requests to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered. At the other extreme, the token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded.

The delay from the moment a process needs to enter a critical region until its actual entry also varies for the three algorithms. When critical regions are short and rarely used, the dominant factor in the delay is the actual mechanism for entering a critical region. When they are long and frequently used, the dominant factor is waiting for everyone else to take their turn. In Fig. 5-16 we show the former case. It takes only two message times to enter a critical region in the centralized case, but \(2(n - 1)\) message times in the distributed case, assuming that messages are sent one after the other. For the token ring, the time varies from 0 (token just arrived) to \(n - 1\) (token just departed).

Finally, all three algorithms suffer badly in the event of crashes. Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system. It is ironic that the distributed algorithms are even more sensitive to crashes than the centralized one. In a fault-tolerant system, none of these would be suitable, but if crashes are very infrequent, they might do.

### 5.6 DISTRIBUTED TRANSACTIONS

A concept that is strongly related to mutual exclusion is that of a transaction. Mutual exclusion algorithms ensure that a shared resource such as a file, printer, and so on, is accessed by at most one process at a time. Transactions have in common that they also protect a shared resource against simultaneous access by several concurrent processes. In particular, transactions are used to protect shared data. However, transactions can do much more. In particular, they allow a process to access and modify multiple data items as a single atomic operation. If the process backs out halfway during the transaction, everything is restored to the point just before the transaction started. In this section we take a closer look at the concept of a transaction, and in particular concentrate on a transaction's capabilities for synchronizing multiple processes to protect shared data.
5.6.1 The Transaction Model

The original model of the transaction comes from the world of business. Suppose that the International Dingbat Corporation needs a batch of widgets. They approach a potential supplier, U.S. Widget, known far and wide for the quality of its widgets, for a quote on 100,000 10-cm purple widgets for June delivery. U.S. Widget makes a bid on 100,000 4-inch mauve widgets to be delivered in December. International Dingbat agrees to the price, but dislikes mauve, wants them by July, and insists on 10 cm for its international customers. U.S. Widget replies by offering 3 15/16 inch lavender widgets in October. After much further negotiation, they finally agree on 3 959/1024 inch violet widgets for delivery on August 15.

Up until this point, both parties are free to terminate the discussion, in which case the world returns to the state it was in before they started talking. However, once both companies have signed a contract, they are both legally bound to complete the sale, come what may. Thus until both parties have signed on the dotted line, either one can back out and it is as if nothing ever happened, but at the moment they both sign, they pass the point of no return and the transaction must be carried out.

The computer model is similar. One process announces that it wants to begin a transaction with one or more other processes. They can negotiate various options, create and delete entities, and perform operations for a while. Then the initiator announces that it wants all the others to commit themselves to the work done so far. If all of them agree, the results are made permanent. If one or more processes refuse (or crash before agreement), the situation reverts to exactly the state it was in before the transaction began, with all side effects on files, databases, and so on, magically wiped out. This all-or-nothing property eases the programmer's job.

The use of transactions in computer systems goes back to the 1960s. Before there were disks and online databases, all files were kept on magnetic tape. Imagine a supermarket with an automated inventory system. Every day after closing, a computer run was made with two input tapes. The first one contained the complete inventory as of opening time that morning. The second one contained a list of the day's updates: products sold to customers and products delivered by suppliers. The computer read both input tapes and produced a new master inventory tape, as shown in Fig. 5.17.

The great beauty of this scheme (although the people who actually had to live with it probably did not realize it at the time) is that if a run failed for any reason, all the tapes could be rewound and the job restarted with no harm done. Primitive as it was, the old magnetic tape system had the all-or-nothing property of a transaction.

Now look at a modern banking application that updates an online database in place. The customer calls up the bank using a PC with a modem with the intention
of withdrawing money from one account and depositing it in another. The operation is performed in two steps:

1. Withdraw an amount \( a \) from account 1.
2. Deposit amount \( a \) to account 2.

If the telephone connection is broken after the first step but before the second one, the first account will have been debited but the second one will not have been credited. The money vanishes into thin air.

Being able to group these two operations in a transaction would solve the problem. Either both would be completed, or neither would be completed. A key issue is therefore rolling back to the initial state if the transaction fails to complete. What we really want is a way to rewind the database as we were able to do with the magnetic tapes. This ability is what a transaction has to offer.

Programming using transactions requires special primitives that must either be supplied by the underlying distributed system or by the language runtime system. Typical examples of transaction primitives are shown in Fig. 5-18. The exact list of primitives depends on what kinds of objects are being used in the transaction. In a mail system, there might be primitives to send, receive, and forward mail. In an accounting system, they might be quite different. READ and WRITE are typical examples, however. Ordinary statements, procedure calls, and so on, are also allowed inside a transaction.

