

Recall the definition of conflicting operations that we gave in Chapter 10. Two operations $O_{ij}(x)$ and $O_{kl}(x)$ (i and k not necessarily distinct) accessing the same database entity x are said to be in *conflict* if at least one of them is a write. Note two things in this definition. First, read operations do not conflict with each other. We can, therefore, talk about two types of conflicts: *read-write* (or *write-read*), and *write-write*. Second, the two operations can belong to the same transaction or to two different transactions. In the latter case, the two transactions are said to be *conflicting*. Intuitively, the existence of a conflict between two operations indicate that their order of execution is important. The ordering of two read operations is insignificant.

We first define a *complete schedule*, which defines the execution order of all operations in its domain. We will then define a schedule as a prefix of a complete schedule. Formally, a complete schedule S_T^c defined over a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ is a partial order $S_T^c = \{\Sigma_T, \prec_T\}$ where

1. $\Sigma_T = \cup_{i=1}^{n} \Sigma_i$.
2. $\prec_T \supseteq \cup_{i=1}^{n} \prec_i$.
3. For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, > either $O_{ij} \prec_T O_{kl}$, or $O_{kl} \prec_T O_{ij}$.

The first condition simply states that the domain of the schedule is the union of the domains of individual transactions. The second condition defines the ordering relation as a superset of the ordering relations of individual transactions. This maintains the ordering of operations within each transaction. The final condition simply defines the execution order among conflicting operations.

Example 11.1

Consider the two transactions from Example 10.8. They were specified as

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x + 1$		$x \leftarrow x + 1$
	Write(x)		Write(x)
	Commit		Commit

A possible complete schedule S_T^c over $T = \{T_1, T_2\}$ can be written as the following partial order (where the subscripts indicate the transactions):

$$S_T^c = \{\Sigma_T, \prec_T\}$$

where

$$\Sigma_1 = \{R_1(x), W_1(x), C_1\}$$
$$\Sigma_2 = \{R_2(x), W_2(x), C_2\}$$

Thus

$$\Sigma_T = \Sigma_1 \cup \Sigma_2 = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x)\}$$

and

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

which can be specified as a DAG as depicted in Figure 11.1; Note that consistent with our earlier adopted convention (see Example 10.7), we do not draw the arcs that are implied by transitivity [e.g., (R_1, C_1)]. Also note that we omit the data items that these operations operate on since this is obvious from the context.

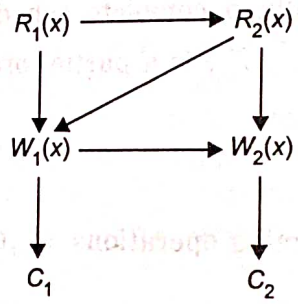

Figure 11.1. DAG Representation of a Complete Schedule

At this point we should mention a convention that is commonly employed to simplify the notation of a schedule. It is often specified as a listing of the operations in Σ_T, where their execution order is relative to their order in this list. Thus S_T^c can be specified as

$$S_T^c = \{R_1(x), R_2(x), W_1(x), C_1, W_2(x), C_2\}$$

Because of its simplicity, we use the latter notation in the remainder of this chapter.

A schedule is defined as a prefix of a complete schedule. A prefix of a partial order can be defined as follows. Given a partial order $P = \{\Sigma, \prec\}$, $P' = \{\Sigma', \prec'\}$ is a *prefix* of P if

1. $\Sigma' \subseteq \Sigma$;
2. $\forall e_i \in \Sigma'$, $e_1 \prec' e_2$ if and only if $e_1 \prec e_2$; and
3. $\forall e_i \in \Sigma'$, if $\exists e_j \in \Sigma$ and $e_j \prec e_i$, then $e_j \in \Sigma'$.

The first two conditions define P' as a *restriction* of P on domain Σ', whereby the ordering relations in P are maintained in P'. The last condition indicates that for any element of Σ', all its predecessors in Σ have to be included in Σ' as well.

The real question that needs to be asked is: What does this definition of a schedule as a prefix of a partial order provide for us? The answer is simply that we can now deal with incomplete schedules. This is useful for a number of reasons. From the perspective of the serializability theory, we deal only with those operations of transactions that conflict rather than with all operations. Furthermore, and perhaps more important, when we introduce failures, we need to be able to deal with incomplete schedules, which is what a prefix enables us to do.

The schedule discussed in Example 11.1 is special in that it is complete. It needs to be complete in order to talk about the execution order of these two transactions' operations. The following example demonstrates a schedule that is not complete.

Example 11.2

Consider the following three transactions:

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

A complete schedule S^c for these transactions is given in Figure 11.2, and a schedule S (as a prefix of S^c) is depicted in Figure 11.3.

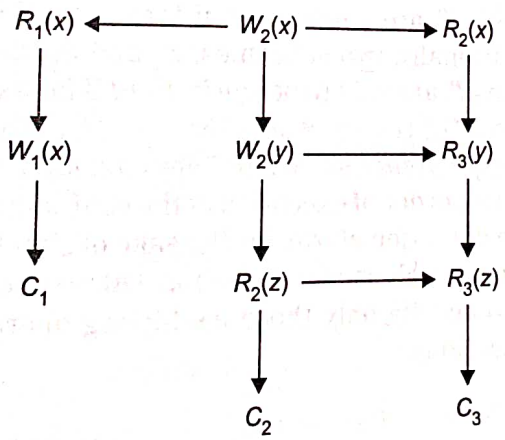

Figure 11.2. A Complete Schedule

If in a schedule S, the operations of various transactions are not interleaved (i.e., the operations of each transaction occur consecutively), the schedule is said to be *serial*. As we indicated before, the serial execution of a set of transactions maintains the consistency of the database. This follows naturally from the consistency property of transactions: each transaction, when executed alone on a consistent database, will produce a consistent database.

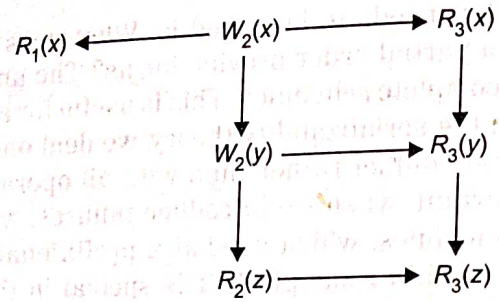

Figure 11.3. Prefix of Complete Schedule in Figure 11.2

Example 11.3

Consider the three transactions of Example 11.2. The following schedule,

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

is serial since all the operations of T_2 are executed before all the operations of T_1 and all operations of T_1 are executed before all operations of T_3. One common way to denote this precedence relationship between transaction executions is $T_2 \prec_s T_1 \prec_s T_3$ or $T_2 \rightarrow T_1 \rightarrow T_3$.

Based on the precedence relationship introduced by the partial order, it is possible to discuss the equivalence of schedules with respect to their effects on the database. Intuitively, two schedules S_1 and S_2, defined over the same set of transactions T, are *equivalent* if they have the same effect on the database. More formally, two schedules, S_1 and S_2, defined over the same set of transactions T, are said to be *equivalent* if for each pair of conflicting operations O_{ij} and O_{kl} $(i \neq k)$, whenever $O_{ij} \prec_1 O_{kl}$, then $O_{ij} \prec_2 O_{kl}$. This is called *conflict equivalence* since it defines equivalence of two schedules in terms of the relative order of execution of the conflicting operations in those schedules. In the definition above, for the sake of simplicity, we assume that T does not include any aborted transaction. Otherwise, the definition needs to be modified to specify only those conflicting operations that belong to unaborted transactions.[1]

Example 11.4

Again consider the three transactions of Example 11.2. The following schedule S' defined over them is conflict equivalent to S given in Example 11.3:

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

We are now ready to define serializability more formally. A schedule S is said to be *serializable* if and only if it is conflict equivalent to a serial

[1] For the sake of completeness, we should also point out that there is another form of equivalence, called *view equivalence*. The concept of serializability can also be defined based on view equivalence. We will not dwell on this point further since conflict equivalence is a more useful concept to work with in concurrency control.

schedule. Note that serializability roughly corresponds to degree 3 consistency, which we defined in Section 10.2.2. Serializability so defined is also known as *conflict-based serializability* since it is defined according to conflict equivalence.

Example 11.5

Schedule S' in Example 11.4 is serializable since it is equivalent to the serial schedule S of Example 11.3. Also note that the problem with the uncontrolled execution of transactions T_1 and T_2 in Example 10.8 was that they could generate an unserializable schedule.

Now that we have formally defined serializability, we can indicate that the primary function of a concurrency controller is to generate a serializable schedule for the execution of pending transactions. The issue, then, is to devise algorithms that are guaranteed to generate only serializable schedules.

Serializability theory extends in a straightforward manner to the nonreplicated (or partitioned) distributed databases. The schedule of transaction execution at each site is called a *local schedule*. If the database is not replicated and each local schedule is serializable, their union (called the *global schedule*) is also serializable as long as local serialization orders are identical. In a replicated distributed database, however, the extension of the serializability theory requires more care. It is possible that the local schedules are serializable, but the mutual consistency of the database is still compromised.

Example 11.6

We will give a very simple example to demonstrate the point. Consider two sites and one data item (x) that is duplicated in both sites. Further consider the following two transactions:

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

Obviously, both of these transactions need to run at both sites. Consider the following two schedules that may be generated locally at the two sites:

$$S_1 = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$$
$$S_2 = \{R_2(x), W_2(x), C_2, R_1(x), W_1(x), C_1\}$$

Note that both of these schedules are serializable; indeed, they are serial. Therefore, each represents a correct execution order. However, observe that they serialize T_1 and T_2 in reverse order. 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 1 while it is 15 at site 2. This violates the mutual consistency of the two local databases.

Mutual consistency requires that all the values of all replicated data items be identical. Schedules that can maintain mutual consistency are called *one-copy serializable* [Bernstein and Goodman, 1985]. Intuitively, a one-copy serializable global schedule has to meet the following conditions:

1. Each local schedule should be serializable.
2. Two conflicting operations should be in the same relative order in all of the local schedules where they appear together.

The second condition simply ensures that the serialization order be the same at all the sites where the conflicting transactions execute together. Recall that concurrency control algorithms ensure serializability by synchronizing conflicting accesses to the database. In replicated databases, the additional task of ensuring one-copy serializability is usually the responsibility of the *replica control protocol*.

Let us assume the existence of a data item x with copies $x_1, x_2, \ldots, x_n$. We will refer to x as the *logical data* item and to its copies 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 each read on the logical data item x [Read(x)] to a read on one of the physical data item copies x_j [Read(x_j)]. Each write on the logical data item x, on the other hand, is mapped to a set of writes on a (possibly proper) subset of the physical data item copies of x. Whether this mapping is to the full set of physical data item copies or to a subset is the basis of classifying replica control algorithms. In this chapter, and for the most part in this book, we consider replica control protocols that map a read on a logical data item to only *one* copy of the data item, but map a write on a logical data item to a set of writes on *all* physical data item copies. Such a protocol is commonly known as the *read-once/write-all* (ROWA) protocol.

The common complaint about the ROWA protocol is that it reduces the availability of the database in case of failures since the transaction may not complete unless it reflects the effects of the write operation on all the copies (more on this in Chapter 12). Therefore, there have been a number of algorithms that have attempted to maintain mutual consistency without employing the ROWA protocol. They are all based on the premise that one can continue processing an operation as long as the operation can be scheduled at a subset of the sites which correspond to a majority of the sites where copies are stored [Thomas, 1979] or to all sites that can be reached (i.e., available) [Bernstein and Goodman, 1984] and [Goodman et al., 1983]. There are still other protocols that perform the updates on an identified master copy of the replicated data item and then propagate these updates to other replica copies *lazily*.

11.2 TAXONOMY OF CONCURRENCY CONTROL MECHANISMS

There are a number of ways that the concurrency control approaches can be classified. One obvious classification criterion is the mode of database distribution.

Some algorithms that have been proposed require a fully replicated database, while others can operate on partially replicated or partitioned databases. The concurrency control algorithms may also be classified according to network topology, such as those requiring a communication subnet with broadcasting capability or those working in a star-type network or a circularly connected network.

The most common classification criterion, however, is the synchronization primitive. The corresponding breakdown of the concurrency control algorithms results in two classes: those algorithms that are based on mutually exclusive access to shared data (locking), and those that attempt to order the execution of the transactions according to a set of rules (protocols). However, these primitives may be used in algorithms with two different viewpoints: the pessimistic view that many transactions will conflict with each other, or the optimistic view that not too many transactions will conflict with one another.

We will thus group the concurrency control mechanisms into two broad classes: pessimistic concurrency control methods and optimistic concurrency control methods. *Pessimistic* algorithms synchronize the concurrent execution of transactions early in their execution life cycle, whereas *optimistic* algorithms delay the synchronization of transactions until their termination. The pessimistic group consists of *locking-based* algorithms, *ordering* (or *transaction ordering*) *based* algorithms, and *hybrid* algorithms. The optimistic group can, similarly, be classified as locking-based or timestamp ordering-based. This classification is depicted in Figure 11.4.

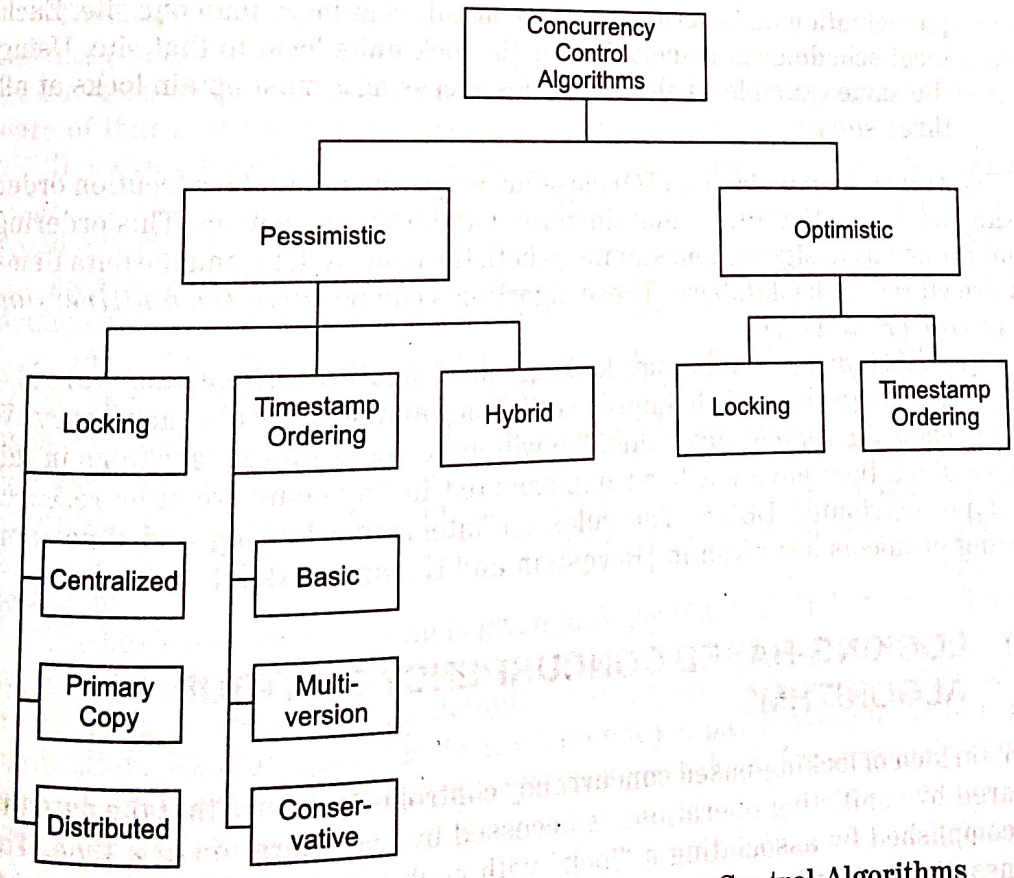

Figure 11.4. Classification of Concurrency Control Algorithms

In the *locking-based* approach, the synchronization of transactions is achieved by employing physical or logical locks on some portion or granule of the database. The size of these portions (usually called *locking granularity*) is an important issue. However, for the time being, we will ignore it and refer to the chosen granule as a *lock unit*. This class is subdivided further according to where the lock management activities are performed:

1. In *centralized locking*, one of the sites in the network is designated as the primary site where the lock tables for the entire database are stored and is charged with the responsibility of granting locks to transactions.
2. In *primary copy locking*, on the other hand, one of the copies (if there are multiple copies) of each lock unit is designated as the primary copy, and it is this copy that has to be locked for the purpose of accessing that particular unit. For example, if lock unit x is replicated at sites 1, 2, and 3, one of these sites (say, 1) is selected as the primary site for x. All transactions desiring access to x obtain their lock at site 1 before they can access a copy *of* x. If the database is not replicated (i.e., there is only one copy of each lock unit), the primary copy locking mechanisms distribute the lock management responsibility among a number of sites.
3. In *decentralized locking*, the lock management duty is shared by all the sites of a network. In this case, the execution of a transaction involves the participation and coordination of schedulers at more than one site. Each local scheduler is responsible for the lock units local to that site. Using the same example as above, entities accessing x must obtain locks at all three sites.