BEGIN_TRANSACTION and END_TRANSACTION are used to delimit the scope of a transaction. The operations between them form the body of the transaction. Either all of these operations are executed or none are executed. These may be system calls, library procedures, or bracketing statements in a language, depending on the implementation.

Consider, as an example, the process of reserving a seat from White Plains, New York, to Malindi, Kenya, in an airline reservation system. One possible route is White Plains to JFK, JFK to Nairobi, and Nairobi to Malindi. In Fig. 5-19(a) we see reservations for these three flights being made as three different operations.
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Mark the start of a transaction</td>
</tr>
<tr>
<td>END_TRANSACTION</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT_TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, or otherwise</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, or otherwise</td>
</tr>
</tbody>
</table>

Figure 5-18. Example primitives for transactions.

Now suppose that the first two flights have been reserved but the third one is booked solid. The transaction is aborted and the results of the first two bookings are undone—the airline database is restored to the value it had before the transaction started [see Fig. 5-19(b)]. It is as though nothing happened.

BEGIN_TRANSACTION
  reserve WP → JFK;
  reserve JFK → Nairobi;
  reserve Nairobi → Malindi;
END_TRANSACTION

BEGIN_TRANSACTION
  reserve WP → JFK;
  reserve JFK → Nairobi;
  reserve Nairobi → Malindi full ⇒
END_TRANSACTION

Figure 5-19. (a) Transaction to reserve three flights commits. (b) Transaction aborts when third flight is unavailable.

The all-or-nothing property of transactions is one of the four characteristic properties that transactions have. More specifically, transactions are:

1. Atomic: To the outside world, the transaction happens indivisibly.
2. Consistent: The transaction does not violate system invariants.
3. Isolated: Concurrent transactions do not interfere with each other.
4. Durable: Once a transaction commits, the changes are permanent.

These properties are often referred to by their initial letters, **ACID**.

The first key property exhibited by all transactions is that they are **atomic**. This property ensures that each transaction either happens completely, or not at all, and if it happens, it happens in a single indivisible, instantaneous action. While a transaction is in progress, other processes (whether or not they are themselves involved in transactions) cannot see any of the intermediate states.

Suppose, for example, that some file is 10 bytes long when a transaction starts to append to it. If other processes read the file while the transaction is in progress, they see only the original 10 bytes, no matter how many bytes the transaction has appended. If the transaction commits successfully, the file grows instantaneously
to its new size at the moment of commitment, with no intermediate states, no matter how many operations it took to get it there.

The second property says that they are **consistent**. What this means is that if the system has certain invariants that must always hold, if they held before the transaction, they will hold afterward too. For example, in a banking system, a key invariant is the law of conservation of money. After any internal transfer, the amount of money in the bank must be the same as it was before the transfer, but for a brief moment during the transaction, this invariant may be violated. The violation is not visible outside the transaction, however.

The third property says that transactions are **isolated** or **serializable**. What it means is that if two or more transactions are running at the same time, to each of them and to other processes, the final result looks as though all transactions ran sequentially in some (system dependent) order. We return to serializability below.

The fourth property says that transactions are **durable**. It refers to the fact that once a transaction commits, no matter what happens, the transaction goes forward and the results become permanent. No failure after the commit can undo the results or cause them to be lost. Durability is discussed extensively in Chap. 7.

### 5.6.2 Classification of Transactions

So far, we have basically considered a transaction as a series of operations that satisfy the ACID properties. This type of transaction is also called a **flat transaction**. Flat transactions are the simplest type of transaction, and are most often used. However, flat transactions have a number of limitations that have led to alternative models. Below we discuss two important classes: nested transactions and distributed transactions. Other classes are discussed extensively in (Gray and Reuter, 1993).

**Some Limitations of Flat Transactions**

The main limitation of flat transactions is that they do not allow partial results to be committed or aborted. In other words, the strength of the atomicity property of a flat transaction also is partly its weakness.

Consider again booking a flight from New York to Kenya, as shown in Fig. 5-19. Suppose that the entire trip was being sold as a relatively cheap single package deal, for which reason the three parts were grouped into a single transaction. At the time we discover that only the last part cannot be booked, it may be decided to still confirm the reservations of the first two parts. For example, we may have also found out that it was already hard enough to reserve the flight from JFK to Nairobi. Aborting the entire transaction would mean that we would have to make a second attempt to reserve a seat on that flight, which by then may fail.
Consequently, what we need in this case is to only partially commit the transaction. Flat transactions do not allow this.