The *timestamp ordering* (TO) class involves organizing the execution order of transactions so that they maintain mutual and interconsistency. This ordering is maintained by assigning timestamps to both the transactions and the data items that are stored in the database. These algorithms can be *basic TO*, *multiversion TO*, or *conservative TO*.

We should indicate that in some locking-based algorithms, timestamps are also used. This is done primarily to improve efficiency and the level of concurrency. We call this class the *hybrid* algorithm. We will not discuss these algorithms in this chapter since they have not been implemented in any commercial or research prototype distributed DBMS. The rules for integrating locking and timestamp ordering protocols are given in [Bernstein and Goodman, 1981].

11.3 LOCKING-BASED CONCURRENCY CONTROL ALGORITHMS

The main idea of locking-based concurrency control is to ensure that the data that is shared by conflicting operations is accessed by one operation at a time. This is accomplished by associating a "lock" with each lock unit. This lock is set by a transaction before it is accessed and is reset at the end of its use. Obviously a

lock unit cannot be accessed by an operation if it is already locked by another. Thus a lock request by a transaction is granted only if the associated lock is not being held by any other transaction.

Since we are concerned with synchronizing the conflicting operations of conflicting transactions, there are two types of locks (commonly called *lock modes*) associated with each lock unit: *read lock* (*rl*) and *write lock* (*wl*). A transaction T_i that wants to read a data item contained in lock unit x obtains a read lock on x [denoted $rl_i(x)$]. The same happens for write operations. It is common to talk about the *compatibility* of lock modes. Two lock modes are compatible if two transactions which access the same data item can obtain these locks on that data item at the same time. As Figure 11.5 shows, read locks are compatible, whereas read-write or write-write locks are not. Therefore, it is possible, for example, for two transactions to read the same data item concurrently.

	$rl_j(x)$	$wl_j(x)$
$rl_j(x)$	compatible	not compatible
$wl_j(x)$	not compatible	not compatible

Figure 11.5. Compatibility Matrix of Lock Modes

The distributed DBMS not only manages locks but also handles the lock management responsibilities on behalf of the transactions. In other words, the users do not need to specify when data needs to be locked; the distributed DBMS takes care of that every time the transaction issues a read or write operation.

In locking-based systems, the scheduler (see Figure 10.4) is a *lock manager* (LM). The transaction manager passes to the lock manager the database operation (read or write) and associated information (such as the item that is accessed and the identifier of the transaction that issues the database operation). The lock manager then checks if the lock unit that contains the data item is already locked. If so, and if the existing lock mode is incompatible with that of the current transaction, the current operation is delayed. Otherwise, the lock is set in the desired mode and the database operation is passed on to the data processor for actual database access. The transaction manager is then informed of the results of the operation. The termination of a transaction results in the release of its locks and the initiation of another transaction that might be waiting for access to the same data item.

The basic locking algorithm is given in Algorithm 11.1. In Figure 11.6 we give the type declarations and the procedure definitions that we use in the algorithms of this chapter. Note that the definitions in Figure 11.6 are at a high level of abstraction. Also, in Algorithm 11.1, we do not yet pay any attention to how the commit and abort operations of transactions are serviced. Those complications are not added until Chapter 12.

```
declare-type
    Operation: one of Begin_Transaction, Read, Write, Abort, or Commit
    DataItem: a data item in the distributed database
    TransactionId: a unique identifier assigned to each transaction
    DataVal: a primitive data-type value (i.e., integer, real, etc.)
    SiteId: a unique site identifier
    Dbop: a 3-tuple of      {a database operation from the application program}
        opn : Operation
        data : DataItem
        tid : TransactionId
    Dpmsg: a 3-tuple of                  {a message from the data processor}
        opn : Operation
        tid : TransactionId
        res : DataVal
    Scmsg: a 3-tuple of                          {a message from the scheduler}
        opn : Operation
        tid : TransactionId
        res : DataVal
    Transaction ← a 2-tuple of
        tid : TransactionId
        body : a transaction body as defined in Chapter 10
    Message ← a string of characters that are to be transmitted
    OpSet: a set of DbOp's
    SiteSet: a set of SiteId's
    WAIT(msg:Message)
    begin
        {wait until a message arrives}
    end
```

Figure 11.6.

Preliminary Definitions for the Upcoming Algorithms

Algorithm 11.1 *Basic LM*

```
declare-var
    msg : Message
    dop : Dbop
    Op : Operation
    x : DataItem
    T : TransactionId
    pm : Dpmsg
    res : DataVal
    SOP : OpSet
begin
    repeat
    WAIT(msg)
    case of msg
```

```
    Dbop :
    begin
      Op ← dop.opn
      x ← dop.data
      T ← dop.tid
      case of Op
      Begin_transaction:
      begin
        send dop to the data processor
      end
      Read or Write:
      begin                                          {requires locking}
        find the lock unit lu such that x ⊆ lu
        if lu is unlocked or lock mode of lu is compatible with Op then
        begin
            set lock on lu in appropriate mode
            send dop to the data processor
        end
        else put dop on a queue for lu
        end-if
      end
      Abort or Commit:
      begin
        send dop to the data processor
      end
    end-case
  Dpmsg :                      {acknowledgment from the data processor}
  begin                                            {requires unlocking}
    Op ← pm.opn
    res ← pm.result
    T ← pm.tid
    find lock unit lu such that x ⊆ lu   release lock on lu held by T
    if there are no more locks on lu and
      there are operations waiting in queue to lock lu then
    begin
      SOP ← first operation from the queue
      SOP ← SOP ∪ {O|O is an operation on queue that can lock lu
        in a compatible mode with the current operations in SOP}
      set the locks on lu on behalf of operations in SOP
      for all the operations in SOP do
        send each operation to the data processor
      end-for
    end-if
  end
 end-case
 until forever
end. {Basic LM}
```

The locking algorithm that is given in Algorithm 11.1 will not, unfortunately, properly synchronize transaction executions. This is because to generate serializable schedules, the locking and releasing operations of transactions also need to be coordinated. We demonstrate this by an example.

Example 11.7

Consider the following two transactions:

T_1:		T_2:	
	Read(x)		Read(x)
	$x \leftarrow x + 1$		$x \leftarrow x * 2$
	Write(x)		Write(x)
	Read(y)		Read(y)
	y $\leftarrow$ y $-$ 1		y $\leftarrow$ $y * 2$
	Write(y)		Write(y)
	Commit		Commit

The following is a valid schedule that a lock manager employing the algorithm of Algorithm 11.1 may generate:

$$S = \{wl_1(x), R_1(x), W_1(x), lr_1(x), wl_2(x), R_2(x), w_2(x), lr_2(x), wl_2(y), R_2(y), W_2y), lr_2(y), C_2, wl_1(y), R_1(y), W_1(y), lr_1(y), C_1\}$$

Here, $lr_i(z)$ indicates the release of the lock on z that transaction T_i holds. Note that S is not a serializable schedule. For example, if prior to the execution of these transactions, the values of x and y are 50 and 20, respectively, one would expect their values following execution to be, respectively, either 102 and 38 if T_1 executes before T_2, or 101 and 39 if T_2 executes before T_1. However, the result of executing S would give x and y the values 102 and 39. Obviously, S is not serializable.

The problem with schedule S in Example 11.7 is the following. The locking algorithm releases the locks that are held by a transaction (say, T_i) as soon as the associated database command (read or write) is executed, and that lock unit (say x) no longer needs to be accessed. However, the transaction itself is locking other items (say, y), after it releases its lock on x. Even though this may seem to be advantageous from the viewpoint of increased concurrency, it permits transactions to interfere with one another, resulting in the loss of total isolation and atomicity. Hence the argument for *two-phase locking* (2PL).

The two-phase locking rule simply states that no transaction should request a lock after it releases one of its locks. Alternatively, a transaction should not release a lock until it is certain that it will not request another lock. 2PL algorithms execute transactions in two phases. Each transaction has a *growing phase*, where it obtains locks and accesses data items, and a *shrinking phase*, during which it releases locks (Figure 11.7). The *lock point* is the moment when the transaction has achieved all its locks but has not yet started to release any of them. Thus the lock point determines the end of the growing phase and the beginning of the shrinking phase of a transaction. It is a well-known theorem [Eswaran et al., 1976]

that any schedule generated by a concurrency control algorithm that obeys the 2PL rule is serializable.