As another example, consider a Web site in which a hyperlink is implemented as a bidirectional reference. In other words, if a Web page $W_1$ contains a URL to a page $W_2$, then $W_2$ knows that $W_1$ refers to it (see, e.g., Kappe, 1999). Now suppose a page $W$ is moved to another location or replaced by another page. In that case, all hyperlinks to $W$ should be updated, and preferably in a single atomic operation, or otherwise there will (temporarily) be dangling references to $W$. In theory, a flat transaction can be used here. The transaction consists of updating $W$ and a series of operations, where each operation updates a single Web page containing a hyperlink to $W$.

The problem, however, is that such a transaction may take hours to complete. Not only may pages referring to $W$ be scattered across the Internet, there may also be thousands of them that need to be updated. Doing each update as a separate transaction is no good, for in that case some Web pages may have correct links, while others will not. A possible solution in this case is to commit updates, but also to keep the old $W$ for those pages whose link has not yet been updated.

**Nested Transactions**

Some of the limitations mentioned above can be solved by making use of **nested transactions**. A nested transaction is constructed from a number of subtransactions. The top-level transaction may fork off children that run in parallel with one another, on different machines, to gain performance or simplify programming. Each of these children may also execute one or more subtransactions, or fork off its own children.

Subtransactions give rise to a subtle, but important, problem. Imagine that a transaction starts several subtransactions in parallel, and one of these commits, making its results visible to the parent transaction. After further computation, the parent aborts, restoring the entire system to the state it had before the top-level transaction started. Consequently, the results of the subtransaction that committed must nevertheless be undone. Thus the permanence referred to above applies only to top-level transactions.

Since transactions can be nested arbitrarily deeply, considerable administration is needed to get everything right. The semantics are clear, however. When any transaction or subtransaction starts, it is conceptually given a private copy of all data in the entire system for it to manipulate as it wishes. If it aborts, its private universe just vanishes, as if it had never existed. If it commits, its private universe replaces the parent’s universe. Thus if a subtransaction commits and then later a new subtransaction is started, the second one sees the results produced by the first one. Likewise, if an enclosing (higher-level) transaction aborts, all its underlying subtransactions have to be aborted as well.
Distributed Transactions

Nested transactions are important in distributed systems, for they provide a natural way of distributing a transaction across multiple machines. However, nested transactions generally follow a logical division of the work of the original transaction. For example, the transaction by which three different flights needed to be reserved as shown in Fig. 5-19, can be logically split up into three subtransactions. Each of these subtransactions can be managed separately and independent of the other two.

However, a logical division of a nested transaction into subtransactions does not necessarily imply that all distribution is taken care of. For example, the subtransaction handling the seat reservation from New York to Nairobi, may still have to access two databases, one in each city. In this case, the subtransaction can no longer be subdivided into smaller subtransactions, because, logically, there are none; a reservation itself is an indivisible operation.

In this case, the situation that we are faced with is that of a (flat) subtransaction that operates on data that are distributed across multiple machines. Such transactions are known as distributed transactions. The difference between nested and distributed transactions is subtle, but important. A nested transaction is a transaction that is logically decomposed into a hierarchy of subtransactions. In contrast, a distributed transaction is logically a flat, indivisible transaction that operates on distributed data. This difference is illustrated in Fig. 5-20.

![Diagram of nested and distributed transactions](image)

Figure 5-20. (a) A nested transaction. (b) A distributed transaction.

The main problem with distributed transactions is that separate distributed algorithms are needed to handle the locking of data and committing the entire transaction. Distributed locking is discussed below. A detailed presentation of distributed commit protocols is deferred until Chap. 7, where we discuss fault tolerance and recovery mechanisms, to which commit protocols belong.
5.6.3 Implementation

Transactions sound like a great idea, but how are they implemented? That is the question we will tackle in this section. To simplify matters, we consider transactions on a file system. It should be clear by now that if each process executing a transaction just updates the file it uses in place, transactions will not be atomic and changes will not vanish magically if the transaction aborts. Clearly, some other implementation method is required. Two methods are commonly used, which are discussed in turn below.

Private Workspace

Conceptually, when a process starts a transaction, it is given a private workspace containing all the files to which it has access. Until the transaction either commits or aborts, all of its reads and writes go to the private workspace, rather than directly to the file system. This observation leads directly to the first implementation method: actually giving a process a private workspace at the instant it begins a transaction.

The problem with this technique is that the cost of copying everything to a private workspace is prohibitive, but various optimizations make it feasible. The first optimization is based on the realization that when a process reads a file but does not modify it, there is no need for a private copy. It can just use the real file (unless it has been changed since the transaction started). Consequently, when a process starts a transaction, it is sufficient to create a private workspace for it that is empty except for a pointer back to its parent’s workspace. When the transaction is at the top level, the parent’s workspace is the file system. When the process opens a file for reading, the back pointers are followed until the file is located in the parent’s (or further ancestor’s) workspace.