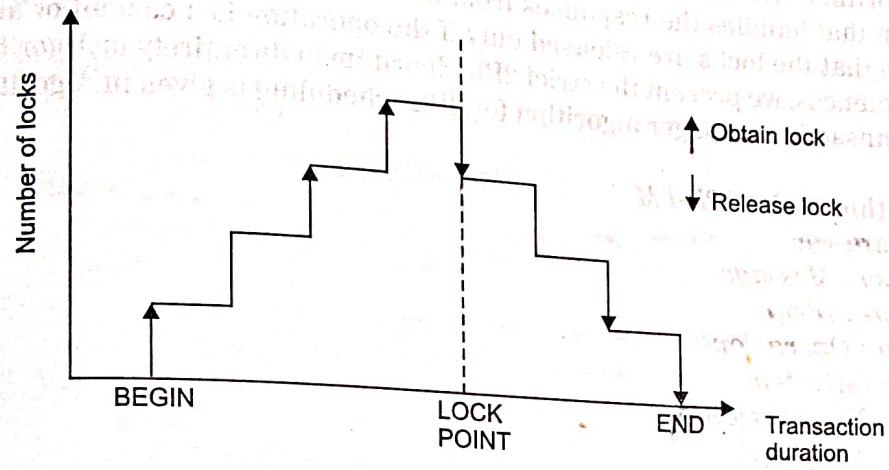

Figure 11.7. 2PL Lock Graph

Figure 11.7 indicates that the lock manager releases locks as soon as access to that data item has been completed. This permits other transactions awaiting access to go ahead and lock it, thereby increasing the degree of concurrency. However, this is difficult to implement since the lock manager has to know that the transaction has obtained all its locks and will not need to lock another data item. The lock manager also needs to know that the transaction no longer needs to access the data item in question, so that the lock can be released. Finally, if the transaction aborts after it releases a lock, it may cause other transactions that may have accessed the unlocked data item to abort as well. This is known as *cascading aborts*. Because of these difficulties, most 2PL schedulers implement what is called *strict two-phase locking*, which releases all the locks together when the transaction terminates (commits or aborts). Thus the lock graph is as shown in Figure 11.8.

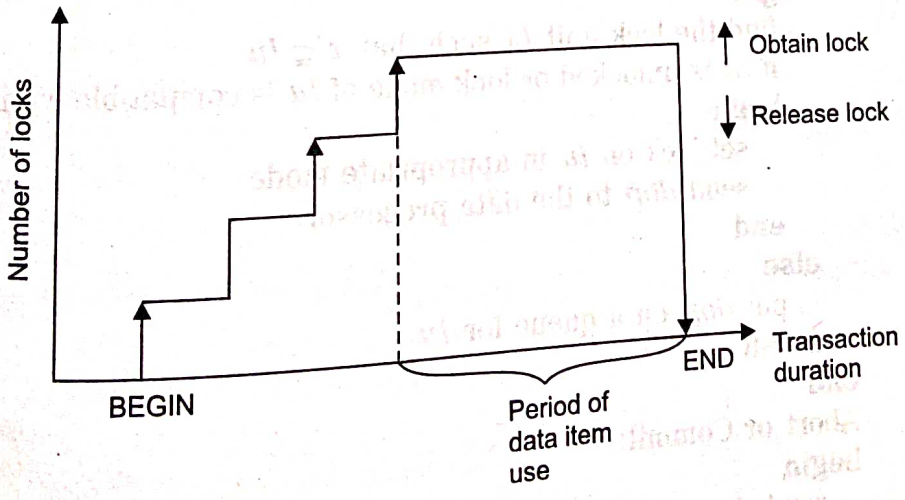

Figure 11.8. Strict 2PL Lock Graph

The strict 2PL lock manager requires minimal modification of the algorithm of Algorithm 11.1. In fact, the only modification that is necessary is to change the section that handles the responses from the data processor. This is necessary to ensure that the locks are released only if the operation is a commit or abort. For completeness, we present the strict 2PL algorithm in its entirety in Algorithm 11.2. The transaction manager algorithm for 2PL scheduling is given in Algorithm 11.3.

Algorithm 11.2 *S2PL-LM*

```
declare-var
  msg : Message
  dop : Dbop
  Op : Operation
  x : DataItem
  T : TransactionId
  pm : Dpmsg
  res : DataVal
  SOP : OpSet
begin
  repeat
    WAIT(msg)
    case of msg
      Dbop :
      begin
        Op ← dop.opn
        x ← dop.data
        T ← dop.tid
        case of Op.
          Begin_transaction:
          begin
            send dop to the data processor
          end
          Read or Write:
          begin                                    {requires locking}
            find the lock unit lu such that x ⊆ lu
            if lu is unlocked or lock mode of lu is compatible with Op then
            begin
              set lock on lu in appropriate mode
              send dop to the data processor
            end
            else
              put dop on a queue for lu
            end-if
          end
          Abort or Commit:
          begin
            send dop to the data processor
```

```
            end
          end-case
        Dpmsg :
        begin
          Op ← pm.opn
          res ← pm.result
          T ← pm.tid
          if Op= Abort or Op=Commit then
          begin
            for each lock unit lu locked by T do
            begin
              release lock on lu held by T
              if there are no more locks on lu and
                 there are operations waiting in queue for lu then
              begin
                SOP ← first, operation from the queue
                SOP ← SOP ∪ {O|O is an operation on queue that
                                can lock lu in a compatible mode with
                                the current operations in SOP}
                set the locks on lu on behalf of operations in SOP
                for all the operations in SOP do
                  send each operation to the data processor
                end-for
              end-if
            end-for
          end-if
        end
      end-case
    until forever
  end. {S2PL-LM}
```

Algorithm 11.3 *2PL-TM*

```
  declare-var
    msg : Message
    Op : Operation
    x : DataItem
    T : TransactionId
    O : Dbop
    sm : Scmsg
    res : DataVal
    SOP : OpSet
  begin
    repeat
      WAIT(msg)
      case of msg
        Dbop :
```

```
        begin
          send O to the lock manager
        end
        Scmsg :                          {acknowledgment from the lock manager}
        begin
          Op ← sm.apn
          res ← sm.result
          T ← sm.tid
          case of Op
            Read:
            begin
              return res to the user application      (i.e., the transaction)
            end
            Write:
            begin
              inform user application of completion of the write
              return res to the user application
            end
            Commit:
            begin
              destroy T's workspace
              inform user application of successful completion of transaction
            end
            Abort:
            begin
              inform user application of completion of the abort of T
            end
          end-case
        end
      end-case
    until forever
  end. {2PL-TM}
```

We should note that even though a 2PL algorithm enforces conflict serializability, it does not allow all schedules that are conflict serializable. Consider the following schedule taken from [Agrawal and El-Abbadi, 1990]:

$$S = w_1(x)r_2(x)r_3(y)w_1(y)$$

S is not allowed by 2PL algorithm since T_1 would need to obtain a write lock on y after it releases its write lock on x. However, this history is serializable in the order $T_3 \rightarrow T_1 \rightarrow T_2$ [Agrawal and El-Abbadi, 1990]. The order of locking can be exploited to design locking algorithms that allow schedules such as these [Agrawal and El-Abbadi, 1990].

The main idea is to observe that in serializability theory, the order of serialization of conflicting operations is as important as detecting the conflict in the first place and this can be exploited in defining locking modes. Consequently, in addition

to read (shared) and write (exclusive) locks, a third lock mode is defined: *ordered shared*. Ordered shared locking of an object x by transactions T_i and T_j has the following meaning: Given a schedule S that allows ordered shared locks between operations $o \in T_i$ and $p \in T_j$, if T_i acquires o-lock before T_j acquires p-lock, then o is executed before p. Consider the compatibility table between read and write locks given in Figure 11.5. If the ordered shared mode is added, there are eight variants of this table. Figure 11.5 depicts one of them and two more are shown in Figure 11.9. In Figure 11.9(a), for example, there is an ordered shared relationship between $rl_j(x)$ and $wl_i(x)$ [denoted as $rl_j(x) \Rightarrow wl_i(x)$] indicating that T_i can acquire a write lock on x while T_j holds a read lock on x as long as the ordered shared relationship from $rl_j(x)$ to $wl_i(x)$ is observed. The eight compatibility tables can be compared with respect to their permissiveness (i.e., with respect to the histories that can be produced using them) to generate a lattice of tables such that the one in Fig. 11.5 is the most restrictive and the one in Fig. 11.9(b) is the most liberal.

	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	compatible	not compatible
$wl_i(x)$	ordered shared	not compatible

(a)

	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	compatible	ordered shared
$wl_i(x)$	ordered shared	ordered shared

(b)

Figure 11.9. Commutativity Table with Ordered Shared Lock Mode

In the above example, the write lock on behalf of T_i is said to be *on hold* since it was acquired after T_k acquired its read lock on x. The locking protocol that enforces a compatibility matrix involving ordered shared lock modes is identical to 2PL, except that a transaction may not release any locks as long as any of its locks are on hold. Otherwise circular serialization orders can exist.

11.3.1 Centralized 2PL

The 2PL algorithm discussed in the preceding section can easily be extended to the (replicated or partitioned) distributed DBMS environment. One way of doing this is to delegate lock management responsibility to a single site only. This means that only one of the sites has a lock manager; the transaction managers at the other sites communicate with it rather than with their own lock managers. This approach is also known as the *primary site* 2PL algorithm [Alsberg and Day, 1976].

The communication between the cooperating sites in executing a transaction according to a centralized 2PL (C2PL) algorithm is depicted in Figure 11.10. This communication is between the transaction manager at the site where the transaction is initiated (called the *coordinating* TM), the lock manager at the central site, and the data processors (DP) at the other participating sites. The participating sites are those at which the operation is to be carried out. The order of messages is denoted in the figure.

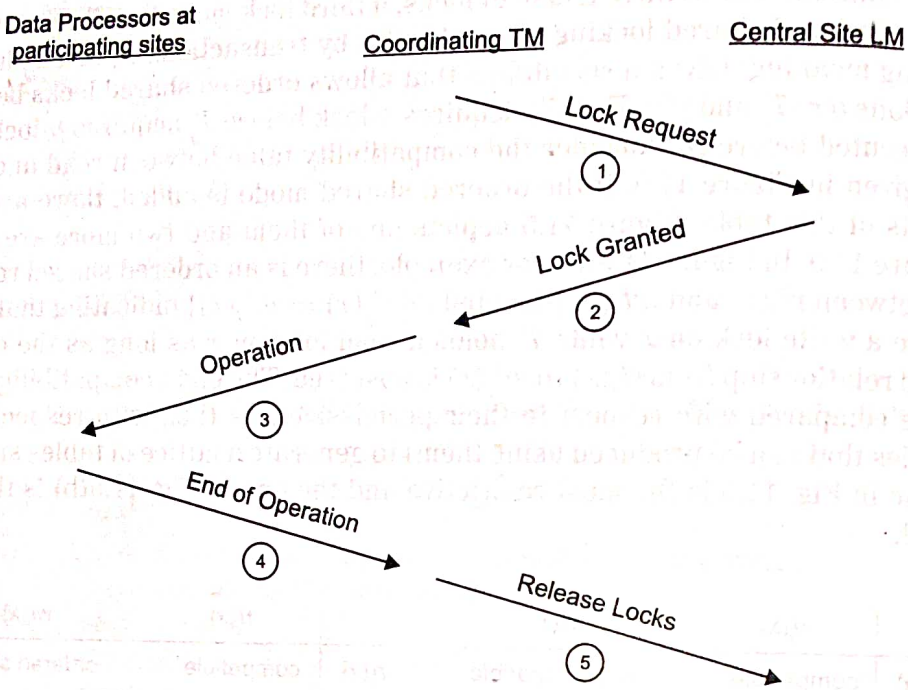

Figure 11.10. Communication Structure of Centralized 2PL

An important difference between the centralized TM algorithm and the TM algorithm of Algorithm 11.3 is that the distributed TM has to implement the replica control protocol if the database is replicated. The C2PL-LM algorithm also differs from the strict 2PL lock manager in one major way. The central lock manager does not send the operations to the respective data processors; that is done by the coordinating TM.

The centralized 2PL transaction management algorithm (C2PL-TM) that incorporates these changes is given in Algorithm 11.4, while the centralized 2PL lock management algorithm (C2PL-LM) is shown in Algorithm 11.5.

Algorithm 11.4 *C2PL-TM*

```
declare-var
    T : Transaction
    Op : Operation
    x : DataItem
    msg : Message
    O : Dbop
    pm : Dpmsg
    res : DataVal
    S : SiteSet
begin
    repeat
        WAIT(msg)
```

```
case of msg
  Dbop :
  begin
    Op ← O.opn
    x ← O.data
    T ← O.tid
    case of Op
      Begin-Transaction:
      begin
        S ← θ
      end
      Read:
      begin
        S ← S ∪ {the site that stores x and has the lowest
                 access cost to it}
        send O to the central lock manager
      end
      Write:
      begin
        S ← S ∪ {S_i | x is stored at site S_i}
        send O to the central lock manager
      end
      Abort or Commit:
      begin
        send O to the central lock manager
      end
    end-case
  end
  Scmsg :                    {lock request granted on locks released}
  begin
    if lock request granted then
      send O to the data processors in S
    else
      inform user about the termination of transaction
    end-if
  end
  Dpmsg :
  begin
    Op ← pm.opn
    res ← pm.result
    T ← pm.tid
    case of Op
      Read:
      begin
        return res to the user application (i.e., the transaction)
      end
```

```
            Write:
            begin
              inform user application of completion of the write
            end
            Commit:
            begin
              if commit msg has been received from all participants then
              begin
                inform user application of successful completion of
                  transaction
                send pm to the central lock manager
              else                    {wait until commit msg comes from all}
                record the arrival of the commit message
              end-if
            end
            Abort:
            begin
              inform user application of completion of the abort of T
              send pm to the central lock manager
            end
          end-case
        end
      end-case
    until forever
  end. {C2PL-TM}
```

Algorithm 11.5 *C2PL-LM*

```
declare-var
  msg : Message
  dop : SingleOp
  Op : Operation
  x : DataItem
  T : TransactionId
  SOP : OpSet
begin
  repeat
    WAIT(msg)        {The only msg that can arrive is from coordinating TM}
    Op ← dop.opn
    x ← dop.data
    T ← dop.tid
    case of Op
      Read or Write:
      begin
        find the lock unit lu such that x ⊆ lu
        if lu is unlocked or lock mode of lu is compatible,with Op then
```

```
            begin
              set lock on lu in appropriate mode
              msg ← "Lock granted for operation dop"
              send mag to the coordinating TM of T
            end
            else
              put Op on a queue for lu
            end-if
          end
          Commit or Abort:
          begin
            for each lock unit lu locked by T do
            begin
              release lock on lu held by T
              if there are operations waiting in queue for lu then
              begin
                SOP ← first operation (call O) from the queue
                SOP ← SOP ∪ {O|O} is an operation on queue that can lock lu
                   in a compatible mode with the current operations in SOP}
                set the locks on lu on behalf of operations in SOP
                for all the operations O in SOP do
                begin
                  msg ← "Lock granted for operation O"
                  send msg to all the coordinating TM's
                end-for
              end-if
            end-for
            msg ← "Locks of T released"
            send msg to the coordinating TM of T
          end
        end-case
      until forever
    end. {C2PL-LM}
```

One common criticism of C2PL algorithms is that a bottleneck may quickly form around the central site. Furthermore, the system may be less reliable since the failure or inaccessibility of the central site would cause major system failures. There are studies that indicate that the bottleneck will indeed form as the transaction rate increases, but is insignificant at low transaction rates ([Özsu, 1985a], [Koon and Özsu, 1986]). Furthermore, sharp performance degradation at high loads is observed in other locking-based algorithms as well.

11.3.2 Primary Copy 2PL

Primary copy 2PL (PC2PL) is a straightforward extension of centralized 2PL in an attempt to counter the latter's potential performance problems, discussed above.

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. The transaction managers then send their lock and unlock requests to the lock managers that axe responsible for that specific lock unit. Thus the algorithm treats one copy of each data item as its primary copy.

We do not give the detailed primary copy 2PL algorithm since the changes from centralized 2PL are minimal. Basically, the only change is that the primary copy locations have to be determined for each data item prior to sending a lock or unlock request to the lock manager at that site. This is a directory design and management issue that we discussed in Chapter 4.

Primary copy 2PL was proposed for the prototype distributed version of INGRES [Stonebraker and Neuhold, 1977]. Even though it demands a more sophisticated directory at each site, it also reduces the load of the central site without causing a large amount of communication among the transaction managers and lock managers. In one sense it is an intermediate step between the centralized 2PL that we discussed in, the preceding section and the distributed 2PL that we will cover next.