When a file is opened for writing, it can be located in the same way as for reading, except that now it is first copied to the private workspace. However, a second optimization removes most of the copying, even here. Instead of copying the entire file, only the file’s index is copied into the private workspace. The index is the block of data associated with each file telling where its disk blocks are. In UNIX, the index is the inode. Using the private index, the file can be read in the usual way, since the disk addresses it contains are for the original disk blocks. However, when a file block is first modified, a copy of the block is made and the address of the copy is inserted into the index, as shown in Fig. 5-21. The block can then be updated without affecting the original. Appended blocks are handled this way too. The new blocks are sometimes called shadow blocks.

As can be seen from Fig. 5-21(b), the process running the transaction sees the modified file, but all other processes continue to see the original file. In a more complex transaction, the private workspace might contain a large number of files instead of just one. If the transaction aborts, the private workspace is simply
deleted and all the private blocks that it points to are put back on the free list. If the transaction commits, the private indices are moved into the parent's workspace atomically, as shown in Fig. 5-21(c). The blocks that are no longer reachable are put onto the free list.

This scheme also works for distributed transactions. In that case, a process is started on each machine containing a file that is to be accessed as part of the transaction. Each process is given its own private workspace as described above. If the transaction aborts, all processes simply discard their private workspace. On the other hand, when the transaction commits, updates are propagated locally, at which point the transaction as a whole completes.

Writeahead Log

Another common method of implementing transactions is the writeahead log. With this method, files are actually modified in place, but before any block is changed, a record is written to a log telling which transaction is making the change, which file and block is being changed, and what the old and new values are. Only after the log has been written successfully is the change made to the file.

Fig. 5-22 gives an example of how the log works. In Fig. 5-22(a) we have a simple transaction that uses two shared variables (or other objects), $x$ and $y$, both initialized to 0. For each of the three statements inside the transaction, a log record is written before executing the statement, giving the old and new values. These values are separated by a slash in Fig. 5-22(b)-(d).
x = 0;
y = 0;
BEGIN TRANSACTION;
\[x = x + 1;\]
\[y = y + 2;\]
\[x = y \times y;\]
END TRANSACTION;

Figure 5.22. (a) A transaction. (b)-(d) The log before each statement is executed.

If the transaction succeeds and is committed, a commit record is written to the log, but the data structures do not have to be changed as they have already been updated. If the transaction aborts, the log can be used to back up to the original state. Starting at the end and going backward, each log record is read and the change described in it undone. This action is called a rollback.

Again, this scheme is also seen to work for distributed transactions. In that case, each machine keeps its own log of changes to its local file system. Rolling back in the case of an abort requires that each machine rolls back separately to restore the original files.

5.6.4 Concurrency Control

So far, we have explained the essence of achieving atomicity of transactions. Achieving atomicity (and durability) in the presence of failures is an important topic that we will discuss in Chap. 7, as it is related to more than only transactions. The properties of consistency and isolation are basically handled by properly controlling the execution of concurrent transactions, that is, transactions that are executed at the same time on shared data.

The goal of concurrency control is to allow several transactions to be executed simultaneously, but in such a way that the collection of data items (e.g., files or database records) that is being manipulated, is left in a consistent state. This consistency is achieved by giving transactions access to data items in a specific order whereby the final result is the same as if all transactions had run sequentially.

Concurrency control is best understood in terms of three different managers which are organized in a layered fashion as shown in Fig. 5.23. The bottom layer consists of a data manager that performs the actual read and write operations on data. The data manager is not concerned about which transaction it is performing a read or write. In fact, it knows nothing about transactions.

The middle layer consists of a scheduler and carries the main responsibility for properly controlling concurrency. It determines which transaction is allowed to pass a read or write operation to the data manager and at which time. It does so
by scheduling individual read and write operations in such a way that isolation and consistency of transactions are met. Below, we discuss scheduling based on the use of locks, and scheduling based on the use of timestamps.

The highest layer contains the transaction manager, which is primarily responsible for guaranteeing atomicity of transactions. It processes transaction primitives by transforming them into scheduling requests for the scheduler.

The model shown in Fig. 5-23 can be adopted for the distributed case as shown in Fig. 5-24. Each site has its own scheduler and data manager, together responsible for ensuring that local data remain consistent. Each transaction is handled by a single transaction manager. The latter communicates with the scheduler of individual sites. Depending on the concurrency control algorithm, a scheduler may also communicate with remote data managers. We return to the distribution of concurrency control below.

**Serializability**

The main purpose of concurrency control algorithms is to guarantee that multiple transactions can be executed simultaneously while still being isolated at the same time. This means that the final result should be the same as if the transactions were executed one after the other in some specific order.