11.3.3 Distributed 2PL

Distributed 2PL (D2PL) expects the availability of lock managers at each site. If the database is not replicated, distributed 2PL degenerates into the primary copy 2PL algorithm. If data is replicated, the transaction implements the ROWA replica control protocol.

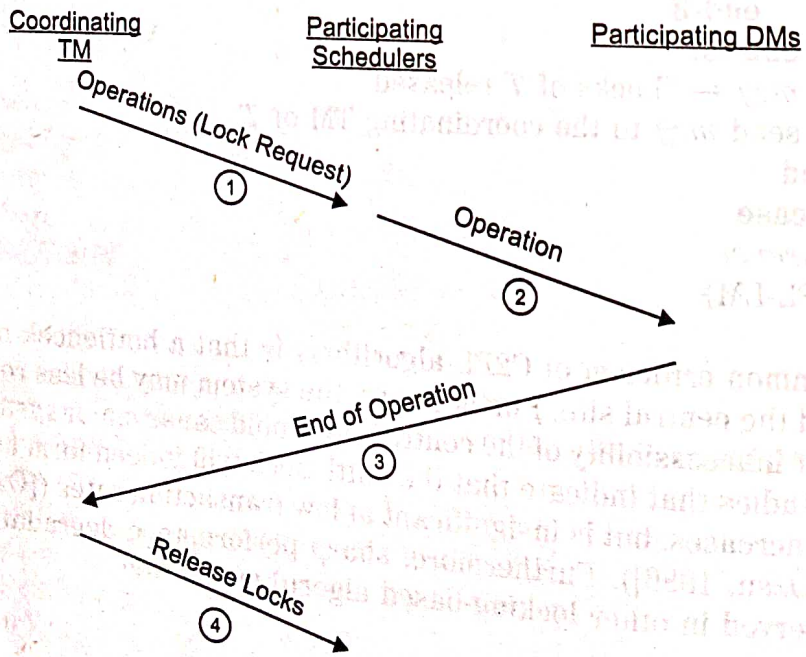

Figure 11.11. Communication Structure of Distributed 2PL

The communication between cooperating sites that execute a transaction according to the distributed 2PL protocol is depicted in Figure 11.11. Notice that Figure 11.11 does not show application of the ROWA rule.

The distributed 2PL transaction management algorithm is similar to the C2PL-TM, with two major modifications. The messages that are sent to the central site lock manager in C2PL-TM are sent to the lock managers at all participating sites in D2PL-TM. The second difference is that the operations are not passed to the data processors by the coordinating transaction manager, but by the participating lock managers. This means that the coordinating transaction manager does not wait for a "lock request granted" message. Another point about Figure 11.11 is the following. The participating data processors send the "end of operation" message to the coordinating TM. The alternative is for each DP to send it to its own lock manager who can then release the locks and inform the coordinating TM. We have chosen to describe the former since it uses an LM algorithm identical to the strict 2PL lock manager that we have already discussed and it makes the discussion of the commit protocols simpler (see Chapter 12). Owing to these similarities, we do not give the distributed TM and LM algorithms here. Distributed 2PL algorithms are used in System R* [Mohan et al., 1986] and in NonStop SQL ([Tandem, 1987], [Tanderm, 1988] and [Borr, 1988]).

11.4 TIMESTAMP-BASED CONCURRENCY CONTROL ALGORITHMS

Unlike the locking-based algorithms, timestamp-based concurrency control algorithms do not attempt to maintain serializability by mutual exclusion. Instead, they select, a priori, a serialization order and execute transactions accordingly. To establish this ordering, the transaction manager assigns each transaction T_i a unique *timestamp*, $ts(T_i)$, at its initiation.

A timestamp is a simple identifier that serves to identify each transaction uniquely and to permit ordering. *Uniqueness* is only one of the properties of timestamp generation. The second property is *monotonicity*. Two timestamps generated, by the same transaction manager should be monotonically increasing. Thus timestamps are values derived from a totally ordered domain. It is this second property that differentiates a timestamp from a transaction identifier.

There are a number of ways that timestamps can be assigned. One method is to use a global (systemwide) monotonically increasing counter. However, the maintenance of global counters is a problem in distributed systems. Therefore, it is preferable that each site autonomously assign timestamps based on its local counter. To maintain uniqueness, each site appends its own identifier to the counter value. Thus the timestamp is a two-tuple of the form <local counter value, site identifier>. Note that the site identifier is appended in the least significant position. Hence it serves only to order the timestamps of two transactions that might have been assigned the same local counter value. If each system can access its own system clock, it is possible to use system clock values instead of counter values.

With this information, it is simple to order the execution of the transactions' operations according to their timestamps. Formally, the timestamp ordering (TO) rule can be specified as follows:

TO Rule Given two conflicting operations O_{ij} and O_{kl} belonging, respectively, to transactions T_i and T_k, O_{ij} is executed before O_{kl} if and only if $ts(T_i) < ts(T_k)$. In this case, T_i is said to be the *older* transaction and T_k is said to be the *younger* one.

A scheduler that enforces the TO rule checks each new operation against conflicting operations that have already been scheduled. If the new operation belongs to a transaction that is younger than all the conflicting ones that have already been scheduled, the operation is accepted; otherwise, it is rejected, causing the entire transaction to restart with a *new* timestamp.

A timestamp ordering scheduler that operates in this fashion is guaranteed to generate serializable schedules. However, this comparison between the transaction timestamps can be performed only if the scheduler has received all the operations to be scheduled. If operations come to the scheduler one at a time (which is the realistic case), it is necessary to be able to detect if an operation has arrived out of sequence. To facilitate this check, each data item x is assigned two timestamps: a *read timestamp* [$rts(x)$], which is the largest of the timestamps of the transactions that have read x, and a *write timestamp* [$wts(x)$], which is the largest of the timestamps of the transactions that have written (updated) x. It is now sufficient to compare the timestamp of an operation with the read and write timestamps of the data item that it wants to access to determine if any transaction with a larger timestamp has already accessed the same data item.

Architecturally (see Figure 11.5), the transaction manager is responsible for assigning a timestamp to each new transaction and attaching this timestamp to each database operation that it passes on to the scheduler. The lock manager is responsible for keeping track of read and write timestamps as well as performing the serializability check.

11.4.1 Basic TO Algorithm

The basic TO algorithm is a straightforward implementation of the TO rule. The coordinating transaction manager assigns the timestamp to each transaction, determines the sites where each data item is stored, and sends the relevant operations to these sites. The basic TO transaction manager algorithm (BTO-TM) is depicted in Algorithm 11.6. The schedulers at each site simply enforce the TO rule. The scheduler algorithm is given in Algorithm 11.7.

Algorithm 11.6 *BTO-TM*

declare-var

T : *Transaction*
Op : *Operation*
x : *DataItem*

```
    msg : Message
    O : Dbop
    pm : Dpmsg
    res : DataVal
    S : SiteSet
  begin
    repeat
      WAIT(msg)
      case of msg
        Dbop :                    {database operation from the application program}
        begin
          Op ← O.opn
          x ← O.data
          T ← O.tid
          case of Op
            Begin_Transaction:
            begin
              S ← ϕ
              assign a timestamp to T [ts(T)}
              end
        Read:
        begin
          S ← S∪ {the site that stores x and has the lowest access cost to it}
          send O and ts(T) to the scheduler at S
        end
        Write:
        begin
          S ← S ∪ {S_i | x is stored at site S_i}
          send O and ts(T) to the schedulers at S
        end
        Abort or Commit:
        begin
          send O to the schedulers in S
        end
      end-case
    Scmsg :                       {the operation must have been rejected by a scheduler}
  begin
    msg ← "Abort T"
    send msg to schedulers in S
    restart T                                  {will assign a new timestamp}
  end
  Dpmsg :
  begin
    Op ← pm.opn res ← pm.result   T ← pm.tid
    case of Op
      Read:
```

```
        begin
          return res to the user application (i.e., the transaction)
        end
        Write:
        begin
          inform user application of completion of the write
        end
        Commit:
        begin
          inform user application of successful completion of transaction
        end
        Abort:
        begin
          inform user application of completion of the abort of T
        end
          end-case
        end
      end-case
    until forever
end. {BTO-TM}
```