In Fig. 5-25(a)-(c) we have three transactions that are executed simultaneously by three separate processes. If they were to be run sequentially, the final value of \( x \) would be 1, 2, or 3, depending upon which one ran last (\( x \) could be a shared variable, a file, or some other kind of entity). In Fig. 5-25(d) we see various orders, called **schedules**, in which they might be interleaved. Schedule 1 is actually serialized. In other words, the transactions run strictly sequentially, so it meets the serializability condition by definition. Schedule 2 is not serialized, but is still legal because it results in a value for \( x \) that could have been achieved by
running the transactions strictly sequentially. The third one is illegal since it sets \( x \)
to 5, something that no sequential order of the transactions could produce. It is up
to the system to ensure that individual operations are interleaved correctly. By
allowing the system the freedom to choose any ordering of the operations it wants
to, provided that it gets the answer correct, we eliminate the need for program-
ners to do their own mutual exclusion, thus simplifying the programming.

\[
\begin{align*}
\text{BEGIN TRANSACTION} & \quad \text{BEGIN TRANSACTION} & \quad \text{BEGIN TRANSACTION} \\
& x = 0; & x = 0; & x = 0; \\
& x = x + 1; & x = x + 2; & x = x + 3; \\
\text{END TRANSACTION} & & \text{END TRANSACTION} & \text{END TRANSACTION}
\end{align*}
\]

(a) (b) (c)

Time →

| Schedule 1 | x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3; | Legal |
| Schedule 2 | x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 2; | Legal |
| Schedule 3 | x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3; | Illegal |

(d)

Figure 5-25. (a)-(c) Three transactions \( T_1, T_2, \) and \( T_3 \). (d) Possible schedules.

To understand schedules and concurrency control, it is not necessary to know
exactly what is being computed. In other words, it does not matter whether the
value of \( x \) is incremented by 2 or 3. What does matter is that the value of \( x \) is being changed. Consequently, we can represent transactions as a series of read and write operations on specific data items. For example, each of the three transactions \( T_1, T_2, \) and \( T_3 \) shown in Fig. 5-25(a)–(c), respectively, can be represented as the series

\[
\text{write}(T_i,x); \text{read}(T_i,x); \text{write}(T_i,x)
\]

The whole idea behind concurrency control is to properly schedule conflicting operations. Two operations conflict if they operate on the same data item, and if at least one of them is a write operation. In a read-write conflict exactly one of the operations is a write. Otherwise, we are dealing with a write-write conflict. Note that it does not matter whether conflicting operations are from the same transaction or from different transactions. It is important to note that two read operations never conflict.

Concurrency control algorithms can generally be classified by looking at the way read and write operations are synchronized. Synchronization can take place either through mutual exclusion mechanisms on shared data (i.e., locking), or explicitly ordering operations using timestamps.

A further distinction can be made between pessimistic and optimistic concurrency control. Fundamental to pessimistic approaches is Murphy's law: if something can go wrong, it will. In pessimistic approaches, operations are synchronized before they are carried out, meaning that conflicts are resolved before they are allowed to happen. In contrast, optimistic approaches are based on the idea that, in general, nothing will go wrong. Operations are therefore simply carried out and synchronization takes place at the end of a transaction. If at that point it turns out that conflicts occurred, one or more transactions are forced to abort. In the following pages, we study two pessimistic and one optimistic approach. An excellent overview of various mechanisms is given in (Bernstein and Goodman, 1981).

Two-Phase Locking

The oldest and most widely used concurrency control algorithm is locking. In the simplest form, when a process needs to read or write a data item as part of a transaction, it requests the scheduler to grant it a lock for that data item. Likewise, when a data item is no longer needed, the scheduler is requested to release the lock. The task of the scheduler is to grant and release locks in such a way that only valid schedules result. In other words, it needs to apply an algorithm that provides only serializable schedules. One such algorithm is two-phase locking.

In two-phase locking (2PL), which is illustrated in Fig. 5-26, the scheduler first acquires all the locks it needs during the growing phase, and then releases them during the shrinking phase. More specifically, the following three rules are obeyed, as explained in (Bernstein et al., 1987):
1. When the scheduler receives an operation $open(T,x)$ from the transaction manager, it tests whether that operation conflicts with any other operation for which it already granted a lock. If there is a conflict, operation $open(T,x)$ is delayed (and thus also transaction $T$). If there is no conflict, the scheduler grants a lock for data item $x$, and passes the operation to the data manager.

2. The scheduler will never release a lock for data item $x$, until the data manager acknowledges it has performed the operation for which the lock was set.

3. Once the scheduler has released a lock on behalf of a transaction $T$, it will never grant another lock on behalf of $T$, no matter for which data item $T$ is requesting a lock. Any attempt by $T$ to acquire another lock is a programming error that aborts $T$.

It can be proven (Eswaran et al., 1976) that if all transactions use two-phase locking, all schedules formed by interleaving them are serializable. This is why two-phase locking is widely used.

In many systems, the shrinking phase does not take place until the transaction has finished running and has either committed or aborted, leading to the release of locks as shown in Fig. 5-27. This policy, called strict two-phase locking, has two main advantages. First, a transaction always reads a value written by a committed transaction; therefore, one never has to abort a transaction because its calculations were based on a data item it should not have seen. Second, all lock acquisitions and releases can be handled by the system without the transaction being aware of them: locks are acquired whenever a data item is to be accessed and released when the transaction has finished. This policy eliminates cascaded aborts: having to undo a committed transaction because it saw a data item it should not have seen.
Both two-phase locking and strict two-phase locking can lead to deadlocks. If two processes each try to acquire the same pair of locks but in the opposite order, a deadlock may result. The usual techniques apply here, such as acquiring all locks in some canonical order to prevent hold-and-wait cycles. Also possible is deadlock detection by maintaining an explicit graph of which process has which locks and wants which locks, and checking the graph for cycles. Finally, when it is known in advance that a lock will never be held longer than \( t \) sec, a timeout scheme can be used: if a lock remains continuously under the same ownership for longer than \( t \) sec, there must be a deadlock.

There are several ways that the basic two-phase locking scheme can be implemented in a distributed system. The assumption is that the data that is operated on are distributed across multiple machines. In centralized 2PL, a single site is responsible for granting and releasing locks. Each transaction manager communicates with this centralized lock manager, from which it receives lock grants. When a lock has been granted, the transaction manager subsequently communicates directly with the data managers. Note that in this scheme, data items may also be replicated possibly across multiple machines. When the operation has completed, the transaction manager returns the lock to the lock manager.

In primary 2PL, each data item is assigned a primary copy. The lock manager on that copy's machine is responsible for granting and releasing locks. Primary 2PL works essentially the same as centralized 2PL, except that locking has been distributed across multiple machines.

Finally, in distributed 2PL, it is assumed that data may be replicated across multiple machines. In contrast to primary 2PL and centralized 2PL, the schedulers on each machine not only take care that locks are granted and released, but also that the operation is forwarded to the (local) data manager. In this sense, distributed 2PL comes much closer to the basic 2PL scheme, but which is now executed at each site where the data reside.

A classical treatment of two-phase locking for database systems and concurrency control in general can be found in (Bernstein et al., 1987).
Pessimistic Timestamp Ordering

A completely different approach to concurrency control is to assign each transaction \( T \) a timestamp \( ts(T) \) at the moment it starts. Using Lamport's algorithm, we can ensure that the timestamps are unique, which is important here. Every operation that is part of a transaction \( T \), is timestamped with \( ts(T) \). Furthermore, every data item \( x \) in the system has a read timestamp \( ts_{RD}(x) \) and a write timestamp \( ts_{WR}(x) \) associated with it. The read timestamp is set to the timestamp of the transaction that most recently read \( x \), whereas the write timestamp is that of the transaction that most recently changed \( x \). Using timestamp ordering, if two operations conflict, the data manager processes the one with the lowest timestamp first.

Now suppose that the scheduler receives an operation \( \text{read}(T,x) \) from transaction \( T \) with timestamp \( ts \), but that \( ts < ts_{WR}(x) \). In other words, it notices that a write operation on \( x \) has been performed after \( T \) started. In that case, transaction \( T \) is simply aborted. On the other hand, if \( ts > ts_{WR}(x) \), it is correct to let the read operation take place. In addition, \( ts_{RD}(x) \) is set to \( \max\{ts, ts_{RD}(x)\} \).

Likewise, assume the scheduler receives a write operation \( \text{write}(T,x) \) as part of transaction \( T \) with timestamp \( ts \). If \( ts < ts_{RD}(x) \), it can only abort transaction \( T \), because the current value of \( x \) has been read by a more recent transaction. Transaction \( T \) is simply too late. On the other hand, if \( ts > ts_{RD}(x) \), it is in order to change the value of \( x \), as no younger transaction has yet read it. Also, \( ts_{WR}(x) \) is set to \( \max\{ts, ts_{WR}(x)\} \).

To better understand timestamp ordering, consider the following example. Imagine that there are three transactions, \( T_1 \), \( T_2 \), and \( T_3 \). \( T_1 \) ran a long time ago, and used every data item needed by \( T_2 \) and \( T_3 \), so all their data items have read and write timestamps set to \( ts(T_1) \). Transactions \( T_2 \) and \( T_3 \) start concurrently, with \( ts(T_2) < ts(T_3) \).