Algorithm 11.7 *BTO-SC*

```
declare-var
  msg : Message
  dop : SingleOp
  Op : Operation
  x : DataItem
  T : TransactionId
  SOP : OpSet
begin
  repeat
    WAIT(msg)
    case of msg
      Dbop :    {database operation passed from the transaction manager}
      begin
        Op ← dop.opn
        x ← dop.data
        T ← dop-tid
        save initial rts(x) and wts(x)
        case of Op
          Read:
          begin
            if ts(T) > rts(x) then
            begin
              send dop to the data processor
              rts(x) ← ts(T)
```

```
            end
            else begin
              msg ← "Reject T"
              send msg to the coordinating TM
            end
            end-if
          end {of Read case}
          Write:
          begin
            if ts(T) > rts(x) and ts(T) > wts(x) then
            begin
              send dop to the data processor
              rts(x) ← ts(T)
              wts(x) ← ts(T)
            end
            else begin
              msg ← "Reject T"
              send msg to the coordinating TM
            end
            end-if
          end {of Write case}
          Commit:
          begin
            send dop to the data processor
          end {of Commit case}
          Abort:
          begin
            for all x that has been accessed by T do
              reset rts(x) and wts(x) to their initial values
            end-for
            send dop to the data processor
          end {of Abort case}
        end-case
      end
    end-case
  until forever
end. {BTO-SC}
```

As indicated before, a transaction which contains an operation that is rejected by a scheduler is restarted by the transaction manager with a new timestamp. This ensures that the transaction has a chance to execute in its next try. Since the transactions never wait while they hold access rights to data items, the basic TO algorithm never causes deadlocks. However, the penalty of deadlock freedom is potential restart of a transaction numerous times. There is an alternative to the basic TO algorithm that reduces the number of restarts, which we consider in the next section.

Another detail that needs to be considered relates to the communication between the scheduler and the data processor. When an accepted operation is passed on to the data processor, the scheduler needs to refrain from sending another incompatible, but acceptable operation to the data processor until the first is processed and acknowledged. This is a requirement to ensure that the data processor executes the operations in the order in which the scheduler passes them on. Otherwise, the read and write timestamp values for the accessed data item would not be accurate.

Example 11.8

Assume that the TO scheduler first receives $W_i(x)$ and then receives $W_j(x)$, where $ts(T_i) < ts(T_j)$. The scheduler would accept both operations and pass them on to the data processor. The result of these two operations is that $wts(x) = ts(T_j)$, and we then expect the effect of $W_j(x)$ to be represented in the database. However, if the data processor does not execute them in that order, the effects on the database will be wrong.

The scheduler can enforce the ordering by maintaining a queue for each data item that is used to delay the transfer of the accepted operation until an acknowledgment is received from the data processor regarding the previous operation on the same data item. This detail is not shown in Algorithm 11.7.

Such a complication does not arise in 2PL-based algorithms because the lock manager effectively orders the operations by releasing the locks only after the operation is executed. In one sense the queue that the TO scheduler maintains may be thought of as a lock. However, this does not imply that the schedule generated by a TO scheduler and a 2PL scheduler would always be equivalent. There are some schedules that a TO scheduler would generate that would not be admissible by a 2PL schedule.

Remember that in the case of strict 2PL algorithms, the releasing of locks is delayed further, until the commit or abort of a transaction. It is possible to develop a strict TO algorithm by using a similar scheme. For example, if $W_1(x)$ is accepted and released to the data processor, the scheduler delays all $R_j(x)$ and $W_j(x)$ operations (for all T_j) until T_i terminates (commits or aborts).

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 schedulers 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 to the transaction managers at the other sites, instead of to the schedulers. 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. Our discussion follows that of [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 schedulers 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 which 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 schedulers 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.

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

$$ts(T_i) < ts(x_r) < ts(Tj)$$

then $W_i(x)$ is rejected.

A scheduler that processes the read and the write requests of transactions according to the rules noted above is guaranteed to generate serializable schedules. 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.

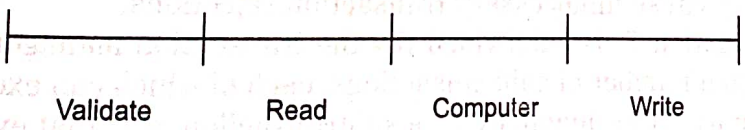

Figure 11.12. Phases of Pessimistic Transaction Execution

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

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

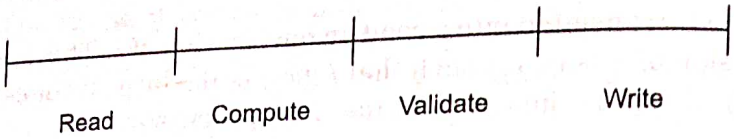

Figure 11.13. Phases of Optimistic Transaction Execution

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.

It is possible to design locking-based optimistic concurrency control algorithms (see [Bernstein et al., 1987]). However, the original optimistic proposals ([Kung and Robinson, 1981], [Thomas, 1979]) are based on timestamp ordering. Therefore, we describe only the optimistic approach using timestamps. Our discussion is brief and emphasizes cc cepts rather than implementation details. The reasons for this are twofold. First, most of the current work on optimistic methods concentrates on centralized rather than distributed DBMSs. Second, optimistic algorithms have not been implemented in any commercial or prototype DBMS. Therefore, the information regarding their implementation trade-offs is insufficient. As a matter of fact, the only centralized implementation of optimistic concepts (not the full algorithm) is in IBM's IMS-FASTPATH, which provides primitives that permit the programmer to access the database in an optimistic manner.

The algorithm that we discuss was proposed in [Kung and Robinson, 1981] and was later extended for distributed DBMS [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 subtransaciion of T_i that executes at site j. Until 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 Ty 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.

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

[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.

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

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 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}.

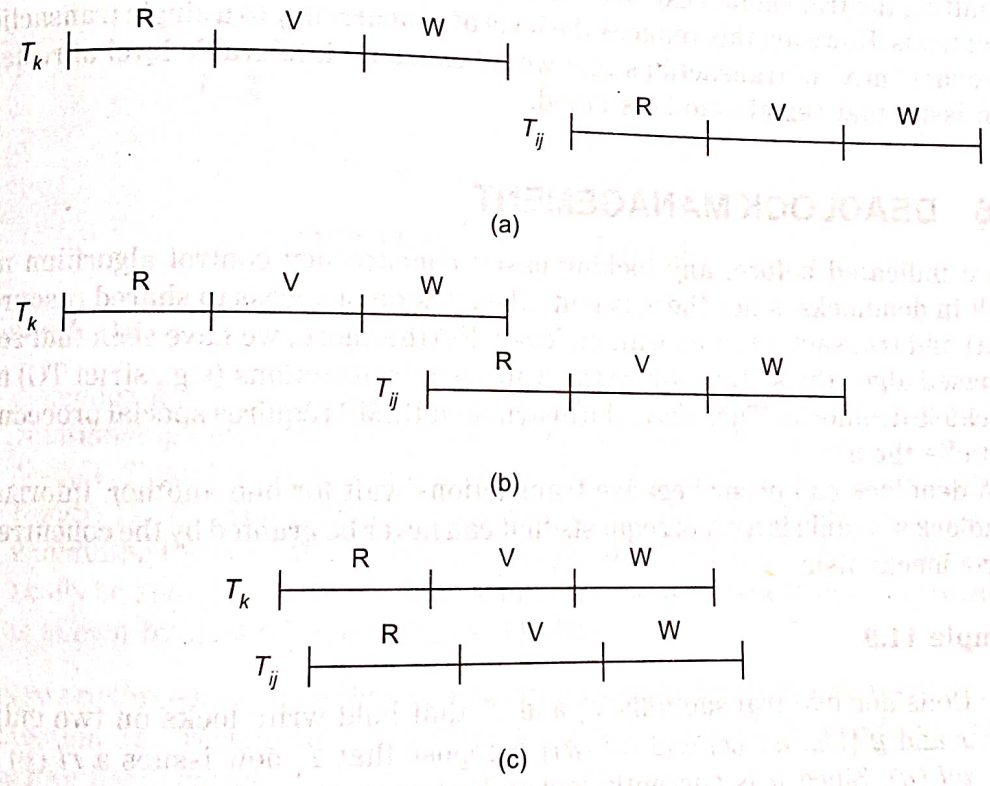

Figure 11.14. Possible Execution Scenarios

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 the optimistic concurrency control algorithms is their potential to allow a higher level of concurrency. It has been shown [Kung and Robinson, 1981] that when transaction conflicts are very rare, the optimistic mechanism performs better than locking. 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-far 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 arc $T_i \rightarrow T_j$ exists in the WFG if transaction T_j is waiting for T_j to release a lock on some entity. Figure 11.15 depicts the WFG for Example 11.9.