Let us first consider \( T_2 \) writing a data item \( x \). Unless \( T_3 \) has snuck in already and committed, both \( ts_{RD}(x) \) and \( ts_{WR}(x) \) will have been set to \( ts(T_1) \), and thus less than \( ts(T_2) \). In Fig. 5-28(a) and (b) we see that \( ts(T_2) \) is larger than both \( ts_{RD}(x) \) and \( ts_{WR}(x) \) (\( T_3 \) has not already committed), so the write is accepted and done tentatively. It will become permanent when \( T_2 \) commits. \( T_2 \)'s timestamp is now recorded in the data item as a tentative write.

In Fig. 5-28(c) and (d) \( T_2 \) is out of luck. \( T_3 \) has either read (c) or written (d) \( x \) and committed. \( T_2 \)'s transaction is aborted. However, it can apply for a new timestamp and start all over again.

Now look at reads. In Fig. 5-28(e), there is no conflict, so the read can happen immediately. In Fig. 5-28(f), some interloper has gotten in there and is trying to write \( x \). The interloper's timestamp is lower than \( T_2 \)'s, so \( T_2 \) simply waits until the interloper commits, at which time it can read the new file and continue.

In Fig. 5-28(g), \( T_3 \) has changed \( x \) and already committed. Again \( T_2 \) must abort. In Fig. 5-28(h), \( T_3 \) is in the process of changing \( x \), although it has not
committed yet. Still, \( T_2 \) is too late and must abort.

Timestamping has different properties than locking. When a transaction encounters a larger (later) timestamp, it aborts, whereas under the same circumstances with locking it would either wait or be able to proceed immediately. On the other hand, it is deadlock free, which is a big plus.

The basic timestamp ordering has several variants, notably conservative timestamp ordering and multiversion timestamp ordering. Details can be found in (Gray and Reuter, 1993; and Oszu and Valduriez, 1999).

**Optimistic Timestamp Ordering**

A third approach to handling multiple transactions at the same time is **optimistic concurrency control** (Kung and Robinson, 1981). The idea behind this technique is surprisingly simple: just go ahead and do whatever you want to without paying attention to what anybody else is doing. If there is a problem, worry about it later. (Many politicians use this algorithm, too.) In practice, conflicts are relatively rare, so most of the time it works all right.

Although conflicts may be rare, they are not impossible, so some way is needed to handle them. What optimistic concurrency control does is keep track of which data items have been read and written. At the point of committing, it checks all other transactions to see if any of its items have been changed since the transaction started. If so, the transaction is aborted. If not, it is committed.

Optimistic concurrency control fits best with the implementation based on private workspaces. That way, each transaction changes its data privately, without interference from the others. At the end, the new data are either committed or released, leading to a relatively simple and straightforward scheme.
The big advantages of optimistic concurrency control are that it is deadlock free and allows maximum parallelism because no process ever has to wait for a lock. The disadvantage is that sometimes it may fail, in which case the transaction has to be run again. Under conditions of heavy load, the probability of failure may go up substantially, making optimistic concurrency control a poor choice.

As pointed out in (Oszu and Valduriez, 1999), research on optimistic concurrency control has primarily focused on nondistributed systems. In addition, it has hardly been implemented in commercial or prototype systems, making it hard to evaluate it against the other approaches we discussed.

5.7 SUMMARY

Strongly related to communication between processes is the issue of how processes in distributed systems synchronize. Synchronization is all about doing the right thing at the right time. A problem in distributed systems, and computer networks in general, is that there is no notion of a globally shared clock. In other words, processes on different machines have their own idea of what time it is.

There are various ways to synchronize clocks in a distributed system, but all methods are essentially based on exchanging clock values, while taking into account the time it takes to send and receive messages. Variations in communication delays and the way those variations are dealt with, largely determine the accuracy of clock synchronization algorithms.

In many cases, knowing the absolute time is not necessary. What counts is that related events at different processes happen in the correct order. Lamport showed that by introducing a notion of logical clocks, it is possible for a collection of processes to reach global agreement on the correct ordering of events. In essence, each event \( e \), such as sending or receiving a message, is assigned a globally unique logical timestamp \( C(e) \) such that when event \( a \) happened before \( b \), \( C(a) < C(b) \). Lamport timestamps can be extended to vector timestamps: if \( C(a) < C(b) \), we even know that event \( a \) causally preceded \( b \).

Because there is no notion of shared memory in a distributed system, it is often hard to determine exactly what a system’s current state is. Determining the global state of a distributed system can be done by synchronizing all processes so that each collects its own local state, along with the messages that are currently in transit. The synchronization itself can be done without forcing processes to stop and collect their state. Instead, what is called a distributed snapshot, can be collected while the distributed system continues to operate.

Synchronization between processes often requires that one process acts as a coordinator. In those cases where the coordinator is not fixed, it is necessary that processes in a distributed computation decide on who is going to be that coordinator. Such a decision is taken by means of election algorithms. Election algorithms are primarily used in cases where the coordinator can crash.
An important class of synchronization algorithms is that of distributed mutual exclusion. These algorithms ensure that in a distributed collection of processes, at most one process at a time has access to a shared resource. Distributed mutual exclusion can easily be achieved if we make use of a coordinator that keeps track of whose turn it is. Fully distributed algorithms also exist, but have the drawback that they are generally more susceptible to communication and process failures.

Related to mutual exclusion are distributed transactions. A transaction consists of a series of operations on (shared) data, such that the transaction is either carried out completely, or not at all. In addition, a number of transactions can be executed simultaneously so that the overall effect is as if the transactions had been carried out in some arbitrary but sequential order. Finally, a transaction is durable, meaning that if it completes, its effects are permanent.

PROBLEMS

1. Name at least three sources of delay that can be introduced between WWV broadcasting the time and the processors in a distributed system setting their internal clocks.

2. Consider the behavior of two machines in a distributed system. Both have clocks that are supposed to tick 1000 times per millisecond. One of them actually does, but the other ticks only 990 times per millisecond. If UTC updates come in once a minute, what is the maximum clock skew that will occur?

3. Add a new message to Fig. 5-7 that is concurrent with message A, that is, it neither happens before A nor happens after A.

4. To achieve totally-ordered multicasting with Lamport timestamps, is it strictly necessary that each message is acknowledged?

5. Consider a communication layer in which messages are delivered only in the order that they were sent. Give an example in which even this ordering is unnecessarily restrictive.

6. Suppose that two processes detect the demise of the coordinator simultaneously and both decide to hold an election using the bully algorithm. What happens?

7. In Fig. 5-12 we have two ELECTION messages circulating simultaneously. While it does no harm to have two of them, it would be more elegant if one could be killed off. Devise an algorithm for doing this without affecting the operation of the basic election algorithm.

8. Many distributed algorithms require the use of a coordinating process. To what extent can such algorithms actually be considered distributed? Discuss.

9. In the centralized approach to mutual exclusion (Fig. 5-13), upon receiving a message from a process releasing its exclusive access to the critical region it was using, the coordinator normally grants permission to the first process on the queue. Give another possible algorithm for the coordinator.
10. Consider Fig. 5-13 again. Suppose that the coordinator crashes. Does this always bring the system down? If not, under what circumstances does this happen? Is there any way to avoid the problem and make the system able to tolerate coordinator crashes?

11. Ricart and Agrawala's algorithm has the problem that if a process has crashed and does not reply to a request from another process to enter a critical region, the lack of response will be interpreted as denial of permission. We suggested that all requests be answered immediately, to make it easy to detect crashed processes. Are there any circumstances where even this method is insufficient? Discuss.

12. How do the entries in Fig. 5-16 change if we assume that the algorithms can be implemented on a LAN that supports hardware broadcasts?

13. A distributed system may have multiple, independent critical regions. Imagine that process 0 wants to enter critical region A and process 1 wants to enter critical region B. Can Ricart and Agrawala's algorithm lead to deadlocks? Explain your answer.

14. In Fig. 5-17 we saw a way to update an inventory list atomically using magnetic tape. Since a tape can easily be simulated on disk (as a file), why do you think this method is not used any more?

15. In Fig. 5-29(d) three schedules are shown, two legal and one illegal. For the same transactions, give a complete list of all values that x might have at the end, and state which are legal and which are illegal.

16. When a private workspace is used to implement transactions on files, it may happen that a large number of file indices must be copied back to the parent's workspace. How can this be done without introducing race conditions?

17. Give the full algorithm for whether an attempt to lock a file should succeed or fail. Consider both read and write locks, and the possibility that the file was unlocked, read locked, or write locked.

18. Systems that use locking for concurrency control usually distinguish read locks from write locks. What should happen if a process has already acquired a read lock and now wants to change it into a write lock? What about changing a write lock into a read lock?

19. With timestamp ordering in distributed transactions, suppose a write operation \( write(T_1, x) \) can be passed to the data manager, because the only, possibly conflicting operation \( write(T_2, x) \) had a lower timestamp. Why would it make sense to let the scheduler postpone passing \( write(T_1, x) \) until transaction \( T_2 \) finishes?

20. Is optimistic concurrency control more or less restrictive than using timestamps? Why?


22. We have repeatedly said that when a transaction is aborted, the world is restored to its previous state, as though the transaction had never happened. We lied. Give an example where resetting the world is impossible.