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.

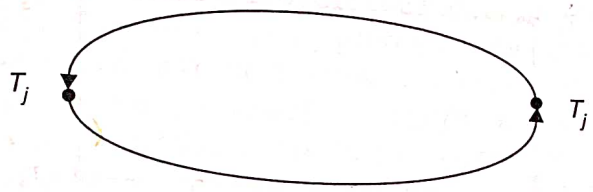

Figure 11.15. A WFG Example

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.[4] In the remainder of this section we discuss each approach in more detail.

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

[4] Of course, there is a fourth alternative that the system ignore deadlocks and require either that the application programmer deal with it or that the system be restarted. However, we obviously do not consider this to be a serious alternative.

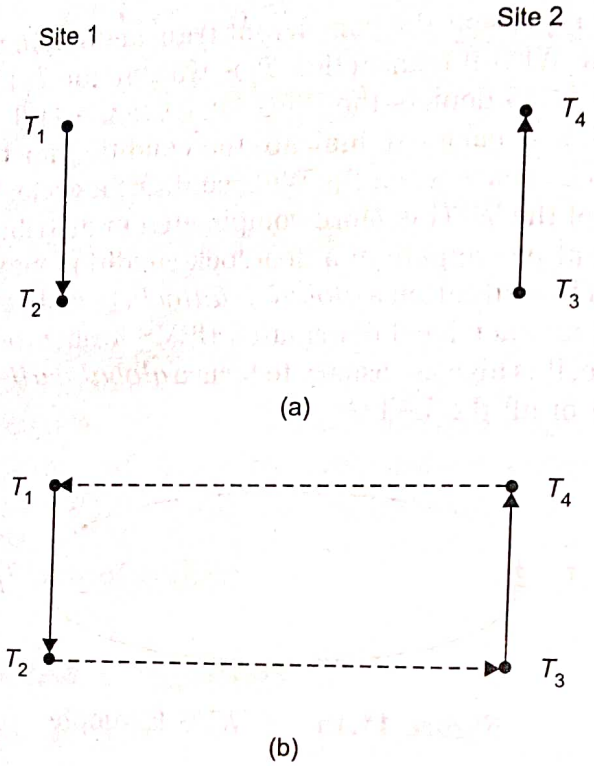

Figure 11.16. Difference between LWFG and GWFG

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.[5]

11.6.2 Deadlock Avoidance

Deadlock avoidance schemes either employ concurrency control techniques that will never result in deadlocks or require that schedulers detect potential deadlock

[5]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.

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 runtime 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. We should first indicate that there are three fundamental methods of detecting distributed deadlocks. These are commonly called *centralized, distributed,* and *hierarchical deadlock detection.*

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

situations in advance and ensure 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 nonpreemptive 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. It follows the following rule:

> **WAIT-DIE Rule.** If T_i requests a lock on a data item that is already locked by T_j, T_j 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:

> **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_j.

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

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.

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 local WFG. Each site also sends its local WFG 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, detects it, where x is either one of 1, 2, 3, or 4.

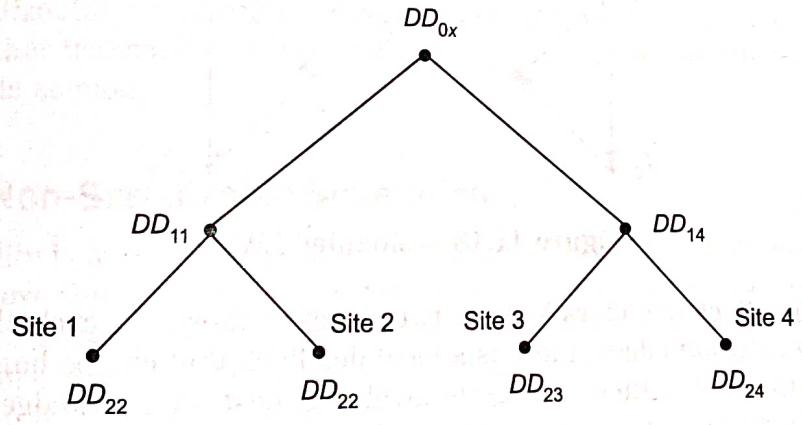

Figure 11.17. Hierarchical Deadlock Detection

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 nontrivial modifications to the lock and transaction manager algorithms.

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 which communicate their local WFGs 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 local WFG 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 local WFGs.
2. The edges in the local WFG which show that local transactions are waiting for transactions at other sites are joined with edges in the local WFGs 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.

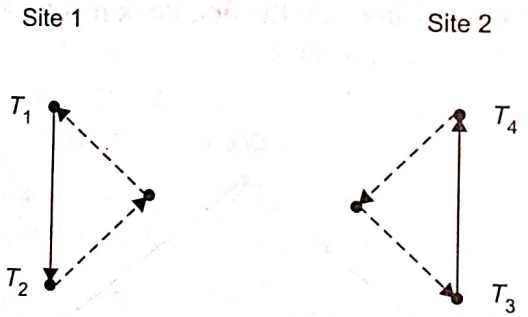

Figure 11.18. Modified LWFGs

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 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. The algorithm proposed in [Obermarck, 1982] 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 $T_i \rightarrow \ldots \rightarrow T_j$. 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 Schedules

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 schedules 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 schedule 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. In [Garcia-Molina, 1983], transactions are classified 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 in [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.

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 consider closed nested transactions [Moss, 1985] first. 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 sate, 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.

Prom 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. These subtransactions

commit commands do not have the same semantics as given in Algorithms 11.5 and 11.6.

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:

T_1: Withdraw(o, x)
Deposit(p, x)

T_2: Withdraw(o, y)
Deposit(p, y)

The notation here is that each T_i withdraws x 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. This results in a layered abstraction of transaction execution as depicted hi Figure 11.19 which is taken from [Weikum, 1991].

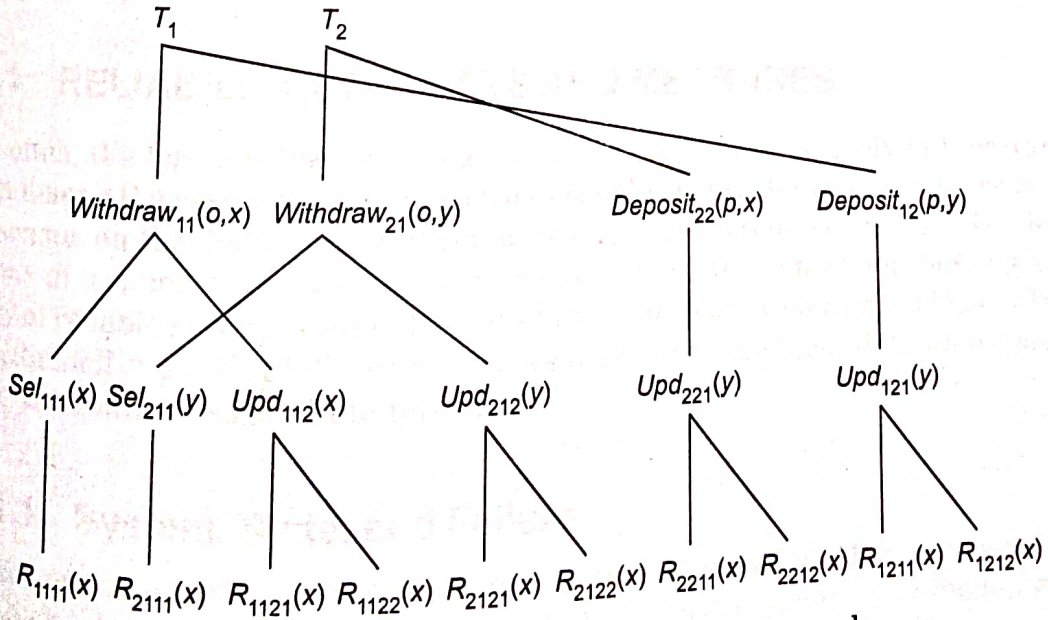

Figure 11.19. Multilevel Transaction Example

REVIEW QUESTIONS

11.1 Explain serializability theory with an example.
11.2 What is classification of concurrency control algorithm?
11.3 Explain the compatibility matrix of lock modes.
11.4 Explain timestamp-based concurrency control algorithms.
11.5 Explain optimistic concurrency control algorithms.
11.6 How do you perform deadlock management?
11.7 Explain "relaxed" concurrency control.